Teaching Introductory Programming Using Graphics as a Domain

Doctoral Dissertation submitted to the

Faculty of Informatics of the Università della Svizzera italiana

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

presented by

Luca Chiodini

under the supervision of

Prof. Matthias Hauswirth

August 2025

Dissertation Committee

Carlo Alberto Furia	Università della Svizzera italiana, Switzerland
Marc Langheinrich	Università della Svizzera italiana, Switzerland
Quintin Cutts	University of Glasgow, UK
Mark Guzdial	University of Michigan, USA
Johan Jeuring	Utrecht University, Netherlands

Dissertation accepted on 25 August 2025

Research Advisor	PhD Program Director
Prof. Matthias Hauswirth	**Prof. Walter Binder / Prof. Stefan Wolf**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Luca Chiodini
Lugano, 25 August 2025

To teachers who go above and beyond

Abstract

An increasingly wider and more diverse population is learning to program, with interests and motivations that often differ from traditional ones. Instructors try to cater to these new needs by designing learning experiences that go beyond classic domains and include, for example, multimedia. Graphics, in particular, has emerged as an attractive domain. However, this shift is not exempt from criticisms: programming graphics can be an engaging activity, but it might actually distract from the intended learning goals.

This dissertation aims to show that it is possible to design an approach to teach programming using graphics as a domain, thereby sustaining the engagement, without neglecting fundamental aspects of programming such as abstraction and problem decomposition. We start by reviewing existing approaches to graphics used in introductory programming, highlighting a number of pitfalls. We then present the PyTamaro approach: a Python library with a design that eschews these pitfalls, an unplugged introduction to programming based on the library, and a dedicated web platform that integrates pedagogical features to leverage the strengths of the approach.

The PyTamaro approach is evaluated in a controlled experiment using the popular turtle graphics as a baseline. Both groups reported high engagement. On transfer to questions outside the domain of graphics, we found few differences, despite the fact that the PyTamaro group had practiced on tasks isomorphic to those in the post-test.

We then conducted a case study with five Swiss high school teachers who have adopted the PyTamaro approach to teach programming. The study analyzed why they decided to adopt our approach, examined their teaching materials in depth, and collected the experience of using PyTamaro with their students. In summary, teachers recognized PyTamaro as a novel and engaging approach to graphics, used it to introduce most programming concepts, and emphasized the definition of functions as a means of abstraction. The case study also highlighted certain issues: teachers had to create their own materials, explained problem decomposition only in the domain of graphics, and struggled to reconcile the ideas of (im)mutable variables and constants.

Overall, the PyTamaro approach shows the feasibility of teaching introductory programming in an engaging way, emphasizing abstraction and decomposition. Its current use at three educational levels paves the way for future empirical investigations.

Acknowledgements

This thesis would simply not exist without Matthias Hauswirth. It was Matthias who, at the start of a pandemic, offered me the opportunity to become his doctoral student and successfully persuaded me to accept the offer. It was Matthias who involved me in a unique program to train high school teachers in Switzerland and who allowed me to explore the possibility of creating an educational graphics library to support that program. And it was again Matthias who dedicated so much effort over several years to turn a small library into a major project that spread across Switzerland. His relentless dedication to teaching well is outstanding. His honest approach to research manages to offset my criticisms of academia. Thank you, Matthias, for the guidance, the availability, the freedom, and the inspiration you gave me each day of these years.

I had the honor of spending a semester in Finland during a research visit, which enabled me to carry out an essential experiment for this thesis; the participants were students in a course taught by Kerttu Pollari-Malmi. For the Finnish period I need to thank Juha Sorva and the entire research group led by Lauri Malmi; in particular, Otto Seppälä and Arto Hellas for their assistance in the study. Juha's commitment to teaching shines through his writing, which also inspired parts of this dissertation. The table of contents of this thesis is designed to read as an extended abstract of sorts, and does injustice to Juha's. Thank you and sorry, Juha.

I acknowledge the financial support of the Swiss National Science Foundation: the grant 184689 funded part of my research and the entire Finnish experience.

I am grateful to each member of my dissertation committee—professors Carlo Alberto Furia, Marc Langheinrich, Quintin Cutts, Mark Guzdial, and Johan Jeuring—for accepting the invitation even with their busy schedules, and especially for their thoughtful feedback on an earlier proposal that led to this thesis.

I would not have been able to carry out a relatively large project alone. At different points, members of the research group—Andrea Gallidabino, Igor Moreno Santos, and Joey Bevilacqua—helped each with their competencies and all with a laugh. The stronger connections to programming language theory are the merit of Igor. Even with some philosophical disagreements, our collaboration in these years has been nothing short of remarkable. Many Bachelor's and Master's students also contributed to my

project. The precise attribution is within the dissertation; this serves as a collective expression of gratitude.

The project described in this dissertation would not make sense without teachers. I will refrain from making names here, in part because the list would be too long (and that is simply amazing!) with the risk of forgetting someone, in part because some of them went the extra mile, agreed to be scrutinized in a case study, and should remain anonymous to the extent that is possible. Working together with so many teachers who voluntarily decided to adopt an innovation and bear the extra load that comes with it has been one of the best experiences during these years. May this appreciation also extend to some great teachers who shaped me in elementary, middle, and high school. This thesis also exists because of their passion.

I feel grateful for my parents. They cannot fathom what I have been "studying" for all these years of my life, but nonetheless allowed me to pursue this long journey without having to worry about anything other than studying. That is a privilege, and they deserve praise. *Grazie.*

My sister and my friends helped me stay balanced and offered joyful distractions that are essential to endure a demanding journey. The existence of Mikko tempers my nihilism. Thank you, all.

Contents

Part I

Prologue

Chapter 1

Here Is an Introduction to This Thesis

This first chapter introduces the context of this dissertation, formulates the thesis statement, and describes the structure of this document.

1.1 More and more people are learning to program

Recent years have witnessed a surge in the number of people who are learning to program, including historically underrepresented populations [159]. The typical student of a programming course used to be primarily someone enrolled in a computer science degree program, but this is rapidly changing. A course that includes some form of programming is nowadays part of almost every university-level degree, and increasingly also of high school curricula. As a locally relevant example, in 2018 the Swiss Federal Council accepted a new regulation by the Conference of Cantonal Ministers of Education that introduces informatics as a required subject in all high schools[1] in Switzerland, starting from the 2022/2023 school year [251].

An immediate consequence of this trend of increasing student numbers is that more learners require more teaching resources. For example, this may materialize in the need for training and hiring more computer science teachers, or in the need of universities to scale up the number of teaching assistants or parallel sections (i.e., additional offerings of the same course) to accommodate larger classes.

Although these practical issues cannot be discounted, the phenomenon also has deeper implications that deserve further investigation. There are qualitative differences between the traditional population of a programming course and the new one. First, there is a difference in terms of *interest*: many programming courses are now

---

[1]This dissertation primarily uses the terms "high school" and "informatics", but some readers may find respectively the terms "upper-secondary education" and "computer science" more standard. Within this work, the terms are interchangeable.

aimed at people who have not chosen to study the subject but have been "forced to". Their motivation to study computer science, especially programming, often starts at a particularly low level, a factor known for being detrimental to learning [210]. Second, the new population might call for different *learning objectives*, i.e., what a learner who has profitably attended the course should know. The vast majority of this new audience will not pursue a career in computer science: is there nonetheless something worth learning for them? The question leads to extensive discussions on the essence of computer science that goes beyond vocational needs and is still important for every person. The ACM Computer Science Curricula [156] and Wing's seminal paper on "computational thinking" [281] offer complementary perspectives on the matter.

Guzdial notes that computing, of which programming is a key component, has not been introduced in schools for the reason of training future programmers. After all, that is only one of the possible careers: "the vocational argument is a weak reason to teach everyone computing—not everyone needs to be a professional programmer" [106].

Turing Laureate Peter Naur offered a profound argument for teaching informatics to the broad population, as quoted in a passage by Caspersen [40]:

> Once informatics has become well established in general education, the mystery surrounding computers in many people's perceptions will vanish. This must be regarded as perhaps the most important reason for promoting the understanding of informatics. This is a necessary condition for humankind's supremacy over computers and for ensuring that their use do not become a matter for a small group of experts, but become a usual democratic matter, and thus through the democratic system will lie where it should, with all of us.

This dissertation reflects this view, placing a special emphasis on programming as one of the central intellectual activities within the field of computing. Carrying out the activity of programming in the first person helps to reduce the aura of mystery (the "magic-ness") that surrounds computers, demystifying them.

This goal becomes even more important as computers acquire increasingly more sophisticated capabilities, as proven by the recent advances in "artificial intelligence" technologies. Companies whose existential purpose is to sell these technologies to customers actively try to muddy the waters with evocative terminology for the masses. A healthy relationship with these systems should instead be one in which the citizen is in a dominating—and not subjugated—position. And given that students are the citizens of tomorrow, teaching programming is essential for democracy.

1.2 Programming is often taught using mathematics as a domain

How is programming taught to novices? When someone who knows at least a little bit of programming is asked about their first program, the answer is almost always the same: "My first program printed 'Hello, World!' on a screen". The unanimous response shows how influential the programming language C and its prime textbook have been: the "Hello, World!" program is featured on the first chapter of *The C programming language* textbook [222]. At the same time, the undefeated popularity of the example almost half a century later is a sign that, once ingrained, teaching practices are remarkably hard to change.

Historically, the programs used in introductory courses are drawn from the domain of mathematics. On the one hand, this traces back to the origin of computing and computers [269]. On the other hand, elementary mathematics is included in every school curriculum and, as a consequence, it is part of the background of all students. These two factors arguably led to programming being taught using mathematics as a privileged domain.

As a common example, one can think about the explanation of a sorting algorithm: students are familiar with integer numbers, which seem to work well as an unobtrusive environment in which to apply the algorithm. As a slightly more advanced reference, one can look at the first chapter of the influential textbook *Structure and Interpretation of Computer Programs* [3]. The first two sections are already permeated with examples taken from the domain of mathematics. The textbook features, among others, Newton's method to find the square roots, the factorial and Fibonacci functions, an algorithm for fast exponentiation, Euclid's algorithm to compute the greatest common divisor, and primality tests.

1.3 Other domains can be used to teach programming

When beginners are not proficient with relatively sophisticated mathematical concepts, programming exercises in the domain of mathematics are mostly confined to basic arithmetic operations. These rather uninteresting exercises are still in widespread use, despite having been criticized by some as "tedious and dull" [105].

Papert and Solomon already pointed this out back in 1971: "Why should computers in schools be confined to computing the sum of the squares of the first twenty odd numbers?" [196]. As a remedy, they proposed that students could use the computer "to produce some actions". They built a robot with the semblance of a turtle and designed a language to give commands to it, such as "move forward" or "turn left". The same

idea was then replicated virtually on a screen, enabling students to program without the need for a physical robot. The on-screen turtle conceptually carries a pen and leaves a trace of its movements, effectively producing a drawing.

Since then, the idea of giving commands to a "turtle" to draw graphics has become popular in introductory programming. Nowadays, a "turtle graphics" library is available for almost every programming language (e.g., Python's `turtle` [216]). Acknowledging the attractiveness and the motivational benefits of graphics and other media, educators started to leverage these domains as interesting areas for teaching programming.

Beyond turtle graphics, many other approaches have been used to teach programming using graphics. For example, Guzdial designed a course for students graduating in majors other than computer science that uses images and sounds [105]. Schanzer et al. developed Bootstrap, a curriculum for schools in which students learn programming also by writing programs that produce graphics, which are then used to create a game [231].

1.4 A domain is not necessarily a context, which some criticize

The previous sections use the term *domain* to describe the application area in which the programming tasks are situated. The term *context* is also widely used in the education literature. Sometimes, authors seem to use context as a loose synonym for domain. One could then speak of a "contextualized approach to programming using graphics". Other times, context is used with a richer meaning, drawing from contextualism in philosophy and instructional design [94]. A course could be said to use, say, graphics as a context when programming examples use graphics consistently and coherently, problems from the application area of graphics are used to motivate the introduction of new concepts, students' projects are based on graphics, students meet external experts from the domain, and more [62].

Contextualized approaches are praised for being more engaging and helping with retention, but they are also sometimes criticized for providing a narrow view of computer science, confined to one application area, and adding distractions that reduce the time spent on the actual content [107]. Conversely, decontextualized approaches have been lauded for their generality and flexibility [169].

Guzdial tried to clarify the meaning of a decontextualized approach: "[...] one where we (are) teaching computer science in a way that can be easily applied to any application area. When we teach students to swap two variables, or to implement a binary search, or to sort an array, we are teaching decontextualized knowledge about

computer science" [107]. It is important, however, not to conflate simply using examples situated in a domain with adopting a full-fledged contextualized approach. Even some of the approaches described as decontextualized are still embedded in a domain, albeit minimally. A classical example from a course on algorithms could be sorting an array of numbers: if "decontextualized" is used to mean "detached from any domain", then this programming task would still not qualify as such, as it requires an understanding of numbers (including their representation, which can in turn affect the complexity in time and space) and of ways to compare them. As DeClue aptly points out: "learning always takes place in a context, whether that context is named and studied [...] or the context is ignored" [70].

The benefits of a decontextualized or, better, domain-less approach rest on the assumption that learners can acquire knowledge independently from any domain and then *transfer* it whenever needed to the domain at hand [107]. However, teaching from the beginning without any concrete domain is extremely challenging, as it immediately demands abstract reasoning. Indeed, pedagogies typically interleave content with different degrees of context, using a range of examples that vary from very concrete and embedded in a domain to very abstract and domain-less [177].

1.5 This thesis claims that graphics can be a suitable domain

Viewed under this lens, many existing pedagogies are already using a domain to teach introductory programming: mathematics. This thesis claims that:

> It is feasible to teach introductory programming using graphics as a domain, in an engaging way, emphasizing abstraction and problem decomposition.

Abstraction and problem decomposition are two of the central "skills", "facets", or "components" that many use to characterize "computational thinking" [238]. As we shall see in the next chapter, they are essential for programming but often neglected in introductory approaches.

1.6 This thesis presents six main contributions

This dissertation is organized into parts and presents six main contributions towards the thesis claimed above. Table 1.1 schematizes the contributions as six high-level questions and answers. Refined research questions will be presented throughout the relevant chapters of the dissertation.

Question	Answer
What pitfalls are there in existing approaches?	Three groups of pitfalls (Chapter 4).
How can a graphics library be designed to avoid the pitfalls?	Design of the PyTamaro library and teaching approach (Chapter 5).
How can one teach the approach initially without computers?	Unplugged activities with TamaroCards (Chapter 6).
How can pedagogical software tools be designed to further support the approach?	Design of the web platform (Chapter 7), including the Toolbox (Chapter 8) and the Judicious documentation (Chapter 9).
How do learning and engagement compare between the new approach and an established one?	Randomized controlled experiment (Chapter 10).
How are teachers using the new approach in practice?	Multiple-case study (Chapter 11).

Table 1.1. Mapping high-level questions to their answers in dissertation chapters.

In Part II, we focus on the state of the art. Chapter 2 discusses some problems with how programming is taught, highlights the importance of abstraction and decomposition, and reviews the literature on why graphics has potential as an engaging domain. Chapter 3 reviews the different existing approaches to using graphics as a domain for teaching programming. Chapter 4 presents the first main contribution: it critically analyzes existing approaches in light of the above goal, revealing multiple pitfalls.

In Part III, we present the PyTamaro approach with three main contributions:

- Chapter 5 presents the design of a Python library, implementing an approach that eschews the pitfalls.

- Chapter 6 illustrates TamaroCards, a tangible notional machine to introduce programming with PyTamaro through unplugged activities.

- Chapters 7 to 9 present a web platform to showcase and work on PyTamaro-based programming activities, with dedicated features that leverage the strengths

of the minimalist library to promote abstraction and provide a gradual documentation system.

Part IV presents two empirical investigations of the PyTamaro approach:

- Chapter 10 reports on a randomized controlled experiment that evaluates the proposed design, using the popular turtle graphics approach as a baseline.

- Chapter 11 describes a multiple-case study with five Swiss high school teachers to analyze why they decided to adopt our approach, how they integrated it into their teaching materials, and their experience with students after using it.

Part V wraps up this dissertation: Chapters 12 and 13 respectively conclude this dissertation and outline directions for future work. Part VI is just for the appendices.

Although this dissertation reports on an approach to teaching introductory programming, it is not intended as a programming tutorial and assumes prior familiarity with programming.

1.7   This thesis subscribes to pragmatism and uses multiple research methods

Quantitative and qualitative research paradigms are often portrayed as mutually exclusive, and purists from both sides declare the other side as conducting invalid research.

Coming from a positivist philosophy of science, "Quantitative purists believe that social observations should be treated as entities in much the same way that physical scientists treat physical phenomena. Further, they contend that the observer is separate from the entities that are subject to observation. Quantitative purists maintain that social science inquiry should be objective. [...] According to this school of thought, educational researchers should eliminate their biases, remain emotionally detached and uninvolved with the objects of study, and test or empirically justify their stated hypotheses." [136]

Qualitative purists reject positivism and "argue for the superiority of constructivism, idealism, relativism, humanism, hermeneutics, and, sometimes, postmodernism. These purists contend [...] that it is impossible to differentiate fully causes and effects, that logic flows from specific to general (e.g., explanations are generated inductively from the data), and that knower and known cannot be separated because the subjective knower is the only source of reality" (ibid.).

The debate is heated to the point that some argue for declaring a complete incompatibility between the two views. This "incompatibility thesis" stems from the belief

that the underlying epistemological paradigms are incompatible, and therefore the re-
search methods are incompatible [127].

This tension is present not only in science broadly, but also within computing edu-
cation research in particular. Brown and Guzdial [33] recently discussed the apparent
dichotomy in programming education research between the quantitative analysis of
large datasets ("big data") and the in-depth qualitative analysis of a small number of
participants in a study ("rich data"). The two different types of data analysis, each with
passionate supporters and vocal detractors, are sometimes viewed as incompatible, but
they offer valuable complementary perspectives on phenomena in programming edu-
cation.

Howe [127] and Johnson and Onwuegbuzie [136] argue in favor of pragmatism [204]
as a philosophy of science. Pragmatism views knowledge "as being both constructed
and based on the reality of the world we experience and live in", "endorses fallibilism"
in that "current beliefs and research conclusions are rarely, if ever, viewed as perfect,
certain, or absolute" and regards theories as true "based on how well they currently
work" [136].

Pragmatism supports using multiple research methods. This is often referred to
as "mixed methods research". Mixed methods research could be viewed as a third,
alternative paradigm in educational research, moving beyond quantitative and quali-
tative research and recognizing both of them as useful to answer research questions.
In practice, things are not so categorical: "if you visualize a continuum with qualitative
research anchored at one pole and quantitative research anchored at the other, mixed
methods research covers the large set of points in the middle area" [136].

The research approach used in this dissertation subscribes to pragmatism as a phi-
losophy of science and embraces multiple research methods. Appropriate justifications
for the use of each method are provided when discussing how each part of the data
collected in empirical studies is analyzed.

In addition to empirical studies, this dissertation critically examines prior work
in programming education and introduces new pedagogical software tools. For these
parts, our arguments are grounded in prior research in computing education (e.g., on
the struggles faced by novices in learning to program) and in programming languages
theory (e.g., on claims of how certain language features are more complex than others).

Dewey [73], a prominent pragmatist, suggested substituting "truth" with the ex-
pression "warranted assertibility". The notion of a "warrant" comes from "the legal
sphere, where a warrant is an authorization to take some action, e.g., to conduct a
search; to obtain a warrant the investigator has to convince a judge that there is suffi-
cient evidence to make the search *reasonable*" [207].

Echoing Sorva, for this thesis "I am satisfied to say that our results lead to warranted
beliefs rather than truths" [244]. The scientific community is the ultimate judge of the

legitimacy of this research, and will eventually complement or replace it with new theories and findings.

1.8 Parts of this thesis are based on published work

Some chapters of this dissertation are based on peer-reviewed, published work. Specifically:

- Chapters 3 to 5 are a revised and significantly expanded version of: Luca Chiodini et al. "Teaching Programming with Graphics: Pitfalls and a Solution". In: *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E*. SPLASH-E 2023. New York, NY, USA: ACM, Oct. 2023, pp. 1–12. ISBN: 979-8-4007-0390-4. DOI: 10.1145/3622780.3623644 [55].

- Chapter 8 has been published as: Luca Chiodini et al. "The Toolbox of Functions: Teaching Code Reuse in Schools". In: *Proceedings of the 6th European Conference on Software Engineering Education*. ECSEE '25. New York, NY, USA: Association for Computing Machinery, June 2025, pp. 185–189. ISBN: 979-8-4007-1282-1. DOI: 10.1145/3723010.3723029 [49].

- Chapter 9 borrows from and extends: Luca Chiodini et al. "Judicious: API Documentation for Novices". In: *Proceedings of the 2024 ACM SIGPLAN International Symposium on SPLASH-E*. SPLASH-E 2024. New York, NY, USA: Association for Computing Machinery, 2024, pp. 1–9. ISBN: 979-8-4007-1216-6. DOI: 10.1145/3689493.3689987 [54].

- Chapter 10 presents a study that first appeared in: Luca Chiodini et al. "Two Approaches for Programming Education in the Domain of Graphics: An Experiment". In: *The Art, Science, and Engineering of Programming* 10.1 (Feb. 2025), 14:1–14:48. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2025/10/14 [56].

Additionally, during the doctoral program, the author of this dissertation has authored other publications in the area of teaching introductory programming. On misconceptions, he has worked on an inventory of programming language misconceptions [52], a self-check tool for teachers on these misconceptions [51], and a study of misconceptions among high school teachers [48]. On expressions, he investigated their role while learning Java [53] and the use of a notional machine to assess the understanding of expressions [26]. Finally, he has worked on methodological issues with ad hoc instruments in experiments to assess programming competence [50].

1.9 Several people deserve to be acknowledged for their contributions

In addition to the original text written for this dissertation, all the publications that form the base of this document have been led by the author of this dissertation. Nonetheless, due acknowledgment should be given to the valuable work of each publication's co-authors, who also contributed to the writing: Matthias Hauswirth, Juha Sorva, Arto Hellas, Otto Seppälä, and Joey Bevilacqua.

The initial version of TamaroCards (Chapter 6) was created by Matthias Hauswirth and Igor Moreno Santos. The middle school curriculum using TamaroCards described in Section 6.7 is a collaboration with Rahel Ehinger, who enthusiastically taught it together with Davide Frova.

The Hour of Code curriculum, hosted on the PyTamaro Web platform and described in Section 7.7, was authored by a team of students: Agnese Zamboni, Davide Frova, Jamila Oubenali, and Giorgia Lillo.

This dissertation also describes a number of pedagogical software tools. The author of the thesis has led the work and developed all the main parts, but several contributions deserve explicit acknowledgment:

- The PyTamaro library (Chapter 5) benefits from improvements by Matthias Hauswirth, Davide Frova, Fabio Marchesi, Peiyu Liu, and Joey Bevilacqua. The French localization is due to the work of Arnaud Fauconnet and Romain Edelman.

- The PyTamaro Web platform (Chapter 7) has also been improved by Matthias Hauswirth, Joey Bevilacqua, and Davide Frova. Alen Sugimoto worked on the initial implementation of curricula. Raffaele Perri prototyped a feature to show the estimated duration of an activity, Giovanni Elisei prototyped an automatic system to generate multilingual cards, and Anuj Kumar prototyped code comprehension questions. Alessandra Sasanelli and Amedeo Zappulla explored how to analyze the code collected on the web platform. Nathan Coquerel developed a live environment for TamaroCards. Igor Moreno Santos provided valuable feedback throughout.

- An early prototype of the Judicious documentation system (Chapter 9) was developed by Simone Piatti.

Part II

Teaching Introductory Programming With Graphics

Chapter 2

Teaching Introductory Programming Is Challenging

This chapter describes some of the key challenges faced by teachers of introductory programming courses, motivates the need for simplicity in the choice of programming language, and justifies the focus on abstraction and decomposition. The last two sections introduce graphics as a domain that can address one of the challenges: how to engage learners.

2.1   Learning to program is difficult for many

Programming involves constructing programs using a formal language, known as a programming language.

The knowledge required for programming can be divided into three levels, usually referred to as syntactic, conceptual, and strategic knowledge [19]. At the lower level, syntactic knowledge refers to the syntax of a specific programming language. Java, for example, requires a semicolon to terminate each statement. Conceptual knowledge refers to a model of the computer, to understand which actions are possible and happen after specific commands. For example, an assignment statement evaluates the expression on the right-hand side of the assignment and stores the resulting value at a memory location. Strategic knowledge involves techniques that leverage syntactic and conceptual knowledge to solve problems. For instance, it is necessary to devise a plan to compose simple actions to solve a problem such as computing the average of a set of grades. In some sense, this kind of knowledge is the specialization to programming of general problem-solving skills [180].

When considering the activity of programming, one usually thinks about the highest of these three levels: strategic knowledge. However, writing a correct program

that can be executed requires mastering all three kinds of knowledge, which are inter-related [19]. Acquiring this knowledge to use it fruitfully is challenging for many.

Guzdial [109] provides an account of how expectations to teach students how to program consistently fall short. A famous problem used in many experiments is the so-called "rainfall problem", proposed in 1983 by Soloway et al. [242]. It is a relatively simple programming task that asks to repeatedly read in numbers representing rainfall measures from the user until a sentinel value terminates the sequence, and then compute and print the average rainfall (excluding that final sentinel value). In the study, conducted with students at an elite university, the fraction of correct solutions in the various experimental groups was as low as 14 %.

Since then, the study has been repeated in multiple universities, in multiple countries, and with different programming languages. Results were similarly disappointing [179] and became emblematic of how students worldwide are failing to learn programming in any decent way. More recently, some studies painted a less dramatic picture. The emphasis on problem decomposition typical of the "functional programming" paradigm appears to lead more students to successfully complete the task [91]. The different variants of how the problem has been formulated, the focus on corner cases, and the differences in what has actually been taught in the introductory courses, may also have contributed to an overemphasis of the extremely poor results on this specific problem [235].

However, other pieces of evidence rule out the hypothesis that difficulties in programming are a mere hallucination of some researchers. The "attrition rate", i.e., the percentage of students who enroll in an introductory programming course and fail, is depressingly high. Beaubouef and Mason [20] anecdotally report attrition rates among first-year computer science students as high as 30–40 % at many universities. Bennedsen and Caspersen [25] surveyed instructors at various institutions, finding large differences among courses, with a mean of 33 % of students not succeeding. A systematic review of the literature conducted by Watson and Li [271] confirmed this number: the analysis of 161 introductory programming courses from 51 institutions in 15 different countries revealed a pass rate of 67.7 %.

2.2 Programming misconceptions abound

Several threads of research have identified *misconceptions* held by novice programmers. Documenting which incorrect conceptions beginners hold can help programming teachers better support their students, ultimately lowering the failure rates reported above.

The knowledge of a teacher can be divided into content knowledge (knowledge

about the subject taught) and pedagogical knowledge (general knowledge about how to teach) [237]. Misconceptions are one part of a teacher's "pedagogical content knowledge". This category of knowledge includes "ways of representing and formulating the subject that make it comprehensible to others [...]" and "an understanding of what makes the learning of specific topics easy or difficult" (ibid.), such as knowing which pre-conceptions—often misconceptions—students bring with them. An awareness of the misconceptions can help teachers recognize the need and develop instructional strategies to explicitly confront them, with the ultimate goal of overcoming them.

Back in 1986, Du Boulay [77] summarized a number of misconceptions leading to errors that students face frequently. Issues included lacking a model of the execution environment (i.e., the "computer" or the "machine") that one is programming for, poor analogies that soon become incoherent, error messages that are impossible to understand at the beginner's level, and misunderstanding of how specific constructs of certain programming languages (such as an assignment statement) work.

In the same year, Pea [201] proposed a single root cause that would explain many bugs encountered by the students. He termed it "superbug": "the idea that there is a hidden mind somewhere in the programming language that has intelligent, interpretative powers". Despite instructors pointing this out at the beginning of their courses that computers require well-specified instructions to perform any task, in their programming, students behave "as if the programming language is more than mechanistic" [77].

Sorva [244, Appendix A] collected more than a hundred misconceptions from over a dozen specific studies. As Sorva himself admits, the catalog lumps together misconceptions at different levels of abstraction. The original researchers of the studies also used terms such as difficulties, mistakes, bugs, misunderstandings.

In 2017, Qian and Lehman [217] conducted a literature review on the topic of programming misconceptions in introductory programming. They proposed a definition of misconceptions as "errors in conceptual understanding", and structured their analysis of the difficulties dividing them according to whether they pertain to syntactic, conceptual, or strategic knowledge.

More recently, Chiodini et al. [52] proposed a narrow definition for a subset of programming misconceptions, which they termed "programming language misconceptions" and defined as "statements that can be disproved by reasoning entirely based on the syntax and/or semantics of a programming language". They published an online inventory of misconceptions that are divided by programming language to acknowledge that some claims that are wrong in one language may be correct in another. The inventory contains hundreds of misconceptions observed in students, linking to many of the new studies that continue to provide corroborating evidence: misconceptions abound in beginner programmers.

2.3   Educators have long searched for the next simpler programming language

Given that programs need to be expressed using a programming language, programming teachers have been placing great expectations on the advent of new languages. The right programming language could perhaps avoid an entire class of problems and make it easier for beginners to learn how to program without all the difficulties described in the previous two sections.

In part, this desire has been met by programming languages and programming techniques that operate at increasingly higher levels of abstraction.

It is not the intention of this short section to provide a comprehensive account of the history of programming languages, a topic well worth a book of its own [276]. Instead, we will briefly review the evolution of languages that have been adopted for introductory programming.

Historically, programming was done at a level close to the hardware, with a family of languages normally grouped under the generic term "assembly language". Instructions in these languages closely correspond to instructions the underlying machine can execute. Programmers need to understand the architecture of the machine they are programming for: as an example, one needs to be aware of the size of the machine's memory registers.

In the 1950s, a team led by Backus introduced Fortran [14], a language that enabled programmers to abstract away from the machine language and write code using mathematical formulas and control structures like loops and conditional statements. The language was compiled by a "translator" (which would now be called a compiler) that was able to generate optimized assembly code. Until the 1970s, Fortran was popular as a language for introductory programming. As late as 1971, Ralston [218] noted that "Fortran is a quite adequate vehicle for the student to use in writing programs" and can serve as the language for the first course in programming.

A significant shift occurred in the late 1960s and the 1970s, with calls from influential computer scientists to transition to "structured programming" [67]. This style of programming emphasizes disciplined use of higher-level control structures such as selection and repetition, code blocks, and modular functions, prohibiting arbitrary jumps to specific sections of the program code [74].

ALGOL [13], a language first introduced in 1960 and with several evolutions (e.g., ALGOL 68), embodied the principles of structured programming and has also been used for educational purposes [193]. Pascal [132] and PL/I [124] have also been widely used to teach novice programmers.

The C programming language [141] gained popularity in the 1980s. C was devel-

oped to implement parts of the Unix operating system. It offers control structures to achieve structured programming, while still allowing programmers to manage memory manually. A survey among universities in the USA in 1995 showed that, with the exception of Pascal, C was the most common language for the introductory programming course [163].

The trend of object-oriented programming led to the creation and adoption of C++ and Java, two programming languages that dominated introductory programming courses since the late 1990s [126] until recently.

A separate thread of high-level programming languages developed starting from Lisp [178], a language that emphasized functions and was influenced by Church's lambda calculus [57]. Scheme, one of Lisp's dialects, has been used as an introductory programming language [83], also due to the influential *Structure and Interpretation of Computer Programs* textbook [3].

2.4 Python is increasingly popular for introductory programming

The changes in which language to choose for the introductory programming course were only partially motivated by the newer language being "better" in some objective sense than the one previously used.

In some sense, one could argue that fundamental programming concepts can be expressed in many languages, and thus the choice of which language to use is at best of secondary importance, if not irrelevant. However, one of the key concerns of instructors is attending to the *authenticity* factor.

Authenticity is not confined to the programming language, as many other factors contribute to it, such as which problems learners are required to solve and in which domain. However, the programming language is certainly one of the key and most visible factors.

Learners are more likely to be interested in learning a subject if they consider it *relevant* (see Albrecht and Karabenick [8] for a review of studies on relevance, and a critical discussion of some objections). Many students, especially those who deliberately choose to study computer science at a university, recognize programming as a skill that will be necessary in their future careers. Industries gradually update the programming languages in use, and with a cascading effect, this has repercussions on the programming language taught in universities.

At the time of writing this dissertation, the TIOBE index[1], an indicator of the "popu-

---

[1] https://www.tiobe.com/tiobe-index/

larity" of programming languages, reports Python as being the most popular language, with a large gap to the next most common ones. GitHub, a platform used by developers to host programming code, reports that in 2024 Python became the most popular programming language for projects hosted on GitHub [245]. The language enjoys a vast set of libraries that enable its effective use in many domains, including "artificial intelligence" with its recent expansion.

The industrial relevance, the vast choice of available libraries, and the alleged simplicity—more a myth than a fact, as we discuss below—led to Python becoming particularly common in introductory programming. An update in 2021 to the survey among universities in the USA showed how Python and Java are the two most common languages used for introductory programming, with the popularity of Python growing significantly [239].

The popularity of Python is also spreading at educational levels that come before the university. Curricula, such as the framework curriculum for informatics in Swiss high schools, usually do not prescribe a specific programming language. Virtually all Swiss high school teachers, however, use Python as the common language for the mandatory programming part. Other languages are relegated to more advanced and specific usages, such as JavaScript to program for the web.

2.5   But Python is not a simple language

One of the most commonly trumpeted arguments in favor of Python's adoption is its supposed "intuitiveness" or "minimal syntax". By contrast, Java has a reputation for being verbose and less ideal for introductory programming.

A simple 'Hello, World!' example, which is still the choice for many teachers as the very first program, requires creating a class, declaring inside it a `main` method, and calling the `println` method on the `out` object, a static field in the `System` class. Novices are usually not overwhelmed with this complex explanation from the beginning: teachers usually hand out code that students need to accept without fully understanding it. In this sense, it is correct to claim that an 'Hello, World!' example in Python does not require any of that.

Developments in recent years have weakened this argument. First, Java has introduced a Read-Eval-Print-Loop (REPL) called `jshell` [85], which can be used to interactively evaluate expressions, without requiring I/O. This interactive shell can be exploited in pedagogical settings [212]. Second, after some years of experimentation, Java 25 introduced "compact source files" and "instance main methods" [133], two new features of the language that explicitly address the pain points described above, with beginners in mind.

Setting this opinionated debate aside, we can analyze whether claims of simplicity hold by reviewing together a Python program that solves a problem commonly used in typical introductory programming. Our task, dubbed "Even or odd", is to ask the user for a number and print whether the number entered is even or odd.

There are many styles in which this program can be written: Listing 1 shows a reasonable Python solution for the problem.

```python
n = int(input("Enter a number: "))
if n % 2 == 0:
    print(n, "is even")
else:
    print(n, "is odd")
```

Listing 1. A Python program to print whether a number entered by the user is even or odd.

For the beginner programmer, writing this five-line program is often much harder than people with some programming experience would expect. Teachers may not realize this due to the expert blind spot [189]. After all, "it reads like English". A caricature explanation, attempting to include Python's keywords and functions, may go along these lines:

> It's easy. We start from the beginning. We ask the user as an input to enter a number, we turn it into an int and save it in the variable n. If n divided by 2 has remainder zero, we print that n is even, else we print that n is odd.

This deceptively short program, however, involves several language features:

- Function calls. There is a specific syntax necessary to call a function: the function name must be followed by a pair of parentheses, with arguments in it. The program uses functions with one and two arguments. Students need to learn that arguments must be separated by commas. And even if it is not apparent from the source code, the `print` function supports an arbitrary number of arguments.

- Nesting of calls. The call to `input` is nested inside the call to `int`, and students need to handle nesting (a general property of expressions, here in the context of function calls).

- Data types. Even though the program does not feature any explicit type annotations, it requires an understanding that the `input` function returns a value of

type `str`, which must be converted to `int` if one later needs to manipulate it as a number. The `print` function, however, supports displaying values of arbitrary type (in the example, an integer and a string).

- Literals. The program uses both string and integer literals. Python's syntax requires string literals to be enclosed in quotes. The string literal used as an argument for `input` ends with a space, which will also be printed because it is part of the literal.

- Assignments and variables. The program assigns the result of evaluating an expression on the right-hand side of the = to the variable `n` on the left.

- `if` statement. The program uses an `if`/`else` statement to take a decision. The `if` statement requires specifying a boolean condition, which must be followed by a colon, and the same is true after the `else` keyword. The code to be executed in each of the two cases must be indented: each line needs to be prefixed by a number of spaces. Unlike in other parts of the program, such as after the comma used to separate arguments, spaces here matter and are essential to write a correct program. In Python, indentation is not just a convention, but is part of the required syntax.

- Operators. The condition in the program uses the % operator to compute the remainder of the division, using a symbol that is unfamiliar to most non-programmers, or worse, one they associate with a different meaning. The comparison between the result and `0` is achieved using the equality operator ==, a sequence of two equal signs.

The simple logic ("algorithm") to determine whether a number is even or odd, i.e., checking if the remainder of the division by two yields zero, does not stand out in this program. It is mixed with operations to perform input and output, with the former being a blocking operation that pauses the program waiting for external input. This also makes it harder to explain that programs are simply executed by the computer one instruction at a time, proceeding automatically line after line.

The program we just discussed was only intended to serve as a basic counterpoint to some popular and yet unsubstantiated claims surrounding the simplicity of Python. Research thoroughly demonstrates this claim beyond the simple didactic example discussed here.

Politz et al. [211] developed a formal semantics of Python, and in doing so revealed a number of complex aspects of the language. Issues related to scope, in particular, affect many aspects across the entire programming language, including those that ought to be reasonably considered orthogonal to it.

Johnson et al. [134] showed that students exhibit a significant number of misconceptions in Python, even about basic language constructs. Mutation, sharing, and overloading are frequently used in Python and thus also by beginners. Sometimes, these complex concepts "hide" behind seemingly innocuous programs, such as the += operator used on lists, which are mutable by default. Programs written by novices contain subtle bugs, which are hard for instructors to explain, and hinder the acquisition of the fundamental concepts. The pervasiveness of objects and references is also a common source of errors [183].

Finally, it does not help the discourse the fact that "simple" and "easy" are used almost interchangeably in casual conversations. Hickey [120], among others, argues for a clear distinction between the two terms and for favoring simplicity (from the Latin *simplex*, the antonym of complex) over ease. The quality of being simple, such as a programming language that does not intertwine (*complect*) unrelated features, should be preferred to the quality of being easy, a subjective judgment based on familiarity. A language we have spent a lot of time with may not be objectively simple, but will appear easy—to us.

2.6   A sublanguage focused on expressions is a sensible starting point

The last two sections established that programming languages such as Python, which are used by professionals, have the appeal of authenticity, but often also the downside of complexity in terms of the number of language features and their interactions.

One way to tackle this complexity is to opt out of professional programming languages and work with languages developed with education in mind. Two examples are Logo [1] and Scratch [221].

Another way is to explicitly define a coherent subset of a professional programming language so that novices do not have to deal with the complexity of the full language. It is usually sensible to define not only one subset, but a sequence of subsets, so that students can graduate from one subset to the next one, gradually adding complexity until they reach the full language (or the desired level). Steele applied this principle of *language growth* to English in a remarkable talk at OOPSLA [246].

A subset of a programming language is often called a *sublanguage*. Computing education has a long tradition of creating sublanguages, which have been defined for many programming languages.

Holt and Wortman [125] created teaching versions for PL/I; Pagan [193] did the same for Algol 68. The *How to Design Programs* textbook [82] introduced sublanguages of Racket referred to as "student languages". For Java, Roberts [223] defined one sub-

set ("Mini-Java") and Gray and Flatt [102] a sequence of teaching languages. Heeren et al. [115] created Helium, a compiler for a subset of Haskell 98. More recently, Hermans [118] created Hedy, an educational language with a syntax that gradually evolves to reach the one of Python. Anderson et al. [11] defined subsets of JavaScript to enable the JavaScript version of the *Structure and Interpretation of Computer Programs* textbook [4].

It is no coincidence that many of these sublanguages have expressions at their core. Expressions are syntactic phrases that are constructed compositionally and can be evaluated to produce a value [209]. "Functional" programming languages, such as Racket or Haskell, undoubtedly put expressions front and center. But even in languages that are not considered predominantly functional, expressions play an essential role. This can be shown both theoretically and in practice. First, a study by Chiodini et al. [53] presented the grammar of a hypothetical version of Java without expressions, in which little is left when expressions cannot be used. Second, an analysis of Java projects written by students showed that 53 % of tokens in the source code of the median project were part of expression constructs (ibid.).

Despite their importance, expressions are not always adequately covered in teaching. Duran et al. [78] report anecdotal evidence that "many programming teachers and introductory textbooks do not emphasize expressions and evaluation, except when it comes to arithmetic and logic". Chiodini et al. [53] substantiated this claim by analyzing six introductory Java textbooks. The results revealed a number of misleading explanations of expression constructs that go beyond arithmetic, such as array accesses or class instance creations, which are presented with ad hoc rules instead of leveraging the compositional nature of expressions.

2.7 Abstraction and decomposition are fundamental in programming

The notion of abstraction has been subject to a myriad of definitions, even just within the field of computer science [184, Ch. 2]. A widely shared characterization, compatible with the meaning used in this dissertation, is abstraction as a process of recognizing similarities and ignoring differences (ibid.).

Abstraction is central to programming. Without the power of abstraction, the capabilities of a programmer are limited by their brain. Dijkstra presented this as a widely recognized truth already back in 1972: "We all know that the only mental tool by means of which a very finite piece of reasoning can cover a myriad of cases is called 'abstraction'; as a result the effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer." [75].

The centrality of abstraction extends from programming to human thinking in general. Hoare wrote: "in the development of our understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction" [121]. Hudak entitled a section of his programming textbook "Abstraction, Abstraction, Abstraction". The attention-grabbing title answers the question "What are the three most important ideas in programming?" [129], referring to three important types of abstraction commonly practiced by programmers.

Problem decomposition is similarly fundamental and closely related to abstraction. Milewski describes decomposition, along the corresponding *composition* of subsolutions to solve an overall problem, as "the essence of programming" [182].

Problem decomposition extends beyond programming and is indeed a general technique. In his classic book on how to solve problems, Polya suggests "decomposing and recombining" [213] as one of the strategies to deal with problems.

Programming educators acknowledge this importance. Using a Delphi process, Goldman et al. [99] attempted to characterize which concepts instructors deem important and difficult in introductory programming. "Abstraction/Pattern recognition and use" and "Functional decomposition, modularization" are both ranked as very important (8.8 and 9.3 respectively, on a scale up to 10) and difficult (9.0 and 7.9). Mirolo et al. [184] provide a broad overview of abstraction in computer science education, including some strategies to teach and assess it.

Abstraction and problem decomposition are also core components of what has recently been termed "computational thinking" [281, 238]. In 2008, Wing described abstraction as "the essence of computational thinking" [280]. Yet, a fundamental and general argument in favor of decomposition and abstraction was already put forward in 1690 by the philosopher Locke [165]:

> The acts of the mind, wherein it exerts its power over simple ideas, are chiefly these three:
> 1. Combining several simple ideas into one compound one, and thus all complex ideas are made.
> 2. The second is bringing two ideas, whether simple or complex, together, and setting them by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of relations.
> 3. The third is separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all its general ideas are made.

2.8   Language models may raise the importance of abstraction and decomposition

As discussed, nearly all educators concede that abstraction and decomposition are important topics [99], but some refrain from teaching them explicitly during the first introductory course and defer them instead to more advanced courses. After all, one could argue, there are already signs that students struggle in programming courses: not everyone succeeds (Section 2.1), and even writing very short programs such as Listing 1 can be extremely challenging (Section 2.5).

To complicate matters even further, recent years have witnessed a surprising increase in the capabilities of language models, which students are increasingly using to solve all sorts of tasks, including programming problems.

2.8.1   Models are great at generating code

A statistical model for a language represents the probability of a given sentence to occur [24]. Once the model is created, one can then use it to make predictions about the word that would most likely follow any given sequence of words (i.e., the word having the maximum likelihood of appearing next, as determined by the model). Historically, language models have been used to model *natural* languages, but they can also be employed to model and generate text of a *programming* language. The problem of generating language is inherently difficult due to the curse of dimensionality [24]: the number of parameters of such a model explodes even with relatively modest vocabulary sizes and lengths of the sequence of words considered for the prediction.

More recently, the increase in computing power, the availability of large datasets, and new architectures for neural networks [264] have enabled the training of language models with an increasingly large number of parameters. In 2020, Brown et al. [36] trained a model with 175 billion parameters, called GPT-3, that achieved surprising levels of performance in several different domains without extensive task-specific fine-tuning. Even without being specifically trained for programming, GPT-3 already has rudimentary capabilities for generating basic programs starting from Python docstrings.

To improve its performance, in 2021 researchers have fine-tuned it using publicly available code on GitHub [46]. On a benchmark consisting of 164 programming problems, the fine-tuned model solved 38 % of problems with its first prediction and 78 % within 100 tries. In a typical introductory programming course, students' tasks are remarkably similar to the examples on which the model has been trained: an exercise often consists of just implementing one function using the programming language re-

quired by the course, given a natural language description of its behavior and possibly some input-output pairs to clarify the problem and its corner cases.

2.8.2 Models ace introductory programming tasks

In 2022, Finnie-Ansley et al. [90] investigated the performance of this fine-tuned model on 23 programming questions used as a summative assessment in an introductory programming course at their university, as well as on 6 different variations of the "Rainfall problem" (Section 2.1). The model performed moderately well on the "Rainfall problem", averaging between 19 % and 63 % of the available points [90, Tab. 4], and would score in the top quartile of the class when measuring its performance on the assessment problems.

Since then, the capabilities of Large Language Models (LLMs) have increased even more. GPT-4 and coeval models are able to solve programming tasks that previous generations of models struggled on. Savelka et al. [230] showed that GPT-4 is able to solve a significantly larger fraction than its predecessors of multiple-choice questions and programming tasks typically used in introductory and intermediate courses in Python.

This surprising performance raises profound questions on how assessments should work for beginner programmers. Here, we leave the assessment concerns on the side to speculate on what, if anything, is still worth learning in programming.

2.8.3 What remains of programming then?

Attempting to predict the future is always a risky endeavor, particularly at times when technology changes rapidly, as LLMs did over the last three years. Claims are at risk of soon being disproved by the course of events.

However, the impact of LLMs is too big to be ignored in a dissertation written in 2025 that focuses on teaching introductory programming. In the introductory chapter of this dissertation, we echoed quotes from prominent Turing laureates to justify the importance of teaching informatics to everybody. And yet, there now exists a technology that can generate program code— one of the central activities of computing—at will and without any effort.

Merely having code written in a programming language cannot be the ultimate goal of programming education. We already built tools, LLMs, that can do that instantly.

If there is still any value left in teaching programming, it must be in something that goes beyond the final artifact (i.e., the code). Instead of focusing on the *result* of programming, the emphasis should probably be placed on the *process* of programming.

What does the process of programming look like, even if the ultimate result of it were to be a prompt to be fed to a LLM? Any nontrivial problem to be solved needs

to be understood and divided into smaller problems until those become manageable. This may seem like an academic perspective on programming, but programming practitioners in industry recognize this as well. In the words of one of them: "If you work as a professional developer, that is the bulk of the work you get paid to do: breaking down problems" [79].

The importance of learning how to decompose problems is acknowledged even by leading executives at companies that sell services based on LLMs and have a clear business interest in pushing the narrative that "learning to code is no longer needed". When asked what should one learn instead, the chief of an "artificial intelligence" company suggested three skills: "learn how to think, learn *how to break down problems*, learn how to communicate clearly [...]" [254, 3'37"].

Once we reach a problem that is small enough, LLMs can solve it by producing new code. Inherently, a language model works by repeatedly generating one token after the other. Developers who use code completions powered by a LLM in their Integrated Development Environment (IDE) know this phenomenon. Generating new code is just one keyboard shortcut away, and models easily generate many concrete variations of existing code.

Without adequate forms of abstraction, the code quickly becomes hard to correctly extend or fix even for the model itself (on top of being unmaintainable by humans in the long run). A programmer needs to recognize the need for abstraction, to know which means to do so are offered by the programming language, and to introduce them at suitable moments.

The hypothesis we suggest here is that LLMs may further increase the importance of abstraction and problem decomposition. Time will tell whether this proposition proves accurate. This dissertation does not offer any conclusive proof of this hypothesis: it assumes—based on reasoned justifications, but perhaps also reflecting the author's aspirations—that abstraction and decomposition are worthwhile learning goals, and presents an approach to teaching them.

2.9   We need to engage a diverse population of learners

The "Even or odd" program of Listing 1 is not only more complex than it appears at first glance (Section 2.5) and readily solvable with a modern LLM (Section 2.8.2): it also solves a fundamentally uninteresting problem.

Complete novices are often promised that one of the great powers that comes with learning to program is the ability to build compelling projects. However, solving interesting tasks demands programming skills that are difficult to acquire at the very beginning.

Inexperienced students may be disappointed when presented with a problem like "Even or odd". Determining whether a number is even or odd is a computation that they can already do in their heads, and there is little to be gained from having a computer solve the same problem. Running their own solution on a computer does not unlock new capabilities. In other words, many may not see the *relevance* of programming.

This issue is more important now than before, when programming was taught mostly to students who had actively chosen to study a scientific discipline at a university. Programming is now part of mandatory courses in computer science in high school, in which all students from different backgrounds and interests are asked to learn the rudiments of the discipline (Section 1.1).

The search for strategies to engage this diverse population of learners is ongoing. One possible strategy is to give more space to creativity; this can be implemented through various pedagogical techniques and in many domains, such as robotics, games, and music [236]. Below we consider one domain in particular: graphics.

2.10   Using graphics is a way to engage novices

Engagement has long been sought by educators, as it has been robustly correlated with positive outcomes of improved learning, student success and development [260].

Multiple studies report positive effects on engagement from graphics-based pedagogies. Sloan and Troy [241] introduced a new introductory course ("CS 0.5") before the traditional first course in programming ("CS1") and observed good success in retaining students using the *Media Computation* approach [105]. Porter and Simon [214] discuss how three combined instructional changes made to an introductory programming course (using the *Media Computation* curriculum, adopting peer instruction, and encouraging pair programming) retained almost one-third more students (from 51 % to 82 %). At the same university, the curriculum also increased pass rates for students majoring in Computer Science [240].

After having accumulated a decade of experience with media computation curricula, Guzdial [108] summarized their observations on the impact on engagement. The highlights of these findings include gains in student engagement, which led to a drastic reduction in failure rates (from 50 % to under 15 %). Moreover, female participation in the courses increased, stabilizing above 40 %.

Courses that adopt turtle graphics also seem to enjoy similar boosts in motivation. Bakar et al. [15] used turtle graphics in an introductory programming course in Java and observed high levels of motivation in students, measured across four dimensions: attention, relevance, confidence, and satisfaction. Santana and Bittencourt [228] used turtle graphics to gradually transition from block-based programming to Python and

noted how "students remained enthusiastic about the course" despite the challenges. Summarizing a number of studies with Logo, Clements and Meredith [58] note that teachers observed increased student self-esteem and confidence, provided that students are given enough autonomy.

At least for this aspect, the literature seems almost unanimous: graphics-based approaches increase engagement.

Chapter 3

There Are Many Approaches to Using Graphics

This chapter reviews related work on using graphics for introductory programming. It first clarifies what is meant by graphics in programming education in this dissertation, and then presents existing approaches by grouping them into three "families".

3.1 The scope is textual programming languages with graphical output

There are multiple senses in which programming education can be related to graphics. This section clarifies the scope of our work.

A fundamental separation exists between *graphical* and *text-based* programming. Doing "graphical programming" commonly means using a visual programming language to create programs *graphically*, instead of *textually*. Scratch [171] is an example of a visual programming language popular in educational contexts at the school level. Scratch programs are composed of "blocks" connected visually. Alice [61] is another example of a block-based programming environment in which students create 3D graphics to be used for animations, story-telling or games.

The success of block-based visual programming languages led to the creation of Blockly [199], a library that facilitates the creation of other block-based languages. Scratch also inspired Snap! [113], a block-based language in which users can define their own blocks. Block-based programming languages can be effective in reducing the occurrence of certain classes of issues in novices [274]

This dissertation, however, focuses on *text-based* programming languages, which are used more frequently in high school and university-level educational contexts.

Even programs written using text-based programming languages can involve "the domain of graphics" to different extents. For instance, a program can produce "static" graphics (2D or 3D), "animated" graphics, or "interactive" graphics—going towards Graphical User Interafaces (GUIs).

Creating interactive graphical programs has also been explored as a possibility to teach introductory programming (e.g., [81, 172]). However, in this review of approaches we restrict our focus to 2D graphics, textual programming languages, and tools designed with education in mind; already within this scope, there is radical variation in designs. GUI libraries, game engines, and block-based programming are thus excluded here. Some of the libraries mentioned below do support some form of user interaction, but that aspect is out of scope for this thesis.

Table 3.1 summarizes the four combinations that arise from combining two types of programming languages (textual or visual) and two types of output (textual or graphical). Each combination is exemplified with a possible programming task.

		Output	
		Textual	**Graphical**
PL	**Textual**	Add two numbers in Java	Draw a square in Python
	Visual	Say "Hello, World!" in Scratch	Draw a triangle in Snap!

Table 3.1. Example programming tasks for each combination of type of Programming Language (PL) and Output. The highlighted combination is the focus of this thesis.

3.2 We review three approaches using the classical example of a house

To illustrate, we pick a representative library from each family and use it to write a tiny Python program that creates the time-honored example graphic of a "house" as shown in Figure 3.1. The house consists of a single "floor", represented by a square, atop which sits a "roof", an equilateral triangle.

Figure 3.1. A house with a single floor and a roof (adapted from Abelson and diSessa [2], originally from Papert and Solomon [196]).

3.3 Graphics can be drawn using global coordinates on a canvas

School geometry introduces the two-dimensional Cartesian coordinate system, which describes a point (x, y) by its horizontal and vertical offset from the origin $(0, 0)$ of the plane. Indeed, many graphics libraries used for teaching novices are centered around the Cartesian coordinate system: shapes are drawn on an empty canvas of a certain size by specifying their positions with coordinates. Examples include Java2D [144], `acm.graphics`, and the Portable Graphics Library derived from the latter [224], as well as Processing [219], which is popular in the field of digital art.

Listing 2 draws a house with `cs1graphics` [100] for Python, another example of a library in this family. We create an empty canvas (implicitly of size 200×200) and then add shapes to it, specifying their absolute positions in a global coordinate system with the origin located at the top-left corner.

```
paper = Canvas()
floor = Square(100)
floor.moveTo(50, 137)
paper.add(floor)
roof = Polygon([Point(0, 87), Point(50, 0), Point(100, 87)])
paper.add(roof)
```

Listing 2. Drawing a house with the `cs1graphics` library.

However, not all libraries in this family require calling methods on objects. For example, Designer [123], another educational graphics library for Python, creates images with plain function calls and modifies their state with subscripts that update attributes, as in `floor['x'] = 50`.

3.4   Graphics can be drawn controlling a turtle

Turtle graphics was introduced by Papert as a simple way to draw with computers as early as elementary school [195]. It is based on a metaphor, sometimes made tangible with a robot, of the programmer controlling a "turtle" that carries a "pen". The turtle follows a sequence of commands, leaving a trace that results in a drawing.

Originally introduced with Logo, turtle graphics has become widespread in introductory programming. Libraries exist for several programming languages, and some are even included in languages' standard libraries. This is the case for Python's `turtle`, with which we draw the house in Listing 3.

```python
for _ in range(4):
    forward(100)
    right(90)
left(60)
for _ in range(3):
    forward(100)
    right(120)
```

Listing 3. Drawing a house with Python's `turtle` library.

Commands express movements relative to the turtle's current state. The effect of a command such as `forward(100)` depends on the turtle's location and direction.

In contrast to canvas-based libraries, there are no global coordinates. The perspective is *local*, relative to the turtle. Papert argued that turtle geometry is learnable because it is "body syntonic" [194]: the turtle's perspective is the same as the learner's.

However, most turtle libraries allow some form of positioning using global coordinates, partially going against Papert's principle of body syntonicity. Python's `turtle` offers `goto` and `setpos` to move the turtle to an absolute position (and possibly draw a line), and `teleport` to jump to absolute coordinates without leaving a trace on the canvas.

An attentive reader probably noticed that the program in Listing 3 only draws the outline of the house. Turtle graphics focuses on drawing *lines*, as opposed to *shapes*. When the end point of a sequence of lines matches its beginning, a line forms a closed path and effectively represents the outline of a shape. This shape can be colored as shown in Listing 4, using the `begin_fill` and `end_fill` functions to denote the shape to be filled.

Throughout this dissertation, we will use turtle graphics to draw outlines of shapes, enabling a fairer comparison to the other approaches. Nonetheless, "emulations" are

```
fillcolor("yellow")
begin_fill()
for _ in range(4):
    forward(100)
    right(90)
end_fill()
```

Listing 4. Filling a square with Python's `turtle` library.

possible in both directions: turtle's closed paths can be filled, and shape-based approaches can mimic lines by drawing thin rectangles of the desired width.

This dissertation focuses only on controlling one turtle at a time by calling simple functions. Some turtle environments, including the one offered by Python, allow multiple turtles which are modeled as objects and commanded with method calls. Caspersen and Christensen [42], for example, described a pedagogical Java implementation that uses methods and objects.

3.5   Graphics can be treated as values to compose

Both families presented so far produce graphics by "direct rendering", but that is not the only conceivable way to create graphics. A different approach was proposed in 1982 by Henderson [117], who introduced a purely functional way to describe pictures. The textbook *Structure and Interpretation of Computer Programs* adopted the idea with a "picture language" [3, Sec. 2.2.4].

Finne and Peyton Jones later proposed the idea of *composing* pictures from primitives using what they called "combinators" [89]. Some of these principles have been adopted in the textbook *How to Design Programs* [82]: an "image teachpack" [18] accompanies the book and is available as a Racket library. Felleisen and Krishnamurthi discuss how the teachpack enables students to construct algebraic expressions that "consume and compute pictorial values" [84].

Libraries in this family treat images as values. There are functions that produce primitive shapes, such as rectangles and circles. Other functions combine images into more complex ones; for example, an image may be placed `above` or `beside` another. Images can only be composed, not mutated.

The libraries in this family tend to be written for languages that embrace the "functional" programming paradigm. In Listing 5, we build the house with Pyret [256], an educational programming language whose syntax is similar to Python's.

The Pyret environment offers a REPL capable of rendering graphics. At the end of

```
floor = square(100, "solid", "yellow")
roof = triangle(100, "solid", "red")
house = above(roof, floor)
```

Listing 5. Drawing a house with Pyret's `image` module.

the program in Listing 5, `house` is an expression whose evaluation produces a graphical value, visible to the student within the environment.

Chapter 4

Existing Approaches Have Pitfalls

This chapter analyzes the existing approaches in light of the main goal of this thesis: teaching introductory programming in an engaging way, emphasizing abstraction and problem decomposition.

We refine this goal into three specific goals that drive our analysis of the existing approaches. First, we focus on problem decomposition, stating as a goal that libraries should enable students to decompose a graphic into a sub-graphic without introducing unwanted dependencies (a goal we dub *clean problem decomposition*, Section 4.1). Second, we consider whether there are pitfalls that hinder *meaningful engagement* (Section 4.2): students should be engaged with programming not frivolously, but to learn essential aspects such as abstraction. Third, we argue that approaches should be suitable for introductory programming and thus offer a *manageable complexity* (Section 4.3) that enables them to be used with beginners.

RQ Given the three above guiding goals, what pitfalls are there in using existing graphics approaches for introductory programming education?

Table 4.1 summarizes the eight pitfalls we identified in the existing approaches according to these three guiding goals. In the next sections, we explain and justify each pitfall by drawing on the research literature on computing education.

4.1 Decomposing a problem cleanly is hard

Decomposition—along the corresponding *composition* of sub-solutions to solve an overall problem— has been considered "the essence of programming" [182]. Problem decomposition is also at the core of "computational thinking" [281]. However, a look at student programs reveals that achieving proper decomposition and modularization is difficult for novices [142].

Table 4.1. Pitfalls identified in the libraries which represent the three families. ✗ means a pitfall is present, (✗) denotes partiality.

Guiding Goal	Pitfall	**Coords on Canvas** cs1graphics	**Turtle Graphics** Python's turtle	**Compositional Graphics** Pyret's image
	Global coordinates	✗	(✗)	
Clean Problem Decomposition	Turtle's state		✗	
	Local coordinates		✗	✗
	External graphics	✗		✗
Meaningful Engagement	Rich API	(✗)	(✗)	✗
	Scaling	✗		✗
Manageable Complexity	Extra language features	✗		
	Mutability	✗		

For decomposition to be effective, subproblems need to be *independent*. A subproblem that is tightly coupled to other subproblems cannot be solved in isolation. The main promise of decomposition is being able to reason locally and focus on each subproblem separately. Without independence, we have to keep multiple interacting subproblems in mind simultaneously, which increases our cognitive load. Good decomposition is also closely linked to *abstraction* and *reuse*: if we create the right abstraction (e.g., a function) to solve a (sub)problem, we can reuse the abstraction when the solution is needed again.

As we discuss the pitfalls, we again resort to the "house" from Figure 3.1 as a small-scale running example. The problem of drawing the house features two smaller subproblems: drawing the roof and drawing the floor; their solutions can be combined to solve the overall problem.

A problem is decomposed *cleanly* when subproblems do not depend on each other, except for dependencies that are desired because they are inherently part of the problem (e.g., we may want to keep the roof and the floor of the house at the same width). These latter dependencies should be made explicit. When an approach introduces a hidden, unwanted dependency, it violates the goal of clean problem decomposition.

4.1.1 Global coordinates break independence

In Listing 2, coordinates such as (50, 137) implicitly depend on the origin of the plane, a globally shared "zero" that acts as a reference point. The two subproblems do not have independent subsolutions. When only one subproblem changes, this becomes an issue.

Consider an increase in the height of the roof, from 87 to 120 for example. This also requires a change to the position of the floor, whose center should be moved to (50, 170) instead of (50, 137).

Decomposition should produce subproblems that are independent from each other. We would like the roof problem to be independent from the floor problem, so that we get local reasoning: we do not want to worry about the roof when writing the code for the floor. A global coordinate system (Section 3.3) breaks this promise.

4.1.2 Turtle state also breaks independence

One of the original goals of "turtle geometry" [195] was to eliminate the problems caused by global coordinates. The turtle provides a *local* perspective to drawing. Commands like "move forward" and "turn right" represent movements and rotations relative to the turtle's current position and direction. This makes it easier to extract the first three lines of Listing 3 and create a reusable procedure to draw a square. Similarly, the last three lines can be extracted into a procedure that draws an equilateral triangle.

We would now like to use these smaller procedures as "building blocks in more complex drawings" [2]. Things are not so simple, however, as shown by the extra command on Line 4 of Listing 3: `left(60)`. To compose the subsolutions we need to know the turtle's position and heading after executing the first procedure; we may also need to adjust that state with "interface steps" [2, 111] before executing the second procedure. This happens because the turtle's position, heading, and pen status constitute a *global state*. The turtle's state is not local to every procedure; it is shared, mutated, and kept across procedures.

Harvey [111] pointed out that programs are "much easier to read and understand if each procedure can be understood without thinking about the context in which it's used". However, turtle functions may only compose cleanly with extra instructions (the "interface steps"), adding to the programmer's burden. Harvey's partial patch for this issue is to have a higher-order procedure reset the turtle's heading (ibid.). While an experienced programmer might consistently apply this patch, it seems unlikely that novices could and would. And in any case, one still needs to deal with the rest of the turtle's state, which includes its position.

4.1.3   Local coordinates are prone to misuse

Listing 5 (in Pyret) exhibits clean decomposition: the roof and floor are created independently, then combined as desired into a house. In this simple example, above does the job: it succinctly conveys the programmer's intent. However, not all graphics consist of shapes next to each other. As a silly variation on the theme, imagine the same house with the roof collapsed and lying in front of the ground floor. With Pyret's library, overlay-align("center", "bottom", roof, floor) does the trick, specifying that the composite image should have the two originals with their bottom edges aligned at the center. Aligning enables the creation of a big class of more interesting graphics.

One might rightfully object that not all graphics are that simple and these combinators are still not general enough. And indeed, for maximal freedom in overlaying images, Pyret's library also offers multiple combinators that involve exact offsets. For instance, overlay-xy "initially lines up the two images upper-left corners and then shifts img2 to the right by dx pixels, and then down by dy pixels". Some curricula encourage beginners to use such functions that operate on local (relative) coordinates. For example, students might need them to place a rocket at a certain height onto a scene [84].

Unfortunately, anecdotal evidence shows that students who know of these functions frequently misuse them, wielding them where simpler combinators would suffice. An image can act as a background "scene" to overlay other images on it at specific positions. This effectively reintroduces a shared origin, mimicking the global coordinate system.

The very teaching materials can exacerbate the issue by suggesting this problematic usage of local coordinates (e.g., [28, Step 11] and [234, p. 73]). Listing 6 reproduces an example [28, Step 11] taken from an "Hour of Code" activity by Bootstrap, a leading curriculum that embraces Pyret. Despite the availability of a function that would align the two images on their left edge, the activity suggests that learner use coordinates. This couples the alignment location with the size of the two images, marring the otherwise clean decomposition.

```
eye = circle(30, "outline", "black")
pupil = circle(10, "solid", "black")
googly-eye = put-image(pupil, 10, 30, eye)
```

Listing 6. A googly eye with local coordinates, created with Pyret's image library.

4.2   Learners' engagement should be meaningful

4.2.1   External graphics may lower motivation

When a student creates their first graphical program from scratch, their sense of empowerment is often palpable: it is not too hard to write a program that displays images, not just characters in a terminal! Without too much effort, the student draws a house or a tree or a flag and declares victory. That joy may quickly give way to disappointment when the student realizes how hard it is to build graphics from basic shapes and make them look decent compared to the slick artwork they see every day on their smartphone. Suddenly the house does not look that nice.

In order to recover the initial excitement, or perhaps to quickly enable non-trivial graphics for a simulation or game, teachers often explain how students can *import external images* into their creations. For example, `cs1graphics` allows this through the `Image` class and Pyret's `image` library has an `image-url` function. Designer [123] even offers an `emoji` function to include Unicode emoticons. There are lots of fun graphics out there, and letting students select custom images, possibly to be used as sprites in a game, has been shown to contribute to their sense of ownership over the resulting program [233]. This therefore sounds like a great feature (and it can be, in the right context[1]).

However, our anecdotal experience suggests that once this possibility is revealed, many students perceive writing a program to create a graphic as much less interesting. They frequently spend time searching for fancy images online and lose focus on what the graphical programming was intended to highlight.

4.2.2   Rich APIs shift the emphasis from programming to libraries

Alphonce and Ventura, introducing an educational graphics library, remark on a common complaint: "non-standard libraries are a waste of time since students will not use them outside... the one course" [9]. Their retort is also typical: "we are not teaching students the library, we are teaching students object-orientation using the library as a supportive mechanism" (ibid.).

Every library introduced adds something to what students need to learn. How much is added varies significantly, depending on factors such as API size. Since time is scarce, it is important that students invest as much of theirs as possible on learning core content, rather than memorizing the minutiae of a particular API or poring over

---

[1]Supporting external graphics is a valuable feature especially if the approach is focused on manipulating existing images, for example with *Media Computation* [105], rather than creating new ones from scratch. This dissertation focuses on the creation of graphics.

documentation.

Consider Pyret's `image` module. Just for the purpose of placing an image on top of another there are seven functions: `overlay`, `underlay`, `overlay-align`, `overlay-xy`, `overlay-onto-offset`, `underlay-align`, and `underlay-xy`. There are also ten functions for specifying triangles in various ways.

Rich APIs also deprive students of opportunities to learn. For example, students could benefit from writing a function that creates a square or places many images in a row. The absence of a function to scale forces to abstract and produce graphics parameterized on the size. But if novices find canned solutions for these problems in a library, they are less motivated to re-implement the solutions.

Libraries for novices should serve the needs of their target audience. Experienced programmers' convenience of always having the right function at hand can harm learners' engagement with programming.

4.2.3   Scalable graphics reduce the need for abstraction

Together with problem decomposition, abstraction is often highlighted as a defining element of "computational thinking" [281]. Devlin argues that *"computing is all about constructing, manipulating, and reasoning about abstractions"* [72]. However, moving from the concrete to the abstract is a significant challenge. For instance, students have trouble defining functions when solving problems that require abstraction [110].

Graphical programs offer many opportunities for abstraction. For instance, our house-drawing program could be parameterized with respect to roof color (or wall color, or both). This would likely produce a generic function that takes the color as a parameter and creates a house accordingly. such a function could then be called in multiple places with a suitable argument. As an example, size is one obvious aspect of a house that we may wish to parameterize—to abstract. Students often want to experiment rapidly and repeatedly change the size of a house they just drew. In all the programs of Chapter 3, a change in this single aspect necessitates multiple changes to code: Listing 2 needs seven edits, Listings 3 and 5 two each. On a small scale, students experience what professionals call an issue of *maintainability*.

Students may then be encouraged to *abstract* by defining a general function that creates a house of a given size or by extracting the size into a constant used throughout the program. Is it really true that this laborious refactoring is needed? Not necessarily. Pyret's `image` module contains a function to *scale* an image; `cs1graphics` allows *zooming* the entire canvas. The availability of such functions undermines clean abstractions: instead of parameterizing their houses with a size, learners can just insert a new function call at the end to perform scaling and produce the desired visuals. No abstraction skill is then needed or practiced.

4.3   Complexity should be kept under control

Part of what is necessary to understand what a program does is understanding the programming language features it uses. A programming language can be seen as an aggregation of features [154]. In turn, each feature can be defined by grammar rules and semantics to determine the meaning (when formalized, the semantics can be expressed for example in terms of inference rules [209]).

This section points out two pitfalls that may expose absolute novices to complexity that is difficult to manage. This complexity can arise either because of the *number of programming language features* used in the programs read and written by learners, or because those programs use features *intertwining aspects* that can be orthogonal, such as state and time.

4.3.1   At the beginning, language features should be minimized

Beginner programmers often resort to "bricolage": extensive trial-and-error whose *"manifestation [...] is endless debugging: try it and see what happens"* [23]. While experimentation is certainly valuable, ineffective experimentation is common and leads to frustration and poor learning outcomes. A student writing a program that produces the correct output is no guarantee of understanding [140, 160]. To ensure that learners understand the source code they write, one must be careful to introduce new programming language constructs at a pace that novices can keep up with.

Graphics libraries vary in which language constructs they require. For example, Listing 2 features instantiations with and without parameters, method invocations, and lists. Introducing all these constructs "from the first day" [100], as the library's authors suggest, may be feasible in some contexts and under some definition of what it means to introduce a construct. We argue that this approach is incompatible with the goal of learners understanding the concepts in the code they write [183]. In most circumstances, students will need to accept parts of code as "something you need to write" with the promise that "one day you will understand".

Graphics can be an excellent domain for learning object-oriented programming; some authors explicitly advocate it (e.g., [9, 100, 42]). But at the very beginning of an introductory course, object-oriented language features (and others) add complexity. To understand the third line of Listing 2, for example, students need to understand function invocations *and* how functions (which are actually methods) relate to the objects on which they are invoked.

4.3.2   Mutability makes it harder to reason about programs

Mutable state is often introduced early in introductory programming, even though it breaks referential transparency and demands a more complex mental model for reasoning about programs [262]. There is extensive research (e.g., [243, 217, 52, 167, 48]) on misconceptions that novices have about assignments, both with primitive values and references [170], and to (mutable) objects in general [122].

Some graphics libraries, too, model images as objects with a mutable state. Consider Listing 2 and a student who wishes to add another floor to the house. A fairly typical novice intuition would be to reuse the existing square (that `floor` refers to) and to make a copy of it. The student may then write `floor2 = floor` to create the copy and `floor2.moveTo(50, 237)` to move the new floor to the desired location. A puzzling, unexpected result awaits such a student.

As illustrated, mutable state is closely associated with *aliasing,* another concept that is fundamental but not easy: difficulties abound even among upper-level undergraduates at a prestigious university [93]. A recent study by Strömbäck et al. [248] investigated 397 students from different programs and years and consistently found that they were unable to predict the outcome of short programs that involved aliasing, parameter passing, and scope.

For these reasons, we consider mutable state a pitfall in the design of graphics libraries for complete beginners.

Part III

The PyTamaro Approach

Chapter 5

PyTamaro Is a Library Designed to Avoid the Pitfalls

The previous chapter described eight pitfalls when using graphics to teach introductory programming. This chapter describes a design that avoids those pitfalls.

RQ How can a graphics library be designed to support *clean problem decomposition*, *meaningful engagement*, and *manageable complexity*, avoiding pitfalls?

The design is implemented in PyTamaro[1], a minimalist Python library publicly available as open source at <https://github.com/LuCEresearchlab/pytamaro>. We illustrate the key aspects of the PyTamaro design gradually, alongside a teaching approach that builds on the library's strengths.

5.1 This is an initial example with PyTamaro

Listing 7 below shows how to draw a simple graphic with PyTamaro, resorting once more to the house example of Figure 3.1. Unlike the code listings in Chapter 3, Listing 7 is a standalone Python program ready to be executed, complete with an import statement and a function call to display the resulting graphic.

  All the programming language constructs used in Listing 7 can be reasonably explained from the very beginning. This example makes no use of method calls, lists, tuples, or even strings. In fact, understanding it requires knowing the same programming language constructs needed to use the standard library to compute the square

---

[1]We pronounce PyTamaro as PyTàmaro. The name pays homage to Monte Tamaro, a mountain near the author's home university, located between Lake Maggiore and Lake Lugano. With this in mind, does the logo shown on the GitHub page look familiar?

```python
from pytamaro import rectangle, triangle, yellow, red, above,
    ↪ show_graphic
floor = rectangle(100, 100, yellow)
roof = triangle(100, 100, 60, red)
house = above(roof, floor)
show_graphic(house)
```

Listing 7. Drawing a house with PyTamaro.

root of a number (one of the introductory examples used outside the domain of graphics).

PyTamaro belongs to the family that treats graphics as values to be composed (Section 3.5). Therefore, when students reason about graphical programs, they can rely on the same mental model that they use for expressions that operate on numbers [84]. This design choice avoids three previously identified pitfalls: there is *no global coordinate system* and *no stateful turtle,* and all *graphics are immutable*.

5.2   The design encourages the definition of abstractions early on

Listing 7 looks similar to the earlier Listing 5 written using Pyret's image library, and indeed the two share many key aspects.

However, PyTamaro does not include a function for even a primitive shape such as a square. This is a deliberate design choice so as to *avoid a rich API* (Section 4.2.2). It is a tempting pitfall: as a library author, it is easy to pack in all sorts of function variants that are convenient for certain use cases—and users often appreciate rich APIs.

Since abstraction is so central to programming, we should immerse novices in it early. One valuable way to do that is to have learners *use* abstractions that somebody else defined; this is a good place to start and an instance of the "consume before produce" principle in instructional design [41]. We argue that novices also need early opportunities to *define* their own abstractions [149] and should be placed in situations that *beg* for them to define some.

The absence of a square function in PyTamaro introduces one such situation into our context of simple graphics. Having first built some graphics that include squares, learners begin to relate to the inconvenience of having to specify each width and each height of each square rectangle. This motivates the definition of a general function like that in Listing 8, which creates a square with a given side length and color.

```python
def square(side, color):
    return rectangle(side, side, color)
```

Listing 8. A function to create a square with PyTamaro.

Implementing such a function is challenging for many beginners [130], but the generalization pays off later when the function can be conveniently reused to solve bigger problems. When drawing a bigger graphic, one does not want to worry about the details of how to build such a basic shape. At that point, a square function comes in handy.

This build-for-reuse approach confronts the temptation to write throw-away code. Instead, students are encouraged to build their own "toolbox", adding functions they implemented and deem useful. Over time, they create increasingly interesting graphics that are nevertheless entirely based on their very own code. We will present in detail this idea of a toolbox in Chapter 8.

An auxiliary benefit of custom functions is that they help gradually introduce type annotations, which are optional in Python. In our approach, students first encounter types in PyTamaro's documentation. They learn that red is a name for a value of type Color and that above operates on two parameters of type Graphic and returns a value of type Graphic. Later, we encourage explicit type annotations as in Listing 9. We have multiple reasons for introducing type annotations to beginners: one is that types guide the design of programs [80]; another is that types help catch errors early. IDEs with a static type checker for Python (e.g., Thonny [12] for education) show warnings when types do not match, giving students early feedback. Types can also help with conceptual learning. For example, beginners have trouble with the distinction between returning a value and printing (or otherwise reporting) a result inside a function [147]; types make the distinction explicit and checkable.

```python
def equilateral_triangle(side: float, color: Color) -> Graphic:
    return triangle(side, side, 60, color)
```

Listing 9. A function to create an equilateral triangle, with type annotations.

5.3   Graphics enable visual problem decomposition

PyTamaro's design does not expose any coordinate system and does not maintain state: this supports clean decomposition into independent subproblems.

5.3.1   Visual decomposition starts from basic examples

Graphics can make decomposition *visually* apparent and relatively easy[2] to intuit. This can be leveraged in teaching. Consider the emblem of the International Red Cross at the top of Figure 5.1. It is easy to discern *visually* that the emblem consists of a red cross on a square white field. The "big" problem of drawing the entire emblem decomposes into two subproblems: drawing a red cross and drawing a white field. The subproblems' solutions compose into a solution for the whole problem, just like the two visuals compose to produce the combined image. There is a direct mapping between visual components and problem decomposition; a student who engages in creating these graphics is *meaningfully engaged* with computing.

Crosses are not a PyTamaro primitive, so how do we draw one? We can visually observe that the cross is made up of two bars (a horizontal bar and a vertical bar). We must repeatedly break down our problem into smaller ones until we reach elementary problems: a powerful *recursive* process for problem solving, which can be illustrated as a tree as in Figure 5.1. Visualizing the entire (de)composition as a tree can help to reify the (de)composition process, supporting explanations and discussions in class.

This hierarchical decomposition into independent subproblems which are then composed is enabled by two key aspects in PyTamaro. First, each graphic is built without a notion of where it lies. There is no global coordinate system, and we can reason about the *properties* of a graphic (e.g., that the horizontal bar has a certain width and height) without having to think about *where* the graphic will be positioned (i.e., its coordinates). Second, functions like `above` or `overlay` produce composite graphics that are just like primitive ones in that they, too, can be further composed.

The decomposition of a problem into subproblems that are given explicit names or descriptions closely corresponds to the educational practice of giving labels to "subgoals" when trying to accomplish a bigger goal [43]. This technique of "subgoal labeling" originated in mathematics education but has also been adopted in programming education, with a number of studies showing improvements at least in some contexts [174, 186, 175].

---

[2]Claiming that something comes easy to people is always a slippery slope. The ability to decompose a 2D graphic can probably be considered a component of the broader spatial skills, which are known to vary across individuals [116]. Parkinson and Cutts [197] investigated the relationship between these abilities and programming, noting how the cognitive functions involved in spatial skills are also involved in programming. This dissertation does not explore further the relationship with spatial skills, but Section 5.4 provides a perspective using programming language theory that may also explain the relationship between programming and spatial skills described by Parkinson and Cutts.
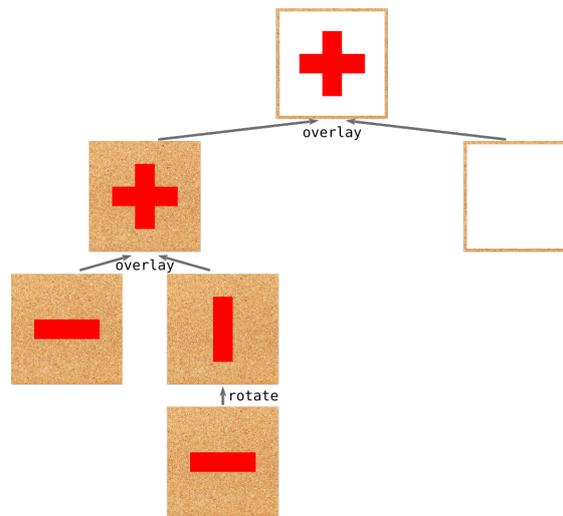
Figure 5.1. Hierarchical (de)composition of the IRC emblem. The bars' "cork board" background indicates transparency.

5.3.2   There are multiple ways to (de)compose

Even many simple graphics can be (de)composed in multiple equally valid ways. Functions in PyTamaro that combine graphics are designed so that equally valid solutions indeed result in graphics that are equivalent.

Just like the binary operators that children learn in basic algebra operate on two numbers, PyTamaro's composition functions operate on two graphics. For example, the `beside` function places two graphics next to each other, just like the `+` operator adds two numbers. However, unlike the addition of numbers, the combination of graphics is *not commutative*. It is visually obvious that overlaying the white field on the red cross would not have been an equally valid way to compose the emblem in Figure 5.1.

Now consider the Italian flag shown twice at the top of Figure 5.2. We can trivially discern that it is made of three rectangular bands; our elementary subproblems are to draw those bands. It is slightly less obvious how to compose the rectangles, given that we only have functions to combine *two* graphics. Figure 5.2 shows two equally valid compositions: we may first join the green and white rectangles into a single graphic, then join this composite with the red; or we may first join the white and red, and then compose the result with the green. The equivalence of multiple ways to compose is guaranteed by the *associativity* of PyTamaro's composition functions. This is like adding numbers: we can sum three numbers in two ways. No matter whether we start by adding the first or the last two numbers, the result is the same.

Similarly, the rotation of a graphic by a given angle *distributes* over the composition
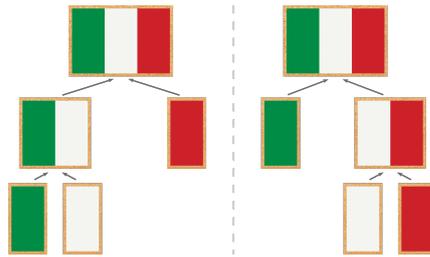
Figure 5.2. Two different but equally valid ways to compose the Italian flag exploiting the associativity of `beside`.

of graphics. This means that we obtain an equivalent graphic no matter whether we first individually rotate two graphics and then compose them, or we first perform the composition and then rotate the composite.

PyTamaro's design satisfies certain algebraic properties to empower students with maximum flexibility in the different ways equivalent graphics can be composed. A side benefit is that graphics is an interesting domain other than numbers in which teachers and learners may revisit these properties.

Educators also hope that some transfer also happens in the other direction, i.e., programming should help to learn algebra. The Bootstrap curriculum is deliberately developed around this idea and, unlike other approaches, has shown some positive evidence of this transfer [232]. Algebraic properties tend to be memorized without much understanding, perhaps because most of the students will not encounter them again in a domain that is different from numbers. There is some evidence that at least two different examples are needed to induce an appropriate mental schema [98].

5.3.3 A flexible combinator enables (de)composing more elaborate graphics

Not all graphics can be composed just by placing basic shapes next to each other or overlaying them on their centers. How can we have learners create more intricate graphics with PyTamaro and still *avoid the pitfall of local coordinates* (Section 4.1.3)?

In PyTamaro, every graphic has a *pinning position*, a designated point where it may connect to other graphics. Graphics are `composed` by aligning these positions. Pinning is invisible but readily explained by a visual analogy, a "notional machine" [86] with a cork board, a pin, and paper cutouts (cf. Goldwasser and Letscher [100]). In our teaching, we have used this analogy both onscreen and in tangible, unplugged activities.

A graphic is represented by a paper cutout, such as a red square, and pinned to the

cork board. The place where the pin is stuck is the graphic's pinning position. When we rotate a graphic, we do so around the pin. When we `compose` two graphics, we align their pinning positions and stick a pin there through both; we then staple the cutouts together and consider the result an inseparable composite. Figure 5.3 illustrates the (de)composition of a less obvious graphic using pins.
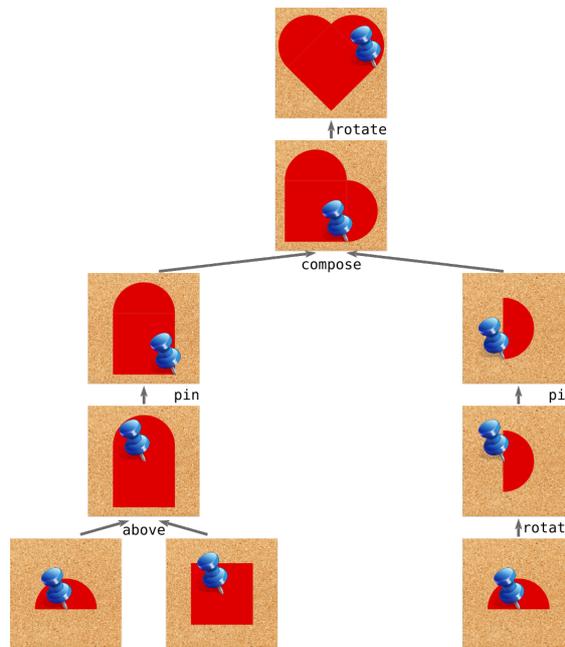
Figure 5.3. Drawing a heart with `pin` and `compose` (leaves are PyTamaro primitives).

In the left sub-tree, we place the semicircle `above` the square. In the right sub-tree, we perform a rotation. Before joining the two sub-trees using `compose`, which aligns two graphics on their pinning positions, we need to "move" the pin. We create a new graphic on the left with the pin at its bottom-right corner, a new graphic on the right with the pin at its bottom-left corner, and then compose. Finally, we can just `rotate` the composite graphic to obtain a heart.

Since there are no explicit coordinates in PyTamaro, there are restrictions on where a pinning position may be placed. As things stand, PyTamaro creates shapes with a sensible default pinning position (e.g., the centroid for a triangle) and has nine standard options for adjusting it (e.g., `top_right`). These nine options are determined by the graphic's bounding box.

Once learners familiarize themselves with this way of composing, they can draw many more challenging and interesting graphics. Examples include tile-based worlds composed entirely starting with PyTamaro's primitives, such as a Pac-Man maze. The

right-hand side of Figure 5.4 depicts a Pac-Man maze built entirely by composing PyTamaro's primitives; the maze is just one example of the many tile-based worlds that can be drawn. A key part of the solution is the *decomposition* of the world into the various possible tiles. The left side of Figure 5.4 shows four "corner tiles", two "straight tiles", two tiles for the ground containing a "dot" and a "pill", and a tile with the Pac-Man character. Drawing each group of tiles becomes an *independent* subproblem, whose solution can then be easily combined with others.
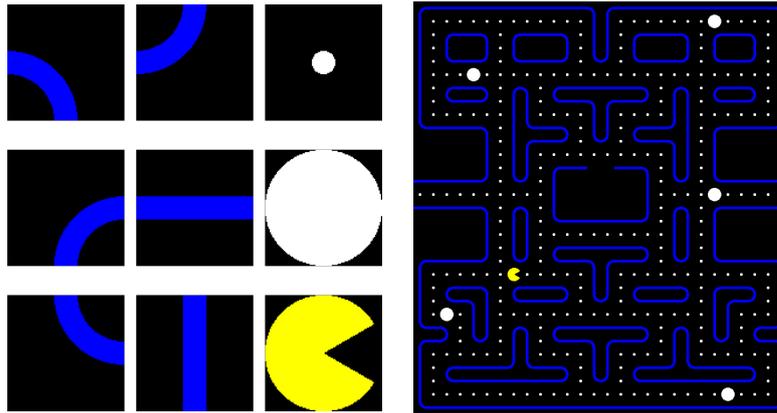
Figure 5.4. A Pac-Man maze (right) created out of tiles (left), which are in turn composed from PyTamaro's primitives.

5.3.4   Clean decomposition means no unwanted dependencies

The house example of Listing 7 showed a first basic example of visual problem decomposition. The floor and the roof are two subproblems, which we solve separately in the second and third lines of the program.

A careful reader may note that, in a sense, the two subsolutions are not fully independent. The hardcoded numeric literal 100 is used to determine the width of both the floor and the roof. As introduced in Chapter 4, the goal of achieving clean problem decomposition means that the approach should not introduce unwanted dependencies.

To determine whether a dependency in the *program* is wanted or unwanted, we need to carefully examine the statement of the *problem*. For the house example, the problem was presented informally with Figure 3.1. If the problem specifies, implicitly or explicitly, that the width of the roof and the floor should be the same, this requirement becomes part of the problem and needs to be encoded in the program as well.

Listing 7 uses hardcoded numeric literals to match all the other programs presented in Chapter 3. If the width requirement just discussed has to be respected, this wanted

dependency should be made explicit (e.g., by introducing a named constant `width`). Similar cases occur frequently when one desires to use the same color in many sub-graphics: the color is a wanted dependency that should be made explicit. This does not violate the principle of clean problem decomposition, which aims to prevent *unwanted* dependencies that are not part of the problem and are introduced because of the approach.

5.4 The structure of the graphic informs the structure of the program

The "blank page syndrome" is a feeling experienced by a writer who finds it challenging to commence new work. The empty page feels intimidating.

A similar feeling has been reported for novice programmers. Learners may have watched their teacher write programs, and may have modified or completed some programs themselves. In front of an empty editor, designing a new program from scratch can feel intimidating, with no clear direction for a starting point.

The *How to Design Programs* textbook [82] introduces learners to a systematic approach to program design, helping them avoid the "blank page syndrome". The textbook offers a "design recipe": a guided sequence of steps leading to a template for what the program should look like.

The template depends on the structure of the data. Indeed, a key lesson of the textbook may be summarized as *program structure follows data structure*. For example, when the "input" consists of different cases, the program will contain a conditional with a branch for each case. When the program needs to deal with a recursively-defined list, it will need a conditional to distinguish between the empty list and the case where there is more to process. Several refinements of the design recipe are introduced throughout the book, to handle different kinds of data. Felleisen et al. warn that the "design recipe" is not enough to cover all kinds of programs, admitting that some advanced programs require non-obvious insights [82, Part V].

Gibbons [97] pointed out that the templates for some of these more advanced programs can actually be systematically derived when also considering the structure of the output data. In short, *the program structure follows the structure of both input and output data*.

Paying attention to the structure of the output data is particularly relevant for programs that use PyTamaro to create graphics. The structure of the "input" is often simple: typically the desired sizes, angles or colors for certain parts of the graphics. The structure of the "output"—the final graphic—dominates because of its rich, visible structure. All the examples presented earlier in this section focused on visually

decomposing the graphic to identify its constituent parts.

This gives graphics an important pedagogical advantage over other kinds of simple data common in introductory programming, such as numbers[3]. Except for overlapping parts, the structure of a graphic is directly visible for the learner in the problem statement. Decomposing a graphic is thus not a frivolous process confined to the graphics domain, but a process that follows a fundamental principle of program design. The decomposition of the graphic can directly inform the structure of the program that creates it.

5.5   Abstraction arises from similarities and differences

This section reviews two fundamental mechanisms to teach abstraction with PyTamaro. The two techniques are not novel per se: they are fundamental in programming. Here, we illustrate them using the domain of graphics, which exploits the power of graphics of being visual to hopefully make them more understandable.

5.5.1   We can give a name to identical graphics

"One of the most basic ideas in programming—for that matter, in everyday life—is to name things" [129]. Giving a name to things is a first form of abstraction: not only can we avoid having to remember a specific value, but we can also use it multiple times in our program.

In the graphics domain, when we decompose a graphic, we may give names to the different parts. We did that in Listing 7: we gave the name `roof` to the red triangle, `floor` to the yellow square, and we even named the entire graphic `house`.

Names become extremely important when a sub-graphic occurs *identically* more than once in a graphic. Consider the pair of eyes shown in Figure 5.5 and assume we cannot use names.

Figure 5.5. A graphic showing a pair of eyes.

Listing 10 shows the single expression necessary to create a pair of eyes. This expression appears to be a convoluted way to express what we see in Figure 5.5: there

---

[3]In principle, a lone number could also be treated as structured data (as a Peano number: 2 is the successor of the successor of zero), but it is exceedingly rare to do so, especially in introductory programming.

```
beside(
    overlay(ellipse(100, 100, black), ellipse(200, 200, green)),
    overlay(ellipse(100, 100, black), ellipse(200, 200, green))
)
```

Listing 10. A single expression to create a pair of eyes.

are two eyes and they are identical in every way. When we leverage the power of giving names to expressions, we can write the expression to create an eye only once and name it. From that point on, we can use that name in place of the whole expression. Listing 11 shows an example of this process, introducing the name `eye`, which can then be used twice in the expression in the second line.

```
eye = overlay(ellipse(100, 100, black), ellipse(200, 200, green))
eyes = beside(eye, eye)
```

Listing 11. Introducing a name to abstract Listing 10.

This process is not merely a trick to avoid typing (or copy-pasting) a few characters. Working with abstractions is necessary to build even minimally more elaborate graphics: more broadly, to write programs that solve any problem that is not trivial.

Some IDEs offer this operation and present it as a refactoring operation called "extract constant" or "extract variable". Throughout this text, we will mainly use *constant* to refer to this abstraction, even though the language construct used in Python is technically a variable that can be reassigned.

Debates on terminology aside, the important bit is the ability to abstract by introducing mnemonic names to refer to an expression and being able to later use those names.

5.5.2 We can create a function for similar graphics with few differences

The simple definition of a constant only works for identical graphics. It is common, however, to have the need to create multiple graphics that share a lot of similarities and only differ by a few details.

Books for children and newspapers sometimes feature a curious puzzle, dubbed "Spot the difference"[4], which asks the reader to spot all the differences between two

---

[4]A visual example can be found at `https://en.wikipedia.org/wiki/Spot_the_`

images that are identical except for a few differences. This game of "similarities and differences" can also be "played" by students when they recognize that two graphics have a lot in common.

Figure 5.6 shows two variants of the Pac-Man sprite, which are created with the code in Listing 12. A circular sector is created with the first radius "pointing towards 3 o'clock", and therefore needs to be rotated by a suitable angle to resemble Pac-Man.

The two graphics contain many similarities: they are both in the shape of a circular sector, they both share the same radius, and they are both yellow. One clear difference is the angle of the circular sector (280 degrees in one case, 340 in the other). The angle by which the circular sector is rotated is also different (40 and 10 degrees, respectively, assuming that the sector starts from "3 o' clock" and opens clockwise).
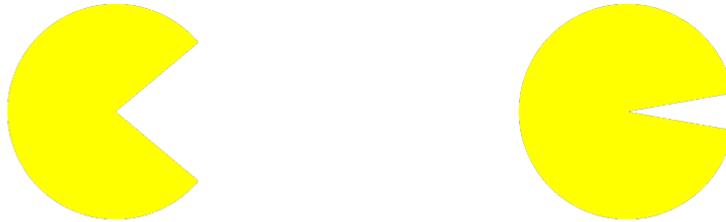
Figure 5.6. Two graphics representing the Pac-Man sprite with either a wide open mouth or an almost closed mouth.

```
pacman_open = rotate(40, circular_sector(200, 280, yellow))
pacman_closed = rotate(10, circular_sector(200, 340, yellow))
```

Listing 12. A program to create the two graphics of Figure 5.6.

Once we have recognized similarities and differences, abstraction is relatively straightforward. We can define a *function* whose body (i.e., the expression in the **return** statement) consists of the code in common between the two expressions, with similarities left as placeholders: `rotate(..., circular_sector(200, ..., yellow))`. We give a name to each difference, which becomes a parameter of the function, and is used in the expression to replace the `...`.

Listing 13 shows the result of this process of abstraction. The two expressions in Listing 12 have been replaced by a call to the newly introduced `pacman` function, which is able to create Pac-Man sprites with an arbitrary angle for the circle of the body and an arbitrary angle of rotation.

`difference`.

```python
def pacman(body_angle, rotation_angle):
    return rotate(rotation_angle,
                  circular_sector(200, body_angle, yellow))

pacman_open = pacman(40, 280)
pacman_closed = pacman(10, 340)
```

Listing 13. Introducing a function to abstract Listing 12 with one parameter for each difference.

Students may still rightfully object that there is only one true difference between the two graphics of Figure 5.6: how wide is the aperture of Pac-Man's mouth. There is indeed a relationship both between the mouth angle and the angle for the body (together they form a full circle) and the mouth angle and how much the circular sector needs to be rotated so that the mouth opens in the center (the rotation needs to be half of the mouth's angle).

Listing 14 shows the final result of this abstraction process. The pacman function has only a single parameter, the opening angle of the mouth, from which the two previous parameters can be computed.

```python
def pacman(mouth_angle):
    body_angle = 360 - mouth_angle
    rotation_angle = mouth_angle / 2
    return rotate(rotation_angle,
                  circular_sector(200, body_angle, yellow))

pacman_open = pacman(80)
pacman_closed = pacman(20)
```

Listing 14. Improving the abstraction of Listing 13: a Pac-Man can be drawn by specifying a single parameter.

This example was discussed here to illustrate a non-trivial case of abstraction. A simpler example, also based on graphical values, is presented in the chapter "From Repeated Expressions to Functions" of the textbook *A Data-Centric Introduction to Computing* [92]. After programming concrete cases of simple triband flags (e.g., the Italian and the French flags), learners can play this game of "similarities and differences" and identify that the differences only lie in the colors of the three stripes, which should
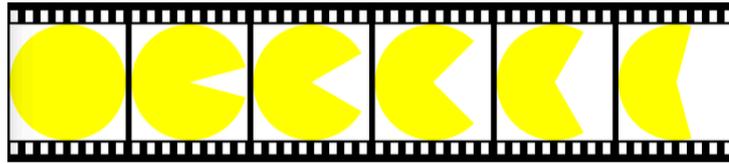
Figure 5.7. A filmstrip showing frames of Pac-Man with an increasingly open mouth.

become the three parameters of a generic function.

5.5.3   Functions can then be used to produce animation frames

An animation consists of a sequence of frames reproduced in rapid succession, fast enough to give our eyes the illusion of movement. Frames in an animation constitute an example of many graphics that are almost identical, except for a few differences. In the previous section, we discussed how that motivates the need to introduce a function to abstract over the differences.

Figure 5.7 plays once more with the Pac-Man sprite, showing it in six different frames with its mouth progressively more open. The angle at which the mouth is open is actually the only difference between the frames. We can use the pacman function implemented in Listing 14 to generate each frame, placing the sprite on a squared white background, aligned on the left.

The PyTamaro library offers a show_animation function to create and visualize an animation from a list of frames.

Initially, beginners can spell out the elements manually using a list literal (e.g., [frame(0), frame(20), ...]).

```
def frame(mouth_angle):
    background = rectangle(400, 400, white)
    return compose(pin(center_left, pacman(mouth_angle)),
                   pin(center_left, background))

frames = [frame(angle) for angle in range(0, 180, 20)]
show_animation(frames)
```

Listing 15. Extending Listing 14 to show an animation of a Pac-Man opening its mouth.

Once learners are familiar with lists, teachers can also introduce the range function

to generate a list of numbers[5] and then turn this list of numbers, representing the mouth angles, into graphics. This transformation could be performed using a loop, the `map` function, or a list comprehension as shown in Listing 15, depending on the teacher's intended learning goals.

The animated result of `show_animation` cannot be represented on a static page, but the reader should imagine the frames of Figure 5.7 being reproduced in a rapid sequence, in a loop, transmitting the illusion of movement.

5.6   PyTamaro can be used to create meaningful graphics

Teaching programming with PyTamaro does not need to be limited to flags or dry geometric shapes. Students can also create data visualizations that are meaningful to them; they may then synergistically explore the insights from the visualizations together with interesting aspects in the programs that draw them.
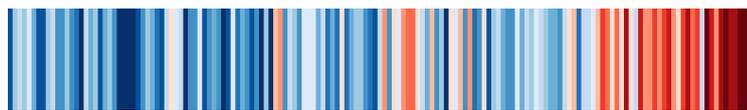
Figure 5.8. "Warming stripes" created with PyTamaro.

As an example, learners can draw the famous "warming stripes" visualization of temperature anomalies over time. Figure 5.8 shows an example created with PyTamaro, based on the data from the MeteoSwiss. There are several questions involved in creating such a graphic; here we only explore an important one related to PyTamaro's design. One of the tasks is placing a sequence of colored rectangles side by side, a problem that occurs frequently. It is worthwhile to define a general function `beside_many` to solve the problem once and add it to one's "toolbox" (like `square` from Section 5.2).

Depending on the educational context, this function may be implemented using a `for` loop as shown in Listing 16, recursion as shown in Listing 17, or the higher-order `reduce` function, which is shown in Listing 18.

All the three versions correctly handle the corner cases where the list is empty or contains just one graphic. The implementation is short and elegant, as it exploits the possibility of creating an empty graphic with no area. Composing an empty graphic with any other graphic simply results in the latter unmodified; this is no different from adding 0 to any number. In algebra, this distinguished element is called an *identity*.

---

[5]The `range` function technically produces a "sequence", an object of class `range`. One can iterate over such an object, or turn it into a full-fledged list with `list`.

```python
def beside_many(graphics: list[Graphic]) -> Graphic:
    result = empty_graphic()
    for graphic in graphics:
        result = beside(result, graphic)
    return result
```

Listing 16. An implementation of `beside_many` using a **for** loop.

```python
def beside_many(graphics: list[Graphic]) -> Graphic:
    if len(graphics) == 0:
        return empty_graphic()
    else:
        return beside(graphics[0], beside_many(graphics[1:]))
```

Listing 17. An implementation of `beside_many` using recursion.

The set of all graphics, together with an identity and an associative binary function for composition, forms a *monoid*. The monoidal flavor in PyTamaro has been explored to an extreme in Yorgey's powerful graphics library for Haskell [286]. That library is aimed at experts and features sophisticated concepts such as envelopes and traces; its extraordinary flexibility comes at the expense of ease for novices. We concur with Yorgey that "library design should be driven by elegant underlying mathematical structures" [286]. PyTamaro's design gives a taste of that elegance and power to novice programmers.

Figure 5.9 shows more examples of different kinds of graphics that can be meaningful to certain groups of learners. Creating a program to draw the periodic table of elements can reinforce concepts learned in a different subject (e.g., chemistry). Programming even the simple version depicted here requires thinking about the layout of the elements in periods and groups, the sequential atomic number of each element, and the coloring to distinguish between blocks. The pie chart that illustrates the breakdown of area per continent can be meaningful in the context of geography and/or data visualization, whereas a culturally meaningful graphic such as the Swiss railway clock represents a real-world object that may inspire learners, sustaining the engagement.
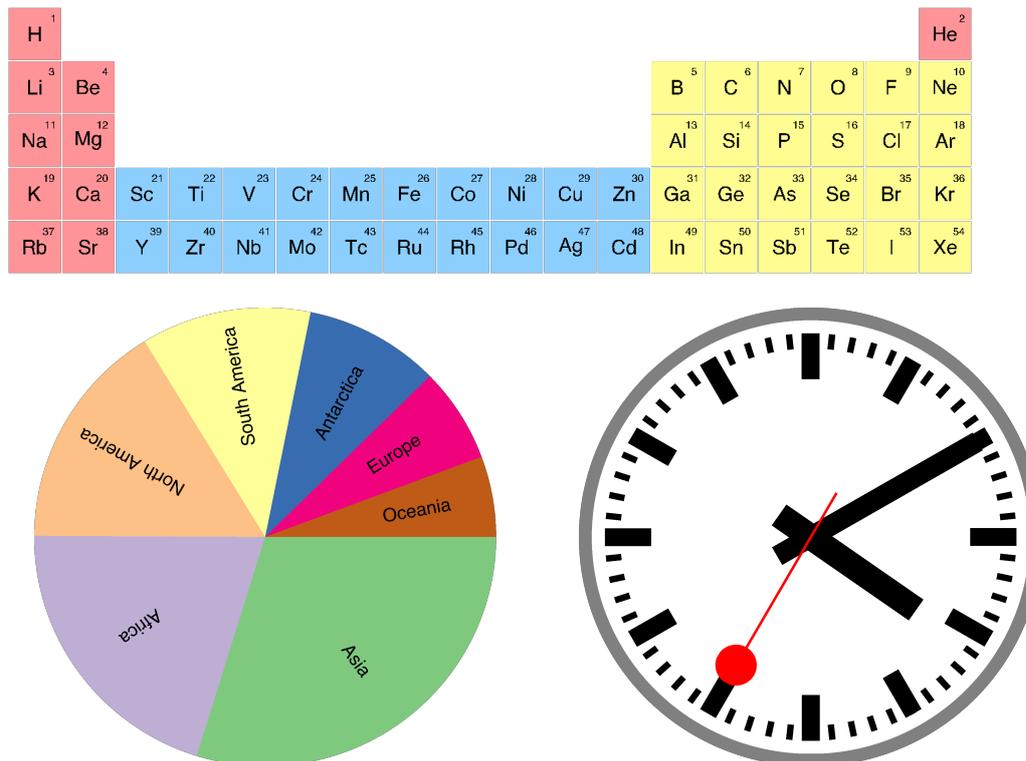
Figure 5.9. Three examples of meaningful graphics created with PyTamaro: the periodic table of chemical elements (top, first five periods only), a pie chart of area by continent (bottom left), and the Swiss railway clock (bottom right).

```python
from functools import reduce
def beside_many(graphics: list[Graphic]) -> Graphic:
    return reduce(beside, graphics, empty_graphic())
```

Listing 18. An implementation of `beside_many` using the higher-order function `reduce`.

5.7 PyTamaro programs only require a subset of Python, but are not limited to it

PyTamaro has been designed to avoid the pitfall of requiring many language features (Section 4.3.1). The initial example program shown in Listing 7 only required an import statement, assignment statements (to define constants), function calls, numeric literals, and the use of constants.

It would be possible to explicitly define a subset of Python in the vein of what Anderson et al. [11] did for JavaScript, who promised to apply the same process to Python. This subset would clearly define all the language features necessary to write programs with PyTamaro, including the definition of constants and functions as the two fundamental forms of abstraction.

The *explicit* use of a sublanguage remains relatively niche within programming education. Dedicated support from tools is necessary to exploit most of its benefits, including tailored error messages and warnings when using a construct outside the sublanguage (e.g., DrRacket's support for language levels [88]). Most instructors *implicitly* use sublanguages as they gradually explain new features of the language.

5.7.1 In a sense, PyTamaro is "functional" programming

Except for the functions that perform output, all PyTamaro functions are pure. Functions always produce immutable values. Behind the scenes, all PyTamaro graphics are represented as an immutable scene graph.

The absence of side effects and mutation operations are defining characteristics of "functional programming". In the "functional" paradigm of programming, "programs consume and produce values, and programming is viewed as the arrangement of functions to compose and decompose values (some have even dubbed functional programming as 'value-oriented programming')" [155].

The PyTamaro program shown in Listing 7 could in all regards be considered an example of "functional programming", especially when contrasted to the other programs shown in Chapter 3. In Listing 7, assignment statements are effectively used to

declare constants, there is no mutation, and the entire computation happens by calling functions and composing the results.

5.7.2   But compartmentalizing programming into paradigms is misguided

Traditionally, programming languages have been divided into "functional", "imperative", "object-oriented", or "logic" languages (and variations thereof: "procedural", "declarative", and so on). The human desire for a clear taxonomy is understandable, but is severely insufficient to characterize all the features each language actually supports. It is true that most programming languages are accompanied by a typical "style" in which most programs are written, but at the same time, it is also possible (and it is commonly done!) to mix and match different styles. Krishnamurthi [154] argues for moving beyond rigid paradigms and thinking more about languages as the result of an aggregation of features. This approach to languages as a combination of features is explored in a textbook [153] and in research [44].

Many commonly used modern programming languages share a significant part of the semantics that enables programming in multiple "paradigms". In fact, we have already encountered a brief but significant example of this phenomenon when presenting three different implementations of `beside_many` in Listings 16 to 18. All three programs are written in Python and require a mechanism to repeat computation. Listing 16 achieves this using a `for` loop: this style of writing programs is arguably typical for Python, pervasive in education, and a signature of "imperative programming". Listing 17 solves the problem with recursion: the approach is atypical for Python's lists, which are akin to vectors, but still leads to a correct implementation. Recursion is often considered the epitome of "functional programming", even though many "imperative" programs use recursion as well. Finally, Listing 18 uses a higher-order function, another signature of "functional programming". However, Python programmers frequently use list comprehensions, which are close cousins of another higher-order function, `map`.

To complicate matters even further, even within the same program, it is common to adopt different styles of programming. Programming courses for beginners in Python often include a section that uses lists. A Python program, for example, may declare a list `numbers` and use the instruction `numbers += [1]` to add a new element. Undoubtedly, this program uses mutable objects, one of the typical characteristics of "object-oriented programming". However, few teachers would claim that an otherwise predominantly "imperative" program is also following the "object-oriented" paradigm.

These simple examples should suffice to prove that it is unwise to claim that "teaching in Python" means "teaching imperative programming". Acknowledging this wide

variety of approaches, PyTamaro does not prescribe a specific "style" of programming or a "paradigm" that teachers must follow at all costs.

At its core, PyTamaro strictly maintains its promise of not using mutation and offering an API that does not expose objects. The teacher is ultimately responsible for making choices for the rest of the program, considering the specific constraints and goals of their context.

5.8 The PyTamaro approach is not confined to an English API in Python

Ideally, a graphics library for introductory programming should be accessible to large audiences. So far, we have described a library for beginners who program in *Python* and are competent in *English*. These assumptions emphatically do not hold for all beginners.

5.8.1 The design can be implemented in other programming languages

First, not all introductory programming courses use Python as a language. PyTamaro was originally implemented in Python to serve the needs of our specific contexts: it was first used as part of teacher training programs in Switzerland, and all those teachers planned to use Python with their students. In Section 2.4, we discussed how Python is currently the language of choice for many introductory programming courses.

The minimalism of PyTamaro's design reduces the cost of porting the library to new programming languages. Our research group developed JTamaro[6], a Java library that is a close cousin of PyTamaro and is used in a first-year university course. At its core, JTamaro follows the exact same design as PyTamaro, offering static methods to create and combine graphics.

5.8.2 PyTamaro is localized for natural languages

Second, students' learning of programming should not be hampered by their native language. Research has shown that the prevalence of English can be a barrier for non-native speakers [104] given that students have to learn the programming language and the foreign natural language at the same time. This is especially true now: students around the world start programming at increasingly young ages and are not necessarily comfortable reading and writing English.

---

[6]https://github.com/LuCEresearchlab/jtamaro

PyTamaro attends to this. The documentation of the library is available in multiple languages, and the entire API is localized. For example, an Italian-speaking student can make a triangle using the function `triangolo`, whose parameters, types, and error messages are localized; a German-speaker can use `dreieck`. (Some examples of using localized APIs are shown in Chapter 11, which reproduces excerpts from teaching materials created by Swiss high school teachers.)

5.9 PyTamaro's minimalism is only in service of learning

The minimalism of PyTamaro's design is intended to sustain the engagement (Section 4.2) and to keep the complexity of the programs low (Section 4.3). However, some readers may find the example programs with PyTamaro presented so far in this chapter not "minimal", under some definition of minimalism.

Minimalism for its own sake is not among our goals. Below we discuss three senses in which the programs presented so far are not minimal, and we argue why they are nonetheless pedagogically justifiable.

5.9.1 Minimal does not mean only one primitive

PyTamaro includes only six functions to create primitive graphics: `rectangle`, `triangle`, `ellipse`, `circular_sector`, `text`, and `empty_graphic`. The first four functions cover basic geometric shapes and have been selected to enable learners to draw common graphics, such as the Pac-Man sprite shown in Figure 5.4.

These functions partially overlap: it is possible to produce the same graphic with appropriate calls to two different functions. A circle, for example, can be produced using the `ellipse` function specifying the width to be the same as the height, or using the `circular_sector` function with an angle of 360 degrees. A hypothetical `elliptical_sector` function would be a generalization that would cover all cases, but explaining its behavior to learners would be more challenging.

Even more, a rectangle could be obtained by composing two triangles. Indeed, a `triangle` function would be the only necessary primitive. Even text rendered with the `text` function could be composed from tiny triangles: computer graphics approaches use this as a tessellation technique, turning each polygon into a triangle.

Working exclusively with triangles would make it extremely challenging for novices to program even simple graphics. Instead, PyTamaro's pedagogical approach to teaching programming favors giving students a few carefully chosen pieces to use as "building blocks".

5.9.2 Minimal does not mean only one combinator

A similar argument applies to PyTamaro's functions to combine graphics. Listing 7 shows a program with PyTamaro that draws a simple house, using the above combinator to place the roof on top of the floor. Together with beside and overlay, these three combinators allow learners to create many graphics without needing particular forms of alignment.

Later on, students discover the need for more flexible alignment options, which are achieved using the compose combinator with the pin function. The functions above, beside, and overlay are special cases of a specific combination of compose and pin to place graphics next to or on top of each other, aligning them at their center.

Listing 19 illustrates how the convenient above function can be "desugared" into a combination of compose and pin. Strictly speaking, this implementation proves the combinator above as superfluous, together with its siblings beside and overlay. However, if PyTamaro did not include them, programming the simple house of Figure 3.1 would have as a prerequisite the explanation of pinning positions. Right at the beginning, when a learner already needs to acquire numerous programming concepts, we would also need to explain an intricacy that is only necessary in the graphics domain. Here too, we carefully weighed the tradeoffs and favored a simpler learning curve at the beginning over the minimalism of a single combinator.

```python
def above(top_graphic, bottom_graphic):
    return pin(center, compose(pin(bottom_center, top_graphic),
                               pin(top_center, bottom_graphic)))
```

Listing 19. Desugaring above into a combination of compose and pin.

5.9.3 Minimal does not mean as few characters as possible

Listing 7 includes on its first line a rather long import statement, which is necessary to use two color constants and four functions from the PyTamaro library. Some programming teachers with Python experience may protest: it is much easier to write `from pytamaro import *`!

A meaningful answer to this objection requires a better specification of what it means for a program to be "easier". If the metric is the number of characters (or the number of lines of code), by all means a "star import" is the clear winner. If instead by "easier" it is meant something along the lines of "a student can precisely understand every part of the program", the matter becomes more subtle.

Common introductory example programs, such as the one to determine whether a number entered by the user is even or odd (Listing 1), often include built-in functions that can be used without any import. Students can end up treating functions such as `input` and `print` like "magic entities", baked in the programming language. Often, they are not perceived by novices as regular functions defined by a programmer and part of the standard library. Requiring students to explicitly list the names they import introduces a small inconvenience—which some code editors alleviate with automatic completion mechanisms—but reduces the illusion that programs work "magically".

5.10   The minimalism also brings limitations

The minimalism of the PyTamaro API introduces some limitations that we discuss here explicitly.

5.10.1   Working with a bounding box can be limiting

A bounding box is the smallest, axis-aligned rectangle that fits the graphic. PyTamaro allows pinning a graphic at nine points along its bounding box. Three points are offered for each dimension: two at the extremes and one in the middle.

In addition, each primitive graphic has its pinning position at a specific point. This usually coincides with one of the nine points determined by the bounding box, but there are exceptions: triangles have their default pinning position on the centroid, and graphics representing text are produced with their pin at the left end of the text baseline.

This reduced set of "points of interest" still allows composing many graphics. For example, triangles can be "stacked" on their centroid, and two rectangles can touch on their edge to form a "staircase". Other graphics, however, are much harder or impossible to express with PyTamaro.

For instance, aligning a square and a circle so that their boundaries are tangent at a 45-degree angle [286, Fig. 4] requires more sophisticated alignment features. Yorgey [286] suggests the use of "envelopes" to find the correct position to place two graphics beside each other along an arbitrary vector, instead of just the two axis-aligned ones.

Moreover, in PyTamaro, once a graphic is composed with another graphic it is only possible to refer to the nine points on the bounding box of the *resulting* graphic. PyTamaro does not provide a way to query the tree of graphics and refer to a point that was accessible only in a sub-graphic. Sophisticated libraries such as Haskell's `diagrams` [286] enable users to attach mnemonic names to certain locations, and to later perform queries on composed graphics using those names. This adds more

flexibility, but also introduces a new set of design questions. Different graphics can contain points with the same name and then be combined, or the same graphic with a named point can be included multiple times in a composition. This introduces the need for a mechanism to disambiguate between these cases.

Overall, PyTamaro is not intended to become a full-fledged graphics library for drawing arbitrarily complex graphics. Given that the presented set of alignment features already enables students to draw a broad class of graphics, we decided against introducing these advanced features, remaining faithful to PyTamaro's pedagogical goals.

5.10.2   A local coordinate system can be reintroduced

Although a local coordinate system can be misused by students (Section 4.1.3), graphics may be inherently specified with some parts at precise coordinates. This is frequently the case for graphics that reproduce a physical or virtual 2D world.

In these cases, the graphic should first be decomposed to identify the most specific part that requires a local coordinate system. The remainder of the graphic can be composed as usual. For the part that requires placing a graphic onto another graphic that acts as a background, one can use a transparent rectangle, sized to match the coordinates, to create a larger graphic that can be appropriately pinned and composed with the background.

When teaching with graphics that use a coordinate system, such as with one of the common libraries that implement a global coordinate system (Section 3.3), understanding where the origin of the coordinate system is located is one of the first steps necessary to draw even the simplest graphics. Students are typically acquainted with coordinates from mathematics at school, where the focus is on the upper-right quadrant of a Cartesian plane. The origin is located in the bottom left corner, with "y" coordinates that grow when going upwards. On the contrary, most libraries adopt the so-called "computer graphics" coordinates, where the origin is located in the upper left corner and "y" coordinates grow going downwards.

Given that PyTamaro does not offer a local coordinate system, its implementation by a student or a teacher needs to explicitly decide between one of the two systems. Listing 20 shows an example of a generic function that uses PyTamaro to place a foreground graphic onto a background graphic at specific coordinates. Depending on the fifth parameter, it uses either the "mathematics" coordinates or the "computer graphics" ones. One can then define specialized functions, like `place_at_maths_coords` and `place_at_graphics_coords`, to work within the desired system.

Figure 5.10 shows the result of placing a small red dot onto a background scene at specific coordinates. The two graphics have been altered to indicate the transparent rectangle as a partially transparent white rectangle. When using an actual transparent

```python
def place_at_coordinates(foreground: Graphic,
                         background: Graphic,
                         x: float, y: float,
                         is_math: bool) -> Graphic:
    offset = rectangle(x, y, transparent)
    foreground_offset = compose(foreground,
        pin(top_right if is_math else bottom_right, offset))
    origin = bottom_left if is_math else top_left
    return compose(pin(origin, foreground_offset),
                   pin(origin, background))


def place_at_maths_coords(foreground, background, x, y):
    return place_at_coordinates(foreground, background, x, y,
    ↪  True)


def place_at_graphics_coords(foreground, background, x, y):
    return place_at_coordinates(foreground, background, x, y,
    ↪  False)
```

Listing 20. Two functions which use PyTamaro to place a graphic at specific coordinates, using either the "mathematical" or the "computer graphics" coordinate system.

rectangle, a graphic can be placed at specific coordinates without altering the background scene.

The possibility of reintroducing a local coordinate system is both a strength and a limitation for PyTamaro. On one hand, an entire class of graphics, such as a blue sky with stars at specific coordinates, can still be drawn within this approach. On the other hand, once the possibility is revealed, students may begin to overuse it or apply it inappropriately to position graphics arbitrarily. This was the pitfall described in Section 4.1.3, which we strove to eschew.

5.11 The PyTamaro approach goes beyond the design of a library

This chapter presented the design of PyTamaro and offered some examples of how it can be used to teach programming. The following chapters complement the design of the library with a pedagogy and software systems that build on PyTamaro's strengths,

(a) "Computer graphics" coordinates.

(b) "Mathematics" coordinates.

Figure 5.10. A red dot placed at coordinates $(120, 40)$ onto a gray background.

focusing on teaching programming to complete beginners, emphasizing abstraction and problem decomposition.

Chapter 6

With TamaroCards, Programming Can Be Introduced Unplugged

After introducing a Python library, we now present an approach for teaching introductory programming using PyTamaro without starting with writing Python code, answering the question:

RQ How can the PyTamaro approach be taught initially without computers, eventually transitioning to a text-based programming language?

This chapter discusses an unplugged approach we dubbed TamaroCards, showing how it supports abstraction and problem decomposition as our focus when learning to program. It presents a systematic process to transition from unplugged activities to correct Python programs. A brief description of a curriculum used in middle school concludes the chapter as a concrete example of TamaroCards usage.

6.1 Programming can be initially taught unplugged

Despite the most popular name with which the field is known, Computer Science is not only about computers. The Computer Science Unplugged project [22] consists of a series of activities aimed at teaching important ideas in Computer Science without the requirement to use a computer. Since then, the original set of activities of 1998 has been evolving and has also been complemented by independent but related educational projects. Nishida et al. [191] attempt to characterize the commonalities among the many different activities: they do not involve computers at all, they are generally based around a game or challenge and possibly include a story, they are kinesthetic (they use physical objects), they involve students directly, and they are easy and inexpensive to set up.

Unplugged activities have become popular and they are now commonly recommended in teacher training materials. Bell and Vahrenhold [21] however note that relatively few studies have empirically demonstrated their effectiveness.

In some cases, the requirement of not using computers in unplugged activities is interpreted as an absence of programming (e.g., [21]). This characterization, however, refers to a narrow view of programming as an activity in which a program is encoded in a programming language—often, a textual programming language—and is then executed by the computer.

Alamer et al. [7] present a set of activities, collectively named "Programming Unplugged", used to teach key programming concepts with positive preliminary results. Hermans and Aivaloglou [119] describe a study in which one of the two groups worked on unplugged activities that covered a few programming concepts before programming in Scratch.

Pedagogies that include unplugged programming activities normally use them as a starting point, before students move on to a "plugged" version of programming [187]. This form of sequencing appears to yield better outcomes [187]. Key elements for the success of these activities seem to involve adequate semantic profiles (unplugged activities start from a "low density" [176]), working within the zone of proximal development [270], and an explicit "notional machine" [86].

6.2 Unplugged programming is related to tangible notional machines

The term "notional machine" was introduced by Du Boulay in an influential paper that argued how students need to learn "the general properties of the machine that one is learning to control" [77]. The somewhat broad definition and the centrality of the issue led to the term being adopted to refer to many related but distinct concepts in computing education research. As Duran et al. points out: "In practice, we have often found it difficult to guess at what colleagues and authors precisely mean when they say or write 'notional machine'" [78].

The perspective adopted in this thesis matches the characterization recently proposed by an international working group: "A notional machine is a pedagogic device to assist the understanding of some aspect of programs or programming" [86]. Fincher et al. [86] collected 43 notional machines used by educators in their teaching practice. Notional machines are categorized according to whether they establish an analogy, such as a variable as a parking space, or they are representations, such as a visual representation of the memory stack of a running program. Representations can be drawn manually or automatically generated by a machine.

Some notional machines are primarily based on or include physical manipulatives. For example, students may use clothespins to hold onto a (paper) value, learning that a variable can only hold one value at a time. As an example of a more advanced topic, a linked list can be made tangible by representing each node and references with a piece of paper. Students can move the paper to illustrate, for example, a reference "moving" over the nodes during a list traversal. Notional machines that belong to this group can be classified as *tangible* notional machines.

Embodied cognition posits that cognitive processes are rooted in the interactions of the body with the world [279]. More specifically, one of the claims that has found some empirical support is that "We off-load cognitive work onto the environment. Because of limits on our information-processing abilities (e.g., limits on attention and working memory), we exploit the environment to reduce the cognitive workload. We make the environment hold or even manipulate information for us, and we harvest that information only on a need-to-know basis." (ibid.). This perspective may justify the potential value of tangible notional machines, which serve us as an external representation that we manipulate with our bodies.

6.3 TamaroCards is a notional machine for PyTamaro expressions

One of PyTamaro's explicit goals is to minimize the number of language constructs needed at the very beginning to create the first graphics. Indeed, after suitable imports, the program shown in Listing 7 to draw a house with PyTamaro only requires using constants, functions, and numeric literals.

However, these three concepts also need to be gradually understood by students, who may only vaguely recognize the concept of a function from mathematics. Even the mere act of typing such a program in an editor can be a problem.

Deviations from the correct syntax of the programming language result in the computer refusing to execute the program. Syntactic errors are often easier to overcome than semantic ones, but cause significant frustration for beginners. Drosos et al. [76] analyzed the sentiment and the Python code written by learners, revealing how `SyntaxError` was the strongest correlating feature with frustration. (The same study also noted that global variables, a seemingly convenient but complex feature of Python, are also a frequent source of frustration.)

Listing 21 is a variant of Listing 7, without the introduction of names, the import statement, and the instruction to output the graphic on the computer. This program simply consists of a single expression that composes graphics to create a house.

```
above(
    triangle(100, 100, 60, red),
    rectangle(100, 100, yellow)
)
```

Listing 21. A single expression to create a house with PyTamaro.

Computing education research has proposed strategies to teach students a viable computational model to work with expressions.

Schanzer proposed to use "Circles of Evaluation", a notional machine that represents each expression as a circle, and the nesting of expression as nesting of circles [234, Fig. 15]. In his work, he also showed an example of using "Circles of Evaluation" for an expression that produces an image [234, Fig. 16]. The notional machine, referred to as a "visual-spatial metaphor", is supposed to "dramatically accelerate the speed at which students pick up the Racket programming language, and eliminate many of the most common syntax errors" [234].

Hauswirth used a related notional machine, called "Expression as Tree" [86, Fig. 28], to bring out the explicit structure of an expression as a tree. Each expression is represented as a node, which may contain "holes" as placeholders for arbitrary subexpressions. Subexpressions are represented as nodes connected with an edge to the parent node. Bevilacqua et al. [26] studied the use of this notional machine in exams by analyzing "Expression as Tree" diagrams hand-drawn by students. A question based on the notional machine captures several mistakes, some of which can be explained by already documented misconceptions or suggest new ones [26].

6.3.1 TamaroCards uses physical cards to represent programming constructs

To learn the first programming concepts with PyTamaro without the burden of Python's syntax, we developed the TamaroCards notional machine. The first version was created for a summer workshop dedicated to teachers who were considering the adoption of PyTamaro.

TamaroCards is a *tangible* notional machine, to be used unplugged. Each construct in the programming language corresponds to a physical card. Figure 6.1 shows an example of three cards. The literal 250 is simply written on a piece of white paper. Constants are represented with rectangular blue cards, such as PyTamaro's `red` constant for the red color. Red cards with the shape of a rounded rectangle represent functions, such as PyTamaro's `rectangle` function to create a rectangle given its width, height,

and color.

Literal   Constant   Function
250   red   rectangle(■,■,■)
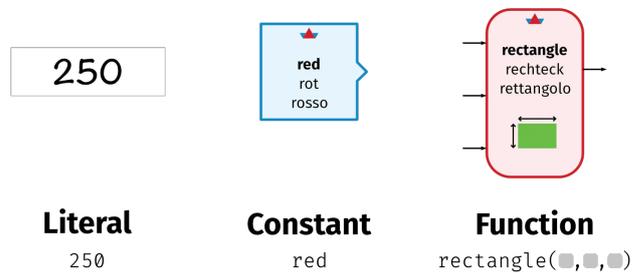
Figure 6.1. Example of three cards in the TamaroCards notional machine.

Constant cards have an arrow on the right, to signal that they stand for a value when used. Function cards have as many arrows on the left as the number of parameters of the function they are representing, and an arrow on the right to indicate that a value is produced when the function is used.

The name of each constant and function card is shown in bold in the center. As explained in Section 5.8.2, the PyTamaro API is available in multiple different natural languages. Cards feature the name in the English API in bold, reflecting its prevalence, but also include the name in German and Italian, the two next most commonly used languages.

Even though the notional machine has not been fully formalized, this explicit correspondence between the elements of the notional machine and the elements of the programming language aims to avoid an unsound mapping between the two [185].

6.3.2 TamaroCards can be seen as a visual programming language

We introduced TamaroCards as a notional machine, but TamaroCards can also be seen as a *visual programming language* for a small language without control-flow and mutable state. The emphasis is on the propagation of values, which correspond to the composition of graphics in PyTamaro. In a sense, thus, TamaroCards is a *data-flow language* [137].

Following the composition approach adopted by PyTamaro, this data-flow visual programming language for novices contrasts with common educational visual programming languages like Scratch [221], which are based on a block paradigm that focuses on "imperative", structured programming.

6.3.3 The house example can be created with TamaroCards

The physical cards are distributed to students at the beginning of an activity. At the very beginning, it is possible to distribute only a subset of all the cards representing the PyTamaro API (e.g., avoiding the more flexible combinator `compose`).

Students work on a table using a drawing surface, such as a sheet of brown paper. To create a graphic, they choose the cards they need and arrange them on the table. With a pencil, students draw lines to establish connections between cards.

Figure 6.2 shows the cards connected together to compose the house. The physical nature of the cards allows them to be rearranged easily, thus enabling multiple strategies to solve problems. In a "bottom-up" approach, a student could recognize the roof, grab the necessary `triangle` card, and start composing the house from there. Alternatively, using a "top-down" approach, a student could start by realizing that the house is made of two pieces one above the other, and start by placing the `above` card on their table.
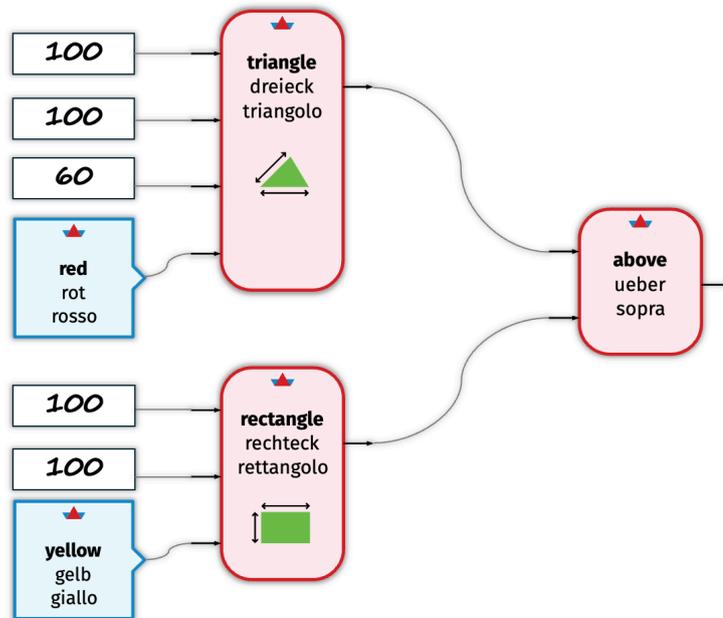
Figure 6.2. Using TamaroCards to compose a house with PyTamaro.

Once the expression is composed, students can evaluate it. The evaluation of a TamaroCards expression needs to start from the leaves. Students can reach the leaves by checking if a card has further connections on its left. The evaluation proceeds from the left to the right. The emphasis is placed on the composition of graphics, which can be drawn by the students on their table using colored pencils, or composed physically using colored paper cut-outs.

Figure 6.3 shows the result of evaluating the TamaroCards expression of Figure 6.2. The final graphic is shown next to the root of the expression (`above`, in this case). Visualizing the intermediate values of the computation can be helpful to find issues, provided that the evaluation is carefully done step-by-step.
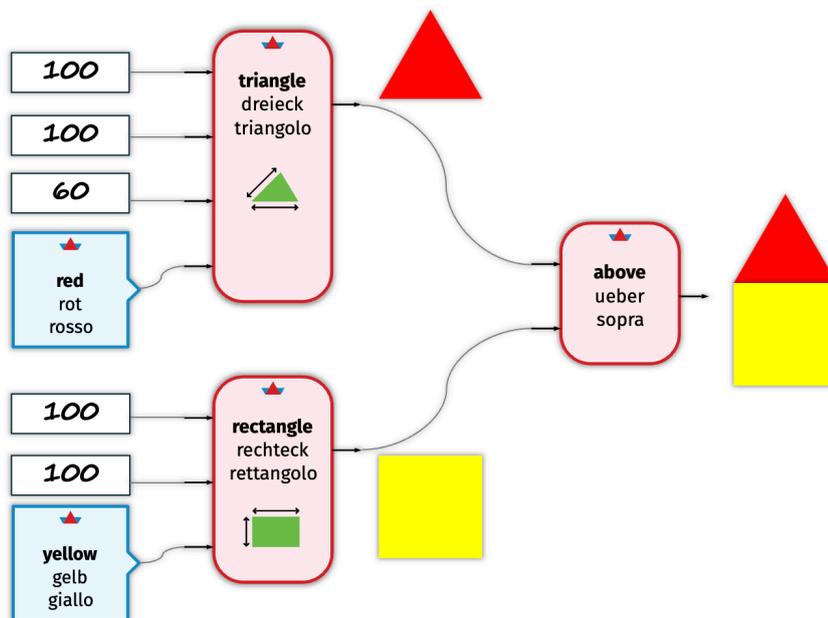
Figure 6.3. Evaluating the graphics of a TamaroCards expression.

Evaluating a TamaroCards expression with an unknown result can also be a fun and instructive activity for pairs or groups of students. Figure 6.4 shows a photograph taken during a summer workshop: students receive a TamaroCards expression without being told what it produces, and they need to evaluate it to determine that it is the composition of a heart.

6.3.4 Cards also serve as documentation

The careful evaluation of a program in TamaroCards requires avoiding second-guessing what is the behavior of each card. In the example of Figure 6.3, it would be uncommon for students to connect the `triangle` card to the second incoming arrow of the `above` card, and viceversa for the `rectangle` card, effectively swapping the arguments to the `above` function. Even so, their intention is to create a house with a roof above the floor, and the final house may still be drawn correctly.

When programming, a teacher would normally correct their student or direct them to the documentation of the API they are using. For this unplugged activity, an *anno-*
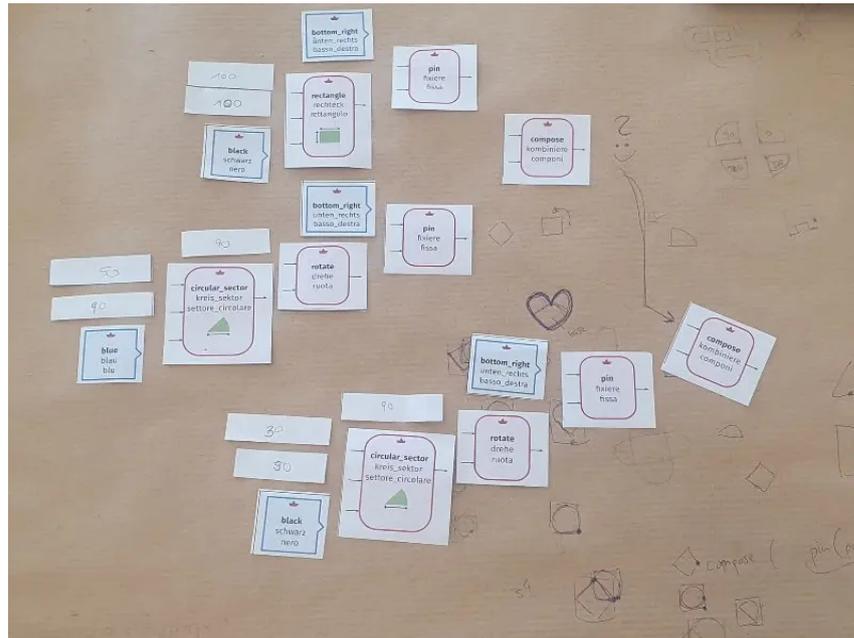
Figure 6.4. Photograph of an evaluated TamaroCards expression taken during a summer workshop.
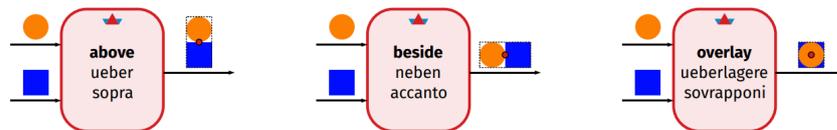
Figure 6.5. Documentation version of TamaroCards for three functions to combine graphics.

tated version of the cards plays the role of documentation. The first card on the left of Figure 6.5 shows a simplified form of documentation for the above function. Unlike *code cards*, which are used to compose the expressions, *documentation cards* feature annotations on the arrows corresponding to the arguments and the arrow for the return value, illustrating an example of usage of the function card.

Documentation cards do not fully specify the behavior of a function, which is usually introduced orally by the teacher. Their purpose is to remind students of the correct behavior, such as the order in which arguments need to be specified. Students can visually see that above produces a graphic with the first graphic placed above the second graphic, and—perhaps less obviously—that overlay places the first graphic in the foreground and the second graphic in the background (third card of Figure 6.5).

The annotations featured on the documentation cards are not included in the code

cards, because they feature concrete graphics that differ from the actual graphics used in the TamaroCards expression created by the student. Using annotated cards also to compose graphics would likely confuse learners.

6.4 Students follow a systematic process from cards to code

Starting with unplugged activities based on TamaroCards is intended to be a stepping stone for learners, who eventually need to transition to using Python and have a computer execute their programs. How can a student who successfully created a house using TamaroCards (Figure 6.2) write a syntactically correct Python expression?

Given the direct correspondence between cards in TamaroCards and constructs in the Python programming language, we can teach a small set of rules to express a TamaroCards expression as Python source code.

The process begins by identifying the root of the expression: the card without any outgoing connection. We need to deal with three types of cards:

- Literal cards are simply (literally!) written as they are. When a literal card shows the number 100, it is also written as '100' in Python code.

- Constant cards are represented by their name, written in the center of the card. A constant card representing PyTamaro's red color is written as 'red' in Python code.

- Function cards also start in Python with their name (e.g., 'above'), followed by an open parenthesis '(', the representation of any card connected to the incoming arrows separated by a comma ',', and a closed parenthesis ')'. To express each card connected on the left, repeat the same process again.

Figure 6.6 shows an instance of the process applied to the TamaroCards expression previously shown in Figure 6.2. This expression is non-trivial to represent in Python, as it already has three different levels of depth. Students can follow a mechanical, systematic process that guarantees that they will always end up with a syntactically correct Python expression, provided that they follow the process diligently.

For illustrative purposes, Figure 6.6 shows each Python token superimposed on the cards. In practice, students would type in Python code directly in an editor on their computer or write it on a separate piece of paper. To avoid losing track of the point of the process completed so far, students can incrementally draw a "path" (sequence of blue arrows in Figure 6.6) over their TamaroCards expression.
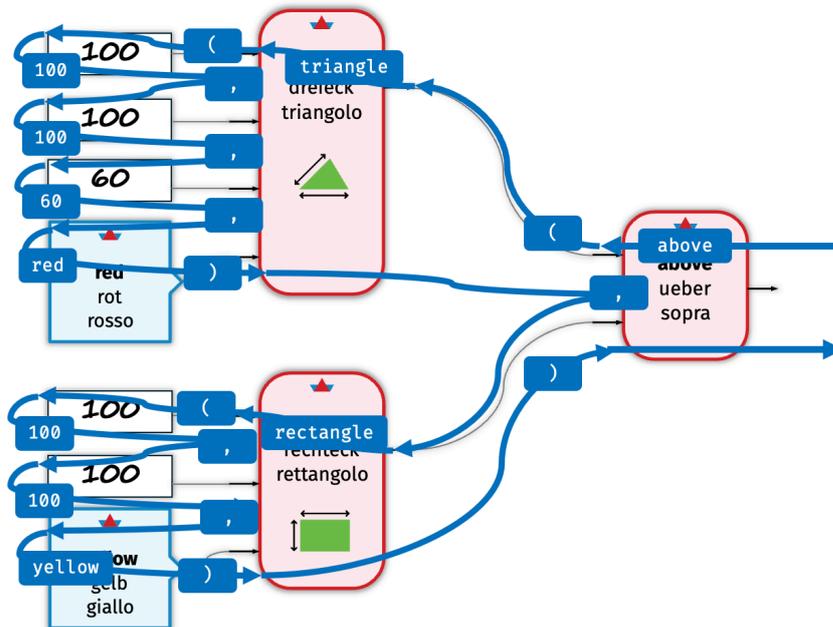
Figure 6.6. Systematic process to turn the TamaroCards expression to create a house into a Python expression.

Students get acquainted with the process at the very beginning and use it systematically. This helps to reduce the frustration around syntactic errors, which are frequent among beginners. Over time, expressions become bigger and the process becomes superfluous. Still, teachers can remind their students to apply the process whenever they incur in a `SyntaxError`.

6.5   TamaroCards can also help with misconceptions

Teaching students to use TamaroCards and systematically translate TamaroCards expressions to Python code can also be helpful to prevent or overcome known programming misconceptions. We briefly discuss three example cases, considering the misconceptions from the `progmiscon.org` inventory [52].

Each expression in TamaroCards is represented with a physical card, including constants and literals. This counters the widespread [48, Tab. 2] NoAtomicExpression misconception: the belief that "expressions must consist of more than one piece"[1] and thus atomic pieces need to be treated differently than "regular", non-atomic expressions.

---

[1] `https://progmiscon.org/misconceptions/Python/NoAtomicExpression/`

The evaluation of a TamaroCards expression proceeds from the leaves to the root. This explicitly combats the OUTSIDEINFUNCTIONNESTING misconception ("Nested function calls are invoked outside in"[2]).

Finally, the systematic process to translate a function card in TamaroCards to Python code addresses the misconception PARENTHESESONLYIFARGUMENT[3], which causes students to omit parentheses when a function has no arguments. This misconception is considered very prevalent by high school teachers [48, Tab. 3].

6.6   Teaching abstraction already starts with TamaroCards

Students need to use a significant number of cards to create even moderately elaborate graphics. When an expression becomes large, representing it with TamaroCards becomes unwieldy. This is no different than with a textual programming language.

In Section 5.5, we discussed how more complex graphics can contain parts that are identical or very similar except for a few differences. In those cases, abstraction helps to reduce the complexity and work with fewer "pieces".

Physical cards make the pain of not using adequate abstractions very tangible. Cards cannot be "copied" easily: if a graphic contains a sub-graphic twice, all the cards required to draw the sub-graphic need to be arranged twice on the table, and also connected twice.

In such a painful situation, the learners demand "a better way to do it". This is the first condition for conceptual change (in its more radical form, accommodation), as theorized by Posner et al.: "there must be *dissatisfaction* with existing conceptions" [215].

TamaroCards attends to this and offers two mechanisms of abstraction. Special care is taken so that they also fulfill the remaining three conditions suggested by Posner et al. [215]: the new abstraction mechanisms need to be *intelligible* for students, appear *plausible* from the beginning to solve the problem with the existing way they solve problems, and seem *fruitful* in enabling students to solve a new class of problems.

The card in Figure 6.7a can be used by a student to define their own constants. Students write the name of the constant in the triangle at the top, before the =. This triangle acts as a "roof" that houses an entire TamaroCards expression, which should be placed immediately below it. Once defined, the card shown in Figure 6.7b can be filled with the name of the constant and normally used to compose other TamaroCards expressions.

Deliberately, the cards for using the constants defined in the PyTamaro library or by the user look alike: they differ only for the PyTamaro logo, present in the former

---

[2]https://progmiscon.org/misconceptions/Python/OutsideInFunctionNesting/
[3]https://progmiscon.org/misconceptions/Python/ParenthesesOnlyIfArgument/

and absent in the latter. This helps students realize that the constants they define can be used in the exact same way as the constants defined in a third-party library.

(a) Definition.   *(b)* Use.
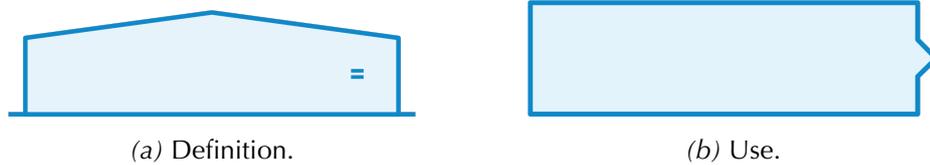
Figure 6.7. Empty cards to define and use a student-created constant.

Functions adopt the same idea. The card in Figure 6.8a defines a function. It is recognizable by the red color with rounded borders and the keyword `def` at the top, helping to bridge to Python's syntax for defining functions. Once defined, functions can be used by supplying the intended arguments to the card of Figure 6.8b.

(a) Definition.   *(b)* Use of a two-parameter function.
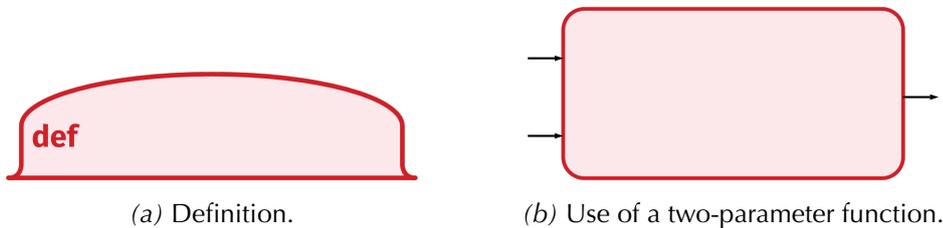
Figure 6.8. Empty cards to define and use a student-created function.

Combined, these features in TamaroCards enable teachers to focus on abstraction even before using the computer.

6.7 We piloted a curriculum in a middle school using Tamaro-Cards

The effectiveness of TamaroCards has not yet been empirically evaluated, but their suitability to be part of an introductory programming course has already been tested in practice.

In close collaboration with a middle school teacher, we developed a curriculum for an elective course on programming. The target audience for this curriculum is 9th graders in Tessin, an Italian-speaking canton in the south of Switzerland.

The curriculum introduces programming with PyTamaro and makes extensive use of TamaroCards[4]. The curriculum consists of a sequence of units.

---

[4]An English translation of the booklets forming the curriculum is available at https://luce.si.usi.ch/composing-python/.

The first unit introduces what it means to program, what is a programming language, and what are its characteristics compared to a natural language.

The second unit discusses problem decomposition using graphics as a domain. Students decompose both elementary and more complex graphics, such as a screenshot of an app on their smartphone, to recognize the elementary graphics.

The third unit focuses on unplugged programming. Students use the cards from TamaroCards to compose simple graphics, such as our recurring example of a house, the Swiss flag, or a pair of eyes. The lessons also teach a form of program tracing: how to interpret a TamaroCards expression to determine the resulting graphic. Figure 6.9 shows an excerpt from the student workbook for this third unit. Students use Tamaro-Cards to build the necessary parts to create the house. In the middle of Figure 6.9, there are cards in their documentation variant.
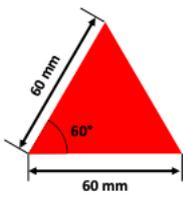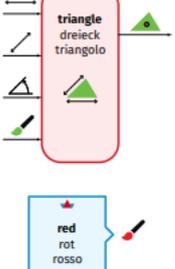
Figure 6.9. Excerpt from the workbook asking students to use TamaroCards to create the roof of a house. Translation: "Program a red, triangular roof", "1. Cut with your scissors", "Library", "2. Create the program with the cards".

In the fourth unit, students learn the process necessary to go from cards to code (Section 6.4) and start programming in Python. Activities include *first* an unplugged part to program with the cards and *then* the corresponding "plugged" part using the computer. The translation process from cards to code is first exercised with one individual shape, before advancing to the more elaborate house example of Figure 6.6. Figure 6.10 shows an excerpt from the workbook that illustrates how to turn a single card into Python code, and then asks students to apply the process on a different example, explicitly recommending drawing the "path" (Figure 6.6).

The execution of Python programs with the PyTamaro library occurs in a web environment which will be described in the next chapter (Chapter 7). We created dedicated activities for this middle school curriculum so that students can take their first steps
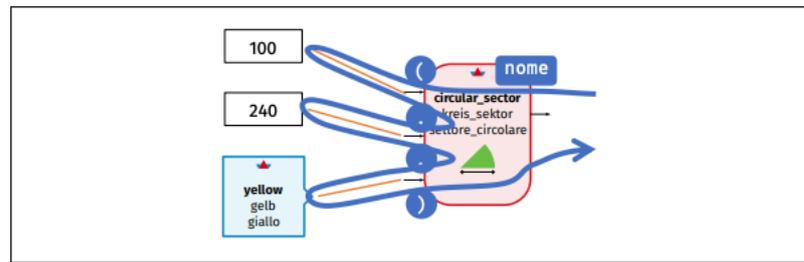
Traduci il seguente programma nel codice Python e disegna la strada:

Programma con le cards	Programma in Python

Figure 6.10. Excerpt from the workbook asking students to translate a TamaroCards expression to Python code. Translation: "Translate the following program into Python code and draw the path", "Program with the cards", "Program in Python".

just by focusing on one single expression (initially not worrying about the import or the output).

The curriculum continues with a unit on colors and one on more advanced forms of composition with `pin` and `compose`. Next, a unit motivates and explains the need to define our own functions to abstract. Figure 6.11 shows an activity from the student workbook that paves the way to functions. Students recognize that two TamaroCards expressions, which are used to create the green and red lights for a traffic light, only differ for the color of the light.

Figure 6.12 shows the definition of a suitable function to create lights of arbitrary colors using TamaroCards. Special emphasis is placed on choosing a suitable name for the *parameter*. Once the function has been defined, it can be used as shown on the left of Figure 6.13. A TamaroCards expression that evaluates to a color, such as the `red` constant, can be used as an *argument* to produce a red light. On the right of Figure 6.13, students are then invited to use the same function to produce a light of a different color.

The final part of the curriculum first introduces lists, which are initially spelled out
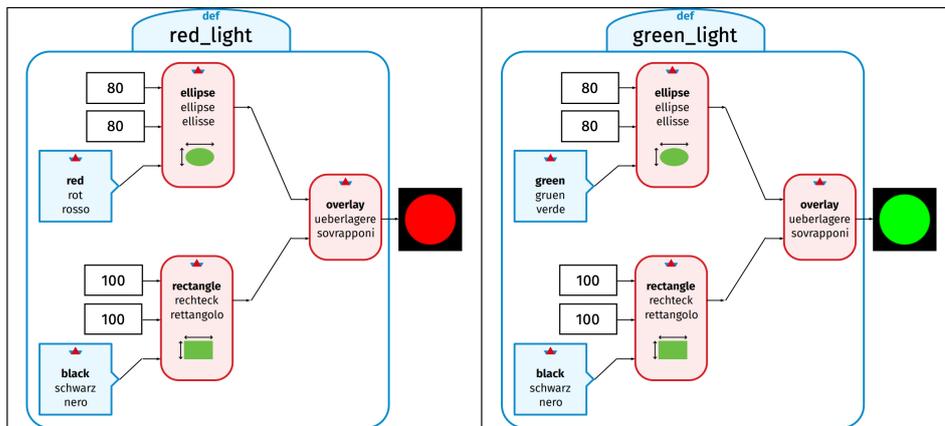
Figure 6.11. Excerpt from the workbook asking students to identify the differences between two similar but not identical constant definitions.

manually with list literals, then presents the `range` function to automatically produce lists of numbers, and finally uses the `map` function to transform those numbers into graphics. Throughout the course, the concepts are practiced with several examples. The course culminates with a project, allowing students to program their own graphic of choice.

This curriculum was piloted in an elective course in a middle school and covered the entire school year 2024/2025. It should not be intended as the reference PyTamaro curriculum, but more as one of the possible ways in which PyTamaro can be used with relatively young students. The teacher in charge of the classes anecdotally reports that her students liked the approach.

The curriculum relies heavily on TamaroCards. The middle school teacher, whose primary subject is mathematics, is very fond of the idea of having students work unplugged before transitioning to the computer. However, she laments that working completely unplugged for the first weeks of the course was a challenge for her students, who were eager to use computers. The experience of working with the cards felt inauthentic for some. The problem became less drastic over time, as students could mix unplugged and plugged activities. Even later on in the course, she encouraged students to resort to reasoning with the cards when they struggled with Python's syntax. Overall, she considers the benefits of using TamaroCards more important than the drawbacks, and plans to keep using the cards in the next editions of the course.
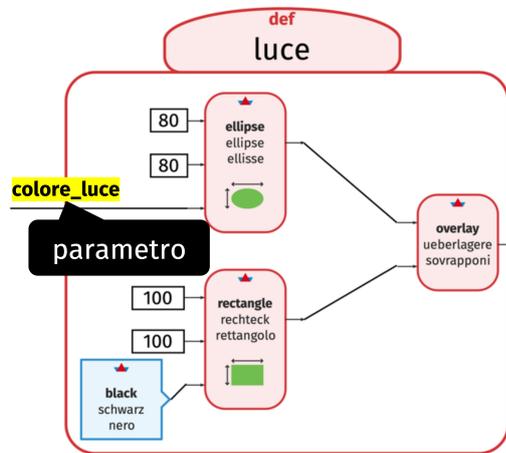
Figure 6.12. Excerpt from the workbook showing students how to define a function with a parameter named `light_color`.
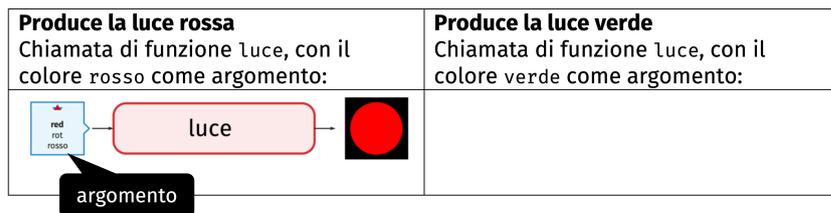
Figure 6.13. Excerpt from the workbook showing students how to use the function defined in Figure 6.12 and asking them to use it again with the color `green` as an argument.

Chapter 7

PyTamaro Web Offers Programming Activities with PyTamaro

Teachers need inspiration for programming activities that they could offer with PyTamaro. Moreover, schools are increasingly adopting a "Bring Your Own Device" policy: students come with an assortment of different devices and operating systems. Using a traditional IDE on a tablet is impractical or impossible. A properly designed web platform has the potential to address both problems.

> **RQ** How can a web platform be designed to offer PyTamaro activities and enable students working on them directly from their browser?

This chapter describes PyTamaro Web, a publicly available web-based Python programming environment, currently hosted at https://pytamaro.si.usi.ch. The platform offers programming activities based on PyTamaro, building on its strengths.

7.1 The platform was created to show example activities to teachers

The first version of the graphics library that later became known as PyTamaro was developed to support the teaching of programming to teachers in Tessin. We developed a number of exercises specifically to serve as assignments for courses in informatics didactics.

Some teachers in other cantons, who attended a different but related training program to qualify for teaching informatics, also completed a few of those exercises.

The initial motivation for developing the web platform was thus collecting and making publicly available this first set of exercises used with teachers, to serve as an

inspiration for activities that teachers could pilot with their students. A page on the platform features a gallery of activities as shown in Figure 7.1, where each activity is represented with the graphic students are asked to program. (The tags that appear under the title of each activity in Figure 7.1 have been added later. They characterize which programming concepts and which graphics concepts are involved in the activity.)

Figure 7.1. A gallery of activities in PyTamaro Web.

Over time, we gradually added more and more programming activities to the platform, with activities at different levels of difficulty and covering a variety of topics.

Additionally, the web platform offers a "Playground" page that can be used as a simple IDE for Python, ready to execute programs that use PyTamaro.

7.2 The computational model is based on notebooks, with key differences

An activity on the PyTamaro Web platform is a guided learning experience. Textual explanations are interleaved with "code cells": regions of the page that offer a code editor in which students can write and execute programs.

This structure is inspired by computational notebooks such as Jupyter [143]. The notebook interface was first popular among mathematicians, and is now used across many fields, including scientific computing, machine learning, and education. Since the beginning, notebooks offered the possibility of combining rich-text explanations with code and its result.

In fact, activities for PyTamaro Web are created and stored in the format of Jupyter notebooks [143]. Many environments, including Visual Studio Code [267], include capabilities to work with Jupyter notebook files.

As mentioned, Jupyter notebooks are commonly used in education: their versatility enables them to be used as an interactive lesson led by a teacher, as an independent lesson followed autonomously by a student, or even for assignments [135]. Johnson, however, warns of a number of pitfalls derived from his observations with students using Jupyter notebooks (ibid.). Two of them appear significant: the presence of hidden state due to out-of-order execution, and impediments to following fundamental principles of code organization, such as modularization. Here we discuss the former; Chapter 8 will deal with the latter aspect.

Code cells in a Jupyter notebook appear as a linear sequence. However, cells can be run and rerun in arbitrary order. This out-of-order execution is known to be a pain point even for moderately experienced users, as it can produce unexpected, puzzling results [45]. An analysis of public notebooks on GitHub revealed that nearly half of them include code cells executed in a different order than listed [225].

Static analysis techniques such as dataflow analysis could detect when code cells need to be rerun, but the problem is intractable in general when using a full-fledged programming language such as Python, where complex mutations are customary. Phipps-Costin et al. [208] suggest instead a simple solution for educational purposes: every time a code cell needs to be executed, all code cells from the beginning are executed sequentially. This may be an unacceptable cost in some contexts, but it is usually reasonable for educational purposes, since it leads to a simple computational model.

```python
from pytamaro import rectangle, red, show_graphic

red_square = rectangle(64, 64, red)
show_graphic(red_square)
```
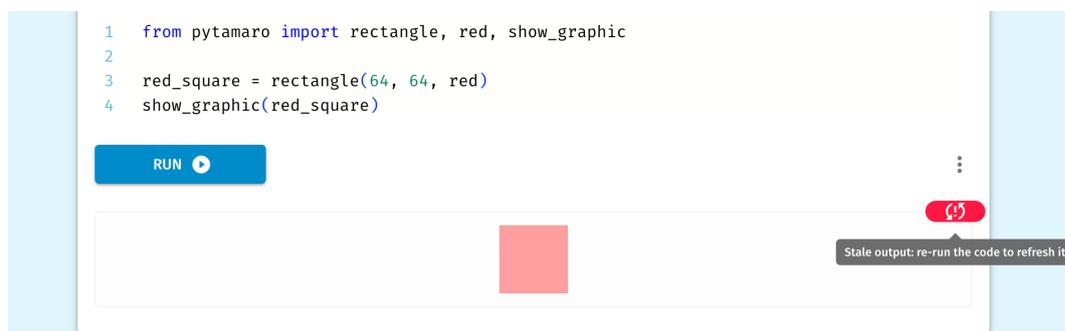
Figure 7.2. Indication of stale output on a code cell in PyTamaro Web.

When the code in a cell is modified after having executed it, the PyTamaro Web platform indicates that the output may no longer reflect the current code and is thus stale. Figure 7.2 shows the former output of a cell, a red rectangle, which becomes partially opaque after the code cell has been edited. A red badge warns that the output is stale. Re-executing the code cell refreshes the output by updating it. In Repartee [208], the same situation is indicated with a dashed border that surrounds the stale output.

If other code cells after the one that was edited have already been executed, all their outputs become stale as well, reflecting the linearity of the computational model.

7.3　Activities can leverage dedicated features to help learners

As mentioned above, an activity for the PyTamaro Web platform consists of a Jupyter notebook file. An additional JSON file contains metadata about the activity (e.g., the concept tags of Figure 7.1).

The Jupyter format interleaves code cells with explanations, which are written in Markdown. In addition to the usual formatting of Markdown documents, the platform also supports custom components using MDX, a superset of Markdown. These custom components are designed to enhance the learning experience.

Some components implement general pedagogical ideas. The *Task* component, for example, highlights the instructions for the learner in a colored box. The *DoNow* component, shown in Figure 7.3, instead requires students to stop and think, preventing them from quickly skimming through the page. It is an extension of the idea featured in the *A Data-Centric Introduction to Computing* textbook [92], where "Do Now" boxes are used throughout the text to encourage the reader to stop and reflect, avoiding a passive read. Unfortunately, incentives are almost nonexistent when the rest of the page is shown, often immediately revealing the answer to the question that was supposed to make the learner think. On PyTamaro Web, the *DoNow* component blurs the rest of the page as shown in Figure 7.3. At a minimum, the user must click an "I did it" button to unlock the rest of the page. While there is no assessment, blurring still introduces friction that can be pedagogically valuable.

Other MDX components are instead specific to PyTamaro. For example, the *API* component can be used to refer to a name from the PyTamaro library. As Figure 7.4 shows, the name is rendered in red and monospace text next to a small PyTamaro logo, helping learners realize that it comes from the library. Clicking the API name opens a popup with the documentation, minimizing the time necessary for novices to remember how a function works.

Embracing the support offered by the PyTamaro library for languages other than

Figure 7.3. A *DoNow* component blurring the remainder of the activity.

Figure 7.4. Two *API* components for PyTamaro's `rectangle` (hovered) and `red`.

English (Section 5.8.2), the web platform also supports activities written in multiple natural languages and localizes the key parts of the user interface. An Italian-speaking student can thus enjoy a programming environment localized in Italian, work on an activity written in Italian, and use the Italian version of the PyTamaro API. (Or, of course, any mix of the above, once they are ready to embrace multilingualism.)

7.4   A curriculum is a guided path through activities

A gallery to showcase activities, as shown in Figure 7.1, can be useful to spark the curiosity of a casual user or provide inspiration to a teacher struggling to decide on which graphics to program.

Effective learning, however, requires a structured path. Programming concepts, language features, and library functions all need to be introduced gradually.

To fulfill this need, the platform enables content authors to create *curricula*. A curriculum is a sequence of activities that students are expected to follow in order.

Some activities may be optional, and sometimes students can select one of multiple alternative activities.

A curriculum is divided into units. The platform does not prescribe the length of a curriculum: some curricula consist of only a handful of activities in a single unit, whereas other teacher-designed curricula cover an entire semester of programming activities with many units.

Figure 7.5 shows the first part of a curriculum dedicated to teaching the basics of programming to complete novices using PyTamaro. The curriculum starts with an introductory unit consisting of a single activity that gives a broad overview of what programming means. Completed activities are marked with a trophy: in Figure 7.5, that is the case for the "Ticino Coat of Arms" activity.

Then, learners execute their first programs to create simple graphics with PyTamaro. The second unit consists of two steps that must be completed in order. The first step requires students to work on the "Rectangles" activity, in which PyTamaro's `rectangle` function is called with different arguments.

For the second step, learners can fulfill the requirement by completing one of the alternatives: an activity to draw a rugby ball, or one to draw a watermelon. In those activities, students learn how to introduce names for graphics and use PyTamaro's `ellipse` function.

The author of a curriculum can also specify a set of *competencies*, learning objectives that students are expected to achieve when completing an activity. These competencies are represented as icons (collapsed at the bottom right of each activity in Figure 7.5). As learners progress through a curriculum, they acquire badges for those competencies as they get practiced. Figure 7.6 shows the competencies acquired after completing the first two activities of the introductory curriculum discussed above. This attempts to combine the appeal and the extrinsic motivation of "badges" with the reinforcement of programming skills.

The platform does not yet support any form of automatic grading for students' work. The emphasis has been on developing as many features as possible that directly help learning, as opposed to the convenience of assessing student work automatically. For this reason, an activity is considered complete simply when all code cells have been executed without errors.

Future developments of the platform could introduce the notion of a reference solution for the activities, which would enable more sophisticated forms of checking. (However, determining the equality of graphics in a way that makes sense pedagogically is all but a trivial task, as discussed by Barland et al. [18].)
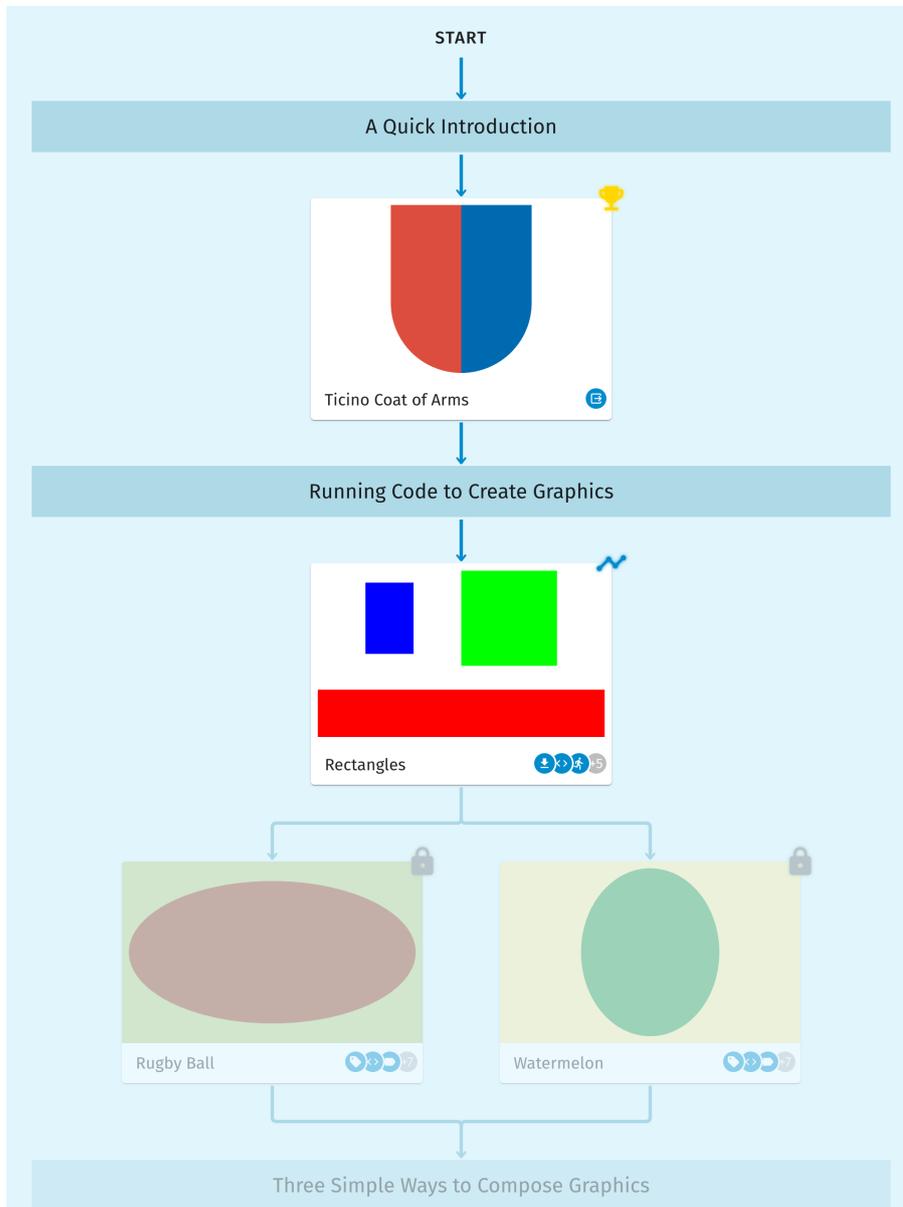
Figure 7.5. The beginning of a curriculum hosted on PyTamaro Web to introduce programming to complete novices using PyTamaro.
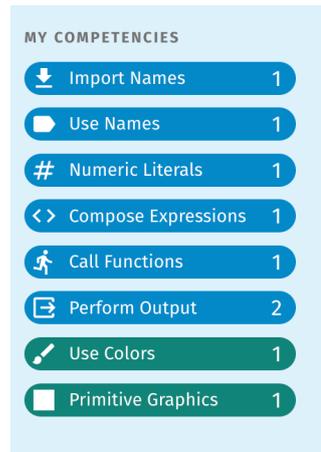
Figure 7.6. Badges for competencies on PyTamaro Web after completing two activities. In blue, competencies related to programming. In green, competencies related to graphics.

7.5 Privacy and pragmatic reasons dictate the platform architecture

The laws regulating data protection in the European Union and in Switzerland impose strict privacy requirements [268] that shaped the architecture we chose for the web platform.

Teachers cannot force their students to register an account on an arbitrary platform, unless specific agreements apply. For this reason, PyTamaro Web was designed to be accessible without the need for any user account. Nevertheless, it is still desirable for users to have a profile, so that they can resume working from where they left, including the code written in the programming activities and the progress in curricula. Modern web technologies enable achieving this combination: browsers include storage space that can be used by web applications, such as `localStorage`.

When a user visits PyTamaro Web for the first time, a profile is created and stored in their browser. The profile is always kept on the client and never stored on the server.

PyTamaro Web is a web application developed in TypeScript using Next.js, a React framework. Without getting lost in the specifics of the implementation, Figure 7.7 gives a bird's-eye view of the architecture.

The web application is generously hosted on university servers. This makes it necessary to minimize the resources needed to run the application in production. To keep the load on our server at a minimum, all the pages of the application are statically generated at build time. An application backend exists solely to accomplish two goals:
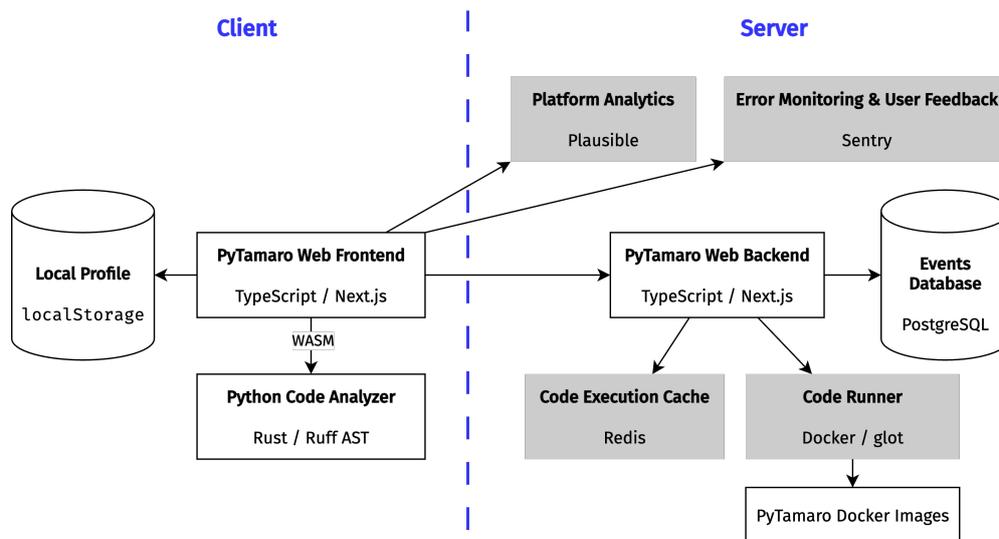
Figure 7.7. Architecture diagram of the PyTamaro Web platform. Gray boxes indicate third-party projects used as is.

- Execute the Python programs. Projects such as Pyodide offer a Python environment running directly on top of the browser JavaScript engine via WebAssembly. However, the PyTamaro library is built on top of the Python port of the C++ Skia graphics library, which cannot be installed in a Pyodide environment. We thus need to execute Python programs that use PyTamaro on our servers. To this end, we use glot[1], a third-party tool to execute untrusted code within ephemeral Docker containers. These containers execute Docker images that include Python, PyTamaro, and a selection of fonts to render text.

- Record user events. Before executing their first program on the platform, users are asked if they consent to voluntarily participate in a data collection project for research purposes. Users are represented with a random, unique identifier. When users give consent, the content of the code cells is stored every time the code is executed, together with some meta information (such as the activity in which the code has been executed, or the initial content of the code cells as created by the activity author). Other kinds of secondary interactions with the platform are also recorded as events.

The structure of the database of events is inspired by Blackbox [35], a database of code submitted by users of BlueJ, an educational IDE for Java [148].

---

[1]https://github.com/glotcode/glot

In the longer term, we are evaluating alternatives to shift the execution of Python programs—an expensive and risky operation—from our server to the user's browser. In the meantime, we introduced a cache using Redis to avoid executing the same code multiple times, alleviating the load on the server.

7.6   Teachers contribute content using version control

On most web platforms, authors create and modify content using a special area of the web platform with access granted only to privileged users. These system are commonly called "Content Management System".

The PyTamaro Web platform statically generates its pages at build time and does not include a system to manage the content via web pages. The reason is twofold. First, such a system requires to handle accounts, a feature we deliberately avoided as argued in the previous section. Second, creating a custom Content Management System has significant development costs we pragmatically tried to avoid.

Instead, teachers contribute their content using regular version control with `git`. Access right management is offloaded to GitHub. The remainder of this section documents how this was technically feasible: uninterested readers can safely jump to Section 7.7.

Figure 7.8 illustrates the steps that take place when a teacher wants to update their content on the PyTamaro Web platform. Each teacher has an account on GitHub. This account grants them *read-only* access (blue arrow of Figure 7.8) to the main `pytamaro-web` repository, and *write* access (green arrow) to a repository dedicated to their content (activities and curricula). The teachers' repositories are integrated in the platform repository as git submodules, which are represented as hexagons in Figure 7.8.

As a first step, teachers clone both the platform repository and their personal repository on their computer. They can work on their content and run the platform locally to preview the final result. At any time, they can commit their changes and push them to GitHub.

Whenever they deem the changes ready to be published on the public platform, they also commit and push their changes to a dedicated, special branch named `live` (Step 2 of Figure 7.8).

GitHub recognizes that a push event occurred on the special branch, and triggers a webhook notification to a server under our control (Step 3). Our server attempts to build the platform with the updated content from the teacher.

When the build succeeds without errors, the changes can be integrated without problems in the main repository. Our server has write access (green arrow, Step 4 of
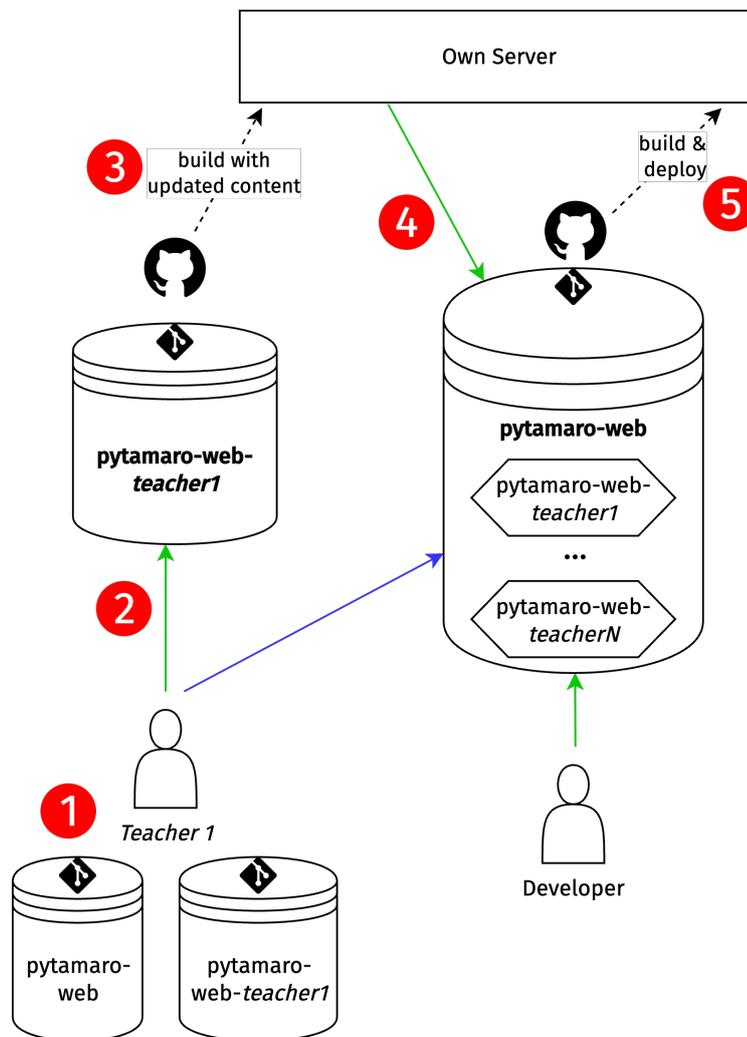
Figure 7.8. The process used to integrate new contributions from a teacher in PyTamaro Web.

Figure 7.8) to the platform repository and can update the commit referenced by the submodule to the latest commit created by the teacher.

In turn, this last push triggers a webhook notification configured on the platform repository: GitHub signals to our server a request to build and deploy the entire platform (Step 5). When this process ends, the changes made by the teacher are visible to everybody on the publicly deployed platform.

Effectively, a teacher's workflow consists only of Step 1 and 2: the remaining steps are handled transparently.

Step 5 of this process is also used by the actual developers of the platform, who have write access to the platform repository and can thus directly commit and push to its special branch to start a new deployment.

7.7 We used the platform for a self-guided Hour of Code curriculum

The "Hour of Code" program aims to bring at least an hour of "programming experience" to millions of students worldwide [278]. The program's website[2] links to short, self-guided tutorials that students can complete in approximately one hour.

The short duration of the activity makes it challenging to teach actual programming content. Despite this, a team of Bachelor's students, supervised by members of the research group, accepted the challenge and developed a short curriculum hosted on the PyTamaro Web platform.

The curriculum is entitled "Program Your Own Castle"[3]. A brief overview showcases the features of the web platform and serves as an example of using PyTamaro to teach programming in a highly constrained environment (i.e., short duration and lack of in-person guidance).

The curriculum mixes history and programming, involving students in a quest to build a fortified castle to defend from invaders. The castle has been modeled after "Fortezza Bellinzona"[4], a set of three medieval fortified castles located in Tessin and part of the UNESCO World Heritage.

The curriculum consists of five units with short activities. In the first unit, students learn to execute code and use PyTamaro's `show_graphic` to output parts of the castle: walls, entrance doors, and battlements. The second unit teaches them how to use names to refer to a graphic. The third unit focuses on passing arguments to call functions. For example, a `drawbridge` function has a parameter that determines whether

---

[2]https://hourofcode.com
[3]https://pytamaro.si.usi.ch/hoc/castle
[4]https://fortezzabellinzona.ch/

the drawbridge is open or closed, and a `battlement` function requires to specify the color and the shape of the merlons (Figure 7.9). The fourth unit is all about lists: students manually create lists to build a tower and a curtain wall by juxtaposing graphics with `above` and `beside`. The final unit consists of just one activity in which students modify a castle like in Figure 7.10 to make it their own.

```
1   from pytamaro import show_graphic
2   from castle_components import battlement
3   from colors import red, yellow, green, cyan, blue, purple, pink, grey
4   from merlons import rectangular_merlon, rounded_merlon, split_merlon
5
6
7   # 🎯 Create a pink battlement with split merlons
8   my_battlement = battlement(pink, split_merlon)
9
10  show_graphic(my_battlement)
```


Figure 7.9. Solution: the `battlement` function is called to produce a pink battlement with split merlons.
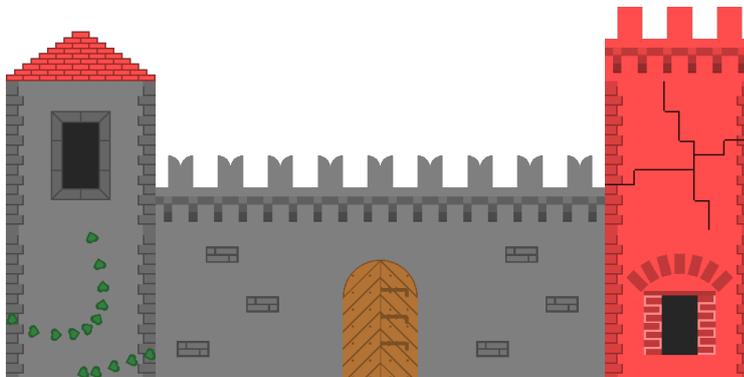
Figure 7.10. Initial version of the castle in the final activity of the "Hour of Code" curriculum.

Students know how to call functions and can modify the arguments to change the color and the type of certain graphics. They also know how to use lists, and can thus add or remove elements both horizontally and vertically.

Given its brief duration, this curriculum could not start from primitive shapes: it would have been impossible for students to build reasonably-looking graphics in such a short amount of time. Nonetheless, it focuses on the *visual decomposition* of a castle into its parts, a seed to teach the skill of problem decomposition.

No part of the castle was created in a raster graphics editor: the authors of the activities programmed all the various graphics with PyTamaro. These graphics effectively constitute an additional library. To support this use case, the PyTamaro Web platform allows authors of activities to bundle an additional set of Python files, which are not shown to the users but included in the execution.

7.8 The platform hosts several activities and curricula

The "Hour of Code" curriculum is a special and atypical use for the PyTamaro Web platform.

Originally, all the activities available on the platform were created by the research group to illustrate possible uses of PyTamaro. The target audience consisted mainly of high school teachers, and possibly of some of their students. Today, the platform hosts many other activities and curricula created by Swiss high school teachers. Some of these activities are an adaptation of the original ones (e.g., translated into a different natural language), while others have been designed anew (e.g., a teacher created a curriculum with activities on optical illusions). Finally, the platform also hosts activities in service of the middle school course described in Section 6.7.

When users consent to anonymously share their data, we collect usage statistics that indicate how much the platform has actually been used. The data collection started on September 2023 and is currently ongoing. Below, numbers refer to data up until July 2025.

The platform currently hosts 215 activities created by the research team, 12 activities for the Hour of Code curriculum, 31 activities for the middle school course, and 275 activities authored by 10 different high school teachers. Some of these activities are part of one or more curricula. Currently, there are 19 curricula created by the research team and 31 curricula developed by high school teachers.

Execution data demonstrate that the platform is in active use: in less than two years, it has executed more than 782 000 Python programs, written by over 22 000 different users.

The collection of this large amount of data written by learners is inspired by Blackbox [35] and will allow us to perform similar analyses on Python code, such as studying errors or language features [32]. This data can also provide a different perspective on how PyTamaro influences the programs novices write *in practice*.

Chapter 8

The Toolbox of Functions Promotes Abstraction

Due to the limited classroom time available, programming lessons often focus on writing small one-off pieces of code. The focus lies on getting students to solve specific problems, and students then throw away their solutions once they are done. No incentive is placed on defining proper abstractions that can be later reused.

> **RQ** How can an approach be designed to support students in reusing their code, practicing abstraction by saving and using the functions they define?

This chapter describes the Toolbox of Functions, an approach to encourage code reuse as a form of abstraction in a motivating and simple way. We implemented the approach as an extension of PyTamaro Web (Chapter 7).

8.1 We should move from code clones to code reuse

When they grow beyond toy examples, programs invariably end up consisting of multiple parts that accomplish similar behaviors. How are these related behaviors implemented in program code? One straightforward option is writing the same or similar code as many times as needed. These duplicate parts of program code are known as *code clones*. Code clones can be classified into different types [150], ranging from being exact duplications of identical chunks of code, to being duplications except for some identifiers, to having some additional or missing parts of code.

Producing code clones is disadvantageous because it leads to maintainability issues [138]. A bug discovered in one clone, or a change needed to accommodate a new functionality, needs to be identified and manually applied individually to each clone.

8.1.1 Code clones are widespread

Despite the disadvantages, programmers frequently duplicate code. The adage "Copy & Paste" embodies this idea. Such an operation is sometimes considered the fastest way to achieve a certain goal.

Because abstraction is considered a fundamental but difficult concept to master in computer science [227], it is perhaps unsurprising that novice programmers avoid abstractions and frequently resort to copy-paste [266, 142].

Studies show that even code written by experienced programmers commonly contains code clones. For example, code clones are common in code examples published on Stack Overflow [17] and in code cells contained in Jupyter notebooks [146]. A large-scale study on GitHub repositories across multiple programming languages found high rates of code duplication in files both within a repository and across repositories [166].

The recent diffusion of powerful language models for code (Section 2.8) and their integration in IDEs reduced the time needed to duplicate a piece of code even further. This applies also to non-exact clones, such as those with replaced identifiers. The language model can quickly recognize the desired pattern even just after typing the first replacement, and can instantly complete the rest of the code.

8.1.2 Code clones can be avoided with code reuse

Software engineering as a discipline realized a long time ago the possibility of writing programs in a modular sense [277, 198]. These "modules" have been variously called subroutines, procedures, or functions; fundamentally, they are abstractions. Modern programming languages support several forms of abstractions, including some more elaborate than the ones mentioned above, such as classes.

Given that this dissertation focuses on introductory programming, we will concentrate on functions, as they are a simple but powerful form of abstraction that is suitable for novices in a school context. Functions can offer "configuration options" [277] through parameters, to accommodate the differences in behavior that are required in different parts of the program.

When the code to achieve a certain functionality is abstracted as a function, it can be *reused* by calling that function in multiple places within the program, every time one needs that functionality.

The next logical step is to reuse code across programs. Indeed, programmers are familiar with the idea of using functions from a *library*. Programming languages come by default with libraries containing several functions deemed useful in many contexts (a library for operations with dates and times, for example).

8.1.3 Environments do not always favor code reuse

Code reuse is however not on the path of least resistance: features of existing IDEs can discourage reuse and instead lead to code clones.

8.1.3.1 Environments stimulate the use of code snippets

Programmers, both novice and experienced ones, can find themselves not having a clear idea of how to solve a problem. In most cases, that specific problem—or a very related one—has already been solved by someone else, and thus the programmer can try to obtain a fragment of code, either to include as is or to adapt with minor modifications [16].

These code snippets can be obtained in different ways from several different sources, such as code-repository sharing platforms (e.g., GitHub), Q&A platforms (e.g., StackOverflow), code-snippets sharing platforms (e.g., GitHub Gist[1]) and language models, that generate code fragments based on the code they have been trained on (which was in turn sourced from the aforementioned platforms).

Moreover, some environments for notebooks, such as Google Colab[2], allow users to directly inject pieces of code from a collection of "snippets": fragments of code to solve recurring programming problems. The environment comes with a selection of pre-written snippets, but also allows users to save and retrieve their own.

8.1.3.2 Environments can offer more advanced templates for code

Online platforms are not the only source of fragments of code: IDEs also provide features to facilitate the repeated insertion of specific *code templates*. Templates are more powerful than snippets because they may contain *holes* to be filled in by the programmer when they want to include them. When code templates become fully concrete, they effectively turn into snippets.

Some IDEs use code templates to assist in writing "boilerplate code", i.e., repetitive patterns of code. For example, the `for` keyword can be automatically expanded to the full skeleton of the for loop statement in a language like C, and the keyword `class` can be turned into a complete class declaration in Java.

More advanced environments also enable developers to define their own custom templates. These features go under different names, such as IntelliJ's Live Templates[3]

---

[1] https://gist.github.com
[2] https://colab.research.google.com
[3] https://www.jetbrains.com/help/idea/creating-and-editing-live-templates.html

and VSCode's Snippets[4].

Code templates can go even further: some IDEs employ static analysis techniques to generate code leveraging information extracted from existing code. For instance, IDEs such as Eclipse or IntelliJ IDEA include a widely used feature that generates an implementation of the `equals` and `hashCode` methods for a class by inspecting the fields declared by the programmer.

8.1.3.3   Scratch offers to remix projects by duplication

Code clones are not exclusive to textual programming languages. A study on Scratch, a popular choice to teach programming in schools using blocks, showed that Scratch projects contain code clones quite pervasively, and that functions as abstractions are rarely used [6].

The Scratch platform promotes taking an entire published project and "remixing"[5] it. This is yet another example of an environment that favors code duplication over reuse: instead of being able to import a well-defined abstraction, such as a sequence of blocks packaged in a function, learners are nudged to "fork" [150] an entire project, ending up with code clones.

8.1.3.4   Multi-file projects can require a complex setup

School teachers use different kinds of IDEs: some choose to work with novice-oriented IDEs such as Thonny [12] or BlueJ [148], some work with full-fledged ones such as Visual Studio Code (VSCode) [267], and others adopt web-based programming platforms, which range from barebone ones like Ideone[6] to sophisticated ones like GitHub's Codespaces[7].

No matter the setup, it is non-trivial to reuse code beyond one single file. IDEs may structure a project into several files, potentially divided into several folders. This complexity is one of the reasons behind the need for a build system: a tool that helps programmers develop programs consisting of multiple files. Reusing code across files requires configuring the environment to handle this setup. This may require configuring the IDE, adopting a precise structure for files and folders, and following the non-obvious importing rules that deal with relative or absolute paths.

The problem is exacerbated when one wants to reuse code across different projects. This requires refactoring code into a separate *library*, which can then be imported

---

[4]https://code.visualstudio.com/docs/editor/userdefinedsnippets
[5]https://en.scratch-wiki.info/wiki/Remix
[6]https://ideone.com
[7]https://github.com/features/codespaces

into multiple projects. Importing libraries outside those that come standard with a programming language is often a source of pain, even for experienced developers.

Understandably, teachers want to eschew all these problems and have their students spend the limited time available on actual programming, as opposed to configuring an environment. As a result, students may be instructed to limit themselves to using only a single file, be that a real file on their device or a virtual one in a web-based text editor.

Unfortunately, this deprives learners of the opportunity to get acquainted with code reuse in the simple context of programs with a modest size. Having students practice code reuse early on teaches them the right practices, which they can then apply to larger programming projects.

8.1.4 Assignments do not always favor code reuse

Teachers of programming courses commonly design assignments for their students to practice programming skills. At the beginning of an introductory course, a programming assignment typically consists of small, independent exercises. As the course progresses, an assignment can become a small project. Still, most assignments do not reuse solutions from earlier work.

Students can find themselves re-implementing the same functionality multiple times, effectively writing code clones across assignments. Instead, learners should be guided to decompose the solution of each assignment into functions, identify the ones that are general and potentially useful for other problems, and use these functions again in the subsequent assignments. This way, students could learn and practice code reuse in a controlled context.

8.2 The Toolbox of Functions is an approach to promote code reuse

A wise handyperson knows the advantage of having the right tool ready to deal with a certain situation, and always carries a *toolbox* containing various useful items. Based on this analogy, we propose that beginner programmers should keep a *Toolbox of Functions* at their disposal: an always-available library of functions that are potentially useful to solve future tasks.

When students recognize the potential general applicability of a function they defined (or the instructions in an assignment recommend doing so), they should add that function to their personal Toolbox of Functions. Adding a function to the Toolbox requires

some polishing to become a reusable abstraction, much like programmers do for larger pieces of software. The learner should identify all the dependencies of the function, such as other functions called or global constants used, and keep them alongside the function they intend to save. Then, students should consider improving the names of the function and its parameters to ensure they are descriptive, annotate the signature with types (for statically typed languages), and add documentation that reminds their future selves of what the function is supposed to do. Optionally, they could also add some tests to ensure that the function they extracted works as intended.

As the Toolbox of Functions grows, teachers can ask students to implement more elaborate programs, counting on the fact that they have already implemented certain functions which are ready to be used. Students can quickly solve parts of a larger assignment by importing functions from their Toolbox, call them with the right arguments, and then focus on the rest of the program.

This practice empowers students with a sense of satisfaction and purpose that derives from reusing—instead of throwing away—the code they had to implement earlier for a different problem. And this is achieved without any code clone or copy-paste activity.

8.3   PyTamaro Web implements the Toolbox of Functions

The Toolbox approach we described is independent of both the programming language and the environment. In this section, we describe one specific instantiation of the approach, as it is implemented in PyTamaro Web (Chapter 7).

As a small-scale running example for demonstration purposes, we will consider a tiny fictional curriculum with two activities, which ask students to draw an "eye" and a "no entry" traffic sign.

8.3.1   A student starts by defining functions normally

When a student solves a programming activity, code clones may appear, possibly due to copy-paste. This often occurs when working with a minimal educational library such as PyTamaro, which does not include functions like `square` or `circle` to encourage students to define their own starting from `rectangle` or `ellipse` (Chapter 5). Activities may deliberately have students *feel the pain* of writing repeated code to motivate the introduction of function definition as a concept.

Code clones can occur within a single code cell or can be scattered across multiple cells. Students can be instructed to observe the similarities between multiple clones and identify the few differences. They can then define a function with a parameter for

each of these differences, and with the body containing the clone, where the differences are replaced by the parameters. Finally, each clone can be replaced with a simple call to the newly defined function.

Defining functions enables code reuse within a single activity. The result of this abstraction process is shown in Figure 8.1a for the "eye" activity, in which students can extract a function to create a circle of a given radius and use it twice to draw the pupil and the iris of the eye. Teachers can scaffold this process by providing interleaved explanations and setting up code cells accordingly.

8.3.2 Functions can be added to the Toolbox

Once the student has defined their own function, they may want to save it for later use by adding it to the Toolbox of Functions.

When the code written by a student contains the definition of at least one function, a button with the icon of a handyperson's tools appears automatically (Figure 8.1a). This automatism is made possible by statically analyzing the Python code directly in the browser (cf. Section 7.5).

Clicking on that button opens a popup with instructions on how to save a function to the Toolbox (Figure 8.1b). Following the approach described in Section 8.2, learners should:

1. Identify all the dependencies of the function they want to save. These may include `import` statements or global constants to define the names that are used in the function body and other functions that are called by the function to be saved. Other pieces of code are unnecessary and should be removed, such as possible calls to the function being saved, as well as any code used for debugging (e.g., `print`).

2. Write a description to remember what the function does, as a minimal form of documentation. When a function contains a docstring written following the PEP 257 convention [205], the Toolbox of Functions uses that string as documentation by default.

3. Implement and execute a short program that calls the function. This serves a dual purpose. First, it serves as a lightweight, non-automatic form of testing, to detect, for example, leftover side effects in the function body, such as debugging statements. Second, it serves as a form of documentation on how to call that function in the future (Figure 8.1c).

Students can then add the function to their Toolbox, without worrying about managing Python files to set up and maintain a library.

```
1   from pytamaro import black, cyan, ellipse, overlay, show_graphic, Graphic, Color
2
3   def circle(radius: float, color: Color) -> Graphic:
4       """Create a circle of the given radius and color."""
5       diameter = radius * 2
6       return ellipse(diameter, diameter, color)
7
8   show_graphic(overlay(circle(20, black), circle(40, cyan)))
```

DONE ✓   ⚒ ⋮

(a) Students define the `circle` function to solve the "eye" activity. They can save it by clicking on the Toolbox button (red arrow).

Implementation

> ⓘ Clean up your code before saving a function to the toolbox. Here are some tips:
>   • Remove any code that is not needed by the function you want to save
>   • Remove calls that produce output (e.g., calls to `print` or `show_graphic`)
>   • Remove unnecessary imports (move them all to the top, combine them and get rid of unneeded ones)

```
1   from pytamaro import black, cyan, ellipse, overlay, show_graphic, Graphic, Color
2
3   def circle(radius: float, color: Color) -> Graphic:
4       """Create a circle of the given radius and color."""
5       diameter = radius * 2
6       return ellipse(diameter, diameter, color)
```

(b) A popup opens. In the first part of the popup, instructions guide students in cleaning up their code to make the function reusable.

Function to save   ┌─ Function ─────────┐   Create a circle of the given radius and color.
                       │ circle   ▾ │
                       └────────────────────┘   Document what this function does

Example that uses this function

```
1   from toolbox import circle
2   from pytamaro import show_graphic, green
3
4   show_graphic(circle(10, green))
```

DONE ✓

SAVE TO TOOLBOX

(c) In the second part of the popup, students have to write minimal documentation, implement and run an example program before being allowed to add the function to the Toolbox.

Figure 8.1. Adding `circle` to the Toolbox of Functions in the "eye" activity.
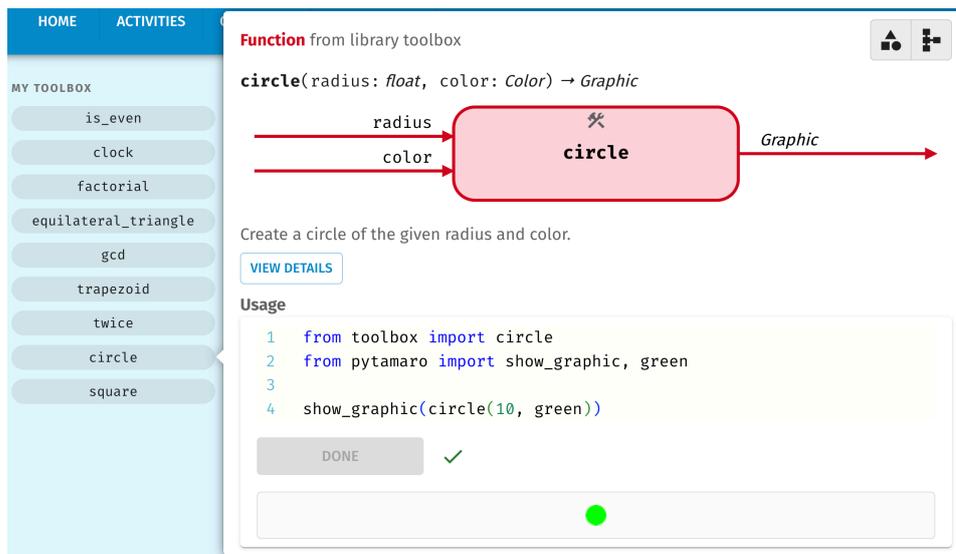
(a) Students explore functions in the Toolbox (left sidebar, always visible while solving activities). Their documentation opens in a popup.

```
1    from toolbox import circle
2    from pytamaro import overlay, rectangle, red, white, show_graphic
3
4    show_graphic(overlay(rectangle(40, 10, white), circle(30, red)))
```


(b) Students can import the `circle` function from the Toolbox just like from any other library and reuse their code in a different activity.

Figure 8.2. Using `circle` from the Toolbox of Functions in the "no entry" activity.

8.3.3   Students can then use functions from their Toolbox

The tools in a physical toolbox are always near a handyperson. Similarly, the functions in the virtual toolbox are near the programmer. On each activity page, the Toolbox of Functions is presented in a sidebar (left side of Figure 8.2a). This quick overview of the Toolbox of Functions realizes Victor's idea of "dumping the parts bucket onto the floor" to "encourage the programmer to explore the available functions" [265]. Students can see that they have a `circle` function that may be useful in an activity where they need to draw a "no entry" traffic sign (Figure 8.2b).

Moreover, by clicking on the corresponding item in the sidebar, students can quickly retrieve the documentation for each of their functions. The documentation for all the

libraries, including the Toolbox of Functions, is shown using a novice-friendly documentation system, which will be introduced in Chapter 9 (right side of Figure 8.2a). The documentation describes the various properties of the function. The function signature is detected automatically and is optionally enriched with the parameter and return types, if the original function was augmented with type annotations. The description and the usage example, instead, come directly from the learner. The example code can be executed in-place, as a reminder of the function's behavior (Figure 8.2a).

Once the student has identified a function to use, they can reap the benefits of code reuse with almost no effort. All that is needed is to `import` the `circle` function from the `toolbox` library and call it with the proper arguments.

8.3.4   The Toolbox grows over time

Functions in the Toolbox behave exactly like all the other functions. They can be used as part of the definitions of new functions, and students can add these new functions to their Toolbox of Functions as well. On a small scale, this showcases how to build more powerful abstractions on top of simpler ones.

Behind the scenes, each function is stored in a separate file, to avoid possible conflicts (e.g., two functions may have been added to the Toolbox of Functions from two activities that defined two different constants with the same name). All the Toolbox functions required for a certain activity are exported from the `toolbox` Python module, which in turn imports the necessary functions. These imports follow the topological sort of the dependencies, to avoid circular imports.

All this complexity is hidden from the student, who can just focus on defining and using functions, fulfilling the goal of practicing abstraction and code reuse without any waste of time.

8.3.5   Students gradually learn to manage their Toolbox

The platform also enables learners to manage their Toolbox of Functions. Learners can search their toolbox, remove functions from the toolbox, and modify existing functions. This is an essential feature to support students who need to fix a bug or make an improvement to their code. At the same time, it offers a sneak peek into the intricacies of library and Application Programming Interface (API) evolution—another important aspect of programming and software engineering [158]—in a controlled setting: whenever the signature (public interface) of a function in the Toolbox of Functions changes, any program using it needs to be updated accordingly.

8.4   We collected initial data on students using the Toolbox in PyTamaro Web

We implemented the Toolbox of Functions in the PyTamaro Web platform during 2023. Since then, we have collaborated with several teachers to explain the idea behind it and motivate its use. We first created example activities to showcase the Toolbox of Functions. Then we collaborated with high school teachers and explained its benefits. Over time, teachers started to integrate the Toolbox approach into their own curricula and activities they use regularly in class.

Focusing only on the last year, usage statistics show that students added more than 1 420 functions to their Toolbox. A total of 32 522 code executions made by 880 different users imported at least one function from their Toolbox into their code. Like for the statistics reported in Section 7.8, the actual user count is likely underestimated, as the platform collects data only from users who give their explicit consent. However, users are simply identified by a randomly generated identifier stored in the browser's local storage, which might be reset at any time, leading to double counts.

Currently, the platform hosts more than 50 different activities that use the Toolbox of Functions, created by 6 different instructors. According to a teacher who actively uses the Toolbox of Functions in their lessons, students are keen on collecting functions and watching their Toolbox of Functions grow. This anecdote suggests that students can perceive curating a Toolbox of Functions as a benefit rather than a chore.

8.5   The idea of the Toolbox can be expanded and empirically evaluated

The Toolbox approach focuses on functions because they are both a fundamental building block to define abstractions and widely taught in schools. However, other kinds of abstractions exist, such as class definitions. The approach of the Toolbox of Functions could be extended to support those as well.

The current model of the Toolbox of Functions has a flat structure. This is a deliberate choice to keep adding to and importing from the Toolbox of Functions as simple as possible, but it can become inadequate when the number of functions grows too large. Some form of namespacing (e.g., Python submodules) could better organize the Toolbox of Functions, but it would require dealing—albeit in a more controlled way—with files and folders, which can be a pain point for students (cf. Section 8.1.3.4).

Changing the signature of a function stored in the Toolbox of Functions may break all the code that depends on it. Students should be warned of the risks, making them

aware of the consequences of changing a public API. An expansion of the Toolbox of Functions could be used to teach more advanced software engineering principles related to code reuse, such as API versioning and the concept of third–party dependencies. This could also open the possibility of introducing a code-sharing mechanism, allowing students to reuse the code already written by their peers.

Finally, a future study could evaluate the effectiveness of the Toolbox of Functions as an approach for teaching code reuse, for example by comparing the ability of identifying and eliminating code clones between students who used the Toolbox of Functions and those who did not.

Chapter 9

Judicious Is a Gradual Documentation System for Novices

Programming is, at its core, using and defining abstractions. Application Programming Interfaces (APIs) are the fundamental mechanism for programs to offer and use abstractions. It should not be surprising then that APIs are ubiquitous. Myers and Stylos [188] observed that "nearly every line of code most programmers write will use API calls" when one considers both public and private APIs.

The pervasiveness of API use in large software engineering projects is undisputed, but the same is true also for small programs common in education. Even without considering third-party APIs, which are sometimes avoided by educators on the grounds of their complexity, didactic programs invariably use APIs offered by the programming language, which are sometimes collectively called the "standard library" of the programming language. It suffices to think about example programs common in introductory programming: determining the length of the hypotenuse of a triangle using Pythagoras' theorem requires a function to compute the square root, and writing a tiny game that asks the user to guess a number requires a function to generate a pseudo-random number. Moreover, these programs need to perform input/output operations, which are also achieved using APIs.

Given the number and the size of software libraries, memorizing all the details of APIs is an impossible task. Worse, students wasting resources on memorizing the inessential is a distraction from the goal of learning how to program.

Educators remind students that all libraries, especially the ones that come standard with the programming language, are extensively documented and this reference documentation can be checked at any time needed. However, the first experience of novice programmers with documentation systems is often frustrating, as these systems are normally targeted at professional developers. They have the significant advantage

115

of being exhaustive, at the expense of including several concepts (such as language features or technical jargon) that novices have not yet learned. They contain a lot of information, which becomes hard to interpret. Figure 9.1 exemplifies these issues, showing the official documentation of Python's `print` function: unless novices use a REPL, they cannot avoid using the function to visualize the output. Unfortunately, its documentation is unapproachable for students at that initial stage, despite their program potentially being as simple as `print("Hello, world!")`. To wit: professional

---

print(*objects*, *sep*=' ', *end*='\n', *file=None*, *flush=False*)

    Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file*, and *flush*, if present, must be given as keyword arguments.

    All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values. If no *objects* are given, `print()` will just write *end*.

    The *file* argument must be an object with a `write(string)` method; if it is not present or `None`, `sys.stdout` will be used. Since printed arguments are converted to text strings, `print()` cannot be used with binary mode file objects. For these, use `file.write(...)` instead.

    Output buffering is usually determined by *file*. However, if *flush* is true, the stream is forcibly flushed.

    Changed in version 3.3: Added the *flush* keyword argument.

Figure 9.1. Official documentation of Python's `print`.

---

documentation systems can be intimidating and overwhelming for the beginner programmer.

Given this situation, we ask:

RQ How can a documentation system be designed to accompany novices gradually as they learn programming?

This chapter first describes existing documentation systems that are used in education. It then draws from prior research in the learning sciences and programming languages to motivate and illustrate in Section 9.2 the design of Judicious, a novel documentation system explicitly designed to accompany novices as they learn to program. Judicious has been specifically built to document the PyTamaro API, but it is effectively a generic documentation system. We compare it to existing systems to discuss tradeoffs and limitations.

9.1 We briefly review documentation and introductory programming

The landscape of documentation systems is rather rich and varied. Some systems come standard with the tooling of a programming language (e.g., Rust's `rustdoc`), while others need to be installed separately. A system can support a single programming language (e.g., `scaladoc` for Scala) or multiple programming languages (e.g., `doxygen` supports C/C++ but also PHP, Python, and many others). Typically, documentation systems allow extracting information from source code in different markup formats (e.g., reStructuredText) and can produce documentation in several formats (e.g., HTML).

9.1.1 There are a number of different documentation systems

We do not conduct a systematic review of all documentation system for programming. Instead, we rather focus on selected systems that have known use in introductory programming, describing them briefly.

9.1.1.1 Javadoc for Java

Java is a popular language for teaching programming. The de-facto standard tool for documenting Java programs is `javadoc` [151], originally developed by Sun. The name Javadoc can also refer to the format used to write Java comments (enclosed within `/**` and `*/`) so that they can be recognized by the tool.

Beginner programmers are likely to encounter web pages generated with `javadoc` when browsing the functionality of one of the many "collection classes" [131].

The pedagogical development environment BlueJ [148] offers special support for Javadoc. Students can quickly use a drop-down menu to generate the HTML version of the documentation for the currently open file. The environment nudges beginners towards writing Javadoc comments by including them in the template for new classes. However, a recent study [34] analyzed programs written with BlueJ and did not find documentation comments half the time, despite them being initially in the template. This suggests that students and educators do not find enough value in writing them, given what they get in return.

9.1.1.2 Scribble for Racket

Scribble is a collection of tools to produce documents and can also serve as a documentation system. Scribble is used to document Racket APIs, including those offered by

the beginner-friendly libraries included in the *How to Design Programs* textbook [82].

The educational programming language Pyret [256] also uses Scribble to document its libraries. The accompanying *A Data-Centric Introduction to Computing* textbook [92] encourages students to consult the documentation to discover the functions available in the libraries.

9.1.1.3   Sphinx for Python

Sphinx is a documentation system originally developed for Python, which at the time of writing is arguably the most common language used in introductory programming. Sphinx has then been extended to support other programming languages. It leverages `docutils` to support multiple markup formats for writing documentation comments.

Students end up visiting HTML pages generated by Sphinx when looking at the documentation of the Python standard library [257] or the API reference of the myriad of third-party libraries available in Python's ecosystem (e.g., `pandas` which is common in "data science" courses [37]).

9.1.1.4   Pylance for Python

Visual Studio Code [267] is currently a popular environment for programming developed by Microsoft. It targets professional developers, but is widely used in education as well [252]. The environment supports multiple programming languages, as language-specific services ("Intellisense") are provided by extensions that communicate with the editor via Language Server Protocol. Microsoft directly provides a Python extension that includes a debugger and Pylance, a separate extension offering type-checking, code completion, and documentation.

Programmers access the documentation by hovering over a name. When Pylance can resolve what that name refers to (e.g., a function), it displays a form of documentation in a modal window as shown in Figure 9.2 for the `sqrt` function imported from the `math` module.
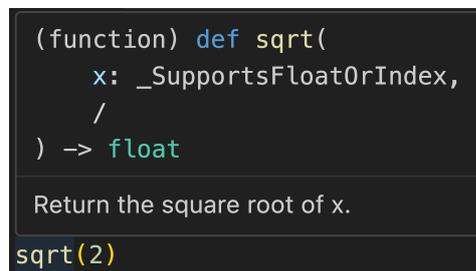
```
(function) def sqrt(
    x: _SupportsFloatOrIndex,
    /
) -> float
```
Return the square root of x.

`sqrt(2)`

Figure 9.2. Pylance's documentation of the `sqrt` function.

9.1.2 API documentation for beginners is sometimes ad hoc

Systems that have been developed primarily with the needs of professional programmers in mind can also be adopted in education. On the one hand, educators may encourage the use of these systems because of their *authenticity*. Students feel that they are familiarizing themselves with tools also used in industry; this can contribute to a positive attitude towards learning. On the other hand, full-fledged systems can quickly overwhelm novices with information that is hard for them to understand.

When the desire to offer a tailored experience prevails, educators may adopt a simpler form of API documentation. This *ad hoc documentation* is normally written manually and included in textbooks or teaching materials.

WebTigerJython's documentation of the `gturtle` Python library [272] is an example of ad hoc documentation for students. The API documentation assumes the form of a table with three columns: a function, a possible abbreviation, and a description in natural language. The function column contains a signature of sorts. For example, the `right` function is listed as `right(angle)` to indicate that it has one parameter (the angle of rotation). The `setPenColor` function is listed as `setPenColor("color")` presumably to indicate that it has one parameter (the new color of the pen) of type `str`.

This form of documentation has the advantage of being completely flexible. The author can decide exactly what to present in the documentation, including which terminology should be used, such that the documentation is just right for the intended context.

Unfortunately, ad hoc documentation comes with major drawbacks. First, as argued above, novices interact with documentation that is not authentic, which may be detrimental to their motivation. Second, manually writing documentation is a time-consuming activity that not every educator can afford, potentially having to resort to one written by someone else for a different context. Third, like with all manually written documents, there is the risk of a lack of consistency: different notations could be used throughout the document, both intentionally and inadvertently, carrying the risk of confusing the novice programmer.

9.2 Judicious is a novel pedagogical documentation system

Can a documentation system be designed to retain most of the benefits of real systems while incorporating sensible pedagogical features?

This section presents Judicious, a minimalist documentation system we developed

to assist beginner programmers during their first steps in learning programming. Judicious is released as open-source software[1] and has been integrated in the PyTamaro Web platform (Chapter 7).

The system targets Python, given its current popularity as a programming language for learning programming (Section 2.4). The focus on a single programming language prevents a combinatorial explosion of the number of features needed to accommodate each language's idiosyncrasies. At the same time, the pedagogical ideas embodied by Judicious are not limited to Python and could be implemented for other programming languages as well.

Judicious's main characteristics revolve around *how documentation is presented*, rather than how documentation can be programmatically extracted from source code. Currently, the system supports manually specified documentation, automatic extraction from source code leveraging Sphinx, and a simplified automatic extraction from source code as described later in Section 9.2.6.

We proceed to illustrate each pedagogical feature of Judicious in turn, presenting a rationale to motivate their need and their design based on extant work from the learning sciences and programming languages research communities.

9.2.1 Judicious includes a diagrammatic representation

From the very beginning, novice programmers have the need to use functions, often the ones included in the standard library. Students first familiarize themselves with the concept of a function in mathematics. Pure functions in programming are precisely functions in the mathematical sense. Schanzer [234] demonstrated that careful pedagogical choices facilitate the transfer of concepts—most prominently, functions—between algebra and programming. We can thus resort to techniques from mathematics education to help learners bridge between the same concept in the two subjects.

A function is commonly explained as a "black box", an opaque machine that ingests something and produces something else. The notion is often visualized with a diagram that represents the machine as a box with an entrance and an exit. This notation has been brought into computer science as well: Harvey uses a "plumbing diagram" to visualize the composition of functions [112, Ch. 2]. This representation is also known as the "Function as Tank" notional machine in the collection presented by Fincher et al. [86].

Figure 9.3 depicts how Judicious includes a diagrammatic representation of the simple `sqrt` function from the `math` module. The function is represented as a rounded rectangle with the function's name at the center. Parameters are depicted as labeled

---

[1]https://github.com/LuCEresearchlab/judicious

incoming arrows from the left, the return value as an outgoing arrow to the right.

This representation deliberately attempts to match very closely the representation used for physical function cards in TamaroCards (Chapter 6), with the intent of helping learners who also used TamaroCards to recognize the same concept.
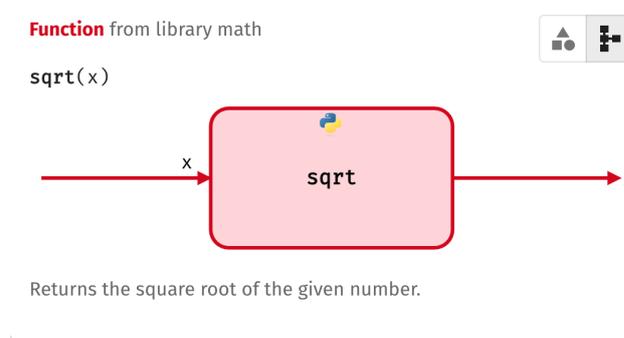
Figure 9.3. Diagrammatic representation of the `sqrt` function.

Multiple (external) representations can aid learning, provided that learners understand the notation and the relationship between the representation and the domain [5]. Indeed, multiple pieces of information are related to the parameter named x in Figure 9.4. As customary in documentation systems, the parameter is shown in the function signature at the top and further described in the list of parameters at the bottom. Judicious clarifies the relationship between the element in the diagram that refers to the parameter (the arrow on the left) and the textual description: when a student hovers over one of these elements, they all get highlighted as shown in Figure 9.4.

9.2.2   Judicious documents one name at a time

One reason why programmers get discouraged from using API documentation is the time it takes to retrieve what one needs, perhaps because it is scattered over multiple places [263]. Professional development environments recognize this need and offer various forms "inline documentation". For example, as discussed earlier in Section 9.1.1.4, Visual Studio Code uses Pylance to show the documentation when hovering over a known name (Figure 9.2).

Judicious offers that convenience to novices, sparing them from opening a separate window to find the name they need in the middle of many others. The system analyzes imported names and shows them in an interactive "documentation bar" above the code editor, allowing students to retrieve the documentation of each name individually.

Figure 9.5 shows how the documentation bar appears for a toy program. Builtin

Function from library math

`exp(x)`

Returns the exponential of the given number.

――――――――――――――― Parameters ―――――――――――――――

x exponent of the formula e^x

――――――――――――――――― Return ―――――――――――――――――

Exponential of the provided number
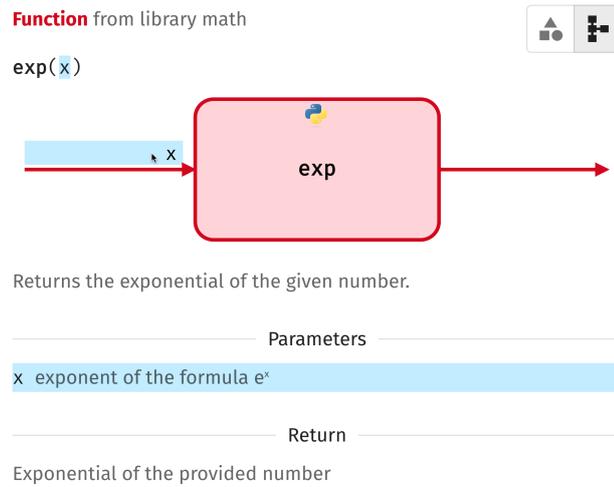
Figure 9.4. Hovering over an element in the textual or diagrammatic representation highlights the corresponding parts in the other representation.

```
Docs: sqrt  range  print

1    from math import sqrt
2    for num in range(10):
3        print(sqrt(num))
```

Figure 9.5. Judicious's documentation bar includes both imported and built-in functions.

functions (e.g., `print`, `range`) are automatically detected when the source code contains a call to them, without the need for imports.

9.2.3   Judicious presents documentation gradually

Computing education researchers have extensively studied the struggles of novices when they begin learning to program (cf. Chapter 2). As part of learning to program, beginners need to learn the syntax and the semantics of a programming language. Programming languages intended for professionals have the big appeal of being used in industry; at the same time, they include a multitude of language features that cannot all be explained at the beginning. The size and complexity of such programming languages can strain students' cognitive load.

A strategy to simplify these languages to reduce the cognitive load is to create smaller languages, also known as sublanguages. Section 2.6 briefly reviewed the his-

(a) Basic view with diagram.

(b) With optional parameters.

(c) With type annotations.

(d) Without diagram.
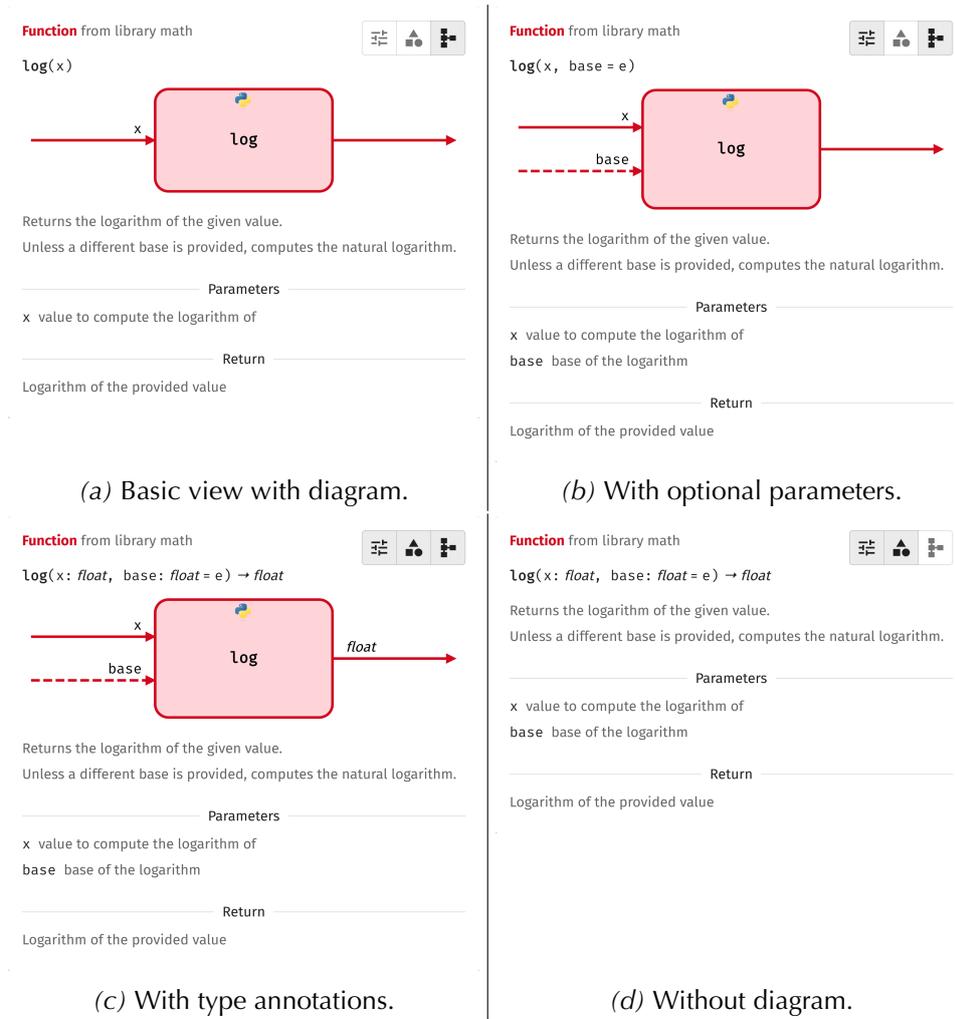
Figure 9.6. A possible evolution of the documentation of the `log` function.

tory of various sublanguages developed over the decades. These sublanguages also affect the development environment: for example, DrRacket programming environment [88] supports Racket's sublanguages by adapting its behavior (e.g., which features are available) depending on the sublanguage currently selected by the student.

Judicious applies this principle to documentation. Like a programming language, the documentation should gradually grow with the beginner programmer.

Figure 9.6 shows a progression of visualization of the same `log` function, included in Python's standard `math` library. Toggle buttons at the top right allow learners to set their preferences, potentially under the guidance of an instructor, for what gets visualized.

At initial settings, the documentation of a function is shown with the diagrammatic representation as in Figure 9.6a. This matches what learners are used to in maths and can be used to introduce the concept of a function.

Once learners understand the basic mechanism of passing arguments to a function, instructors can reveal that many functions in Python can also take optional parameters. These parameters are shown in Figure 9.6c with a corresponding dashed arrow in the diagram.

To understand the behavior of programs, types also serve as a useful form of documentation [209]. Originally a dynamically typed language, Python now supports type annotations since version 3.5. Some educators are reluctant to use them because of the additional syntactic burden. However, types are particularly useful in function signatures to signal in a lightweight way which values the function accepts and which ones it produces. Judicious gives users the choice of whether types should be shown (Figure 9.6c), allowing instructors to introduce them only at the time they feel ready to.

Finally, when learners have mastered the concept of a function, the diagrammatic representation can be hidden, resulting in Figure 9.6d, to get a more compact documentation.

9.2.4   Judicious distinguishes constants from parameter-less functions

Previous research has documented difficulties novices encounter with parameter-less functions and the potential confusion with constants.

Altadmri and Brown [10] analyzed a year's worth of Java compilations from over 250 000 students in the Blackbox dataset and found almost 19 000 instances in which over 10 000 students did not write parentheses after a method call (e.g., when trying to call the `.toString()` method). This study provided quantitative evidence about a mistake already reported by instructors [128]. Relatedly, as already discussed in Section 6.5, the inventory published by Chiodini et al. [52] contains a misconception

named PARENTHESESONLYIFARGUMENT[2] which describes the belief held by some students that () are optional for function calls without arguments.

Judicious attends to this problem and distinguishes between *accessing a constant*[3] and *calling a parameter-less function*. Figure 9.7 and Figure 9.8 contrast the two situations, respectively for `pi` from the `math` module and `random` from the `random` module. The distinction is even more pronounced in the diagrammatic representation: the parameter-less function retains all the characteristics of functions but does not have any incoming arrow on the left; the constant is depicted as a blue rectangle with an arrowhead.

Constant from library math

`pi`

pi

The mathematical constant π = 3.141592...
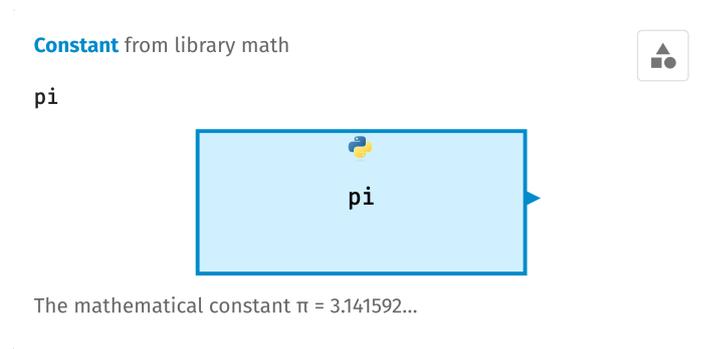
Figure 9.7. Documentation for the constant `pi`.

Function from library random Side Effects

`random()`

random

Returns a random floating point number between 0 (included) and 1 (excluded).

— Return —

A random floating point number between 0 and 1
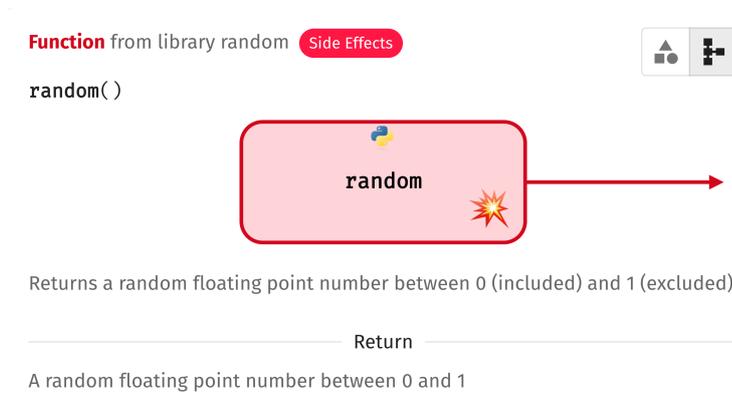
Figure 9.8. Documentation for the parameter-less function `random`.

---

[2]https://progmiscon.org/misconceptions/Python/ParenthesesOnlyIfArgument

[3]Strictly speaking, Python does not (easily) offer immutable variables, but it is pedagogically sensible to treat library variables as such. The official documentation also uses "Constant" as the terminology for this case, e.g., https://docs.python.org/3/library/math.html#constants.

Docs: ⊞greet 🐍print

```python
1    def greet(message, n_bangs):
2        return message + ("!" * n_bangs)
3
4    print(greet("Hi", 3))
```

(a) Function in the documentation bar.

Function from your code

greet(message, n_bangs)

(b) Minimal documentation.

Function from your code

greet(message: *str*, n_bangs: *int*) → *str*
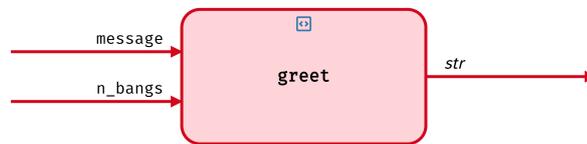
Creates a greeting message ending with the specified number of exclamation marks.

(c) Documentation types and docstring.

Figure 9.9. Documentation of a student-defined `greet` function.

9.2.5 Judicious indicates functions with side effects

Figure 9.8 also features an explosion icon, which is how Judicious denotes a function with side effects. This aspect is too often neglected even by professional documentation systems, despite being essential to highlight the distinction between the general concept of functions in programming and functions in math.

It has been observed that novices struggle to grasp the distinction between returning a value from a function and printing that value inside the function [147]. This distinction is subtle, because didactic programs frequently print the value returned by a function immediately.

It is not enough to distinguish between functions that do not return values (sometimes called "procedures"), because also functions that return values can have side effects that are hard for beginners to see. After all, printing is just one of the possible side-effecting operations. Functions from the `random` library mutate the internal state of the pseudo-random number generator. This behavior violates the mathematical notion of a function ("always produce the same output when the input is the same") and deserves to be pointed out explicitly.

9.2.6 Judicious automatically documents student-defined functions

So far we have described how documentation is presented, without discussing the flip side: how documentation gets generated in the first place. Judicious allows novices to document their functions in a lightweight way. The system leverages an existing Python parser written in Rust and compiled to WebAssembly, running entirely in the user's browser. As soon as a student defines a function, the documentation bar automatically includes it and visualizes the available data (Figure 9.9a). For example, when a student writes code that defines a function `greet`, its documentation is readily rendered as shown in Figure 9.9b.

Apart from a different icon displayed at the top of the diagram, which indicates that this function is imported from student code and not from Python's standard library, every other aspect of the documentation remains the same. This unified design aims to help students understand that the functions they import and use from a library are no different from the ones they learn to define.

Students can then immediately appreciate the usefulness of adding type annotations for the parameters and the return value of a function. The function `greet` shown in Figure 9.9a can be easily turned into Listing 22: it can be augmented with types and a so-called "docstring", a string inserted as the first statement of a function that is treated as a documentation comment. The documentation would then be rendered as shown in Figure 9.9c. The increased intelligibility should prove useful whenever the

```python
def greet(message: str, n_bangs: int) -> str:
    """Creates a greeting message ending with the
    specified number of exclamation marks."""
    return message + ("!" * n_bangs)
```

Listing 22. Example of a function defined in student code. This is the same function shown in Figure 9.9a, with type annotations and a docstring comment.

student needs a refresher on how to use the function they defined a while ago.

9.2.7   Judicious includes usage examples

In addition to reading a textual description of a function's behavior, seeing example usages can also be a way to understand how a function works or to confirm that one's interpretation of the textual description is indeed correct.

Figure 9.10. Two examples shown in Judicious for the `log` function. The second example only appears when the user enables optional parameters.

Figure 9.10 shows a pair of examples for the `log` function from Python's `math` library. Each example features a pair of Python source code and the result of its evaluation.

Judicious' examples maintain the promise of a gradual system, to avoid overwhelming novices. By default, only the first example of Figure 9.10 would be shown, as it uses the mandatory first parameter of the `log` function to compute the log of e. However, the function also has an additional optional parameter to specify the base of the logarithm. When the user enables the optional parameters (Section 9.2.3), a second example "unlocks" and is also shown. The lower part of Figure 9.10 shows how to use `log` to compute a logarithm in base 10.

Examples are also an opportunity to reinforce the distinction between a function's side effects, such as printing to the standard output, and the result of evaluating a

Figure 9.11. Examples in Judicious for Python's `input` function, showing separately the output (I/O) and the result of the evaluation.

function call. Functions typically perform one action or the other, but some built-in Python functions, such as `input`—which is commonly used with beginners—actually perform both actions.

Figure 9.11 shows how Judicious presents two examples for the `input` function. The first example is always shown, because it does not make use of the optional parameter. The user is expected to enter text, which is shown in a rectangle with a dashed border and a keyboard icon. The result of evaluating the function (i.e., the return value) is a string containing the text inserted by the user.

The second example is only shown when optional parameters are enabled (Section 9.2.3) because it provides a string as an argument. That string is printed as output and acts as a "prompt" for the user.

The clear separation in the interface between the I/O operations and the result of evaluating the function conveys the distinction between side effects and evaluation results (Section 9.2.5), which can coexist in the same function.

9.3   PyTamaro's documentation can be fully explored with Judicious

Judicious is a generic documentation system oriented to novices that can be used to teach Python. Its development was motivated by the need to offer novice-friendly documentation for PyTamaro. The minimal set of language features used by the library is fully supported by Judicious. Teachers can thus begin to explain the idea of functions and constants using PyTamaro, and then start using other functions that are part of the Python standard library. Judicious presents them in the same way, to reinforce

the idea that PyTamaro is not a "special version of Python", but merely a library from which one can import functionalities.

A teacher could remind students of how PyTamaro's `rectangle` function works, such as the order in which width and height need to be specified, by pointing students to the documentation shown in Figure 9.12a. If types have been introduced, the version of Figure 9.12b can be shown instead, for example to clarify that the color needs to be a value of type `Color` and not a string.

(a) Initial version without types.   (b) Typed version.
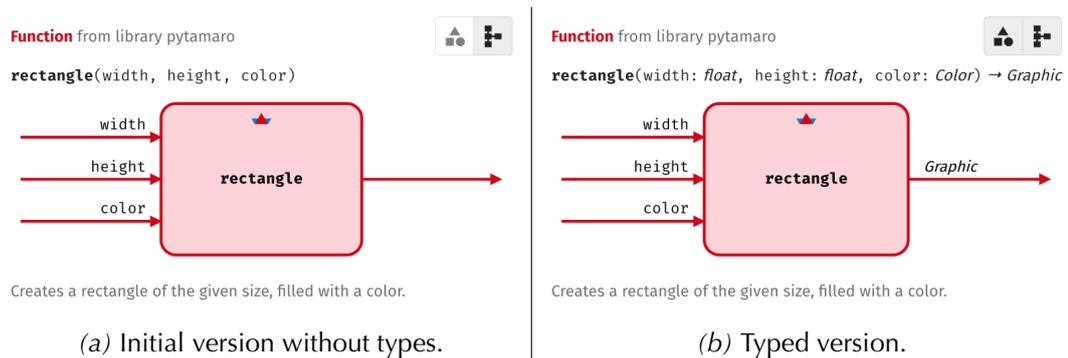
Figure 9.12. Documentation of PyTamaro's `rectangle` function.

PyTamaro's common `show_graphic` function, which is used to output values of type `Graphic`, is marked as a side-effecting function. In its basic usage, a student would call the function simply by passing a graphic as the first and only argument; the Judicious documentation would match this behavior. Later on, for example to debug the pinning position, a teacher may want to expose students to the concept of optional parameters. When **True** is passed as a second argument, `show_graphic` outputs the graphic along with debugging information. As shown in Figure 9.13, the documentation presented by Judicious grows to cover this more advanced usage.
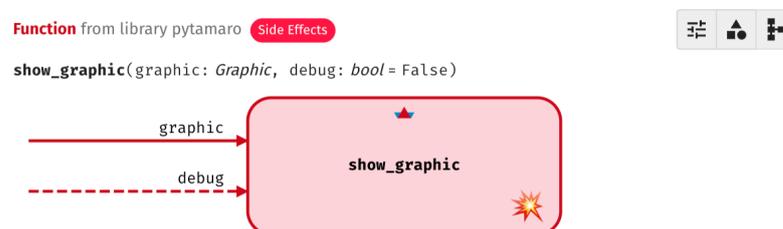
Figure 9.13. Documentation of PyTamaro's `show_graphic` function, with the indication of side effects and the "optional parameters" feature enabled.

9.4 This is how Judicious compares to existing documentation systems

We now compare the characteristics of our system with other documentation systems. Given that programming languages exhibit significant variety in terms of features, we restrict the comparison only to documentation systems for Python. For those systems, we consider the output format that beginner Python programmers are most likely to encounter. We thus compare Judicious's web system to the HTML pages produced by Sphinx (Section 9.1.1.3), which is used in the official Python documentation, and Pylance (Section 9.1.1.4), which is used in Visual Studio Code.

Table 9.1 synthesizes the results of this comparison, which we now analyze in more detail.

Table 9.1. Comparison of three documentation systems for Python.

Features	Sphinx HTML	Pylance VSCode	Judicious Web
§9.2.1: Diagrammatic representation	No	No	Yes
§9.2.2: Single-name documentation	No	Yes	Yes
§9.2.3: Gradual documentation	No	No	Yes
§9.2.4: Functions vs. Constants	Yes	Yes	Yes
§9.2.5: Side Effects indication	No	No	Yes
§9.2.6: Student-defined functions	Yes (heavyw.)	Yes	Yes
§9.2.7: Examples	No	No	Yes
All language features	Yes	Yes	No
Full coverage of libraries	Yes	Yes	No
Extraction from source code	Yes	Yes	Limited
Browsing an entire module	Yes	No	No

9.4.1 Most pedagogical features are unique to Judicious

The upper part of Table 9.1 considers each key pedagogical feature of Judicious described in Section 9.2.

The diagrammatic representation, the user-configurable gradual display of the documentation, and the indication of side effects are three characteristics unique to Judi-

cious. While the first two features are oriented toward novices, it is somewhat surprising to see that the two other systems we analyzed do not report side effects. (However, this is not universally true: Scala programmers conventionally document this distinction by adding parentheses to effectful functions.)

All systems distinguish in some way functions from constants, although only Judicious offers a distinct diagrammatic representation as a further aid. Unlike Pylance and Judicious, which are integrated with the code editor and allow quickly retrieving the documentation for a single name, Sphinx's HTML output is oriented to web pages and shows several names on the same page.

All systems allow documenting functions defined by students (as argued in Section 9.2.6, they are indeed no different from functions present in libraries). Judicious and Pylance offer a lightweight approach: beginner programmers can access a minimalistic form of documentation for a function they just defined with zero extra actions. This differs from systems like Sphinx (or Javadoc), in which the programmer needs to execute a separate tool.

Given that all systems support showing text with markup, it is in principle possible to show examples in each of them. Indeed, some third-party libraries documented using Sphinx include examples as part of the description of their functions. However, currently neither the official Python documentation shown on the website nor the one provided within Visual Studio code provide any example. Examples that are revealed gradually are thus a feature that is unique to Judicious.

9.4.2   Other systems offer certain features not in Judicious

The lower part of Table 9.1 considers other aspects where Judicious falls short in comparison to professional documentation systems. Judicious only covers a small subset of Python's extensive feature set: this enables offering novice-friendly functionalities, but excludes language features that are legitimately needed by proficient programmers. Moreover, Judicious supports a limited form of extraction from source code: it leverages a Python parser to extract functions from student code as described in Section 9.2.6, and exploits Sphinx's `docutils` and `autodoc` utilities to extract documentation from the source code of existing libraries. However, the latter approach cannot be applied to Python's standard libraries because they are partially implemented in C by Python's reference interpreter CPython and are not annotated with standard docstrings. Indeed, the official documentation of Python's `math` library is written manually as a Sphinx document. We had to use the same manual approach to document those libraries in Judicious.

A separate problem, actively investigated in research, is studying how developers *discover* the APIs they need. Multiple studies have revealed intricate retrieval patterns,

often not well supported by existing programming environments and documentation systems [250, 145]. Crichton [64] argues that programmers employ different kinds of leads when searching (e.g., a description of a function behavior in natural language or based on types) and proposes "scanning-oriented" user interfaces. Judicious, like Pylance, is only concerned with the visualization of the documentation of a single name. The PyTamaro Web platform, however, offers dedicated pages to browse the Judicious documentation. It is possible to search for a name and its description, or to explore the set of documented libraries and then access the list of functions, constants, and types available in a specific module.

9.5 The effectiveness of Judicious has not been empirically evaluated

Judicious has been designed by building on prior studies that observed and tackled difficulties novices have when learning to program. We analytically evaluated our system against comparable state-of-the-art alternatives in Section 9.4, but the system still lacks an empirical evaluation to measure its effectiveness in practice. (Although it does not constitute an empirical evaluation, the case study that will be presented in Chapter 11 also explores the use of Judicious by teachers.) We received anecdotal positive feedback from teachers who adopted it with their students on two aspects: the diagrammatic representation, which helped to explain functions, and the ease of retrieving the documentation for a name, which drastically increased student usage of documentation. The latter observation is not entirely surprising, as reducing friction is known to change the behavior of users. For example, in an experiment, Google returned search results with an additional 0.5-second delay and traffic dropped by 20 % [101].

The documentation system itself also has limitations, which we discuss below.

Despite the current popularity as an introductory programming language, Python is a complex language with an extensive number of features [211]. Judicious only supports the small subset of these features to cover an "expression-oriented", "functional" subset of the language that is still meaningful for a CS1 course [11]. The system thus supports functions and constants but does not include classes with their methods. In function definitions, in addition to "regular" parameters, Judicious supports variable-length parameters and parameters with default values, given their pervasive use in Python (even `print` uses all these features!). Less common options (positional-only parameters and "kwargs") are not supported.

The documentation of the Python standard library has been manually written for Judicious, considering the target audience. The writing does not exhaustively describe the functions: for example, it does not include which exceptions might be thrown for in-

valid combinations of arguments, and it reports a simplified version of the complicated types that have been retrofitted to Python. Figure 9.2 exemplifies this predicament: Pylance reports an obscure custom `_SupportsFloatOrIndex` type that accurately describes the type of the parameter; Judicious resorts to `float`, which is inaccurate but more intelligible for novices.

Concerning side effects, we note that Judicious does not run sophisticated program analysis. Functions in the standard library are manually tagged as effectful and no purity analysis is run on student-defined functions.

Part IV

Empirical Investigations

Chapter 10

We Studied Transfer, Engagement, and Code-Related Skills

As discussed in Section 2.10, a number of studies have demonstrated that graphics-based pedagogies have positive effects on engagement [108, 241, 214, 15]. Something that has received comparatively little attention is whether students generalize the concepts they learn in the domain of graphics into programming concepts and transfer them to programming in other domains. Ultimately, even approaches that teach programming using graphics aim to teach students programming skills that they can also apply in other domains.

Papert [194] argued that turtle graphics is an excellent vehicle to teach mathematics, programming, and problem-solving in general. However, Pea and Kurland [203] conducted studies with children using turtle graphics in Logo and did not find the hoped transfer. Planning skills did not improve after a year of programming in Logo [202]. And even for programming skills, the understanding of concepts depended highly on the context. As an example: "a child who had written a procedure using REPEAT which repeatedly printed her name on the screen did not recognize the applicability of REPEAT in a program to draw a square" [203].

To the best of our knowledge, no study has been conducted to empirically evaluate *which* approach best fosters transfer from programming using graphics to programming in general.

Our study seeks insight into both engagement and conceptual transfer. We compare two fundamentally different approaches to graphics: compositional graphics (Section 3.5), as embodied by PyTamaro, and the well-established turtle graphics (Section 3.4).

We ask the following research questions:

RQ1 Is there a difference in conceptual transfer from a short programming tutorial

with a compositional graphics library like PyTamaro or a turtle graphics library to programming outside the domain of graphics?

RQ2 After a tutorial following either approach, are there differences in how students read or write programs?

RQ3 Do the two approaches lead to different levels of student engagement or perceived learning?

10.1   Evaluations of graphics-based approaches and the challenge of transfer

A few studies have found evidence of transfer from graphics-based programming to *mathematics*. Noss [192] carried out an experiment showing that Logo helped to learn certain geometrical concepts. Schanzer et al. [232] presented initial data that show a measurable transfer of skills from a "functional" programming curriculum (which also includes compositional graphics) to algebra when instructional materials are carefully aligned to the concepts normally covered in math classes.

Empirical evidence remains scarce on whether programming with graphics helps with *general programming skills*. Already in the 1980s, Pea and Kurland [203] looked into claims that Logo (and its turtle graphics) helps with problem-solving in general but found little evidence in support; moreover, they found that after 30 hours of programming with Logo, "children's grasp of fundamental programming concepts such as variables, tests, and recursion... was highly context-specific" [203], thus illustrating how difficult it is for learners to transfer their knowledge from a particular context or domain to others. More recently, Guzdial explored what they called the "learning hypothesis" of their media computation curricula, but the results were inconclusive or negative compared to a traditional curriculum [108].

Others have looked into block-based programming environments with graphical affordances, such as Scratch, but found limited evidence of transfer, or even a negative impact for some activities. For example, Grover and Basu [103] found that after an introductory programming course in Scratch, students demonstrated several misconceptions about fundamental programming concepts. Weintrop et al. [273] argue that certain aspects of block-based programming environments are counterproductive for transfer to text-based programming languages, and that this learning trajectory needs to be adequately supported.

10.2 Compositional graphics approaches should have potential for transfer

Chapter 4 identified several trade-offs in graphics-library design for novices. Chapter 5 argued that a compositional graphics library like PyTamaro can be a meaningful alternative to comparable libraries, under the right circumstances and given certain pedagogical goals.

As noted above, PyTamaro is designed to assist in the acquisition of fundamental programming concepts. For instance, few language constructs are needed for writing PyTamaro programs (e.g., no classes and objects), the API is minimal (e.g., no functions for loading external images), and neither mutable state nor coordinate-based operations are present. That is, PyTamaro has *deliberate limitations*: learners are not given access to certain functionality. This design should help manage complexity for beginner programmers, guide them towards better-quality programs, and nevertheless engage them meaningfully not only with graphics but with key computing content that is not specific to the graphics domain. In other words, PyTamaro is claimed to hold the potential for improved transfer.

The decomposition of problems into independent subproblems is key not only to professional programming [247, 198, 182] but broadly to computational thinking [281] and problem-solving [31]. However, analyses of programs written by students show that decomposition and, relatedly, abstraction are not practiced enough [181, 142]. Several of PyTamaro's intended benefits involve these key concepts and skills. Mutable state and "interface steps" (Section 4.1.2) in turtle graphics hinder effective problem decomposition. State makes it harder to reason about a subproblem in isolation: to understand what is the effect of a given piece of code, novices have to mentally reconstruct the state of the turtle. Lewis [164] documented how both school- and college-level students struggle with the turtle's state, leading to issues that are hard to debug. Compositional graphics approaches like PyTamaro eliminate this problem by offering only pure functions that produce immutable graphics.

Treating graphics as values to be composed also facilitates *visual decomposition* (Section 5.3). One may look at an image and identify its components (e.g., the roof and floor in the trivial house example of Figure 3.1), and see how those visual components compose into an overall image; this maps directly to how the subproblems' programmatic solutions compose into an overall program. Learners may be guided to visually decompose graphics and thereby learn how to decompose programming problems and to compose the subprograms that solve them. Visual decomposition is relatively straightforward when objects are immutable and one does not need to consider components' locations as coordinates.

10.3 We used a specific methodology for the randomized controlled experiment

This section describes the overall design of our experiment, its context and participants, the teaching interventions for the two experimental conditions, the surveys the participants answered before and after the intervention, the post-test, and finally our analysis methods.

10.3.1 The procedure included four phases

We designed a randomized, between-subjects experiment with two conditions. In both conditions, the participants worked through a programming tutorial consisting of four "mini-lessons" with a Python graphics library. We selected PyTamaro in its English API version as an example of a compositional graphics library (Section 3.5) and `turtle` from Python's standard library for turtle graphics (Section 3.4).

Since the study took place in a proctored computer laboratory and the participant count was high, we arranged four identical sessions over two weekdays. Each session consisted of four phases (Figure 10.1): a pre-survey, a teaching intervention phase, a post-survey, and a post-test. The teaching intervention was different for the two groups, as was part of the post-test (as explained below); the other phases were identical for both. We allowed participants a maximum of 90 minutes to complete the entire session.

Figure 10.1. The timeline of each session.

Although some of the authors oversaw the sessions, they did not directly teach anything to either group. Instead, the intervention took the form of a self-paced tutorial on a sequence of web pages. This decision was made in an effort to increase the reproducibility of our findings and to eliminate biases in favor of our own library, PyTamaro.

No Pre-Test? By design, our experimental setup did not include a pre-test. This means we cannot compute learning gains, but the decision can nevertheless be justified both methodologically and pragmatically.

First, the differences between the two groups are ironed out because we assigned the large number of participants to the two conditions randomly. This follows the

recommendations of Campbell and Stanley: "While the pretest is a concept deeply embedded in [researchers'] thinking [...] it is not actually essential to true experimental designs. [...] the most adequate all-purpose assurance of lack of initial biases between groups is randomization" [38].

Second, a pre-test on programming could have brought about learning and muddled our results. There is evidence that just taking a test again leads to better learning outcomes [39]. We wanted to avoid that and instead study the effect of the teaching intervention with a compositional graphics library like PyTamaro, checking for transfer using the Turtle group as a baseline.

Third, the time constraints given by the context forced us to consider whether to have a pre-test or a longer teaching intervention. We decided to dedicate a greater fraction of the limited time of the experiment to the intervention.

10.3.2 We recruited participants from a CS1 course

We recruited participants from an introductory programming course (CS1) at a large European research university that is not where PyTamaro originated and where PyTamaro had never been used before. The course is held during the first semester of undergraduate studies and targets non-majors in CS, especially students from other engineering fields. The course uses Python and adopts what might be described as a 'typical imperative programming pedagogy'; it does not feature any graphics-based programming.

None of the present authors are involved in teaching the course. The sessions took place during the third week of the course, outside class hours. During the first two weeks, the course had covered variables, basic I/O, assignment statements, simple arithmetic, and `if` and `while` statements; `for` loops were introduced in the third week. We organized the study very early in the course to limit prior programming knowledge. We ruled out the possibility of running the study even before the course start date: besides our participants having yet to begin their university path, it would have been unfeasible to cover meaningful content in a short teaching intervention with absolute beginners.

We advertised the study during the introductory lecture of the course. Participation was voluntary. Upon completion (but irrespective of performance), the participants were rewarded with a minor amount of course credit and a movie ticket.

A web platform provided all the materials and assessments and took care of randomization and data collection in accordance with the local anonymization policies. Each participant consented to the use of their anonymous data.

10.3.3   We asked participants a pre-survey

The participants answered a pre-survey with questions on three areas. First, we asked standard demographic questions. Second, we gauged their prior knowledge asking how many lines of code they had written before and whether they had ever programmed graphics. Third, we surveyed their attitude towards programming with three Likert items. The full questions are available in Appendix A.1.

10.3.4   We carefully designed a short teaching intervention

The teaching intervention for each group consisted of four "mini-lessons". Each such lesson took the form of a web page and consisted of text, executable snippets of Python code, and a few illustrations. Some of the snippets were ready to execute as-is, but the majority offered only a starting point for the participants to write their own code as instructed.

10.3.4.1   There is an interplay between pedagogy and library

The lessons were designed purposely for this experiment. We tried to align the materials for the two conditions closely, while still adopting a meaningful use of each library. This was difficult, as there is an interplay between the tools one chooses and the pedagogy one may then adopt; each library is associated with a 'typical' pedagogy that tries to match its strengths and minimize its weaknesses.

At one extreme, a library might make a concept inaccessible because there is no support for it. For example, when graphics are not treated as values and functions are essentially procedures, as in turtle graphics, it is impossible to construct nested calls or other composite expressions with graphical values. At the other extreme, a library might make it practically mandatory to teach a certain concept. PyTamaro has, in common use, a parameterless function (`empty_graphic`) and various non-commutative functions that take two parameters (e.g., `above`), effectively forcing learners to deal with these concepts.

Somewhere in the middle of the spectrum, a library may nudge pedagogy towards certain concepts that are particularly compatible with it; for example, PyTamaro does not mandate exploiting associativity and multiple ways to decompose a particular problem, but it invites teachers and learners to explore these topics. Conversely, a library and its standard pedagogies may not particularly need a concept, but the concept may still be introduced despite not being prominent.

We sought a balance within these constraints to keep the experiment fair, especially avoiding favoring PyTamaro.

10.3.4.2 This is the content of the four mini-lessons

The lessons focused on using variables and functions, and on composition in general. Broadly, they emphasized expressions, an essential concept even in non-predominantly functional languages such as Python, but that is often neglected by traditional imperative pedagogies [53] like the one adopted in our CS1 course.

PyTamaro's approach is compositional and exploits expressions. It thus offers the right opportunities to explain these concepts, which were only briefly introduced in the first part of the CS1 course that took place before the experiment.

For both groups, we created materials in two natural language versions: one in English, another in Finnish. Each participant was free to choose whichever of the two; 39 % used the English version. The identifiers in Python code were identical (in English) in both language versions.

The first lesson introduced the idea of a software library and showed how to call library functions. For PyTamaro, functions' parameters and return values were visualized with a "plumbing diagram" similar to Harvey [112, Ch. 2]. For Turtle, animated GIFs showed how the turtle executes a sequence of commands including movements and rotations.

The second lesson guided both groups toward drawing something slightly more interesting: the house from Figure 3.1. The CS1 course had not yet covered function definitions, whose introduction from scratch would have required too much time. We opted to provide both groups with functions such as `square` and `triangle` and focus on their usage.

The third lesson explained how to draw a more complex graphic with two houses and a wall in between, to whet the learners' appetite for constructs that repeat computations. The PyTamaro group practiced combining variables and nesting, whereas the Turtle group focused on the importance of the order in the sequence of commands.

The fourth and final lesson introduced repeated computation with `for` loops, which students had just started practicing in the CS1 course. Both groups drew a "street" of five houses side by side: the PyTamaro group used the parameterless `empty_graphic` function to initialize an "accumulator" variable (cf. zero in summation), the Turtle group added an "interface step" to reposition the turtle at the end of each iteration.

Here we only briefly outlined the contents of the lessons. The complete version for both groups is available in Appendix A.2.

10.3.5 Before the post-test, participants had to complete a post-survey

Immediately after the teaching intervention, we asked the participants to complete another survey. This was to explore RQ3 by eliciting their opinions on the lessons

they just experienced, their level of engagement with programming in the domain of graphics, and whether they had actually perceived to be learning programming.

We formulated four hypotheses for engagement and three for perceived learning. The independent variables are (separately) the approach followed and the gender.

On engagement. *There is a difference in...*

H3a *... how interesting they think the tutorial was.*

H3b *... how fun they find programming graphics.*

H3c *... how much more they like programming graphics over programming in other domains.*

H3d *... how much they would like to learn more with graphics.*

On perceived learning. *There is a difference in...*

H3e *... how much they feel they have learned about programming.*

H3f *... how much they feel they already knew that approach to program graphics.*

H3g *... how much they feel they already knew the programming concepts taught.*

Each hypothesis corresponds to a seven-point Likert item in the post-test, answerable from "not at all true" to "completely true" (Table 10.7). The exact items as presented to the participants are shown in full in Appendix A.3.

10.3.6   The post-test consisted of nine questions

In total, our post-test had nine questions whose themes are listed in Table 10.1. The post-test can be logically divided into two parts (even though this division was not visible to the participants).

The first part was identical for both groups. It consisted of six multiple-choice questions that relate to general programming concepts and RQ1, whose associate hypothesis is the following:

H1 *There is a difference in conceptual transfer between the group that followed a programming tutorial with PyTamaro or with Turtle, as measured on programming tasks outside the domain of graphics.*

We operationalize transfer as participants correctly answering the six multiple-choice questions.

Table 10.1. The themes of our post-test questions (Q1 to Q9) and how they map to the hypotheses derived from our first two research questions.

H	Post-Test	Topic / Task
	Q1	Use a variable more than once in an expression.
	Q2	Nest function calls.
H1	Q3	Use a function with more than one parameter.
	Q4	Call a parameterless function.
	Q5	Exploit an associative operation for multiple solutions.
	Q6	Determine the initial value of a loop's accumulator variable.
H2a	Q7	Tracing: Given a program, determine the size of the result.
H2b	Q8	Writing: Create a program to draw a simple graphic.
H2c	Q9	Modifying: Modify a given program that draws a graphic.

The second part of the post-test relates to our second research question, from which we formulate three specific hypotheses:

After following a programming tutorial with PyTamaro or with Turtle, there is a difference in how learners...

H2a ... *trace an existing program that creates a graphic.*

H2b ... *write a program from scratch to create a graphic.*

H2c ... *modify a given program that creates a graphic to adapt to new requirements.*

10.3.6.1   Q1 to Q6 were multiple-choice questions on programming

The first six questions, Q1 to Q6, were multiple-choice questions unrelated to graphics. The questions targeted expression-related programming concepts (function calls, variable use, composition with operators or nesting) where the PyTamaro approach could yield better transfer, given that these concepts were only explicitly practiced in PyTamaro's teaching intervention. The Turtle group thus served as a baseline with respect to these questions: the Turtle participants had to answer these questions on the basis of whatever they had learned (or failed to learn) prior to the experiment.

Table 10.2 shows the alignment between the six multiple-choice questions and the examples featured in the teaching materials for the PyTamaro group. In the learning sciences literature, these are known as isomorphic tasks (or simply as isomorphs). We will comment further on transfer and its challenges in the Discussion (Section 10.5.3).

The table only summarizes the stems. For completeness and reproducibility, the questions with answer options appear in full in Appendix A.4.

Table 10.2. Each multiple-choice question targeted an abstract concept that should be learned in the teaching intervention phase for the PyTamaro group and then transferred to an isomorphic task in the testing phase (Figure 10.2). For compactness, the tasks from the intervention (left) and the questions (right) are summarized here as single sentences. The full questions and answers can be found in Appendix A.4.

<table>
<tr><td colspan="2" align="center">Abstract Concept</td></tr>
<tr><td>Teaching Intervention with PyTamaro</td><td align="right">Post-Test (Both Groups)</td></tr>
<tr><td colspan="2" align="center">Q1: The same variable can be used more than once in an expression.</td></tr>
<tr><td><code>beside(house, house)</code> is valid</td><td align="right">is <code>print(word + word + word)</code> valid?</td></tr>
<tr><td colspan="2" align="center">Q2: Function calls can be nested.</td></tr>
<tr><td><code>rotate(45,</code>
  <code>rectangle(200, 100, green)</code>
<code>)</code> is valid</td><td align="right">is <code>sqrt(</code>
<code>sqrt(16)</code>
<code>)</code> valid?</td></tr>
<tr><td colspan="2" align="center">Q3: A function can have multiple parameters and their order matters.</td></tr>
<tr><td><code>above(ground_floor, roof)</code>
is valid and different from
<code>above(roof, ground_floor)</code></td><td align="right">is <code>subtract(10, 7)</code>
valid and different from
<code>subtract(7, 10)</code>?</td></tr>
<tr><td colspan="2" align="center">Q4: A function can have zero parameters and calling it still requires parentheses.</td></tr>
<tr><td><code>empty_graphic()</code> is valid</td><td align="right">is <code>fake_random()</code> valid?</td></tr>
<tr><td colspan="2" align="center">Q5: Multiple valid decompositions: if $\otimes$ is associative, $a \otimes (b \otimes c) \equiv (a \otimes b) \otimes c$.</td></tr>
<tr><td><code>beside(house,</code>
  <code>beside(wall, house))</code>
is equivalent to
<code>beside(beside(house, wall),</code>
  <code>house)</code></td><td align="right">is <code>combine("re",</code>
<code>combine("stau", "rant"))</code>
equivalent to
<code>combine(combine("re", "stau"),</code>
<code>"rant")</code> ?</td></tr>
<tr><td colspan="2" align="center">Q6: The initial value of a loop's accumulator variable is the operation's neutral element.</td></tr>
<tr><td>when combining graphics,
initialize <code>result</code> to <code>empty_graphic()</code></td><td align="right">when multiplying numbers,
should <code>result</code> be initialized to 1?</td></tr>
</table>

10.3.6.2   Q7 to Q9 featured programming tasks in the graphics domain

Unlike the questions on general concepts described above, the other three post-test questions could not be identical for the two groups, as the questions involve programming graphics and the two groups learned different approaches for that. We strove for

tasks that are as close to each other as possible and yet respect the idiosyncrasies of each approach. Nevertheless, as the two groups' programs are not identical, this part of our results speaks not only of what the participants learned during the intervention but also of the characteristics of "typical" code written using the two approaches to program graphics.

Below, we consider each of the hypotheses related to RQ2 in turn.

10.3.6.3 Q7 was a tracing task

Question 7 asked participants to trace a program that draws four squares in a two-by-two grid, as shown in Table 10.3. The participants were prompted for the dimensions of the resulting drawing. An answer is considered correct only when both dimensions are correct.

This question checks for differences in how well participants can trace a program (hypothesis H2a).

Table 10.3. Q7: What are the width and height of the resulting drawing?

PyTamaro	Turtle
```a = square(10, black)` `b = square(10, black)` `c = above(a, b)` `d = square(10, black)` `e = square(10, black)` `f = above(d, e)` `g = beside(c, f)` `show_graphic(g)```	```pencolor("black")` `square(10)` `left(90)` `forward(10)` `right(90)` `square(10)` `forward(10)` `square(10)` `right(90)` `forward(10)` `left(90)` `square(10)```

10.3.6.4 Q8 was a program writing task

Question 8 was designed to address the second hypothesis related to RQ2: are there differences between the two approaches when learners write a program from scratch?

The participants were asked to write a Python program to draw the simple graphic in Table 10.4. The PyTamaro group were expected to create a hammer's head and handle with the `rectangle` function, to compose them together, and to rotate the composite graphic. The Turtle group were expected to use a combination of movements and rotations to draw a colored letter T.

Table 10.4. Q8: Write a program to draw the given graphic. (One correct answer is shown for each group.)

PyTamaro	Turtle


```
head = rectangle(120, 30, black)        pencolor("red")
handle = rectangle(40, 200, red)        forward(200)
hammer = above(head, handle)            backward(100)
rotated_hammer = rotate(45, hammer)     right(90)
show_graphic(rotated_hammer)            pencolor("black")
                                        forward(250)
```

10.3.6.5   Q9 was a program modification task

Question 9 gave the participants a program that places five identical houses (like the one in Figure 3.1) next to each other with a `for` loop. This program was the same as the one featured in the fourth mini-lesson.

Successfully modifying the PyTamaro program requires identifying the code that deals with the create-a-house subproblem, and updating the relevant arguments given to `square` and `triangle`. In contrast, modifying the Turtle program also requires the participant to position the turtle correctly before each loop iteration (e.g., editing the call to `forward`). A difference in the success ratio on this task between the groups would validate hypothesis H2c.

This task is meant to challenge the participants on a small scale with issues of maintainability. The turtle approach inherently has one additional challenge because

Table 10.5. Q9: The participants were asked to double the dimensions of the houses.

PyTamaro	Turtle

```
ground_floor = square(100, yellow)        n_houses = 5
roof = triangle(100, 100, 60, red)        for i in range(n_houses):
house = above(roof, ground_floor)             pencolor("yellow")
n_houses = 5                                   square(100)
street = empty_graphic()                       pencolor("red")
for i in range(n_houses):                      left(60)
    street = beside(street, house)             triangle(100)
show_graphic(street)                           right(60)
                                               forward(100)
```

the solutions to the subproblems, such as drawing a single house, cannot be composed independently of their specifics (the width of one house).

10.3.7 We analyzed the data with different techniques

Our study is mostly quantitative. We ran a power test to compute an appropriate minimal sample size, seeking to keep false positives under 5 % ($\alpha = 0.05$) and false negatives under 20 % ($\beta = 0.2$). We speculated on a "medium" effect size ($d = 0.5$) in either direction (two-tailed test). These constraints yielded a minimal sample size of 64 participants per group (cf. Table 2.4.1 in [60]); our participant count (145) exceeds this minimum.

We use parametric tests, trusting the Central Limit Theorem to guarantee normality on our large sample. We do not make assumptions about the direction of effects, or on the equality of variances. We therefore compare means with two-tailed, independent-samples t-tests without the equal-variance assumption—a.k.a. Welch's t-tests.

Following widespread recommendations [249, 258], we report each p-value together with an effect size (Cohen's d). Instead of performing corrections for multiple comparisons and then making claims of statistical significance, we consider p-values and effect sizes together in an attempt to interpret our results' real-world significance [249]. Below, we will note where a p-value is under the traditional threshold for significance $\alpha = 0.05$ or where an effect size is at least "small" ($d \geq 0.20$ as suggested by Cohen [59]).

Some of the questions in our pre- and post-surveys were on a seven-point Likert

scale. We treat the responses to these questions as interval data, while acknowledging that there is no universal consensus on whether it is acceptable to do so [161]. (We adopted a seven-point scale, as simulations show that using more points brings the distribution closer to normal [283].) We thus use the parametric t-test for Likert items; the non-parametric Mann–Whitney–Wilcoxon test would also be appropriate, but it has been shown that the two tests have similar power for almost all distributions [68].

To complement our inferential statistics, we conducted semi-formal analyses of certain student responses (e.g., types of programming errors made), as described below.

10.4   These are the results of our experiment

As noted above, data was collected from four sessions spread over two weekdays. The sessions had 38, 39, 36, and 32 participants, respectively, for a total of 145 participants. The web platform we used automatically assigned participants to groups with equal probability. We ended up with 70 participants in the PyTamaro group and 75 in the Turtle group.

The PyTamaro group spent an average of 47 minutes on the whole session, which is somewhat longer than the Turtle group's 42 ($d = 0.38$, $p = 0.02$). We hypothesized that this might be due to some participants' prior exposure to programming with turtle graphics (Section 10.4.1), so we separately checked only those participants who had never programmed any graphics before; this reduced the difference to roughly three minutes (47 vs. 44; $d = 0.26$, $p = 0.19$).

10.4.1   The pre-survey indicates that most but not all participants were novices

The participants had an average age of 22 years (standard deviation 5.0). 71 participants (49 %) identified as female, 70 (48 %) as male, 0 as non-binary, 2 as other, and the remaining 2 preferred not to disclose gender information.

The vast majority of participants (120, 83 %) reported to have completed the first three rounds of exercises in the CS1 course they were taking. Pre-CS1 experience with programming was uncommon, with some exceptions.

65 participants (45 %) reported having written 0 lines of code before CS1 (excluding any HTML and CSS). 33 (23 %) reported fewer than 50 lines in total, 33 (23 %) fewer than 500, 11 (8 %) fewer than 5 000, and 3 (2 %) over 5 000. When asked whether they had ever written a program that draws graphics, 104 (72 %) participants answered no, 29 (20 %) yes, and the remaining 12 (8 %) were not sure. (We did not directly ask the students which tools they used for programming with graphics,

but given the setting, we expect that Scratch is at the top of the list, and that some would also have done turtle graphics; the compositional graphics approach is likely to be very rare.)

Below, we use the term *novice* for participants who both had written fewer than 500 lines of code and had never programmed graphics before. By this measure, 107 participants (74 %) were novices; they ended up evenly split across the two groups, with 53 novices assigned to PyTamaro and 54 to Turtle.

As expected, given the large sample and randomization, there was no significant difference in self-reported prior programming knowledge between the groups ("lines of code written" on a scale between 0 and 4; $p = 0.59$, $d = -0.08$).

The participants generally expressed positive attitudes to programming. On a scale from 1 to 7, they strongly agreed that "it is useful for me to know programming," with an average rating of 6.21($\pm$0.84). Moreover, they moderately agreed that "programming is fun" (5.34 $\pm$ 1.36) and did not agree that "programming is boring" (2.25 $\pm$ 1.27).

10.4.2 There were no differences in transfer to programming concepts

We rated the answers to the six multiple-choice questions, Q1 to Q6, as either incorrect (including "I don't know") or correct. Table 10.6 shows the results as percentages of correct answers for each question.

Table 10.6. Proportions of correct answers on Q1 to Q6, shown first for all participants and then for novices only.

		Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**
All	PyTamaro % ($N = 70$)	82.9 %	80.0 %	97.1 %	57.1 %	94.3 %	68.6 %
	Turtle % ($N = 75$)	81.3 %	78.7 %	93.3 %	58.7 %	90.7 %	73.3 %
	Delta %	1.5 %	1.3 %	3.8 %	-1.5 %	3.6 %	-4.8 %
	p-value	0.81	0.84	0.29	0.85	0.41	0.53
	Effect size	0.04	0.03	0.18	-0.03	0.14	-0.10
Novices	PyTamaro % ($N = 53$)	79.2 %	81.1 %	96.2 %	56.6 %	92.5 %	60.4 %
	Turtle % ($N = 54$)	81.5 %	74.1 %	94.4 %	51.9 %	90.7 %	70.4 %
	Delta %	-2.2 %	7.1 %	1.8 %	4.8 %	1.7 %	-10.0 %
	p-value	0.77	0.39	0.67	0.63	0.75	0.28
	Effect size	-0.06	0.17	0.08	0.09	0.06	-0.21

Overall, the participants fared decently well on these questions, which targeted basic expression-related programming concepts: average correctness across questions and groups was 80 %. Q3 (multi-parameter functions) and Q5 (equivalence of associative solutions) were solved correctly by more than 90 % of participants. Q4 (calling a parameterless function) and Q6 (initialize a variable for a loop) proved to be the hardest, with aggregate averages of 58 % and 71 %, respectively.

The differences are negligible on all six questions. Only three questions meet the very low bar of 0.10 for effect size: Q3 and Q5 in favor of PyTamaro and Q6 in favor of Turtle. None of these differences are statistically significant.

As shown in the bottom half of the table, we also looked separately into novice performance. Again, we found no major differences. The only effect size to pass the "small effect" threshold was -0.21 in favor of Turtle on Q6, and none of these results are statistically significant either.

We checked for a correlation between the time participants spent on the post-test and their performance on the multiple-choice questions. However, this correlation was effectively zero: $r^2 = 0.02$.

10.4.3 Programming tasks had more diverse results

The last three questions in the post-test targeted the three facets of RQ2. We present the results for each related hypothesis in turn in the next subsections.

10.4.3.1 There was a large difference on tracing

In the PyTamaro group, 90 % of the participants answered their tracing task correctly, whereas only 37 % of the participants in the Turtle group correctly answered their "comparable" task. The difference is statistically significant ($p < 0.001$), and the effect size is very large ($d = 1.29$).

We also looked at novices separately and found a similar trend. 89 % of PyTamaro novices answered correctly, compared to only 33 % for Turtle. Again, the difference was statistically significant ($p < 0.001$) and the effect size large ($d = 1.36$).

10.4.3.2 Both groups performed well on a simple program writing task

Both groups performed rather well on this task. 59 out of 70 (84 %) in the PyTamaro group solved the task correctly (i.e., the program does what was asked), as did 63 of 75 (84 %) in the Turtle group. Looking to understand the student programs in more detail, we semi-formally analyzed the incorrect answers.

The eleven incorrect answers in the PyTamaro group can be characterized as follows. Three participants did not submit a solution; two tried to rotate the individual

graphics before composing them (which leads to issues related to the bounding box); one supplied `rotate`'s arguments in the wrong order; one rotated the hammer in the wrong direction; one had a typographical error; and three had a mix of other issues.

The turtle group had twelve incorrect answers: four participants mixed rotation (e.g., `left`) with movement (e.g., `forward`); two drew the letter 'upside down'; two used wrong lengths; one colored the entire drawing in red; one used strings where integers were called for (e.g., `"250"`); one drew an extra line; and one issue we were unable to classify.

10.4.3.3 Both groups also performed well on a simple program modifying task

On this question, too, both groups performed rather well. 56 out of 70 (80 %) in the PyTamaro group solved the task correctly, and in the Turtle group the number was even higher: 66 of 75 (88 %). The difference is not statistically significant ($p = 0.15$), and the effect size is small ($d = -0.25$).

In the PyTamaro group, several participants changed numbers incorrectly in at least one place. In three cases, `triangle`'s angle argument was also doubled; in one case only the floor was updated, and in one other case only the roof; one participant multiplied numbers by 1.5 instead of 2; three other participants otherwise altered the numbers incorrectly; and one tried to inject an extra instruction in the loop body. Four participants did not submit a solution.

In the Turtle group, we expected to see several failures to update `forward`'s argument (as described in Section 10.3.6.5 above). However, of the nine incorrect solutions, only two were in this category. The other wrong solutions were either not submitted (four cases), had incorrect numerical arguments (two), or inadvertently redefined the name `square` (one).

10.4.4 The post-survey reports engaged students, with some differences

Table 10.7 summarizes the post-survey results, first comparing the groups and then looking at possible gender differences. Each claim, in order, tests the corresponding hypothesis (H3a to H3g).

Overall, the participants liked the activities and found them useful for learning. PyTamaro users were more inclined to say that they "had learned about programming concepts from the lessons" ($d = 0.36$, $p < 0.03$). As expected, PyTamaro's compositional graphics approach was less familiar to the average participant than Turtle ($d = -0.39$, $p < 0.02$). Moreover, we observed small effects in favor of PyTamaro with respect to

Table 10.7. Post-survey results divided by experimental group (left) and gender (right).

Claim (Likert scale from 1 to 7)	PyT.	Tur.	p	d	Fem.	Male	p	d
Engagement:								
I found the preceding lessons interesting	6.06	6.00	0.74	0.06	6.08	5.95	0.46	0.13
Programming with graphics is fun	6.01	5.83	0.33	0.17	5.97	5.86	0.56	0.10
I like programming with graphics more than the text-based programming we have done in the course	4.44	4.56	0.66	-0.08	4.73	4.29	0.11	0.30
I would like to learn more about programming with graphics	6.09	5.87	0.16	0.24	5.97	6.00	0.85	-0.03
Perceived learning:								
I feel that I learned about programming concepts from these lessons	6.07	5.64	0.03	0.36	5.82	5.85	0.88	-0.03
I already knew beforehand how to do graphical programming similar to what was taught	1.64	2.29	0.02	-0.39	2.03	1.88	0.60	0.09
I already knew beforehand all the general programming content	5.26	5.33	0.83	-0.04	5.29	5.28	0.96	-0.01

how much participants now enjoyed programming with graphics ($d = 0.17$) and their desire to continue with graphics-based programming ($d = 0.24$).

Differences between female and male students were minimal, with one exception. The participants' CS1 course relies on traditional programs with text-based console I/O. After the intervention, female participants were more likely to say that they like programming with graphics more than what they have done in CS1 (small-to-medium effect, $d = 0.30$).

Participants across groups largely agreed that "programming with graphics is fun" (average 5.9). This item is not identical to our more generic "programming is fun" item on the pre-test (average 5.4). With that caveat in mind, we note that there is a statistically significant difference between these within-subjects ratings, with a moderate effect size ($d = 0.46$, $p < 0.001$).

10.5   The experimental results need to be discussed

In summary, we found that (1) both graphics-based approaches—compositional and turtle—engaged students and led to high ratings of perceived learning; (2) both groups performed uniformly well on the programming tasks in the post-test, except for a sub-

stantial difference in code tracing; and (3) the groups performed equally on post-test questions on conceptual knowledge. Sections 10.5.1 and 10.5.2 below elaborate on the first two points, respectively. In Section 10.5.3, we discuss the third point in detail as we consider both the factors that may have affected our specific result and, more broadly, the methodological issues that complicate studies such as ours.

10.5.1   Student engagement was high

Both the PyTamaro group and the Turtle group reported high levels of engagement with programming using graphics. They expressed enthusiasm for it, thought it was more likable than traditional non-graphics-based programming, and felt that they had learned programming concepts by engaging with it (Section 10.4.4). Our study thus adds to the body of evidence (Section 2.10) on the value of graphics in introductory programming education. Moreover, PyTamaro's compositional graphics approach appears to yield engagement levels at least on par with the venerable and widely popular turtle approach.

Learner diversity is one of the motivations to introduce graphics in CS1, and pedagogies such as media computation have had positive effects on gender balance [105, 108]. Engaging and retaining diverse students is also an explicit goal for PyTamaro, and two of our results are potentially significant in this light. First, almost half (49 %) of the volunteering participants identified as female, which is substantially higher than the proportion of female students in the CS1 course (ca. 35 %). We had advertised the study as "learning to program graphics in Python," which may have piqued female students' interest in particular. (It is known that there are differences in how female and male students value different domains in programming tasks [173].) Second, female participants especially agreed with the post-survey claim that they prefer programming with graphics to text-based programming (average 4.7 vs. male students' 4.3). This effect was of a small-to-moderate size but did not reach statistical significance ($d = 0.30$, $p = 0.11$). Our experiment was not designed to provide data on long-term impact, but our results do suggest that choosing graphics as a domain for introductory programming may have an immediate impact on stimulating female students' interest.

10.5.2   Differences between groups were scarce, with one exception

Overall, the two experimental groups showed comparable results on the post-test tasks. We start the discussion with the notable exception of the performance difference on the tracing task, which tested hypothesis H2a.

10.5.2.1 The PyTamaro group did better on their tracing task

One of the goals of compositional graphics approaches is to encourage decomposing a problem into independent subproblems so that one may reason about each subproblem in isolation. This is harder to achieve with turtle graphics, as the turtle carries a state that includes its position and heading (Section 3.4). Tracing a turtle program requires one to track the state at each step through a sequence of commands.

While designing the tracing question (Section 10.3.6.3), we considered several aspects in an attempt to ensure a fair comparison. First, both groups used a `square` function they had already practiced during the intervention. Second, the `square` function we gave the Turtle group "behaves nicely": it leaves the turtle facing in the same direction. Third, the PyTamaro code is suboptimal in several ways, compared to the code quality one would typically have with PyTamaro. (Specifically: Each variable is used only once. The `square` function is called four times merely to match the four calls in the Turtle program. Similarly, repetition of the first three lines could have been avoided by reusing c. The side length is not given a meaningful name; the literal `10` is passed directly as an argument. All the variable names are devoid of meaning.)

Our participants performed vastly better on the PyTamaro tracing task than on the Turtle task. This result comes with caveats. For one thing, there was only one tracing question for each participant. For another, the two groups traced different programs. Although we have argued that the two programs are, in a sense, comparable, our reasoning may be called into question. Nevertheless, the very large effect size in favor of PyTamaro (Section 10.4.3.1) suggests that tracing typical turtle graphics code requires more effort and care compared to typical compositional graphics code, even for non-novices. It is important to bear in mind that this finding speaks partially—and perhaps mainly—of the nature of the two programs and their associated learning approaches rather than between-group differences in learning gains; however, that finding, too, is relevant to instructors that employ graphics-based pedagogies.

10.5.2.2 Other differences were largely absent

We found no major differences between the groups on the code-writing task or the code-modifying task (Sections 10.4.3.2 and 10.4.3.3). By and large, both groups performed rather well on these tasks. Despite this result, the possibility certainly remains that the two approaches to programming graphics lead to differences in students' programming skills. However, it may be that observing such differences would require a context where learners practice with a graphics library longer (e.g., over several weeks) and can then demonstrate their abilities on significantly larger exercises. For us, arranging for such an experiment was not feasible in practice; it would have also reduced

the degree of control over the study participants, likely introducing new confounding factors.

Similarly, we observed no substantial differences between the groups' performance on the six post-test questions about expression-related programming concepts (Section 10.4.2). Again, the short length of the intervention may have contributed, but there are many other factors worth considering as well, as we discuss below.

10.5.3 The multiple-choice questions were designed with transfer in mind

Certain programming concepts are prominent in the compositional graphics approach that PyTamaro supports; these include nested function calls and other composite expressions, functions that take various numbers of parameters, and return values, to list a few. In the absence of a validated assessment instrument that focuses on these concepts, we designed an ad hoc one: questions Q1 to Q6 in our post-test (Appendix A.4). We applied this instrument in hopes of showing conceptual transfer from PyTamaro to a non-graphical domain (using turtle graphics as a baseline) but did not find evidence of it.

When designing and applying such an instrument, there are many decisions to be made. Below, we discuss a few of them in order to explain our design, reflect on possible reasons for the lack of observed differences between groups, and, perhaps, to highlight some pitfalls to others engaged in evaluative computing education research.

10.5.3.1 We aimed to stay clear from "Teaching to the Test"

We wanted at least part of the post-test to be identical for both groups. One reason for this was to enable direct comparisons between the groups on the same questions.

Identical questions do not guarantee a fair comparison, however. An inherent problem in evaluating educational innovations is that researchers may unwittingly favor a particular group—especially if they designed the innovation being studied. At an extreme, one group might be taught precisely what the post-test asks. Crichton and Krishnamurthi recently reflected on this while designing interventions to improve a textbook alongside assessments to evaluate the interventions: "If an intervention is too tailored to the specific question being targeted, then learners are likely not forming a robust mental model. We managed teaching-to-the-test by ensuring that interventions did not change the textbook to trivialize the problems under question, e.g., by adding the answer verbatim to the book" [65].

Not providing the exact answers to one group only is a start, but insufficient. If one group's intervention materials closely match the post-test, any success on the test

might be mere memorization. In the words of Perkins and Salomon, "any learning requires a modicum of transfer. To say that learning has occurred means that the person can display that learning later" [206]. In an attempt to capture genuine learning, we designed conceptual questions that were "at a distance" from what the participants experienced during the intervention. We achieved this using a different domain than the graphic one which was used in the teaching materials, staying clear from the risk of "teaching to the test".

10.5.3.2 We studied transfer to isomorphic programs

Within the constraints of a short experiment, one can hardly expect *far transfer* to very dissimilar problems; there is plentiful evidence that far transfer is a lofty goal in general (e.g., [71, 139]). We instead aimed to find *near transfer*, asking the participants to answer questions about programs that are not in the domain of graphics but that are *isomorphic* with what the PyTamaro group was previously taught in that domain.

The diagram in Figure 10.2 illustrates how transfer to an isomorph doesn't happen "directly". Instead, learners are supposed first to acquire an "abstract" understanding and then to apply it in a new context.
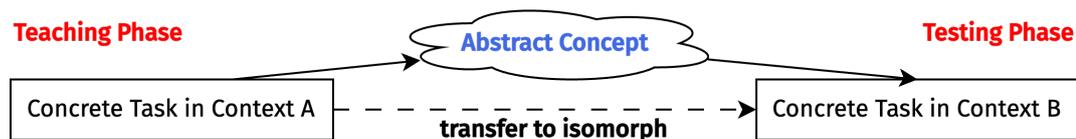
Figure 10.2. Learners are exposed to a concrete task during the teaching phase. During the testing phase, they are assessed on an isomorphic task in a different context. Achieving transfer requires acquiring an understanding of the abstract concept.

We designed the multiple-choice questions with this idea in mind, focusing on the "abstract concepts" that PyTamaro was hypothesized to teach (cf. Table 10.2, that shows the isomorphisms between the PyTamaro intervention and the post-test).

10.5.3.3 Transfer, even to isomorphic tasks, can fail

Contrary to our hypothesis H1, the PyTamaro group, despite having practiced on tasks isomorphic to those in the post-test, did not perform better than the Turtle group. It is well known that transfer is not easy to achieve or to demonstrate through research. The result of this study provides further evidence that programming is no exception in this regard, even when aiming only for near transfer to isomorphic tasks.

One reason why even near transfer often fails is that learners—novices especially—may fixate on the surface characteristics of a task and consequently fail to draw the appropriate connections between tasks and abstract from them. This challenge has been noted, among others, by Perkins and Salomon, who wrote: "Subjects usually do not recognize the connection between one isomorph and the other and hence do not carry over strategies they have acquired while working with one to the other" [206]. Given that our post-test questions were in a different domain, albeit isomorphically so, it is debatable whether we truly tested for near or far transfer.

Transfer is likelier if instruction highlights the relationships between concrete tasks and abstract concepts. For instance, Reed et al. [220] discovered that many people would not transfer from the "Jealous Husbands" problem to the similar "Missionary–Cannibal" without explicit instruction. Our teaching interventions did not consistently and explicitly highlight opportunities for transfer, which may have affected our results. An improved intervention could embrace more deeply the idea of "semantic waves" [177, 66] and guide learners from the abstract to the concrete and then back to the abstract, highlighting the relationships between the two levels of abstraction.

10.6   There are threats to the validity of our study

10.6.1   Students' prior knowledge affects the results

As discussed in Section 10.3.1, the lack of a pre-test is justifiable given the randomization and the large sample, and even has some advantages. Nevertheless, this decision does preclude us from computing "learning gains", and there remains the issue of how prior knowledge affected our participants' performance.

We targeted students who did not have much experience, but not all participants were novices. However, even with only novices included in the analysis, the differences between the two groups were minor. One plausible and perhaps likely explanation for this is that the participants' performance was sufficiently influenced by what they had previously learned in CS1 that any effect of either group's short intervention on the post-test is negligible in comparison.

The participants' CS1 course adopts a "typical imperative" view of programming (loops, etc.) that is more closely attuned to the turtle approach than to PyTamaro's compositional graphics approach, as the latter instead emphasizes the compositional power of expressions that comes with "functional" programming (nested calls, etc.). We speculate that this might have introduced a slight bias in favor of the Turtle group. Similarly, the Turtle group could have been advantaged by prior experience in "computational thinking" or "problem solving" activities that do not involve programming

as such but that match the spirit of turtle graphics (e.g., [47]); such activities are not uncommon in secondary education.

10.6.2 The short study duration limits what can be observed

The experiment was designed to allow participants to work through a programming tutorial with graphics for one hour. As noted in the discussion, the short duration enabled controlling certain variables (e.g., avoiding participants discussing the lessons' content between groups, having the exact teaching materials available for reproducibility), but also significantly limited opportunity to observe effects that are more visible in the medium and long term.

10.6.3 There are threats related to data collection and the instrument

Engagement and prior experience were self-reported and potentially subject to self-report bias.

The items for our surveys and post-test were constructed ad hoc and not validated. Several post-test items were in the multiple-choice format, which can be problematic if the options are not paired with explanations [50]. We mitigated this by (1) including an "I don't know" option, (2) writing longer explanations as part of the options that participants had to pick, (3) administering the instrument beforehand to four students (extraneous to the experiment) as a small-scale pilot study, which helped to refine and clarify the questions.

Nevertheless, our items may have been too "guessable". Although no specific comment was made by the participants during the experiment sessions, we do not possess data that would confirm the absence of misinterpretations of the questions' statements or options.

The items on tracing, modifying, and writing code were designed to be "comparable" between groups but not identical.

10.6.4 Generalization is limited

All our participants come from a single university course, albeit one with students from a wide variety of engineering majors; we cannot comment on how our results might generalize to other groups. Moreover, our subjects were volunteers who received a small compensation for participating, which may have introduced a selection bias.

10.6.5   Students may have some response biases

We solicited answers from participants both in the pre- and the post-survey. Participants did not engage with a "teacher", but may still have skewed their answers due to the social-desirability bias. In particular, the pre-survey asked participants to voluntarily disclose their gender, which is known to possibly affect their behavior (for example with respect to the stereotype threat).

10.6.6   We have an authorship bias as we are PyTamaro's authors

We preemptively compensated for an authorship bias when designing the experiment, disadvantaging our own library in several ways. (1) We placed PyTamaro in an imperative-style CS1 that does not play to PyTamaro's strengths. (2) We used animated (thus possibly slightly more engaging) visuals only in the Turtle materials, as we thought they were needed to provide a high-quality explanation of the turtle's state changes over time. (3) We introduced to both groups the idea of a loop to accumulate a value, despite it not normally being covered in Turtle pedagogy, as we felt it necessary to fairly prepare all participants for one question on the post-test. (4) We wrote the tracing question in a style far from optimal for PyTamaro programs (Sections 10.3.6.3 and 10.5.2.1). (5) We took care of setting up the turtle's drawing environment (e.g., providing a large enough canvas).

Ultimately, we cannot rule out an authorship bias. For transparency, the complete materials used in the study are available as appendices.

10.7   To conclude, we did not find evidence of better transfer with PyTamaro

This chapter presented a randomized, controlled experiment on the use of graphics in teaching programming to beginners. We found that both a compositional graphics approach enacted using the PyTamaro library and a more traditional turtle graphics approach engaged student programmers; female students might find such approaches particularly engaging. We did not find evidence of better transfer from a short PyTamaro session to a post-test on isomorphic tasks outside the graphics domain, compared to the Turtle session which did not feature those tasks. Overall, there were few differences between the two experimental groups. As an exception to that trend, beginners appear to trace compositional graphics code more accurately than "comparable" turtle graphics code.

We have outlined several alternative—or complementary—explanations for our

findings. Further research is needed to test our speculations as well as the generalizability of our results. Future research should look into interventions longer in duration than what we were able to investigate here.

Chapter 11

We Conducted a Case Study With High School Teachers

This chapter describes a multiple-case study conducted with five Swiss teachers who adopted PyTamaro to teach the programming part of the mandatory informatics course in high schools.

11.1 Swiss teachers adopted PyTamaro in different contexts

PyTamaro has been used by its authors as part of multiple teacher training programs throughout Switzerland, both as a way to teach programming for those teachers without prior knowledge of programming and to strengthen the pedagogical content knowledge for everyone.

As the authors of PyTamaro, we were inevitably partial to its adoption in classrooms after the training programs concluded. Nonetheless, each teacher made the final decision freely. Some eventually opted to use it in their curricular activities. To understand the context in which this decision has been made, we need to briefly illustrate the relevant bits of the educational system in Switzerland.

Switzerland consists of 26 cantons, which are member states of the Swiss Confederation. Cantons have significant autonomy in organizing education. Within some general constraints imposed at the federal level, each canton has room to decide when and how much each subject is taught. Consequently, there is considerable variation in both the school years (also known internationally as "grades") during which informatics is taught, and the number of hours devoted to the subject. Each canton also draws up its own curriculum, which contains guidelines for the specific topics expected to be

covered in each subject. The level of detail at which these topics are specified and the level of prescriptiveness again varies from canton to canton.

Each canton has several high schools, and each school has a certain degree of autonomy to develop its own education. Typically, each school defines for each subject its own school-level curriculum, which is based on the cantonal-level curriculum. For the recently introduced informatics course, this process is mostly still ongoing, as the cantonal curricula are also expected to be revised after being tested with students during the first years.

Finally, each individual high school teacher has a significant degree of autonomy in deciding how to implement the school-level curriculum.

This broad autonomy helps explain why five teachers, working in five different schools across multiple cantons, have diverse experiences, interests, needs, and goals. From afar, all five teachers "adopted PyTamaro to teach programming". But a closer look reveals rich and meaningful differences in how this adoption actually took place. The case study we describe here investigates this diversity.

11.2   The pedagogy and the library are interconnected

Developing the teaching materials for the experiment described in Chapter 10 revealed an intricate interplay between an educational library and the pedagogy built around it.

The design of a library *nudges* the users towards a certain style of programming. For example, adopting PyTamaro means favoring expressions over statements, avoiding mutability as well as sophisticated features of the programming language, and defining abstractions early (Chapter 5).

However, there is still a lot of freedom in the pedagogical design of an introductory programming course. At one extreme, a teacher might embrace the library completely and, say, never introduce any language feature that is outside the small set used by PyTamaro. At the other extreme, a teacher could "fight against" the philosophy of the library: for example, the definition of functions could be significantly delayed on the grounds that it is challenging for novices [130].

11.3   Prior work investigated when and how educators adopt innovations

A recent international working group reviewed how teaching innovations get adopted, with a specific focus on Computer Science [253]. They characterized innovations as being *instructional technology*, such as hardware or software used in teaching, *curricu-*

lar innovations, which refers to changes in the actual course content (including which topics are taught and in which sequence), and *pedagogical innovations*, which include new instructional strategies.

Pedagogical innovations may be the easiest to adopt, as teachers normally have a high degree of autonomy over the pedagogy to be used, whereas external constraints may dictate the topics to be covered in the curriculum. Even innovations of this kind are not easy to disseminate, and their propagation does not necessarily indicate that the innovation is enacted as intended. Borrego et al. [29] studied how engineering faculty implemented 11 "innovative" instructional strategies, such as think-pair-share, problem-based learning, or peer instruction. Strategies were characterized by a number of components to measure the "fidelity of implementation", in an attempt to capture how closely classroom practice reflects the original strategy. Overall, there was significant variability depending on the specific strategies: between 11 % and 80 % of instructors spent time on all the required components for a given strategy.

Curricular innovations do not have it easier. Levy and Ben-Ari [162] noted that research-based pedagogical tools have a hard time being adopted by teachers in actual classrooms, despite the hopes of the authors of these software tools. To better understand the reasons behind this common failure, they conducted a phenomenographic study with high school teachers who received training on how to use Jeliot, a tool they developed to visualize Java programs. The study revealed that teachers experienced using the tool in radically different ways: from using it frequently and integrating it deeply within their pedagogy, to a sporadic use that may even conflict with the rest of the materials. When developing a pedagogical innovation, the authors recommend paying attention to its integration into the curriculum, as reflected in the materials, and to ensure that the teacher remains "central" in the class.

In a landscape where changes are rare, there are also some success stories. In 2005, a two-year college in the USA successfully adopted the "Media Computation" curriculum, which was developed at a university [255], leading to improved student retention. Ni et al. [190] interviewed eight instructors who attended workshops that also featured the "Media Computation" and adopted or expressed interest in adopting it. External factors, such as the limited freedom to change the content of a course, hindered the adoption, but one instructor also reported that an initial experiment with the innovative curriculum was successful in terms of student engagement Ni et al. [190].

Fincher et al. [87] collected 99 stories of computing educators who decided at some point to modify their teaching practices, for example by adopting an innovative tool. Nearly all stories reported a change that occurred in a local context, without drawing from outside sources, without conducting an active search for new practices or materials. Personal interactions with peers were a driving force. Fincher et al. [87] called on educational developers and researchers to overcome a naïve model of how

teachers adopt tools, hoping that educators conduct a systematic search for innovations.

11.4   We conducted a case study on how teachers adopt PyTamaro

Using the taxonomy introduced above, the PyTamaro approach can be best described as a curricular innovation. While the approach leverages some software tools (the Python library itself and the web platform with its extensions), these are only in service of—and tightly integrated with—the curricular approach.

11.4.1   Five teachers represent our five cases

Over the last two years, we have been directly working with a number of high school teachers who decided to adopt PyTamaro as part of the mandatory informatics course. This kind of direct connection gave us privileged access to teachers who were willing to integrate an innovation into their teaching. As Section 11.3 discussed, this is quite rare and presented us with a unique opportunity.

We identified six teachers, each of whom would serve as a case in our study. The selection of the cases aimed to ensure representation of different contexts. We knew that each teacher was working at a different school, some were using the PyTamaro Web platform and others were not, some were exploiting the unplugged TamaroCards approach to introduce programming and others were not.

We asked teachers to share with us their teaching materials in the original version, regardless of format: slides, notes, handouts, exams, curricula and activities on the PyTamaro Web platform were all welcome. We also invited teachers to an interview of roughly 90 minutes. To compensate their efforts, in addition to the insights on their materials they could gain in the interview, we promised teachers a monetary reward equivalent to approximately two hours of their salary. This study was approved by the University Ethics Committee.

Of the six teachers we contacted, five accepted our invitation and are part of this case study.

11.4.2   We investigated why teachers adopt PyTamaro and how they translate the approach in their teaching materials

Our investigation was driven by three research questions:

RQ1 What are the factors driving teachers to adopt PyTamaro to teach programming?

RQ2 How are teachers translating the principles embodied by the PyTamaro approach into their teaching?

   – How are problem decomposition and abstraction (e.g., definition of functions, use of the Toolbox) taught?

   – How is repetition (e.g., loops) introduced?

   – How are language features (e.g., nested expressions, lists, methods) used?

   – How is the unplugged approach with TamaroCards used, if at all?

   – What other noteworthy aspects stand out in the teaching materials?

RQ3 What is the experience of students with PyTamaro (e.g., need of learning function definitions early, restrictions in the drawable graphics)?

11.4.3 We collected two different sources of evidence

Yin [284] describes six sources of evidence used in case studies: documentation, archival records, interviews, direct observations, participant-observation, and physical artifacts.

As part of this case study, we decided to collect two of these: documentation and interviews.

We did not collect evidence through direct observations or by becoming an observing participant. Watching the teachers and the students' reactions in real time would have provided unfiltered observations. On the other hand, logistical constraints made it unfeasible to observe an entire year of teaching across multiple schools. On top of that, reflexivity also poses a problem: because students could have behaved differently just knowing that someone was observing and capturing their behavior. Finally, being an observing participant in this context would have required us to become the teacher of those students. This has all the issues described above, being even more time intensive, but would have also fundamentally manipulated the results, given that we would be in control of the various aspects of teaching, including the creation of the materials and their delivery to students.

We now discuss in detail the strengths and weaknesses of the two sources of evidence we decided to collect, selecting the properties described by Yin [284] that are relevant to our specific scenario.

11.4.3.1 Teaching materials serve as documentation

We asked each teacher to share their teaching materials unaltered, exactly as they had been used in class. Yin [284] describes this source of evidence as "documentation",

with four key strengths. First, teaching materials are *stable*: we can review them multiple times and they will contain exactly the same information. (This may seem a trivial property, but it is not a given with other sources of evidence.) Second, documentation is *unobtrusive*: it has not been created for the specific purpose of the case study and is therefore not ad hoc. Third, teaching materials are highly *specific*: they contain the exact wording and code fragments that students have been exposed to, down to the level of individual tokens, enabling a fine-grained analysis that would be impossible when only discussing code at a high level. Fourth, the teaching materials we collect are *broad*, because they cover one or two entire school years.

Our privileged relationship with teachers, alongside whom we worked for a long time, allowed us to eschew two main weaknesses of this source of evidence. Documentation can be difficult to obtain, and access may be deliberately withheld. We believe that these two factors may be key reasons behind the lack of this in-depth analysis of teaching materials actually used by teachers in computing education research.

At the same time, we have to acknowledge two inherent biases for this source of evidence. There is a *reporting* and *selection bias*, which reflects possible biases in teachers who voluntarily or involuntarily did not share the entirety of their materials. For our study, this may manifest in two ways. First, teachers may not be sharing an entire type of materials, such as exams, on grounds of their confidentiality. Second, they may be excluding specific parts of the materials, such as a code example that did not work well with their students, or a slide for which they realized later on contains a mistake that they are unwilling to make public.

11.4.3.2　Individual interviews are targeted and insightful, but suffer from biases

Yin [284] describes two key properties of interviews, which we exploited in our study. First, interviews are *targeted*, because the specific questions asked can focus on the research questions that are of interest. We were able to ask specific questions on topics of interest, such as how teachers dealt with function definitions, as an operationalization of the big ideas of abstraction and problem decomposition. Second, interviews can be *insightful*: a rich source of explanations and personal views. For example, interviews allowed us to investigate the reasons that led a teacher to create a slide in a certain way, or to explain a concept in a specific moment of the school year. Personal views included gathering the teachers' perception of their students on how they reacted to a specific didactic moment, what seemed to elicit their curiosity, and what remained a struggle.

On the flipside, interviews are clearly subject to a number of important biases. First, there are *question* and *response* biases. Questions may be poorly articulated and not well understood by the teacher, or they may be partially "leading questions" to

prompt or encourage the answer that the study authors want to hear. Second, and in a certain sense dual of the previous point, there may be a *reflexivity bias*. The teacher being interviewed may simply default to saying whatever the interviewer wants to hear. This bias is particularly relevant for our study, in which the interviewer is the same person who developed the teaching approach being studied. An overall friendly relationship between the interviewee and the interviewer may exacerbate this problem, as the teacher may not want to compromise the generally positive attitude by saying something that they may perceive as not liked. Third, some of the teachers' responses may suffer from *poor recall* and contain inaccuracies. This problem is also relevant for our study, which straddles three different moments in time: when a teacher created their materials, when the lessons were actually held with students, and when the interview was conducted. The time in between these moments varied for each teacher, but could generally be described as ranging between a couple of months and a year, a period long enough for a person to confuse the memories.

11.4.3.3 Our interviews also included a small assessment part

None of the teachers who participated in the study have an extensive background in Computer Science, such as holding a Master's degree in Computer Science or Informatics. All the teachers attended courses in the national training program to become qualified to teach informatics. However, their backgrounds and interests differ, and it should thus not be surprising that their level of programming maturity varies as well.

Therefore, after a "warm up" section dedicated to the teacher's background, we included in each interview a small assessment component. We asked teachers to give feedback on two small PyTamaro-based programs, pretending that they were programs written by their students. This component was designed to probe their understanding of immutability, abstraction, and decomposition; core skills connected with the principles embodied by PyTamaro. A first program redefines a name multiple times, to probe the understanding of immutability. A second program contains identical or nearly identical expressions that are not abstracted into constants or parameterized functions, to probe the understanding of decomposition and abstraction.

We include an analysis of the teachers' responses to this assessment part in the description of each case. This allows readers to better put into perspective all the other answers given by the teacher in the interview, as well as their teaching materials.

11.4.4 Multiple sources of evidence enable triangulation

Yin warns against the risk of using one single source of evidence to analyze a case. While there have been case studies conducted only with one source of evidence, it is

understood that the first principle to increase the reliability of the findings is to use multiple sources of evidence [284]. For our specific study, we wanted to avoid relying only on the interviews or the teaching materials. We already discussed in Section 11.4.3 the limitations of each specific source of evidence.

Using two or more sources of evidence achieves what is known as *triangulation*. The prefix *tri* suggests that triangulation is often related to three different perspectives. It can be argued that this could be better called a form of *cross-validation*, or cross-checking (this terminology is more standard, for example, in the machine learning community). With this distinction in mind, we will keep using triangulation to follow the tradition of case study research [95].

Patton [200] describes four types of triangulation. An evaluation study can use multiple data sources, multiple evaluators, multiple perspectives on one dataset, and multiple methods. When we refer to triangulation in our study, we are using the first of these four types. We analyze and integrate our two data sources, teaching materials and interviews to achieve *data triangulation*.

An important limitation that needs to be pointed out upfront is that not every aspect is covered by both sources of evidence. For example, when a teacher describes what the effects of a specific part of their materials on their classes were, we can only know what happened from their account during the interview, without having a way to confirm it independently.

11.4.5   We followed a protocol for the interviews

Yin [284] recommends maintaining a case study protocol to increase the reliability of the case study. A protocol should identify the main research questions, which we stated in Section 11.4.2, and describe the high-level questions that guide the interviewer in following the line of inquiry during the interviews. Note that these questions do not always match the exact questions as they are verbalized during the interview. "Case study interviews will resemble guided conversations rather than structured queries. Although you will be pursuing a consistent line of inquiry, your actual stream of questions in a case study is likely to be fluid rather than rigid" [284]. Some researchers classify interviews into "structured interviews", when a protocol or set of questions is rigidly followed, and "unstructured interviews", when no set of questions is defined in advance [275]. Under this perspective, our methodology could be best described as following "semi-structured interviews" [69]. The high-level set of questions was defined in advance. Some of those questions can be verbalized as is during the interviews, while others lead to more specific, contextualized questions adapted during the interview to the situation at hand.

11.4.5.1   Some questions focused on the teacher

- How long have you been teaching in general? How long have you been teach-
  ing programming? How many school years with PyTamaro? Which courses
  (mandatory, elective)? Which students (grades, age)?

- Where and how did you learn programming? Did you attend any regular or ad
  hoc study programs?

- Give feedback on this PyTamaro program as if it were a solution produced by
  one of your students. Why do you think that aspect is problematic or not?

  - Listing 23 is a program that draws the International Red Cross emblem of
    Figure 5.1.

  - Listing 24 is a program that draws a pair of green eyes, as shown in Fig-
    ure 5.5.

```
background = rectangle(320, 320, white)
arm = rectangle(200, 60, red)
arm = overlay(rotate(90, arm), arm)
flag = overlay(arm, background)
show_graphic(flag)
```

Listing 23. Example program using PyTamaro that reassigns to a variable.

```
diameter = 200
small_black = ellipse(diameter / 4, diameter / 4, black)
big_green = ellipse(diameter / 2, diameter / 2, green)
eyes = beside(overlay(small_black, big_green),
↪  overlay(small_black, big_green))
show_graphic(eyes)
```

Listing 24. Example program using PyTamaro with opportunities for abstraction.

11.4.5.2   Other questions investigated the choice of graphics as a domain and Py-
Tamaro

- Why did you adopt PyTamaro over other graphical approaches?

- Do you use PyTamaro "alone", in combination with other approaches for graphics, or in combination with other domains?

- What are the reasons behind your selection of a certain domain (e.g., graphics, math)?

- What are the challenges in introducing a teaching innovation like PyTamaro? (*Interviewer remark, not mentioned in the interview: challenges could include colleagues not adopting it, textbooks already in use, prior experience of colleagues, constraints from the local curriculum.*)

11.4.5.3 We established a template for questions about teaching materials

These questions were adapted for each teacher, based on their teaching materials provided beforehand.

- How do your current teaching materials with PyTamaro differ from older teaching materials with another approach or in another domain? Are there additional topics you are now covering or emphasizing more, or topics you used to cover that are now missing or less emphasized?

- Do you introduce programming concepts with PyTamaro, or do you use PyTamaro as "additional practice" for concepts already introduced in a "traditional way"?

- Why do you explain how to define functions at that point in the materials? Why not sooner/later?

- Why do you introduce a certain concept before another?

- Which pedagogical techniques do you use (e.g., group work, think-pair-share, pair programming, homework, flipped classroom)?

- Do you use unplugged activities? Why or why not? If you use them, do you abandon them at some point?

- Do you use the PyTamaro Web platform to deliver your courses? Why or why not? If you use it, how independent are the students when working through the activities?

After analyzing the teaching materials and before the interview, a set of dedicated questions was prepared for each teacher, based on this template. Appendix B reports the dedicated questions for each case.

11.4.5.4 Some questions discussed the students' experience

- Do you feel your students like or dislike working with PyTamaro? If you used another graphical approach earlier, how does student engagement compare?

- How well can students deal with defining their own functions rather early in the curriculum?

- Do your students tend to write lots of small functions or fewer, bigger functions?

- Do your students feel constrained in what they can do with PyTamaro? Do they feel they have enough choice (e.g., for a possible final project)?

11.4.6 The study suffers from a clear authorship bias, which we tried to mitigate

It is impossible to shy away from the clear authorship bias affecting this case study. The writer of this dissertation is also the author of PyTamaro, who is also the same person who alone conducted the interviews and analyzed the teaching materials for this case study. In the interest of demonstrating the thesis of this dissertation, he is inevitably partial to portraying PyTamaro in a positive light.

This bias is particularly significant for the case study, which relied on interviews. Section 11.4.3.2 already pointed out how interviews as a data source in case studies suffer from a number of biases. The question bias and the reflexivity bias are very significant in our setting.

We acknowledge the presence of these biases in our research. Nonetheless, we tried to compensate for them with several mitigations. First, as recommended by Yin [284], we developed beforehand a protocol to be followed for the interviews (Section 11.4.5). Second, given that some of our questions were dependent on the specific materials provided by each teacher, we transparently provide a case-specific addendum to the protocol in Appendix B. Third, when reporting answers from teachers that appear to repeat back the very words used by the interviewer, we include the exact question as pronounced during the interview in the quoted material. Fourth, despite the infeasibility to publish the entire materials anonymously, we present each case with numerous quotes and figures extracted directly from the interview and the teaching materials.

Ultimately, none of these countermeasures eliminate the authorship bias, but together they allow readers to form a more complete and independent opinion on our research.

11.4.7   We analyzed each case, and across the cases

One compelling characteristic of a case study is the possibility of analyzing a case in depth. Many case studies describe one single case, which is analyzed and discussed in isolation.

In our scenario, we were in contact with more than one teacher, and we knew that the context was very different in each case, and that the context could have a huge influence on teaching decisions. We therefore decided to make our case study a multiple-case study, in which each of the five teachers is a "case". A multiple-case study is not a qualitative study with a small number of participants: we will first analyze and describe each case in isolation over the next five sections. Only afterward will we analyze across cases and present the aggregate findings in a dedicated section. This strategy for reporting the results of a case study follows the second compositional format ("Multiple-case study") recommended by Yin [284, Ch. 6].

We did not follow an "algorithmic" process for the analysis, as it would have been at risk of missing deeper knowledge. As Kvale writes in his influential book on analyzing interviews: "There are no standard methods, no *via regia*, to arrive at essential meanings and deeper implications of what is said in an interview. The demand for a method may involve an emphasis on techniques and reliability, and a de-emphasis on knowledge and validity. The search for techniques of analysis may be a quest for a 'technological fix' to the researcher's task of analyzing and constructing meaning." [157, p. 180]. Demanding that the analysis should lead to the same results no matter the researcher "may lead to a tyranny by the lowest possible denominator: that an interpretation is only reliable when it can be followed by everyone, a criterion that could lead to a trivialization of the interpretations" [157, p. 181] (a positivist view of truth, cf. Section 1.7). Kvale argues that the interpretation of the meaning should go "beyond method and draw upon the craftsmanship of the researcher, on his or her knowledge and interpretative skills" (ibid.). To corroborate this position, Kvale brings an example of analyzing the interview with a chess player, in which "the researcher's knowledge of chess at a higher level than that of the interviewees is a precondition for seeing the solutions they did not see".

Nevertheless, we can provide an account of the methodology we followed. The entire process was conducted by the author of this dissertation.

We collected teaching materials in early 2025. Before the interview, the teaching materials were reviewed to prepare the additional dedicated questions reported in Appendix B. The interviews with the teachers took place between February and April 2025 and were recorded and automatically transcribed.

For the individual analysis of each case, we relied on the third general strategy offered by Yin: "developing a case description" [284, Ch. 5], where each case is richly

described in an attempt to present why and how a certain phenomenon occurred. We used the automatic transcription as a guide, re-watching all the relevant segments of the recording to fix transcription issues and capture additional subtleties (e.g., programming-related terminology). We segmented the interview into coherent parts, which will be presented in individual sections below. The order of the sections in this dissertation does not match that of the interview, as it was rearranged for presentation clarity. In addition to quoted passages from the interviews, the sections occasionally feature excerpts from the teaching materials that illustrate the theme being discussed, anchoring it to the actual teaching materials used in class. We grouped the sections into four high-level blocks that are kept uniform across the cases: the first block introduces the teacher's context, and the next three focus in turn on each of our three research questions (Section 11.4.2). We made an effort to separate the analysis of the teaching materials (RQ2) from the experience with the students (RQ3), but we only partially succeeded, as teachers frequently interleaved comments about both aspects in a single answer.

For the cross-case analysis, we used both the first and the second specific techniques suggested by Yin: "pattern matching" and "explanation building" [284, Ch. 5]. With pattern matching, common themes across subjects are brought together (e.g., "teachers struggle with explaining variables" or "teachers are able to introduce function definitions early"). This is also related to "thematic analysis" [30], a methodology that has been applied in computing education research. With explanation building, we can understand the reasons behind a certain phenomenon (e.g., "teachers managed to create materials with PyTamaro only because they had dedicated time in a training program"). Throughout, we made an effort to attend to all the data we collected, without disregarding parts that might conflict with our beliefs.

The order of presentation of the cases is unimportant. In an effort to preserve the anonymity of the individual teachers to the extent possible in a case study, their names have been replaced with pseudonyms. The illustrative quotes have been left unaltered as much as possible and only minimally redacted to clarify the context, fix certain grammatical mistakes of non-native English speakers, and protect the anonymity of third parties. Omissions for brevity are indicated with ellipses.

Readers in a hurry may want to jump directly to Section 11.10, which presents and discusses cross-case results. However, we are firmly convinced of the value of the rich description of the individual cases of the teachers, which is offered here at a depth that we are not aware has been published before in computing education research.

11.5   The case of Ada

11.5.1   This is Ada's context

11.5.1.1   She has modest programming experience

Ada started teaching approximately 15 years ago. She has been primarily a teacher for German and English, but has occasionally also taught courses that could broadly be categorized as Information and Communications Technology (ICT). She briefly taught a course closer to what she is teaching now when Switzerland offered Informatics at high school level in the 1990s, but cannot recollect with precision the programming language she used. In 2020 she started studying Computer Science in the national retraining program for Swiss teachers.

She cannot pinpoint an exact moment when she learned to program. Some rudimentary teaching in C and Python gave her a first exposure to programming. She started to learn programming properly in Spring 2020, when she wrote her first programs in Java for the two programming courses in the retraining program. Ada actually considered those courses as too advanced and thus not ideal learning experiences for her as a beginner. She then mostly self-taught Python in the next years.

11.5.1.2   She has two colleagues with extensive experience

In her school, there are three informatics teachers. Two of them are dedicated full-time to computer science and have more than a decade of experience teaching the subject. One also has a long industry experience: Ada considers them "super-extra experts".

Ada taught programming for the first time together with a more experienced colleague. She stood on the sidelines and watched them teach. She was effectively a support role and did not actively teach. Programming was taught with Python, without any graphics library. As an environment, they were using Jupyter notebooks with some online platforms that support notebooks or Visual Studio Code. For a while, she also taught with another colleague who was using the same environments, possibly with turtle graphics.

Ada's two colleagues are still following their own approach, with one of the two now trying an approach to teach how to control hardware with simple sensors and actuators.

11.5.1.3   She mainly teaches in the 10th grade

In Ada's high school, the mandatory informatics course is taught in the 9th and 10th grade. There is the possibility to have elective courses later on. At the end of the 9th

grade, students experience a brief ungraded introduction to programming (in which one of the teachers is using turtle graphics) for roughly half of the lessons in the second of the two terms.

In the 10th grade, every week one lesson is held with the full class, while the other lesson is held only with half of the students (and thus repeated twice). This allows students to ask more questions.

In the last term (second term of the second year), teachers propose projects to their students. Each teacher has the freedom to select what kind of project their students will do. For the final project in the second term of the 10th grade, Ada asks her students to create an animation with PyTamaro.

Given that after the 10th grade students only have computer science as an elective course, Ada does not face the problem that some other teachers have when students are remixed into different classes, potentially ending up with a different teacher for the second part of their mandatory course.

11.5.1.4 She gave sensible feedback to two PyTamaro programs

First program Ada begins by noticing in Listing 23 that she also uses this kind of question in her exams. She also notices that there are several extra spaces surrounding certain tokens, similar to how her students write code sometimes. (That was actually an artifact of the chat system used in the interview.) She reports that in practice she does not complain about that with students, unless those spaces are added in the middle of a variable, violating the rules for a valid identifier.

Ada claims she would give a full grade to the program as it produces the correct output, although she admits that the style is not ideal. She believes that this simple graphic is meant for beginners, and line 3 is possibly too complex. She does not mention any issue with the redefinition of `arm`, however:

> **Ada:** Line 3 `arm = overlay...`, this is not easy to read for beginners [...] because of the function call in a function call.
> **Interviewer:** Ah, because they are nested function calls. So you would break [them] into 2 separate instructions, 2 lines...
> **Ada:** I would add a line between 2 and 3 and say `arm2` or `arm_vertical = rotate(90, arm)` and then use `arm_horizontal`, `arm_vertical` in line 3.

Nonetheless, she would pick a sensible and fresh name for the graphic:

> **Interviewer:** And what would you call this composition, the overlaying of the two arms? [...]

Ada: I would call it `cross`. Because it is not yet the flag.

Second program   Ada noticed that line 4 of Listing 24 was particularly long, something that would not fit on a handwritten page (it was actually 79 characters).

She noticed that the code does not have an **import** statement.

As for the previous example, she would give a full mark. She said she might deduct some points from line 4 because everything is on one line, instead of being nicely indented over four lines to make it more readable. While expressing these thoughts, she realized a bigger problem:

> **Ada:** Oh, no, wait! No, no. I would definitely give deductions for style here because it works, but the two eyes with the two concentric circles there should be stored/assigned to one separate variable, I guess `eye`. And then [...] you can put it in one line.
> **Interviewer:** I see. So that would avoid the repetition of `overlay` and it would also make the line shorter...
> **Ada:** It makes it much easier to read because you give it variable names, and I insist on them choosing easy-to-understand variable names. I would even say... `small_black`, what [is that]? That's a pupil. [...]

In summary, Ada acknowledges that there are some issues that she considers regarding "style":

> **Ada:** So, it works. But styling and usage of variables is suboptimal.

11.5.2   On the choice of adopting PyTamaro

11.5.2.1   For her, the training program was essential to develop materials

Ada chose to create PyTamaro-based curricula as part of her retraining program final thesis project. For her, developing her own materials was also a way to learn programming:

> **Ada:** I already use [my colleagues'] material a lot. So at some point I see them basically writing their own books, and I just copy paste. And I also see how they understand what they're doing because they created, and I didn't. And I think that you understand watching how things work or don't work much better if you create something yourself.

But she admits that it would not have been feasible outside this context:

> **Ada:** It's a lot of work. I would not have done that if it had not been for my final paper.

11.5.3 On the teaching materials

11.5.3.1 Here is an overview

Ada created several curricula to cover the entire 10th grade. Two curricula are one a variation of another for an introduction to programming (the second one is a revised version of the first, adjusted to account for slightly different periods of teaching and to include some lessons about what worked and what did not with the first one). She then continues with a curriculum on "functions", one on "repetition and change" (i.e., mutation), and one on the two PyTamaro functions' `pin` and `compose`. Finally, a curriculum on "animations" serves as the basis for the final project.

Some students enter the 10th grade with minimal exposure to programming from the 9th grade, which partially fades over the summer. In the last school year, this first exposure was not with PyTamaro, as the course was taught by a different colleague. The introductory curriculum is therefore really meant for absolute beginners.

Ada is using the PyTamaro Web as the main development environment. This is helpful to deal with the heterogeneous situation caused by the "bring your own device" policy at her school, where students use a mix of laptops and tablets.

Ada gives flexibility to her students: some choose to work in pairs, others prefer to work individually.

11.5.3.2 Function definition is introduced with fading examples

To introduce function definitions to students, Ada starts by showing them a fully implemented function definition in Figure 11.1a paired with some example calls with different arguments, so that students just need to execute the provided code and observe what happens.

Over two activities, she then gradually removes parts of the implemented body as in Figure 11.1b and Figure 11.1c. Ada believes this approach can help with a nontrivial concept such as defining your own function.

> **Interviewer:** Some people argue that defining functions is 'super mega hard'. I wonder, what is your experience after going a little bit through this [curriculum of functions]?
>
> **Ada:** I don't think that defining functions is 'super mega hard'. But the whole concept is mind blowing and the whole bits of things they have to understand about it is mind blowing. Because they've only learned to assign fixed values to variables and then they cannot change those anymore. Some still believe they cannot assign new values to variables. They have to give everything a new name. And now you give them a

```
 7    def viertelkreis(radius: float, farbe: Color) -> Graphic:
 8        return circular_sector(radius, 90, farbe)
```

(a) A function definition, fully implemented.

```
20    def auge(radius: float, farbe: Color) -> Graphic:
21        return beside(
22            rotate(90, ...), # TODO
23            rotate(-90, ...) # TODO
24        )
```

(b) A function definition with a partially implemented body.

```
13    def halbkreis(radius: float, farbe: Color) -> Graphic:
14        return ... # TODO
```

(c) A function definition with a body entirely to be implemented.

Figure 11.1. Three-step fading a function definition.

variable name which is actually a function name and then you give it parameters. 'What is a parameter? What is the difference between a parameter and an argument? Why is it there? Why do you have to give something back? Why can you not access a variable that is in a function from outside a function?' It's just so many things that work and don't work and that I don't want to explain because it's too much, but then they don't understand it. So yes, this fading, as you call it... this works. I still get students that don't understand that they have to now figure things out themselves.

A more theoretical explanation of function definition comes after a bit in a dedicated activity dubbed "Theory". As part of this explanation, shown in Figure 11.2, Ada nicely highlights a function definition, the corresponding entry in the documentation bar that is automatically generated, and an example function call.

Figure 11.2. A figure created by Ada to help explain function definitions and function calls.

She reports struggling in deciding where to best place this material within the curriculum dedicated to functions:

> **Ada:** If you look through the curriculum as it is now, there is the theory
> block in the middle. And I've had that at the very beginning. I've had it
> at the very end... Not happy with any of it yet, because at the beginning
> it's just too much theory, they don't understand any of the context yet. At
> the end it comes way too late. And in the middle, they still don't want to
> hear the theory, they want to do the functions. They don't want to listen
> to me, they want to work. So I have to tell them off for working instead
> of listening to me, which I find the worst thing ever. So what I've now
> done is I have them read through the theory, which I don't think works
> either. That only works with the 'studious' students. That doesn't work
> with the students that are 'efficient'.

An alternative strategy could be to motivate the necessity of introducing functions to abstract over similar pieces of code with the "game of similarities and differences" (cf. Section 5.5). Although it would not solve the challenge of presenting all the concrete ingredients necessary to define a function, it could help students to understand the high-level concept and its purpose.

This different take at introducing function definitions can also be explained with Ada's desire to create her very own materials, without borrowing excessively from the existing ones.

The individual differences among students are still a dominating factor when it comes to defining parameterized functions. Confronted with a function that receives a number and prints all the number from 1 up to that number included, Ada reports that roughly half of the class is able to second guess what the output will be, but is unable to properly explain how the execution proceeds step by step:

> **Ada:** They cannot explain. They can tell me the output, but they can't
> explain it with argument passing on to parameter.

Ada does not seem concerned about exposing students to a function that does not return a value, but is more akin to a procedure. She reports that she has not heard observations on this from her students.

Ada also notices large differences among students even at the end of a year with PyTamaro. In her "Animations" curriculum, an activity asks students to program the animation of a ball, represented as a circle, that moves up and down.

> **Ada:** [An animation] at the end of the day this is just a bunch of pictures
> where things change from one picture to the next, so they would have
> to figure out themselves how to do that. And for some this works really
> well: one student immediately saw the picture [...] you would move the

> ball by adding an invisible pillar at the bottom. Others don't see it [even]
> if I explain it to them. So this [...] 'similarities and differences'... for
> some, they see it immediately, and others you can work them over the
> head with it and they don't see it yet.

This curriculum with animations is a place where Ada sometimes only shows images to students, and so they have to implicitly play the game of "similarities and differences" (Section 5.5) across frames.

11.5.3.3   Offline exercises offer practice for the Toolbox of Functions

Ada created dedicated exercises for students to practice the extraction of a function in order to save it to the Toolbox. The exercise shown in Figure 11.3 asks students to mark the code not needed to save the function `japan` to the Toolbox.

```python
from pytamaro import *
show_graphic(testy(1.1))
for name in namen:
    adressen.append(
        name.lower().replace(" ", ".") + "@firma.ch"
    )
from pytamaro import (
    rectangle, ellipse,
    white, red, overlay,
    Graphic, show_graphic
)
def japan(breite: float) -> Graphic:
    hintergrund = rectangle(breite, breite / 3 * 2, white)
    sonne = ellipse(breite / 4 * 1.3, breite / 4 * 1.3, red)
    return overlay(sonne, hintergrund)
show_graphic(japan(200))
```

Figure 11.3. Starter code for an exercise in which students are asked to mark the pieces of code not necessary for the function `japan`.

> **Ada:** They should realise that they only need one function. So that means
> that the code before the [second] PyTamaro import should all be deleted.
> So no matter what [is] actually there, they should be able to realise that
> they can just ignore it and delete it because it's not used for the graphic
> function.

11.5.3.4   Decomposition was also discussed in older materials with turtle graphics

Ada's older materials, that used turtle, featured a section on "Animation and Decomposition". She is deeply aware of the importance of decomposition as a general process:

> **Ada:** Decomposition is when you take things apart, put them into their individual pieces. It's the same in literature. When you analyse a book or a poem, you take individual pieces and you say this is X, this is Y, this is Z. You put labels on them and then you try to figure out how they work together... and same with graphics. You figure out that this graphic consists of [...] two blobs on top of each other and a line. I guess [I was already focusing on] decomposition because that's what I learnt.

The fact that the materials were over two years old and not really in use anymore meant that Ada could not recollect whether the decomposition process with turtle graphics was supposed to be applied only across multiple frames, or even within the individual graphic (which would be harder to do cleanly in the context of turtle graphics, given the presence of implicit state).

11.5.3.5 Both constants and mutable variables are used

Ada is also using the "variable as a box" metaphor to explain variables. She recalls having heard about it during her training program, when one of the professors talked about the advantages and the disadvantages of that metaphor.

Ada recognizes the clash between the concept of mutable variables in programming and variables in mathematics. She does not exploit, however, the fact she could connect the concept of immutable variables, often referred to as constants in programming, with variables in mathematics:

> **Ada:** The problem really is with they know variables, but from math and in math it's different. It's a different concept.
> **Interviewer:** Right. Because the math variables are more like what we call constants [...]
> **Ada:** Yeah.

On the other hand, Ada's activities talk about constants at the very beginning, for example when referring to the constants for colors, such as `red`.

11.5.3.6 A "Table of Values" is used to explain repetition with loops

Ada's first activity that introduces the concept of repetition is done outside the domain of graphics. She presents the idea with a variable that accumulates the sum of a sequence of numbers. The sequence is generated with the `range` function and iterated over with a `for` loop. She seems convinced that the mathematical domain is easier than the graphics domain to explain this idea, but her argument is not fully valid:

Interviewer: Is repetition first introduced without PyTamaro?
Ada: I think so, yeah. Otherwise they have to understand what an accumulator is, and that is really hard.

The second activity explains repetition using graphics as a domain. Ada uses a "Table of Values" to keep track at each iteration step of the operation and the new value for the accumulator variable. Such a table is presented in the first activity to students (Figure 11.4a) filled with numbers. Students are then expected in the second activity to trace a given code that uses PyTamaro graphics to produce the table shown in Figure 11.4b (which is the reference solution).

element	akk + element	akk (neu)
0	0 + 0	0
1	0 + 1	1
2	1 + 2	3
3	3 + 3	6
4	6 + 4	10
5	10 + 5	15

(a) Accumulating numbers.

(b) Accumulating graphics (`0x0` stands for the size of the empty graphic).

Figure 11.4. "Table of values" with numbers and graphics.

11.5.3.7   She tends to avoid nested expressions

Ada's code examples tend to avoid nested expressions. She is skeptical about the possibility of helping students by relating the evaluation of expressions in mathematics to the one in programming:

> **Interviewer:** I guess [in mathematics] they will see some nested function calls [...]
>
> **Ada:** But you know, this is transfer skills, and transfer skills are super hard. [...] When they learn programming, they're still at the beginning of gymnasium. So they don't excel at that one yet. I mean, if you have those students, that is brilliant, but they are few. They need to learn about that, [it] is not a skill that they are born with.

11.5.3.8   Some of her materials include method calls

The superfluous code in Figure 11.3 contains methods on lists, such as append, and on strings, such as lower. Ada confirmed that she had explained such methods before, but she did not relate or compare them to the simpler functions students are used to:

> **Interviewer:** Do you even connect the two [methods and functions] or do you just say 'these are something else' [...]
>
> **Ada:** [I tell my students:] 'It's *Deus ex machina*, [if you] don't do it like it, it doesn't work. I'm not going to explain it to you.'

She is well-aware of the problem this may cause:

> **Interviewer:** So they just memorize it...
>
> **Ada:** Yes, yes. I don't think this is a good way of teaching, but it's a realistic way of teaching. Because they will see out in the world, these 'dot calls' and it's really tough to then be able to figure out what the state of various variables is, which makes it hard to understand code. Which is why I try not to teach it too much. But on the other hand, they need to learn to use code that they don't understand how it works.

Still, some of these methods, such as the ones on strings, are effectively stateless and could be explained as functions with a special syntax. For example, `"PyTamaro".lower()` can be viewed as a `lower` function taking one parameter: the object onto which the method is invoked (e.g., `lower("PyTamaro")`). This would help to relate method calls, which are unfamiliar to students, to function calls, which they have seen extensively. The Judicious documentation system could be extended to cover these cases and help teachers bridge to methods.

11.5.3.9   She does not use TamaroCards

Ada is not currently using the paper-based approach with TamaroCards to introduce functions, although she still reports using the Judicious documentation on the web platform, which by default includes a diagrammatic representation of functions (Section 9.2.1).

To justify the choice, Ada reports that she is experimenting with her current 10th grade students by having fewer in which they "program" on paper. She recalls doing that only once this year, to explain assigning values to variables. However, her exams also require programming on paper.

11.5.3.10   Her activities on PyTamaro Web end with explicit learning goals

Building upon a suggestion of her final project advisor in the training program, at the end of each activity on the PyTamaro Web platform Ada writes a recap to summarize what a student is supposed to have learned, as shown in Figure 11.5.
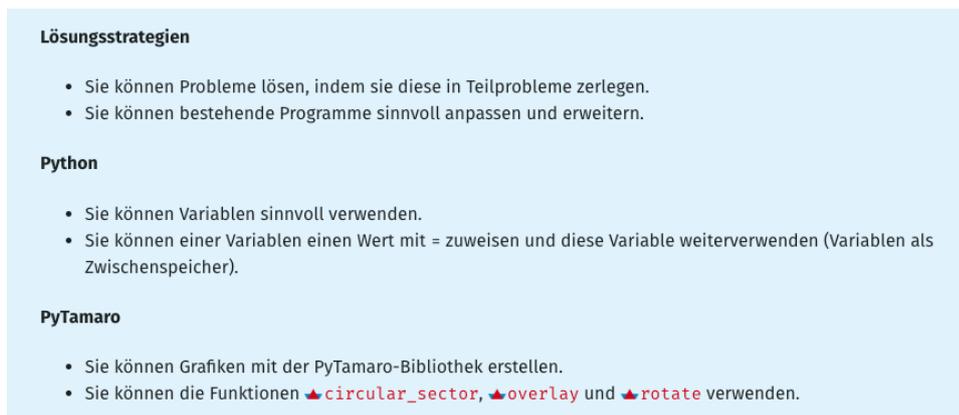
Figure 11.5. Example of learning goals at the end of an activity.

The recap consists of bullet points structured into three major parts: general ideas about programming, knowledge specific to the Python programming language, and knowledge specific to the PyTamaro library. Examples of each category include, in order, being able to decompose a problem into smaller subproblems, knowing the syntax for calling a function in Python, and knowing PyTamaro's above function.

This clear structuring seems to have the potential to clarify the boundary between the programming language and the specific APIs students learn:

> **Interviewer:** [Even after presenting this clear separation,] do you still hear students [saying] 'No, we didn't do Python, we did PyTamaro'?

> **Ada:** I've never heard that. Never heard anyone say 'We did PyTamaro' instead of Python. I do explain what the language is, I do explain what the library is. But the problem is [...] we use multiple libraries, but mainly PyTamaro and I don't think we do any other programming languages apart from SQL [...]. So I do tell them, but they only see actually Python and PyTamaro working together and very rarely do they see Python without PyTamaro at all. So even though I tell them, and even if they repeat after me, that 'Python, PyTamaro' are separate, I don't think that they see that. I'm skeptic about that.

Some of Ada's activities use only libraries that come standard with Python, for example, to play with random numbers or doing string manipulation or mathematical exercises; but she admits that PyTamaro-based activities vastly dominate the others.

Ada does not emphasize too much in class the learning goals at the end of an activity, but she observed that some students actually do read them, because they know those points will be the basis for exams, or just because they are curious to reflect on what they were supposed to have learned.

11.5.4 On the student experience

11.5.4.1 Student attitude varies more individually than by their major

Ada has been teaching computer science to students who choose economics or modern languages as their major. She claims that the profiles between the two classes are very different, but they react similarly to graphics. The variability of individual students within one class is greater than and dominates the differences across classes. Still, she notices a difference in how they approach their coursework:

> **Ada:** In modern languages you mostly get girls [... who] tend to be very studious, so they put in a lot of work. Even if they don't understand it, they put blame on themselves and then they work. Not every girl, of course, and not every boy does that. But then, very often in economics, you get the boys, who are time efficient.

11.5.4.2 Students find listing explicit names to import demanding

One of the snippets of code Ada briefly used in the past for an introduction to turtle graphics had as a first line the statement `from gturtle import *`. With PyTamaro, it is usually encouraged to explicitly list the names one is importing from the library and avoid this wildcard import, that makes things work with some sort of magic. When asked about any positive or negative impact of this choice, Ada focused on the amount of things a novice has to understand at the beginning:

Ada: Well, they were certainly annoyed... when I made them switch to using every single... to import the names individually. And it's also difficult to explain why they have to do it the complicated way, instead of the easy way. I mean, at that moment they're learning so much, because they start from the beginning. I don't bother [them] with having to list everything [...] Because when you're a beginner, the amount of stuff you have to learn is insane. Really insane. You totally forget that [...] once you've been programming for a couple of years, it's insane.

11.5.4.3 Students get creative in the final project with PyTamaro

Overall, Ada feels that PyTamaro gives her students enough freedom to be creative and have agency in their final projects.

Ada: I have students who excel at this and then they don't feel constrained at all, they feel enabled. And other students who would then like to use other libraries inspired by AI code snippets. I wouldn't say that they know other libraries where it would work better. They're so well versed by that point in PyTamaro that this is what they know. So why change system when they're already challenged with putting code together on their own for the first time.

For Ada, the project is really the first moment in which students start from a blank slate and have to write entire programs on their own. She is considering doing more of it:

Ada: I think about doing that earlier... I think it would be good for them.

11.5.4.4 Students can debug PyTamaro programs without a debugger

In her older materials with turtle graphics, Ada showed a screenshot of a debugger. When asked to reflect about it, she realized that debugging is currently lacking as an explicit topic in her current PyTamaro-based curricula.

Ada: So debugging is one of the topics I don't talk about with my current version, and I do think that is something that is missing, because students need to understand what is happening at every level of their code. On the other hand, they cannot rely on extra tool, they have to use their own brain... which is theoretically a good thing.

Nonetheless, she did not make extensive use of a proper debugging tool even before PyTamaro. She also noted that students can use a form of "print debugging" by inserting side effects between instructions, such as calls to `show_graphic`, to inspect the value of the variables.

11.6 The case of Barbara

11.6.1 This is Barbara's context

11.6.1.1 She teaches mathematics and is critical about her programming knowledge

Barbara studied German as her major subject and mathematics as her minor during her university studies. She has been teaching mathematics at a high school for approximately 17 years.

She attended the national retraining program for informatics teachers in Switzerland, but she also had a Java programming course as part of her university studies. She is quite self-critical of her previous programming knowledge:

> **Barbara:** I learned programming with [prof. Hauswirth], with PyTamaro. Before I learned stuff by heart and I reproduced it. But I wasn't able to really do new things.

11.6.1.2 She teaches in the 9th grade to students from different majors

Barbara has been teaching the very basics of programming with MATLAB and a turtle curriculum using TigerJython [259] as part of her math course. In her school, students learn about ICT and a little bit of Scratch in 7th and 8th grade. Afterwards, the core programming part starts, and teachers are free to choose the approach.

Barbara teaches with PyTamaro, as well as a colleague of hers. Another teacher at their school uses JavaScript with the P5 graphics library. The remaining two teachers use turtle graphics with TigerJython.

The programming part spans two-thirds of the 9th grade. Barbara's students come from a variety of majors:

> **Barbara:** Last year I had two classes, one was a 'music' class and the other one was a 'bilingual' class [where] you have to have an average mark of 4.5 [out of 6].
> **Interviewer:** Would you say that there is a difference in how much they liked your stuff?
> **Barbara:** It's easier for the bilingual.
> **Interviewer:** Because they are just better students in general.
> **Barbara:** Yes, and they don't mind the English stuff. A `for` loop doesn't bother them, but other students they have to translate. A little difference, but you feel it.

11.6.1.3 She mostly focused on style when giving feedback to two PyTamaro programs

First program   Barbara correctly notes that the feedback on Listing 23 would depend on what the students have learned by that point:

> **Barbara:** The first thing is... when is that program [written]? Is it after two lessons? Or is it at the end of the first year?

She reports nudging the students towards writing nested code and avoiding giving a name to individual primitive graphics:

> **Barbara:** I always told them we don't do primitives into a constant. At least two or three primitives go into a constant.
> **Interviewer:** Interesting. Why? Why would you say that?
> **Barbara:** Because a lot of students do exactly that, and it doesn't make sense to have just rectangles with a name

Indeed, one of her grading criteria talks about "shallow nesting depth":

> **Interviewer:** Is this related? You want them to understand nesting a little bit and nest their code...
> **Barbara:** Yes, and find the balance between nesting and depth. I mean, if you have 12 primitives in one [expression] it's horrible. [...] [Having both] `arm` and `arm` doesn't make sense; if you do it like that, take `arm` and `arms`. Use sensible names.

While composing nested expressions plays well to PyTamaro's strengths, not giving names to primitives and "inlining" them can lead to code duplication. She offers a solution for the specific problem and discusses "elegance":

> **Interviewer:** But wouldn't that then introduce like quite some code duplication?
> **Barbara:** I wouldn't take `rotate`... [I would just use] `rectangle`, 60, 200... so I don't have code duplication.
> **Interviewer:** Uhm, I see. [...] I'm not sure this is going to work in general.
> **Barbara:** Absolutely not, but that's one thing I want to train with them, to think about what is reasonable in a code. 'What makes it readable? What do I need to do to understand the code? How can I use [as few as possible] variables and functions?' I always talk about elegance. I wouldn't say that is elegant. It works, but it's not elegant.

Second program   Barbara again refers to a vague notion of elegance when assessing Listing 24:

>**Interviewer:** The question is the same, what would you give as feedback?
>
>**Barbara:** About the same: it works, but it's not elegant.

On this second program, she immediately noticed the code duplication, and also complained about poor names:

>**Barbara:** I would define one eye [...] and then `eyes` with a `beside`. The names aren't good; like what does it mean 'big green'? In two weeks, they don't know anymore what 'big green' means. [...] They should use names they can imagine what it is.

She did not bring up the possibility to abstract and create a function to draw a circle, but understood the tip when prompted:

>**Interviewer:** What about the little duplication there, like `diameter / 4`, `diameter / 2` that is repeated for both arguments of `ellipse`? If you were to get rid of that... First, would you? And then, how would you [do it]?
>
>**Barbara:** We start refactoring really early and that is one of the examples we refactor. I would say: 'make a function'.

She then suggested the use of a different primitive, `circular_sector`, to avoid specifying the radius twice. She took a while to find a solution that still uses `ellipse`, potentially because she is used to have a `circle` function in her Toolbox:

>**Barbara:** I don't see what you want me to say... Ah! I know! You mean the `circle`. I haven't seen it! In fact, we relatively quickly introduce the Toolbox and they do have `circle`.

11.6.2   On the choice of adopting PyTamaro

11.6.2.1   The lack of a textbook made her hesitant

At the beginning of her career teaching the newly introduced mandatory informatics course, Barbara did not use PyTamaro: she followed the textbooks that accompany TigerJython. She was not entirely satisfied with them:

>**Barbara:** I wasn't happy with it and I was thinking about doing something else.

She was hesitant about PyTamaro, however:

> **Barbara:** There is no book or stuff, but then I went to the summer camp
> and realized that I can use that stuff [the slides] and develop it.

Indeed, as we shall see in the next section, the backbone of her materials consists
of slides used in a summer training camp with PyTamaro dedicated to teachers and
students. She has now been using PyTamaro materials for two years.

11.6.3   On the teaching materials

11.6.3.1   Here is an overview

Barbara is teaching PyTamaro with an extensive set of slides. She modified and adapted
some of the slides that were used to teach PyTamaro during a week-long summer camp,
and also added additional slides she created herself.

Barbara is also developing a curriculum to teach recursion with PyTamaro. At
the time of the interview, the curriculum was still under development and had not yet
been tried with students. Those activities are more advanced and target students in the
11th grade who choose computer science as an elective subject. They guide students in
writing programs to draw fractals with PyTamaro, such as the Sierpiński triangle and
the Koch snowflake.

11.6.3.2   TamaroCards are used from the beginning

Barbara follows the sequence proposed in the summer training program to introduce
PyTamaro:

> **Barbara:** We introduce primitives, constants, and at the end of the day,
> they already refactor to functions. But they do it all [using the] cards. So
> after one day they know how to write small code blocks and save them
> into a variable or save them into a function. The funny thing is, after one
> day they are really good... and then the holidays come, and they forget
> a lot.

Then, she uses more graphics, such as a traffic light and a seven-segment display, to
practice programming:

> **Barbara:** We always talk about how to do it, they decompose it, then they
> go on Miro board [a collaborative online canvas] and do the PyTamaro
> card stuff.

She instructs them to use the "Playground" of the web platform and define a constant:

> **Barbara:** If it works, they are allowed to refactor it to a function. That's the normal process they learn. Decompose, think about composing and then write it, constant, function.

She reports that the transition from cards to code already happens on the first day of instruction. When asked whether she guides the process or lets students figure it out independently, she confirmed she draws a "path" to teach students how to systematically turn an expression specified with the cards into a syntactically valid Python expression (a process we described in Section 6.4).

11.6.3.3 Students mostly use the Toolbox on the web platform

Barbara's students mostly work using the Playground on the PyTamaro Web platform. She tried to use Thonny [12], but some issues arose when using third-party libraries, possibly caused by the specific setup at her school. Some of the more advanced students use Visual Studio Code [267], but no teacher support is provided for it.

Barbara instructs her students to use the Toolbox—in the form of a local file in Thonny or a virtual file on the web platform—to avoid code duplication and teach abstraction, as shown in Figure 11.6, which reproduces one of Barbara's slides.

Figure 11.6. A slide that compares working with the Toolbox in Thonny and in the PyTamaro Web Playground.

Overall, a good number of students appear to have become familiar with the process of working with the Toolbox:

Barbara: We don't want to have code duplication, so they should use the Toolbox. The better third [of the students get it] really fast, the middle third always forgets to save it [the function] or saves it somewhere else, or has a `show_graphic` in it. I think they got the concept [of saving and then importing].

11.6.3.4 A transition from constant to variables happens when introducing loops

For the first part of her course, Barbara uses "constant" to designate names. Before introducing loops, however, she felt that she had to clarify that actually constants are variables and can change value over time. She explains this with an animation right before loops, as seen in Figure 11.7.
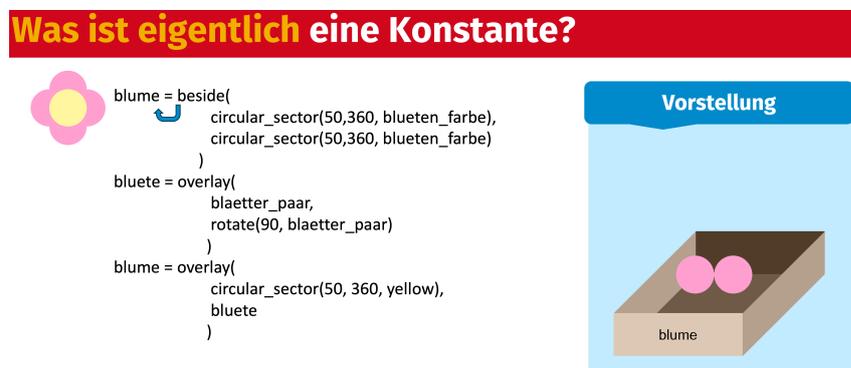
Figure 11.7. A slide featuring the "variable as a box" metaphor with a PyTamaro graphic.

Barbara: Afterwards we go to loops. We normally have constant as something unchangeable. If you save it [the graphic] as `blume`, `blume` is gone, you can't use it anymore. And now we tell them why it is called 'variable' for real, because you can save again and again.

She reflects on the use of "variable as a box" as an analogy, which is known to have problems:

Barbara: And the boxing... I know it's not a very good picture, but we call it a 'magic box'. It's always only one [element] in it, but if you take one out, there is another one in it to take it out... because the box thing has the problem, if you think of having infinite flowers in it, it doesn't make sense at all, but you can take out infinitely often the flower, one flower.

Interviewer: With 'take out', do you mean when you read from a variable?

Barbara: Yeah, yeah. There isn't an infinite amount of flowers in it, but it's one flower you can take out infinitely often.

This new knowledge of variables is then used to achieve repetition with loops:

Barbara: And I think it's very important because we have that constant picture of variables that they are immutable, but then we, suddenly, out of the blue, we save again and again and we save something different. And in the first round when we [I and a colleague] taught loops, we realized that it is a real big problem. That's why we do that thing with the boxes. That helps a little.

11.6.3.5 There are issues with transfer on loops and lists

The "variable as box" analogy is problematic when dealing with mutable objects. Barbara reported covering lists, but not anymore with PyTamaro, in the subsequent year (10th grade):

Barbara: We go to `print`. We work with numbers and characters. We also realized that, for example, the loop processing a list... that's really complicated after PyTamaro, because they always want to work with empty graphics. They don't see... I don't know where the problem is exactly, but that's what we observed.

However, if all the loops students see use an "accumulator" variable that is initialized with an empty graphics, they may conflate the two concepts. It may be useful in the first year to show and contrast multiple approaches.

Barbara: We do that and we also do it really step by step. They have several for loops processing lists, just to read and then to transform to something else, to use it. Like we start with numbers, and they have to change it so they can process characters or strings. And still in the exam last year...

This good example only comes in the 10th grade. Even a small example already in the 9th grade, to introduce already a loop with numbers, could probably help. Time constraints dominate, however, and she feels that her progressive build up towards a graphic with a loop is quite effective:

Barbara: The problem is really we have the loop in a really short time.

Interviewer: I see the [progressive reveal on the slide]: so you build up

> the graphic…
>
> **Barbara:** And they quite get it. And then they use it as you see to make one of the pictures on the next slides. And it works quite well.

11.6.3.6 The PyTamaro curriculum focuses on concepts, but turtle requires less syntax

Given her experience in following a Turtle-based curriculum, Barbara could comment on the differences, positive or negative, that she saw between the two approaches (e.g., covering some concepts more or less extensively). Overall, she was quite critical of the previous approach:

> **Barbara:** I think the main goal of teaching computer science or informatics or programming is to get them understanding how programming works. And I think [the previous approach] doesn't fit at all. I mean, you can have that turtle walking around for the whole year and nobody understands anything about programming.

That was a strong claim, especially because controlling the turtle (physically, as the original idea, or virtually) is closely related to programming a robot, something that students could relate to. In what appears almost like a stream of consciousness, she doubled down on criticizing the previous approach with turtle graphics, contrasting it to PyTamaro:

> **Barbara:** It's about the same when I say "just decompose your language so you can command a robot". I mean, it's "right", it's "left". It's not thinking about why this command works? It's not about writing your own command, because right and left are fixed. You can write a command `circle` or a `square` or something like that, but… I mean, I thought [turtle graphics] in mathematics for eight years or so, and I thought I could program. [*Laughs.*] And then I realized it doesn't help me with Java, it doesn't help me with constants, with variables, with functions, types, stuff like that. You just have that turtle walking around and you think: 'oh, wow, I can program', but what PyTamaro does is… you have to think about what is `yellow`, what is `circular_sector`, what is a function. You have `above`, `beside`, you have higher order functions. I mean, that's programming, that's what I use to program other stuff when I program [in] Java. That comes from my experience. I learned Java with the book "Java ist auch eine insel". It's a horrid book. I had to program mammals and cats and dogs and. I couldn't imagine what that could be. They only barked and ate and slept and… I had a vision

of cats and I thought 'I program a picture of a cat' and it didn't work. So thinking about what I program [...] is given in PyTamaro but not in [other approaches]. I learned programming with PyTamaro because then I got the concept of functions and variables, and defining my own functions and then getting 'ah, you can reuse it!', and stuff like that. I never got that with [turtle]. 'Abstraction'. I never got that. It's a little embarrassing. I taught programming for years and I haven't had a clue myself.

It is not uncommon that at some career stage one teaches concepts without fully understanding them. But Barbara's comments were so favorable to the PyTamaro approach that deserved some pushback. When pressed, she found an aspect that was better when she was following the turtle approach:

Interviewer: What is something you would say was more prominent [or better, in turtle]?
Barbara: I think the frustration was less.
Interviewer: Why would you say that?
Barbara: Because you can have the turtle walking, right 90, forward 100, right 90, forward 100, right 90, forward 100, and you can read it. You can have an endless list of functions, and the turtle did do something. PyTamaro, if I forget a comma, the whole stuff doesn't work.

She noted that the burden imposed by the syntax is higher with PyTamaro, but she was still convinced that it was worth it for her context:

Interviewer: [So with PyTamaro] it's easier to get a syntactic error...
Barbara: Yeah, but I think that's the real thing. TigerJython is, in my opinion, too student-friendly. I mean, you don't have commas. The only thing you have is the indent after `repeat` or after a definition. But that's it. There is not a lot of points where you can get frustrated.

TigerJython modifies Python to introduce a `repeat` statement to simplify the syntax of a loop that repeats a fixed number of times [147]. But for Barbara, this may actually become a downside, if the goal is to progress to "full" Python.

11.6.4   On the student experience

11.6.4.1   Some students still struggle with syntax, despite TamaroCards

Despite using TamaroCards, Barbara reports that this is not solving syntactical issues for all of her students:

> **Barbara:** About half of them really use it because they realize that think-ing about syntax is rather difficult. And if you do it [using the] cards, they can think about the process of composing and decomposing and don't have to think about syntax. But other students, they skip it. And what I see there, they normally have really big issues because they just write `triangle`, `triangle`, and then `above`.

These students are probably struggling with the nesting, using what she calls "natural syntax":

> **Barbara:** And they get really confused because it doesn't work. Play-ground always says 'Perhaps you forgot a comma', 'missing argument' and they say 'all arguments are here, what did I do wrong?'. And sometimes I get them to go back and do the cards stuff and that really helps. And after a while I would say one third is able to directly write code without the cards, and one third is not able after a year [not even with the cards]. [The remaining] are still working with the cards.

Barbara reports, however, that the underlying issue may just be with students who are not practicing programming enough:

> **Barbara:** We have one problem, and we don't have it only with PyTamaro but also the [teacher who uses] P5... that people don't do their home-work. We can't get them to practice, and if they don't practice, they don't get used to syntax and so they're not able to program.

One possible explanation could be that the tasks offered to the students are not at the adequate level, either because they are too hard or too easy. Alternatively, or perhaps additionally, students may not see programming as a relevant activity. If that were the case, the materials could be questioned in terms of their engagement for her audience. Barbara speculates about the underlying causes:

> **Barbara:** I think it has to do with frustration. What we see is that if it doesn't work in the first place, they often say 'it doesn't work!'. And then I say 'read the error message, what does it say?' OK, it says 'You forgot a comma', 'go and look where there is a comma, where is a comma missing? Go through the code'. And they are not able to live with that frustration that in 80 % of the tries it doesn't work. And the problem is you have to stay at the computer for 45 minutes. And try, and try, and try. And be critical with yourself. I guess some of them are not able to say 'OK I made a mistake. That's OK. I try to find it. I try to fix it.'

Programming is known as an activity that requires demanding levels of attention to deal with abstractions and rigorous notations [27]. Barbara believes that some students are not used to it:

> **Barbara:** As a math teacher, I see the same problem with first-year students. But in math, it's not 80 % of the time. It's about 50 % of the time. In programming, it's quite a lot of frustration. I mean, nobody looks for spaces in texts, but if you have a space in the wrong place, the program doesn't work. Or a comma, what does it change if I have German text with or without comma? And now the computer says 'sorry, without that comma, I won't do anything'. They're not used to that.

11.6.4.2 Projects were affected by language models and a restricted set of activities

For the final project, Barbara and a colleague of hers took all the activities available on the PyTamaro Web platform, excluded some of them because they felt were unsuitable, and classified the remaining ones into easy, medium, or hard. Students could then pick two activities to program; solving harder activities awarded more points. This worked quite well for her:

> **Barbara:** We allowed "foreign intelligence", not only AI, but also help from somebody because we had a lot of parents, and friends, and siblings helping. They had to declare if they used somebody else, that's fine with us, but they have to explain the code. And we don't mind if they have... nested for loops three times or whatever, as long as they could explain it. And for my classes it worked fine. They really had a lot of fun. They did do real good stuff. I had only one person who tried to cheat. And one person who had almost the whole code done by AI. But the second one, really worked on it so he could explain it, he said freely: 'the code was generated by AI, but I can explain it'. He got the points for explaining, but not for writing. But it's OK. He knew that before.

But not so well for a colleague of hers:

> **Barbara:** And for another class with [a colleague], about half of them tried to cheat, and I mean... [that's] really stupid. Like they had constant names for real constants written in capitals. And we ask 'why do you do that in capital'? They couldn't answer why. You normally do it [...] like with pi [...] but they haven't learned it with us, so it was obvious. Or they used lists and haven't heard anything about lists. They couldn't explain what 'list name', 'square brackets', 'i' means and stuff like that. It was

> really horrible and they tried to tell us they wrote that by themselves, and that wasn't cool.

Some of Barbara's comments are surprising, because at the time of the interview, most LLMs had not seen much PyTamaro code in their training data and they frequently hallucinated incorrect code. Barbara instead reports that the leading models perform quite well:

> **Barbara:** But the paid [version of] ChatGPT knows [PyTamaro] quite well. I use it myself. And they also have parts of code from the PyTamaro website, like the watermelon [activity]. There is some code you can feed to the AI and it only fills in the missing stuff.

Students were not upset by the limited choice of activities for their final project, but Barbara plans to change this aspect for the current school year:

> **Barbara:** They had quite a lot of fun. But this year we'll do it the other way around. They have to paint a picture first and then program it. Because I think also [that] some activities are too guided and some are not, and it's quite heavy to [decide] to give points or not. It was an experiment and we weren't totally happy.

She still plans to focus on code comprehension and decomposition in the domain of graphics:

> **Barbara:** But that's why we had an oral exam afterwards about their code. They had to explain the code. They had to explain the process, how they worked, how they decomposed, and stuff like that. And quite a lot of points were for that oral exam.

11.7   The case of Charles

11.7.1   This is Charles's context

11.7.1.1   He is a biology teacher with some programming experience

Charles started teaching programming in the school year 2018/2019. He has been a biology teacher, his main subject, for approximately 12 years. He started learning programming during his studies, before becoming a teacher, learning a bit of Java at the university and working with the R language afterwards to do statistics. Before the introduction of computer science as a mandatory subject, he was teaching Java to his students, but he claims to be most familiar with R. He also attended the retraining program, where he mostly took programming courses in Java, with Python being

marginal.

11.7.1.2   He uses PyTamaro in the 9th grade

The mandatory informatics subject happens in the first two years in Charles's school, corresponding to the 9th and 10th grade. Every week he holds a "double lesson" of 90 minutes with the full class. The cantonal curriculum also requires teachers to include parts of ICT, effectively limiting the time for all the other topics, including programming. The programming part lasts roughly 2 months in the 9th and 10th grade. During this second year, Charles's students work on a programming project with a popular Python library for games.

11.7.1.3   He gave good feedback on two PyTamaro programs

First program   At first, Charles praises the structure of the code in Listing 23, which seems in line with what he teaches:

> **Charles:** What I like is the naming. There's quite a lot of naming, like 'background', 'arm', etc... before the graphic is actually shown. And this is actually something that I suggest to my students as well, to do this kind of naming [so that] the code is better readable. Maybe, one problem is [that] it's not very flexible, so [if] you need to change those numbers... it might make sense to use variables there as well, not just for the intermediate steps. At the end of the first-year curriculum, I'd suggest that they transform this into a function, with all of those as parameters.

He then realizes that `arm` as a name is used twice, and suggests a better alternative:

> **Charles:** Well, one thing maybe that's quite... bizarre is that they use the same variable name for arm, for the two states of arm... so in a more like... pure, function-oriented thing, you want to name them differently I think.
> **Interviewer:** So what would you use as a name?
> **Charles:** Well, first horizontal and then vertical arm... Oh no! The second thing, the second arm is actually not an arm. It's already... it's the cross already. It's the wrong name. [...] So they overlay that rotated arm and the arm, and they still call it 'arm'. Well, but 'arms' might be better, or 'cross'.

Second program   On the program of Listing 24, Charles noticed the code duplication and suggested factoring out the common code:

> **Charles:** Well, again, you could transform this into a function. Quite a lot of naming this time. I think the names 'small black' and 'big green' are pretty straightforward. [...] However, I'd rather suggest that they use an intermediate step between 'big green' and the 'eyes'. So maybe first program the eye... because there is so much redundancy in the 'eyes' line.
>
> **Interviewer:** There's a whole part [being] repeated, no?
>
> **Charles:** Yeah, exactly. If you use eye just with `overlay` of `small_black` and `big_green`... you have a smaller code snippet, less redundancy. So you could just glue together your two eyes.

After he suggested again the possibility of removing hard coded numbers and colors, he also noticed an opportunity to further abstract:

> **Charles:** Maybe something else here? Maybe this is also already valid for the program above. You could simplify the ellipse as a circle, a perfect circle. So maybe use a function that creates a circle which uses `ellipse` [with] the same diameter. And the same before with the rectangles.

11.7.2   On the choice of adopting PyTamaro

11.7.2.1   The graphic domain is engaging for many students

Charles works at a gymnasium where students can choose their own specialization. Out of approximately 12 classes, one or two are following a STEM curriculum. All the teachers chose an approach with graphics, be it PyTamaro or turtle graphics:

> **Charles:** At the beginning we have mixed classes and we want to have something that motivates all of them... because I think it's... more like fascinating and it's easier to engage with, even though they are not all motivated in programming first. [...] Actually, most of them want to learn programming, but it's still something that is a to-do for them.

There is a clear difference with the students who chose a mathematics-oriented major, but even those students do not dislike graphics:

> **Charles:** There's a huge difference because the STEM students, they're really motivated for programming. I think with those you could do whatever... But these are the ones who like graphics as well. These are the ones who start experimenting with plenty of different things with the graphics.

Charles developed teaching materials with PyTamaro in preparation for a study he carried out for the final project of the retraining program. Charles taught programming with PyTamaro for the first time during the last school year, when he was conducting the study. He taught two classes in parallel, one with turtle graphics, and the other with PyTamaro. Before that, he was using materials developed at his school by other teachers. These materials were based on turtle graphics. This group of eight informatics teachers is still following the turtle-based curriculum.

11.7.3 On the teaching materials

11.7.3.1 Here is an overview

The teaching materials Charles provided are those created for his study and consist of eight digital handouts to introduce programming using PyTamaro. Starting from unplugged activities with TamaroCards, they proceed to programming in Python, defining functions, using PyTamaro's `pin` and `compose`, repeat computation, work with colors and animations, and end with conditionals.

This year, he has made some adjustments to the materials:

> **Charles:** I did take some takeaways from the experiment and tried to implement them already. So this time I provided them a lot more time at the very beginning. I had this unplugged exercise and this time I had additional 90 minutes, like one of those double lessons, where they could try to transform those unplugged things into code. That was actually absolutely absent before [...] maybe 20 minutes the year before, where they could think about code. But this time I really wanted them to have the entire time to program those things that they already programmed in their mind, with shapes, etc., now with Python. This was quite a change. The second change is that this time, like range and conditionals... [they were] a bit earlier last time. Right after repetition, [...] I moved to colors because [...] I thought that it's actually a more logical successor to sequences [such as lists], because it's basically again a sequence. Or also to create animations, this is also covering sequences.

11.7.3.2 He uses TamaroCards and explains how to turn programs into Python

Charles uses TamaroCards for some unplugged programming activities:

> **Interviewer:** Do they actually do that [composing programs with the cards] physically or do they like drag and drop stuff on their machines?
> **Charles:** No, no, physically. So, paper, scissors, etc... differently col-

ored paper. I usually use some brownish paper as a background and they sometimes glue stuff or they just lay down so they can rearrange. What I always want is that they [...] label the things or they add drawings like arrows and in the end they take a picture and upload [it] as an assignment.

For the entire duration of a 90 minute lesson, students work only with the cards:

Interviewer: Do they use already [...] names, constants, the blue [cards]? Or do they just build a big expression?

Charles: Well [they are] there, but I don't comment this really at first, so they do big expressions. And after a while, especially with the eyes, they get kind of annoyed that it's so repeating, and so complicating, and so much time... And then I introduce names as a 'freedom', so that they'll be less annoyed. This is actually also the introduction of 'giving names', already. This happens maybe at half time, because I want them to feel kind of annoyed first, so they have motivation to actually use names.

Figure 11.8 shows a single, big expression to build the flag of Greenland, while Figure 11.9 shows students using a custom constant they defined (`eye`) to build a pair of eyes.
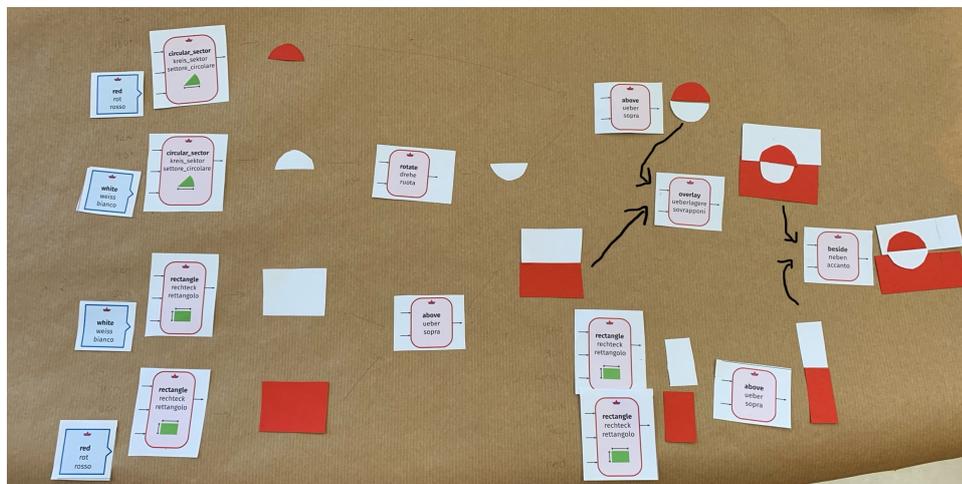
Figure 11.8. Photo of students' work composing the Flag of Greenland in a single expression using TamaroCards.

The second lesson marks the transition from the unplugged activities with the cards to writing Python code. Charles's approach is more ad hoc than the systematic methodology presented in Section 6.4:
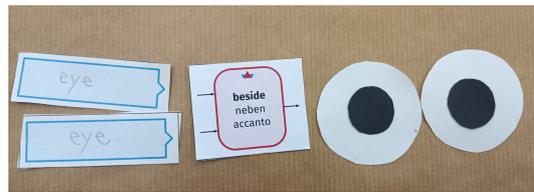
Figure 11.9. Photo of students' work composing a graphic of two eyes with personalized constants using TamaroCards.

> **Interviewer:** What are the instruction for the students to move from the cards to the code?
>
> **Charles:** The instructions are extremely small. It's basically 'this is the documentation about code'. Of course, they kind of know the code already because of the cards, but not all of it. I really propose them to look at the official documentation. [...] They usually do it in teams, in pairs. So it's OK if one codes and has the unplugged material, or the documentation, and then they just need to look at their own unplugged versions. What I did in the meantime is I commented on their unplugged versions they uploaded. So they read those comments, they tried to implement the comments... now directly with code.
>
> **Interviewer:** I see, that makes sense. But how do they know... the card shows a 'rectangle' function with three parameters [...] and then it produces a graphic, but there is also a tiny bit of syntax here [to write in Python]: after the function name you need an open parenthesis, between arguments you need to put commas... how do they figure that out? Do you tell them?
>
> **Charles:** There is this one example with the house where they can already see that. And I keep doing it live on the screen as well, like coding and showing them some of those examples. A mixture of live coding, just showing this example and yeah, supporting them if some questions arise.

This approach could only work because some Python syntax was already shown in a very introductory lesson, before the unplugged activities. In that introductory lesson, Charles showed some bits of syntax: how to import from a module, how parentheses work, what common error messages mean, and so on.

11.7.3.3   He uses a memory diagram to explain variables

To gradually introduce repetition with loops and the necessary concept of variables as references to values, Charles showed his students a comparison between zero to add

numbers (Figure 11.10a) and the empty graphic to compose graphics (Figure 11.10b).

(a) Variable `a` referring to values of type `int`.

(b) Variable `a` referring to values of type `Graphic`.

Figure 11.10. Charles's notes illustrating how a variable gets initialized and mutated.

He uses some kind of memory diagram to show the evaluation of `a = 0; a += 2` and `a = empty_graphic(); a = above(a, rectangle(100, 100, red))`, respectively.

> **Charles:** The drawing [...] where I basically showed them that you could see this variable name as a link to the actual thing that it is. So this variable name now points to the integer `0`. And if you change it with the equal sign, you've actually changed the arrow, so that now the name points to the new thing. [...] Basically first you evaluate the expression `0 + 2`, you get a new 'something' to grasp [*makes a gesture pretending to hold something his hands*], the integer `2`, and now the variable name points to that new thing, to the solution.

11.7.3.4   He explains two different ways to repeat

In the sixth unit of his teaching materials, Charles exposes his students to two different ways of mapping a computation. Figure 11.11 shows how to turn numbers into colored rectangles, using them as hues. The upper part uses a list comprehension, while the lower one a `for` loop that in each iteration adds a new graphic to a list.

> **Charles:** [The] `for` loop it's just a repetition for them from last week, because we did these loops last week. And they have the two code snippets right next to each other, so they can really see the similarity of the new concept above and the old concept below.

```
from pytamaro import *

rectangles = [rectangle(200, 200, hsv_color(color, 1, 1)) for color in range(360)]

show_animation(rectangles, 20)
```

```
from pytamaro import *

rectangles = []
for color in range(360):
    rectangles.append(rectangle(200, 200, hsv_color(color, 1, 1)))

show_animation(rectangles, 20)
```

Figure 11.11. Mapping done two ways: with a for loop and with a list comprehension

The loop example uses a mutable list, which looks harmless in this context but potentially opens the door to further problems. As an alternative, one could use `rectangles = rectangles + [...]`, or the compact version `rectangles += [...]`. This would require introducing the + operator to concatenate lists.

Charles had not fully realized that using a loop this way is different from how it was used in the previous unit. The language construct itself, the **for** loop, is indeed not novel, but the high-level computation one is trying to do is different.

> **Interviewer:** They saw loops [...] but without the append, right?
> **Charles:** Yeah, without it. That's true. That's actually true.
> **Interviewer:** Because I think those loops were printing stuff...
> **Charles:** We had a loop where [we were] gluing things together.
> **Interviewer:** Like reducing, yeah.
> **Charles:** This append... [*stops and thinks for a moment*] is actually just so that it's really similar code to the list comprehension. Doesn't it have a similar... append is also gluing things together.

Indeed, earlier examples combined a mapping computation with the reduction of several graphics into a single one, as the code in Figure 11.12 shows for the rainbow flag.

The students seemed to deal just fine with the list comprehension:

> **Charles:** I think they saw the similarities because between the loop and the list comprehension, that you can like basically place directly the individual images into lists with these list comprehensions. I'm not really sure if they got the `rectangles.append` thing at the bottom, because that's a new concept I think.

```
from pytamaro import *

ROT = rgb_color(255, 0, 0)
ORANGE = rgb_color(255, 140, 0)
GELB = rgb_color(255, 240, 0)
GRUEN = rgb_color(0, 138, 40)
BLAU = rgb_color(0, 80, 255)
VIOLETT = rgb_color(120, 0, 140)

BREITE = 300
HOEHE = 3/5 * BREITE / 6

flagge = ... # TODO

show_graphic(flagge)
```

```
flagge = empty_graphic()
for farbe in [ROT, ORANGE, GELB, GRUEN, BLAU, VIOLETT]:
    flagge = above(flagge, rectangle(BREITE, HOEHE, farbe))
```

Figure 11.12. Starter code for students (left) and reference solution (right) to create a rainbow flag.

11.7.3.5   He does not use the web platform but still adopts the Toolbox approach

Charles does not use the PyTamaro Web platform but the educational IDE Thonny [12], which works in many operating systems. Other teachers in his school also use Thonny.

The IDE is tightly integrated with the exam system used by the school. Charles's exam involve some online coding, in the restricted environment, and some questions that ask students to spot problems or explain the code.

Even though Charles does not use the online platform, he is still adopting the approach of the Toolbox of Functions offline. Students are instructed to create a `toolbox.py` file to store their functions and later import them when needed.

> **Charles:** At the beginning I think the largest problem is the file system. [...] They're not used to file systems anymore, so the knowledge that this has to be placed inside some... [folder] was difficult, I think. But once that got sorted out, it worked pretty well. Well, what they do now, they keep using their Toolbox and that's pretty nice I think. I think it helps understanding those things. So I think this works out pretty well, except those file system challenges.

Charles reports bringing up explicitly in his own teaching that the personal functions defined by students and saved in the Toolbox are no different than the ones coming from the PyTamaro library:

Interviewer: When they do an import from `pytamaro` and then there is also an import from `toolbox`, can they see that it's sort of the same thing?

Charles: I think they can see that... That's hard to test, if they can really see that. But what I experienced talking with them, or at least that's something that I try to highlight... that it's basically the same concept. If someone wrote a file called `pytamaro`, or a library, a bit more complex, it's essentially the same thing. And now we can use that library, that someone else created, and they are free to use their own library [...] I hope at least that they can see similarity here.

11.7.3.6 He uses and praises the Judicious documentation system

While Charles does not use the PyTamaro Web platform as a programming environment, he uses the documentation that is also offered on the platform. The Judicious system (Chapter 9) presents functions from the Python standard library (including the built-in ones) and functions from PyTamaro with the same uniform layout. Charles recommended the Judicious documentation also for his students in the 10th grade, when they use functions from the `random` library to develop a game.

Charles: What I really like is this documentation [that is] now expanding. You showed me a preliminary version about a year ago already [...] I mean, that's really cool. Even for built-in functions, these examples, and visualizations [of the examples], etc... I think that really helped.

11.7.3.7 PyTamaro materials emphasize functions but ignore interactivity

When asked to reflect on the differences between his PyTamaro materials and the turtle-based ones used by his colleagues, Charles brings up function definition as a concept that is emphasized much more. Conversely, he notes the lack of `while` loops and interactive programs with PyTamaro:

Charles: Functions is obviously emphasized a lot more in the PyTamaro curriculum. That's something that's really... I mean, I'm teaching these STEM students as well at the beginning of [9th grade], and I already can spot the difference there regarding functions. That's actually not so intensely covered in the turtle one. But in the meantime, randomness [...] I don't cover that in PyTamaro, and they do it with turtle. I think `while` loops, that's also something I'm not covering.

Interviewer: What do they use a `while` loop for?

Charles: When some turtle moves while some condition is not set, it

continues to move, and then [...] if it vanishes from the window [...] it returns to the center of the screen. [...] `while` is actually pretty common there.

Some colleagues focus on Python's `input` function to create interactive programs:

> **Charles:** Oh and of course the textual input thing. That's something that is not covered in PyTamaro, and they use it also to basically control images generated by the turtle, so that you can input what kind of shape you want and then it creates the things.
>
> **Interviewer:** But in principle you could do the same in your PyTamaro curriculum, right?
>
> **Charles:** Yep, absolutely, yeah.
>
> **Interviewer:** Why did you not do that then?
>
> **Charles:** I think time restriction, because I wanted to focus on these other concepts, and I think they actually need some more time [...] I'm basically running out of time.

11.7.3.8   Complex features are shown to students at the beginning

The introductory material served as a first exposure to Python's syntax, but it also exposed students to some non-trivial features of Python:

> **Interviewer:** [This material] is showing off a little bit of `print`'s features, the fact that you can pass like multiple arguments and you have also optional line terminators and separators. And then also a bit of arithmetic and a little bit of string manipulation... there is also the 'times' operator to 'multiply' a string and repeat it. I was wondering whether you felt that this was working with the students or whether it felt a little bit too much for this `print` function.
>
> **Charles:** I think this actually felt a bit too much for the `print` function. However one of the goals was that they can see that a comma can separate arguments. And I think this is pretty important for later. But maybe it might be a better place to actually do that with PyTamaro instead of the `print` function.

While using functions from multiple domains can actually be positive to avoid that students learn only in one domain, the complexity of the `print` function could make it harder to grasp the general concept of a function.

> **Interviewer:** It's a very powerful function, in some sense. There were these optional arguments, although you introduce them gradually... but

there is also the fact that it's one of these functions that can support a variable number of arguments, and that's sort of a special thing... [...] do you think that they see the connection between, say, `print` as a function and other functions [...] like `sqrt` and `rectangle`. Or do they have a feeling that `print` is something special?

Charles: Oh! That's a nice question! What I try is that they don't see the `print` function as so special, even though [...] it's a kind of impure function that does weird things, but it's still a function. So I'd rather like them to think that it's just a function as the ones they create themselves. I try to talk about that quite often.

Based on his students' experience, Charles does not consider optional parameters a big problem:

Charles: I think the optional arguments... that's something that is pretty intuitive for them. Well, because they see it with the `print` function first. I mean, even the `show_graphic` function in PyTamaro has optional arguments. So I think that's not such a difficult concept.

11.7.4 On the student experience

11.7.4.1 Students feel unconstrained in PyTamaro-based projects

Charles feels that PyTamaro leaves ample room to his students to creatively explore programming, comparatively more than turtle:

Interviewer: Do you think they have enough room with PyTamaro to explore a little bit of stuff on their own... or do they feel heavily constrained in what they can do? What's your feeling?

Charles: It's a strength. I think it was harder before when I did teach the turtle curriculum, then it was like 'oh they had to do these things', and now they go off and create new stuff. So actually it's a positive thing. They try new experimental things and then some of them of course discover the PyTamaro [web] platform and all these activities and they try to do some of them. They feel rather unrestricted.

11.7.4.2 Game programming in the 10th grade without PyTamaro requires complex code

Students in the 10th grade develop a game with Pygame, following a tutorial prepared by other teachers at Charles's school to develop the Snake game. That has positive aspects:

> **Charles:** What I really like is that they learn to think about that main game loop, that everything important happens there, things are prepared and in the end drawn to the screen, and then there is another frame and new things happen. And depending on whether you click somewhere or whether some rectangles touch, something else happens, etc. So I really like that they get a better understanding of that kind of logic with that project.

But on the negative side:

> **Charles:** There is quite a lot of code that needs to be written before anything happens. There are quite a lot of tutorials online for them to choose [...] and then pretty much all of those tutorials quickly introduce object-oriented programming, which [we] didn't look at so far. And then the challenge is like, how much did they actually code themselves and how much is just copy-paste from tutorial, and how much does that help their understanding? And I mean, now these large language models, they just program your Pygame game in a few seconds.

Charles advices his students to use the powerful language models as a coach and ask them to suggest ways to improve it, instead of having them write code from scratch. Interestingly, he reports that the harsh reality of programming, where one little mistake causes the program not to run correctly, triggers in the students the realization that the models are not omniscient:

> **Charles:** Interestingly, this year I have quite a few students that are shocked how bad it is the language model they're using. [...] Usually they use it all the time, everywhere they can, in languages mostly, [and] history [...], they use it like everywhere. And they're totally exhilarated because it's so nice and they don't have to do anything anymore. And because it does everything so well. And suddenly they see that 'oh, it's not perfect', it can program, it can explain, but some things are just... it cannot do that, and they can do it better. And I think that's a pretty nice thing as well.
>
> **Interviewer:** Interesting. I wonder whether that's because [in language classes] they can't really tell the difference between when it's working properly and when it's not...
>
> **Charles:** Yeah, exactly. It's probably that, yeah. And suddenly, because the program doesn't run, they can see that it doesn't work. Whereas in languages, I mean 'it sounds OK, so it's probably right'.

In Charles's experience, the models are unable to produce working programs with Py-Tamaro:

> **Charles:** It doesn't work. It doesn't know PyTamaro yet [...], whereas it knows turtle.

Charles reports that given that his PyTamaro curriculum is quite novel, students do not come to him with code that they do not understand:

> **Charles:** In the game project of course that happens. But they don't pretend that it's their code, because from the beginning I actually motivate them to use help, the Internet, and even LLMs. They're actually quite transparent with that. At least they try to understand. I think that's the key. Actually, that's what I want, because not all of them will end up as programmers, but probably something else.

11.7.4.3 Students do not always see why one should define functions

PyTamaro's approach offers a minimalistic API to encourage students to define their own functions early on. Charles notes that this could help:

> **Charles:** If it's done in this way where they have early on that motivation that it actually helping them with complexity to define functions, that helps and makes the concept less difficult.

But he also reports that students need to be explicitly asked to define functions, otherwise the primary goal of drawing the graphic takes over:

> **Charles:** However, if I go on, after the function [...] if I just look at other things, they usually still ignore the functions and just write the code without functions to quickly generate the graphics. So actually, if they just want to have a graphic without me specifically demanding a function, they ignore the function. They're not like... naturally starting a new graphic with a function. They first try to do the image and then they transform it into a function. So it's still kind of unnatural for them.

This post-hoc definition of functions is indeed not very useful, because an entire graphics will likely not be reused again. One may still define a function to change certain parameters, such as the size, but the real benefits come when one can reuse a previously-defined function to program bigger graphics. Writing individual functions also allows one to test whether a part of their program produces the intended result, before integrating it with other parts, which as a whole become harder to understand and possibly debug.

Interviewer: I guess the tasks that they are asked to solve they are not that big. So maybe that's possibly a reason why... [they are not defining functions for intermediate parts].

Charles: I think it starts at the 'rolling eyes' activity. That's one where I think they see... where different sub-functions, like individual functions, help getting the 'rolling eyes' to work.

Charles is referring to an activity in which students program an animation. Each frame shows a pair of eyes at a slightly different angle. In addition to the function that generates each frame, it is sensible to also decompose each frame. Figure 11.13 shows the starting code Charles created for that activity.

```python
from pytamaro import *
from toolbox import circle

def one_eye(angle: int) -> Graphic:
    return ... #TODO

def two_eyes(angle: int) -> Graphic:
    eyes = beside(one_eye(angle), one_eye(angle))
    w, h = graphic_width(eyes), graphic_height(eyes) # Die Breite und Höhe des Augenpaars
    return overlay(eyes, rectangle(w*1.1, h*1.1, rgb_color(82, 126, 176))) # Für den Hintergrund

eyes = ... #TODO

show_animation(eyes)
```

Figure 11.13. Starting code for an activity to create a rolling eye animation. Functions have type annotations.

11.7.4.4 Type annotations are perceived as comments

As Figure 11.13 shows, Charles normally adds type annotation to the functions he defines. He is unsure about their effectiveness:

Charles: It's there and I tell them that it's there, but I tell them that it's optional and they can also just ignore it. I tried to really introduce it last year and I'm not really sure how well it worked, so this time I'm basically not deleting it, but I tell them that this is there to help them understand what's actually the thing, but that they can just ignore it.

Interviewer: Do you think it helps them? There is an `eye` function with an `angle` parameter and then that `angle` is marked as `int`. Do you think it helps them to see how to use that function, how to call that function?

Charles: I think if it's in an example that I provide them, they can see it and 'oh yeah, it makes sense'. But I think they never think of it on their

own, to use it on their own. It's like basically like a comment. And it's
actually quite similar to comments.

Given that Python environments do not always come with an external type checker,
Charles's observations on the role of type annotations make sense. At the time of the
interview, Thonny integrated mypy[1] as a type checker, but possible warnings were only
shown in a separate panel named 'Assistant'. The PyTamaro Web platform also did
not yet support static type checking, but in late summer 2025 the client-side analyzer
(Section 7.5) was extended to include ty[2], a new type checker, which made it possible
to show type errors directly on the web platform when types do not match. This new
feature should offer a more compelling case for teachers to add type annotations, as
teachers can demonstrate, and not just promise, tangible early feedback when using
types.

11.8 The case of Dorothy

11.8.1 This is Dorothy's context

11.8.1.1 She recently learned programming in the retraining program

Dorothy has been a biology teacher for eight years. She started teaching informatics
as an additional subject roughly one and a half years ago. She learned programming
through the Swiss retraining program for existing teachers. There, she used Java for
the core programming courses, whereas Algorithms and Data Structures was offered
using Python.

11.8.1.2 She uses PyTamaro in the 9th grade with uninterested students

At Dorothy's school, students attend a "media and informatics" class in their 7th and
8th grade, with a focus on ICT. Students then have a proper informatics course in
the 9th and the 11th grade, with two lessons per week. Last year, Dorothy taught in
the 11th grade (not using PyTamaro, covering a bit of "object-oriented programming"
with examples that were mostly using numbers and strings). In the current school year,
she is also teaching in the 9th grade, due to constraints on the number of informatics
teachers.

   The curriculum at her school has been changing over the last years, with the current
version that reserves a little bit more than half a year in the 9th grade for programming.
A second programming part comes in the 11th grade.

---

[1] https://www.mypy-lang.org/
[2] https://github.com/astral-sh/ty

Dorothy reports many students being generally passive and not interested in the content. 11th graders even demanded to cover more ICT, as they saw that as possibly more useful for their final project in the 12th grade. Most male students choose "biology and chemistry" as their major, while most females follow an artistic path.

The new incoming students are now bringing their own device for informatics classes.

11.8.1.3   With help, she managed to give feedback to two PyTamaro programs

First program   Dorothy immediately saw a problem with the program shown in Listing 23:

> **Dorothy:** I would say it's not really thought about, because they twice used `arm`. That's actually one of my exam questions [...] 'look at this code and tell me why it's not really well written' and then I want to hear from them that the same variable is used twice.

She then proceeded with a somewhat confusing explanation of her exam question. When asked to rephrase her thoughts, she seemed to be focused on the possibility of a misleading name, more than the actual mutation of a variable:

> **Interviewer:** I'm not sure I got the point. So what would you say it's the problem for this [...] there is the `arm` variable defined twice, but what is the problem? This code seems to work.
> **Dorothy:** Yeah, it works. I mean if we are... it's a beginners' class, right? Then it's not really good. I tell them always to not have the same variable twice, so that they [are not] gonna get confused.
> **Interviewer:** So you think the problem is mainly because it's confusing for the students to have an inappropriate name.
> **Dorothy:** Yes, yes. But it would also be a good example to explain them or to get rid of a misconception. [...] First of all, variables can hold other values [...], and on the other hand, you can show them as well.

Second program   Before actual feedback, Dorothy noticed that the parameterization shown in the code of Listing 24 is not easy for students at the beginning:

> **Dorothy:** First of all, it's quite a big step already for them to have kind of parameterized the code, because they take `diameter` at the top and just have to change it once if they want to change the size.

To facilitate this process, she exposes her students to code similar to this from the beginning, to make it digestible over a longer period of time.

She did not seem to notice the code duplication, until prompted:

> **Dorothy:** I would say it's a nice code.
> **Interviewer:** I see there is `overlay(small_black, big_green)` twice. Would you change that somehow?
> **Dorothy:** Ah, yeah. True. That could be done. Just write `eye` and then `beside(eye, eye)`.

Immediately, she realized that she also uses this pattern with her students, to teach them that they can use the same variable multiple times within an expression. She shared an excerpt of code with two lines, `auge1 = ellipse(200, 60, red)` and `auge2 = ellipse (200, 60, red)`.

> **Dorothy:** I asked a question about this. It's 'eye 1', 'eye 2', and then take them beside, and I wanted them to reason about that... You just need one eye, and place it beside [itself], so that you don't need code duplication. This is code duplication. That's what I wanted to hear from them, and that would be the answer to your question as well.

11.8.2 On the choice of adopting PyTamaro

11.8.2.1 She saw the value of PyTamaro during the training program

At Dorothy's school, a more experienced teacher created publicly available materials, and a new teacher who is completing the retraining program also adopted those materials. She is wary of that material:

> **Dorothy:** But I cannot deal with it because it's too mathematical... I mean, I understand it completely, but I don't like to have my students be bored about mathematics all the time. And I think they're afraid of informatics to be another subject, which is too comparable to mathematics. I know from some students that they really were afraid of the subject because they thought it's too comparable to mathematics.

This was a key reason for her to motivate students with an approach that had a graphical output. Turtle graphics would also have satisfied this requirement. Some other schools across Switzerland are known to adopt a textbook to guide teachers in using the turtle approach. Dorothy confirmed that she saw that textbook, but she was not compelled:

> **Dorothy:** During my studies, when they showed us these books and they also distributed them freely, I was like 'Oh yeah, I can have a look at it' [...] For the 'informatics didactics' courses, I did one or two projects with

> Turtle Graphics, which was fine but... I mean, you have always to know
> where your turtle sits and where it goes next. So it can also be confusing
> to students as well. [...] I like this stuff that [prof. Hauswirth] showed
> us during one course [...]

Dorothy got exposed to PyTamaro in a course on programming languages concepts during her training. She liked the exercises she was asked to complete, and decided that it could be a good approach also for her students.

11.8.3   On the teaching materials

11.8.3.1   Here is an overview

Dorothy developed two curricula on the PyTamaro Web platform. The first curriculum serves as an introduction to programming with PyTamaro. Students are required to program simple graphics with primitives and combinators, and learn about problem decomposition, variables, operators, types, and using functions. The second curriculum is focused on students defining their own custom functions, but also explores how to create more complex graphics which require pinning with PyTamaro.

In addition to the two curricula, Dorothy also created slides that she uses to drive her lessons. She independently developed most of the slides, while a minority of them are heavily inspired from a slide deck we used during a summer training course.

She still does not have activities to teach loops, but she intends to cover them with PyTamaro-based examples who are currently being developed by a colleague.

11.8.3.2   Programming concepts are introduced using multiple domains

Dorothy's materials mostly use PyTamaro-based examples to introduce concepts. Still, a few of the examples are in different domain, such as an example of defining a function to apply the compound interest formula to compute the amount of money after a given number of years. In that exercise, she reports that her goal is to start from the mathematical formula, show to her students similarities and differences (cf. Section 5.5) when different values are "plugged in", and finally turn those differences into parameters.

Working within (the basics of) the financial domain is somewhat conflicting with the goal of not using mathematical examples. Dorothy justifies her choice on the grounds that she is worried that students conflate the library, the language, and the actual programming concepts:

> **Dorothy:** I wanted to have them the comparison between Python and Py-
> Tamaro because... they also asked me in the defense [of my final project]

> last week: 'Why would you explain all the concepts with PyTamaro'? Are there going to be students who are questioning that? And they're going to go out and tell all their families, friends: 'oh, we are just programming with PyTamaro!'. [The professor at my defense] was concerned that they won't see that they're programming Python.

Dorothy also wants her students to have the same background as the ones taking classes with other teachers, who teach using different approaches, to be equally prepared for the next school years. This also extends to the IDE: Dorothy mainly uses the web platform, for which she has developed activities and curricula, but to be aligned with the other classes, she wants to show "mathematical examples" in IDLE[3], the IDE used by the other teachers.

She remarks that using multiple environments also helps students to distinguish between the editor and the language:

> **Dorothy:** One statement of a student was like: 'oh, it's so great that Python is colored'. For example, `def` is orange [...] within IDLE. But [this student] was like 'Oh, it's Python', but it's not Python that is colored that way, it's IDLE that is giving the color, so they're confused if they have just one IDE.

11.8.3.3   Some function definitions are more subprograms than abstractions of expressions

Figure 11.14 shows a slide that Dorothy created to teach her students how to define functions.

Three increasingly more general versions of the same functions are defined.

> **Dorothy:** They had the exercise to write a `quadrat` function. And then I showed them [these] three and asked them 'which one is nearest to your code', to show them that we can abstract a little more. I mean, the first one is not really abstract.

A confusing aspect of Figure 11.14 is perhaps that the function first declares a variable with almost the same name as the function, and then returns that variable, as opposed to returning directly the expression. Dorothy may have internalized this pattern from other, longer examples during the training courses:

> **Dorothy:** I saw it like this... the programs were longer, but [an instructor] used for example `my_variable` and then he returned `my_variable`. That's why I used another variable and then returned it. That was one of

---

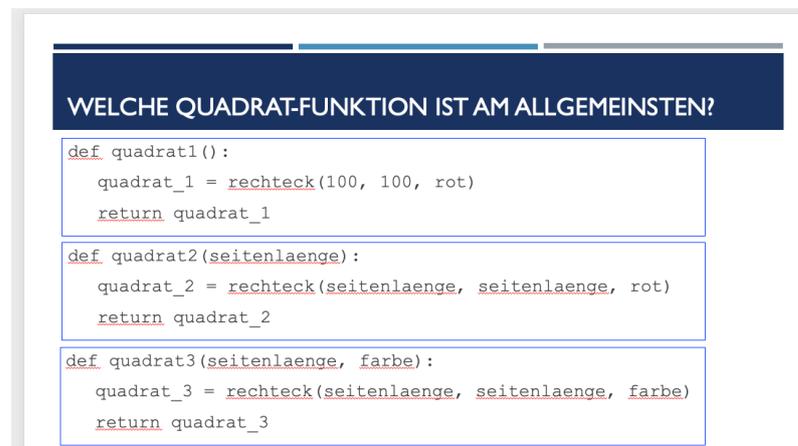[3] https://docs.python.org/3/library/idle.html

Figure 11.14. A slide showing three increasingly generic versions of a function to draw a square.

the really first programs using the definitions of functions.

Interviewer: I see... one aspect is that you have the variable, and the other aspect is that the variable is named very similarly to the function. I thought that may be potentially confusing for the students, because they kind of conflate the variable name with the function name. And maybe they want to call the function using the [name with an] underscore, which is actually the name of the variable inside. [But] given that this is so abstract, it's hard to find a good name for those variables inside.

Dorothy: Yes, it would be better [to] just return, but since it's one of the starting functions I thought it's better for them to [understand that] they just return one thing, not an entire line of...

Dorothy did not seem convinced that it may be advantageous to introduce functions as abstractions over expressions. Instead, she preferred to return an atomic expression and expose students to the possibility of having multiple statements inside a function body, before returning.

11.8.3.4 She uses the Toolbox approach on the web platform, but not offline

Dorothy leverages the Toolbox (Chapter 8) as implemented on the PyTamaro Web platform. She motivates the need for it with the slide shown in Figure 11.15. The translated text recites: "The PyTamaro library is not very large. This requires more creativity and programming. Which in turn boosts your programming skills". Dorothy has thus internalized the minimalistic idea of PyTamaro.

The idea of a Toolbox is general, but Dorothy only seems to use it with PyTamaro
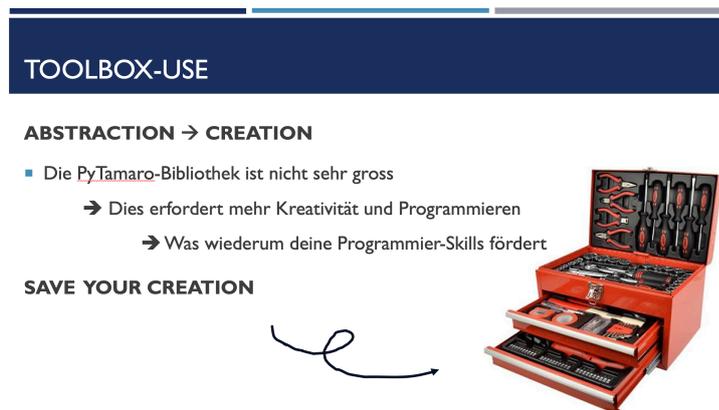
Figure 11.15. A slide motivating the need for saving functions to the Toolbox.

functions. She justifies this choice on the grounds that she moves to a standalone IDE (IDLE) when working on programs that do not require PyTamaro. She had not considered applying the same concept across multiple files when working offline:

> **Dorothy:** I didn't think about it. But it would [allow] something we cannot do in PyTamaro [Web]... for example, `input`, or just get something from another [...] file.

11.8.3.5 She uses TamaroCards to introduce PyTamaro functions

After a first activity on the web platform that introduces the idea of an algorithm, with the analogy of a "recipe", Dorothy explains functions with TamaroCards (Chapter 6). Figure 11.16 is a photograph she took during an actual lesson. She printed huge cards to represent some of the PyTamaro functions to produce graphics, and created the corresponding shapes with cardboard. Below some of the cards, she wrote the corresponding Python expression with chalk (e.g., `ellipse(30, 30, gelb)` under the `ellipse` card.

Students are expected to replicate the same exercise on a smaller scale with regular-sized cards on a sheet of paper.

> **Dorothy:** In the second week I did a group exercise where they had to arrange the primitive functions and reason about what [happens] when I put this value and that value and the third value, it is a red, or is it a big or a small rectangle. That's when they get used to the primitive graphics.

Students then implement these small fragments of code in an activity on the platform. Each student writes code individually, but they can also work in pairs so that both students write code on their device at the same time. From time to time, she demonstrates
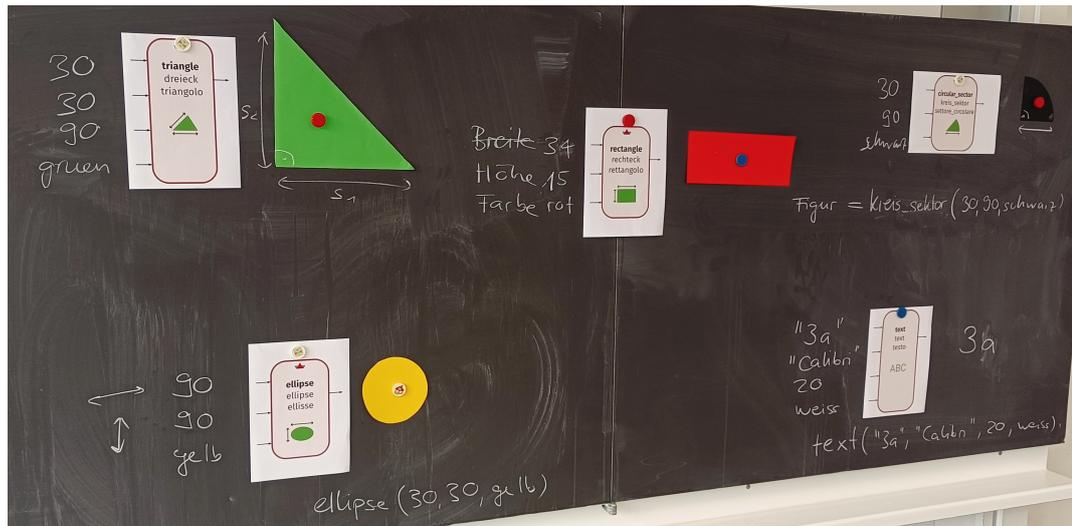
Figure 11.16. Showing the five key PyTamaro primitive graphics using TamaroCards on a blackboard. Written under some cards, the corresponding expression in Python.

some activities by writing code live.

Afterwards, she arranges the cards and connects them to form larger expressions. This illustrates the process of composition, after having decomposed the desired graphic.

11.8.3.6 She uses type annotations extensively

Dorothy frequently adds type annotations to variables, making their type explicit. The code excerpt in Figure 11.17 was written by her and uses the built-in type `int` and the PyTamaro type `Farbe` (German for `Color`).

```
34
35   roteZahl: int = 90
36   grueneZahl: int = 120
37   blaueZahl: int = 240
38
39   farbe1: Farbe = rgb_farbe(roteZahl, grueneZahl, blaueZahl)
40
```

Figure 11.17. An excerpt of code from an activity using type annotations.

Dorothy's first motivation seems to be that students should be exposed to types because there exist statically-typed programming languages that require explicit type annotations, such as Java:

> **Interviewer:** Do you think it helps [showing the types]?
>
> **Dorothy:** Mmh, if it's not helping right now, it's gonna help for the future, I think. Sometimes I tell them 'Python is not the one language, you can also code in different languages' and so I do the comparison between Java and Python and tell them look there you have to be aware what variable stores what value [...]

She also uses small examples to convince students that types matter, even though her demonstrations do not involve static type checking:

> **Dorothy:** In the terminal, we try what is it possible to divide... `floats`? Is it possible to add a number to a string, and there is where we use [types] as well.
>
> **Interviewer:** They see that some operations reveal type incompatibilities... so you have to be aware of types.
>
> **Dorothy:** Exactly, but still Python won't tell which data type you have to put it [as an operand], you can actually put anything, but it might not be able to calculate because of the type.

In a discussion of types in her materials, Dorothy also mentions some Java-specific types. She does not expect her students to have been already exposed to Java, but likes to leave cliffhangers:

> **Dorothy:** Sometimes I like to just place a word or a comparison and tell them 'look, maybe the ones of you who are going to be more into computer sciences... they're going to see that'.

11.8.3.7   Variables are sometimes mutable and sometimes immutable

In an activity that introduces variables, Dorothy borrows an illustration created by a third party to show how a variable works, with a value being placed into a paper box. The illustrative examples are all within the domain of mathematics, showing the numerical content of a variable repeatedly modified.

Surprisingly, even within the same activity that is supposed to introduce variables there are also PyTamaro-based examples, but those do not perform mutation. This suggests a difference between how a typical PyTamaro-based program is written and all the others. Her somewhat generic answer suggests that she finds it challenging to reconcile mutable variables, immutable variables, and constants:

> **Interviewer:** Do you use also with PyTamaro [examples] variables that change, or is that with PyTamaro you mostly use variables that don't

change, so you treat them more as constants?

Dorothy: Mmmh... Mainly as constants, I think. Yeah.

11.8.3.8   Her activities favor shallowly nested expressions

Many of Dorothy's activities contain shallowly nested expressions.

> **Interviewer:** I guess that's sort of a choice... maybe because you believe it's harder to understand the code there when it's nested, maybe it's even you personal experience...
>
> **Dorothy:** Maybe because I struggled as well in the beginning? [*Smiles*.] And I see the 11th graders struggling. They don't get the code which is nested too much. I think since I [now] start from the beginning with the 9th graders, I could also in the end do more nested things.
>
> **Interviewer:** I think there is a point to be made that it's hard to trace deeply nested expressions. It's totally valid.

However, she mentions recommending to some groups who already did a great job in solving an activity that nesting, with a properly indented code, could improve their solution. In a perhaps excessively self-critical way, she realized that showing students how they can organize nested code could help them understand it better.

> **Dorothy:** Sometimes they ask me 'how can I do it better?' [...] and I was like 'Yep, just nest it'. Then you do it with an enter [to indent]. So you go to the next line and then you see like how it's gonna be nested more, always to the right.
>
> **Interviewer:** I see, if they indent properly, that would help them.
>
> **Dorothy:** And some of them were like 'oh, that's great... we can really use it because then we see how it's tracked' [...]
>
> **Interviewer:** So maybe it's something that they didn't see at the beginning, they didn't realize that it was possible to do [it] like that. You can actually break stuff over multiple lines.
>
> **Dorothy:** Yeah. For sure it's also my fault, my liking or not liking [of nesting].

11.8.3.9   She motivates some forms of abstractions with similarities and differences

In an activity, Dorothy introduces what she called "parametrization": avoiding to hard-code numbers and replace them with a symbolic name. Only later in the course, she uses a metaphor learned during the course about "similarities and differences" (Section 5.5), the process one can use to find almost-identical code and abstract it as a function, in which the similar parts become the body of the function, and the differences become the parameters.

Finding repeated numbers and introducing an abstraction using a constant can be an excellent way to expose students to the process when the code is still manageable:

> **Dorothy:** For the defense [of my final project in the retraining program], last week I read a little bit more about parameters and variables and I was like 'oh actually it's not correct to have it called parameterization' [...] because it's still a variable. It's rather maybe a global variable, if you want to say so. Your question is good because it's not thought over that I [do] not use the comparison or the similarities and differences at that point. I wanted to show them already [...] now [that we have] a small code, code that is not really difficult... [...] It's always the same number. [The process is:] Look for the same numbers. Which number can we exchange by just the word? Yeah, it is similarities. You're right.

From this process of abstraction over simple hard-coded numbers, one could move to teach abstraction by defining functions, as discussed in Section 5.5.2.

11.8.3.10   Her materials introduce function definition earlier than her colleagues'

> **Dorothy:** At least the timing is different, so I introduce functions a lot earlier. And functions is the thing that comes last in the 9th grade for the other teachers. They stick to the procedural, sequential programming. I think that before lists, they do branching, and afterwards lists, and then loops, and then functions.

For Dorothy, instead, functions come first, starting with their usage and then their definition. She emphasizes that she would like her students to acquire the same competencies as the other students, as they might end up in another class with other students in the 11th grade.

11.8.3.11   Errors are discussed early on

Early on in her slides, Dorothy presents to her students some typical common errors made by beginners using PyTamaro-based examples. A first slide shows a `NameError` that occurs when students use a function without importing it, a second slide shows an error that results from a function being called without the argument, and finally a third slide, which is shown in Figure 11.18, illustrates a function called with a argument of the wrong type. The `TypeError` signals the attempt to show a color, instead of a graphic.
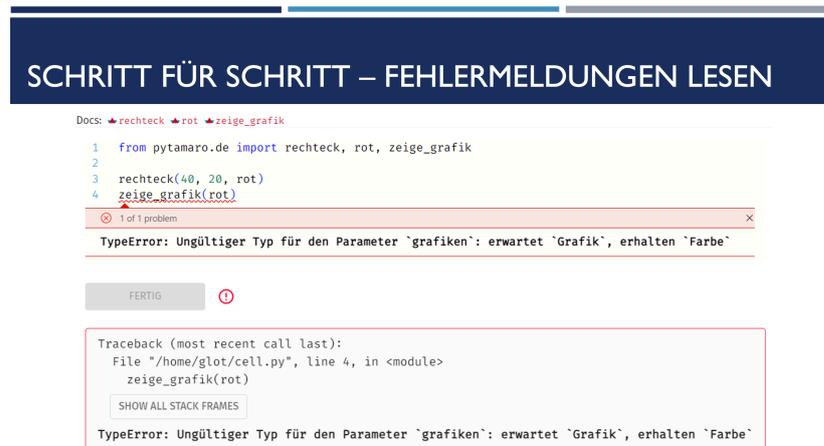
Figure 11.18. A slide showing an error message with a screenshot taken from the PyTamaro Web platform.

> **Dorothy:** I actually planned this after having them always struggle with errors. They were always asking 'but it's not working here', 'it's red', 'it's error', and then I was like 'yeah, read the error, just read it'. 'What does it say?', 'What could you do about it?'. And sometimes they got it [...]
>
> **Interviewer:** Do you think it helps showing the errors?
>
> **Dorothy:** At least to reason about them. I told them 'just read all the... this part, and that part, and then try to figure out where could be the problem'.

As Figure 11.18 shows, Dorothy opted for the German API of the library. She hopes that this can get them up to speed a little bit faster at the beginning, but she admits that she could have also used the English API and students would probably do just fine.

11.8.4 On the student experience

11.8.4.1 Students use functions easily but need help to define them

Dorothy reports heavily relying on the notation of TamaroCards, which shows functions as 'red boxes', to explain what a function is and how one can use it. Her students seem to have no issues with using functions:

> **Dorothy:** Using [functions] is quite fine, actually. With these... I call them 'red boxes', which is actually your graphical version. They take it. You can say 'something is happening inside this red box', we can decide

what it's gonna be worked on, so we put in arguments and something comes out. 'What comes out?' It's a graphic. That's quite easy for them to get it.

Defining functions, on the other hand, requires more support:

> **Dorothy:** I think they just need a good structure. And as you saw these little code [snippets]: do `def`, colon, and then write some code, [and then] `return`. They get it quite fast I think.
> **Interviewer:** I see...
> **Dorothy:** Except of parameterization. [*Smiles*.] That's the biggest deal. They sometimes still forget: 'Ah, I could also do the color', [so] that we can choose the color... They just think about numbers, but it's fine.

When exercises are less guided, students still tend to write big functions. Dorothy reports, for example, that for the final project some students wrote a single, big, parameterized function for their entire graphic.

11.8.4.2 Students get creative in projects and work around the limitations

After 14 weeks of programming, Dorothy has her students work on a project focused on graphics. She partially structures the process for them, to guide them towards defining proper abstractions:

> **Dorothy:** First of all, they sketch it [the graphic] on paper. They have already to reason about 'where can we place parameters?', 'which diameters are gonna be the same?', 'which colors do we like?', 'can we put them as parameters?', 'do we want the colors to be changed?', 'do we want the height to be changed', and stuff like that. They have to write it down. And they have to write down as well what functions they need out of PyTamaro.

This process took roughly three weeks. She reports that some students voluntarily chose to put in some additional time at home as a homework. Overall, Dorothy thinks her students appreciated it:

> **Interviewer:** Did they like this stuff?
> **Dorothy:** [*Nods*.] I thought so! How they behaved... was great in these three lessons.

Overall, Dorothy thinks her students had enough room to explore and draw what they wanted, despite the constraints imposed by the library:

Dorothy: I think they had enough [freedom], and sometimes I could also show them more. For example, they have the transparent color that they might be able to [use to] arrange things through a transparent rectangle.

Some groups ingeniously tried to work around limitations, given that they had still not learned about certain concepts:

Dorothy: Sometimes they came and we could figure out that the code is maybe not really readable, and maybe they could change it like that or this. Two groups actually wanted to do a watch. And they were like: 'how can we arrange it?'. And I was like 'actually, you don't know loops yet, but you could figure out... maybe just put this, and this, and this next to each other, and then take this quarter [of a circle], rotate it half a quarter more, have a half of a watch and then change it again. So they figured out ways which might be complicated in a way, but we can discuss them later on.

11.9   The case of Emil

11.9.1   This is Emil's background

11.9.1.1   He is an experienced biology teacher who recently learned programming

Emil has been a biology teacher for almost twenty years. He attended the national re-training program to teach informatics, where he learned programming primarily using Java. He is currently in the middle of the third year teaching informatics.

11.9.1.2   He teaches with PyTamaro to students in the 9th grade

At his school, students are first exposed to programming in the 7th grade, but the core part is taught in the 9th grade. As a transition period is currently ongoing: students currently in the 9th grade will be the last cohort that did not attend programming lessons in the 7th grade.

The programming part spans roughly two-thirds of a semester, corresponding to approximately 12 weeks.

His school follows a "Bring Your Own Device" policy, with the requirement that students must have a device equipped with touchscreen and a pencil. For him, using web platforms that can work across different devices and operating systems is essentially a requirement.

11.9.1.3   He gave quick and good feedback on two PyTamaro programs

First program   At first, Emil noticed that the program code in Listing 23 lacked proper imports. Then, he immediately noticed that a variable was being reassigned:

> **Emil:** We have two variables with the same name.
> **Interviewer:** Why would you say that [is a problem]?
> **Emil:** Well, because you're overriding what you just had. In line 3, it's not really helpful to name it `arm` because it's actually a cross already.

Second program   When analyzing Listing 24, Emil quickly pointed out the lack of abstraction:

> **Emil:** In line four and five [...] it's rather repetitive. Instead of saying `overlay(small_black, big_green)` [...] again [...] you could come up maybe with a separate name for that.

When students are ready to define their own functions, Emil suggests that they could define one to create a circle given its diameter. He also stresses the importance of giving appropriate names to abstractions, something his students at times struggle with:

> **Emil:** We could have a function with a parameter called `diameter`. But that depends on what the students know already. Would be nicer maybe to have a function called `circle` instead of `ellipse`. It's helpful to give graphics a name and use the name instead of the full function with all its arguments. That's what I see with my students. Some tend to give names for graphics that absolutely make no sense at all. And then they cannot remember what their names actually stand for.

11.9.2   On the choice of adopting PyTamaro

11.9.2.1   PyTamaro enables him to go beyond turtle graphics

Emil claims that his students come to his lessons having already had some exposure to programming in Python with turtle graphics. PyTamaro, for him, is thus an opportunity to offer them different materials:

> **Emil:**   [My students] already have a background, they had these introductory lessons with TigerJython [doing turtle graphics]. It's about to change, and that's why I decided it would be much better to come up with something different.

Emil created a PyTamaro curriculum as his final project for the training program. The current school year is the first year he is using this curriculum; in the previous two

years, he also used TigerJython [259], a variant of Python often used in Swiss schools in combination with turtle graphics.

Graphics-based exercises can appear too simplistic for some students and look detached from authentic problems that could be solved with programming. Using PyTamaro avoids dealing with certain aspects of programming with turtle graphics, such as figuring out the right angles, but it may introduce other forms of accidental complexity (such as the annoyances related to pinning graphics to compose them, even though Emil does not bring that up):

> **Emil:** I got the feedback, from the better students usually [that Tiger-Jython] is not real programming: 'why are we just drawing with a funny turtle?', 'why are we not doing real programming stuff'.
>
> **Interviewer:** Maybe those students have done some extra courses or projects?
>
> **Emil:** Some of them. What I observed is that most of them are not really good at programming, yet they think they are. They failed in my exams quite hard sometimes. They are able to import funny modules from wherever and produce programs that can do fancy things, but when you ask them details concerning variables or so, they actually have no clue about the basics. They just don't want to learn. If you face them with something that's simple, like turtle graphics output or the code behind it, or also PyTamaro, they tend to get bored on one side, but also frustrated they cannot even come up with solutions. With TigerJython tasks... it was often about coming up with the right angles to turn the turtle from left to right, or in the right direction. And that's what they couldn't do. That's why they got frustrated and said 'that's not real programming'. This was just a handful of pupils who had this type of attitude or opinion.

Emil is the only teacher in his school who is currently using PyTamaro. He has three other colleagues who teach informatics. Two of them are also teaching 9th grade students, but they are not using the same materials as he is using.

Emil reports that his colleagues are following a special curriculum using robotics from ETH Zurich. They got the materials from the university, but those materials still required inputs and adaptations because they had not yet been used in practice.

11.9.3   On the teaching materials

11.9.3.1   Here is an overview

As part of his final project for the retraining course, Emil has developed an extensive curriculum on the PyTamaro Web platform. The curriculum features graphics inspired by biology, such as flowers.

It is divided into five main units. The first unit covers fundamental concepts to introduce programming, Python, and PyTamaro. The second unit is titled "modular programming and decomposition" and discusses how to break down a graphic into smaller graphics and how to define custom functions to abstract. It also shows how to use `pin` to create more complex graphics. The third unit consists of five mandatory activities (and some optional ones) to teach repetition, which is achieved using a `for` loop. The fourth unit covers conditionals, achieved using `if` statements. The fifth and last unit completes the course with ideas for a project.

In addition to the curriculum on the platform, Emil also created a small set of slides.

11.9.3.2   A number of unplugged activities use TamaroCards

Emil uses TamaroCards (Chapter 6) for an unplugged introduction to programming. He asks his students to build a number of graphics of increasing complexity purely with the cards:

> **Emil:** They had to build the Swiss flag graphic using paper first... and then harder graphics. The first task we did together. I told them, okay, that's the way to combine them. That's how to bring an output of another function or combinator, `beside` or `ueber` [German for `above`], as an argument into a parameter of another function or combinator.

A minority of students were reluctant to take part in this activity that they may have considered too childish. However, Emil reports that they admitted in the end that the cards can be a useful bridge to textual programming:

> **Emil:** It worked pretty well for some groups. They could do it in groups of two or three people, and some were highly motivated. They liked the idea of using the cards, the glue... others said 'do we really need to do this?', 'can we not just do the coding part?'. But I'd say in the very end also the latter had to accept that it's hard to program without an introductory part with something on paper.

Emil also reports that to solve the more complex tasks, some students took a piece of paper and tried to draw it on paper first, effectively practicing decomposition.

Emil asked his students to complete four tasks with TamaroCards. He had planned to ask them a fifth one, in which they should have composed a graphic of their choice, but students did not want to use cards anymore: they felt it was enough for them. At that point, Emil encouraged them to write the code but not show the output to their neighbors, and make them guess it. The activity worked well, although some students discovered the limitations in how they could position the graphics:

> **Emil:** They really liked [it]. Some of them tried really complicated stuff. I have a Chinese student. He wanted to build a Chinese flag. Many of them were a bit unhappy that you cannot position the graphics wherever you want. At that point, indeed, students had only seen the basic combinators like `beside` or `above` and had not seen more flexible positioning with `pin`.

In one of the very first lessons, Emil showed to his students the slide reproduced in Figure 11.19. It featured the Judicious documentation for PyTamaro's `ellipse` function. Emil did not seem to make explicit the connection between the cards and the diagram visualized in Judicious. The focus was on the difference between the parameter names and the argument expressions, which are shown in the examples:

> **Emil:** The point here was to repeat the terms parameter and argument. At that point, they mixed up arguments and parameters.
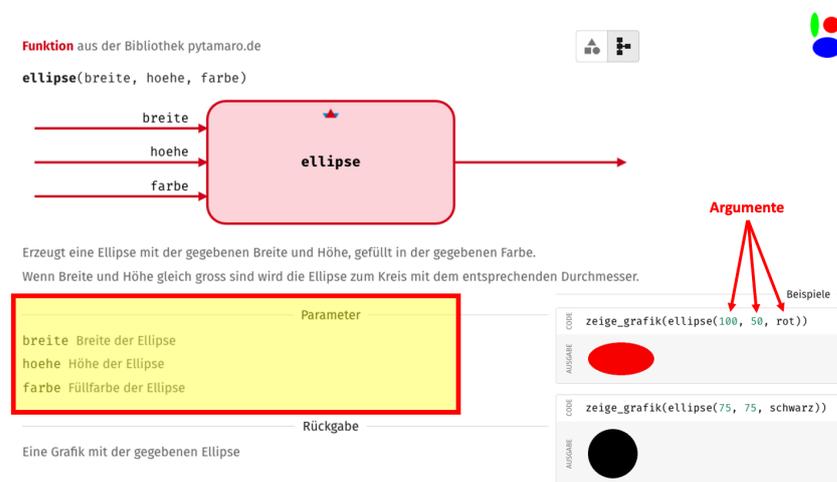

Figure 11.19. A slide created by Emil showing the Judicious documentation for one of PyTamaro's functions, with some annotations.

Overall, Emil claims that his students did not have trouble understanding the notation of the TamaroCards:

> **Emil:** It actually worked pretty well. Of course it's always 80 % are fine, 20 % fail. But in general it worked pretty well, I can say.

However, he is ambivalent on how well the students transferred from the cards to the code:

> **Emil:** That part of the lesson was quite hectic. At least one group did other stuff or didn't want to use the glue, so the cards were moving back and forth, and so there were problems. Other groups used the cards as I told them, they still struggled because they didn't really understand the principles behind it.

He did not seem to tell students explicitly how to turn the cards into syntactically correct Python code, and this may explain part of the difficulties.

To help them, Emil tried an activity in which students had to interpret a given Python code. The whole class was supposed to guess. He admitted that the names were unfortunately rather telling and revealing at times, and thus students probably relied on those names to guess the correct output, instead of accurately tracing the program.

> **Emil:** As soon as these telling names were gone, they were faced with a task where they really have to think about the code. But still it was their very first time where they came across code they haven't seen before, so I was not surprised that it was a hard task for them.

11.9.3.3 Function definition comes early in the curriculum

Emil's curriculum includes the definition of functions before repetitive and conditional computation. He is well-aware of the radicalness of proceeding this way, compared to more established approaches. At the beginning he was fearful, but after having tried his activities with actual students, he sees this as a viable choice:

> **Emil:** For the moment I'm quite happy, but of course that was a big issue right from the beginning. I'd say that's the biggest difference between TigerJython and PyTamaro. I also had discussions with [a university professor]; he said 'you cannot start with functions, that's far too much from the beginning'. But now when you're asking me, I'm quite surprised how well they took it.

11.9.3.4 Decomposition is only discussed using the graphics domain

Emil's materials mostly talked about modularization and decomposition within the domain of graphics. He could not make extensive comparison with decomposition in

other domains, mostly due to time constraints:

> **Emil:** We don't have that many lessons [...] that's the main reason why I had to cut it down to the very basics. It's fantastic to teach it with this graphical output, just as it was with the turtle graphics. Because here you quite easily see what your mistakes are.

As an example, he brings up an activity to draw vertical or horizontal leaves:

> **Emil:** You cannot just switch from `above` to `beside`, because then you see you have still the same ellipse standing upright next to the rectangle standing upright.

During the interview, Emil asked for a recommendation of an example on how to introduce the concept without using PyTamaro. As a common example often used in introductory programming, the bigger problem of computing the solutions of a quadratic equation has as a subproblem computing the discriminant.

11.9.3.5   There is no project due to limited classroom time

In the fifth unit of his curriculum, Emil prepared some more advanced activities meant for those students who are faster than the rest of the class. This additional material is related to biology content.

For the entire class, he has not planned any final project yet, reporting not having enough time for project-based work.

He is free to take this decision as the school has a curriculum which specifies the topics to cover, but not how many lessons per topic. Teachers can therefore be quite independent and free to focus on what they deem most important.

11.9.3.6   All examples use the German API of PyTamaro

Emil uses the German API throughout his materials. He reports making that choice with the intent of helping the students.

However, he admits that sometimes students also see the English API (e.g., on the starting code of the Playground on the PyTamaro Web platform), and if they try to import the German names completing an import statement that uses the English API, this would result in an error.

He is generally unsure about how much the localized API actually helped in practice.

11.9.3.7   Errors are presented at the very beginning

Emil reports having had a better experience with the error messages provided by the TigerJython platform, which features a custom parser to provide specialized error messages for certain common mistakes made by novices [147]. Those messages are also localized.

> **Emil:** My experience is that students get frustrated quite quickly when they don't understand the error. With TigerJython, that's a big plus because their engine gives you quite nicely shortened error messages, often in German. Here students tend to give up or be shocked at the beginning when they see... it's all red, it's all wrong.

Issues with error messages can be exacerbated by notebook-style environments such as the PyTamaro Web platform, where an error in a certain code cell affects all the code coming after, potentially even in different code cells:

> **Emil:** On PyTamaro Web, if you have an error somewhere in line 2 [in an earlier code cell] and you're in a code cell starting from line 20 and you haven't managed to deal with the error in line 2, you still don't get the correct [output].

Emil's observation inspired a new feature on the PyTamaro Web platform. When the traceback for an error refers to a line of code that occurs before the current code cell, a message replaces the error that would be shown (Figure 11.20), informing the student that the source of the problem is in a previous code cell.

Figure 11.20. An error message on the PyTamaro Web platform indicating that the current code cell errors because of an error in one of the previous cells.

Emil reports that his decision to expose students early on to errors seems effective in reducing their fears:

> **Emil:** That's why I wanted to give an introductory chapter saying: errors are completely normal. It's helpful if they don't get frightened. That's actually working pretty well. Some students just don't read, but that's

a general problem. They don't read any sort of instructions. They just
ask me or ask their colleagues, who tell them read what's given in the
instruction. And they still don't do it. They want to have a nice output.

11.9.4   On the student experience

11.9.4.1   Students exhibit creativity with PyTamaro

Despite having created activities on the web platform, Emil still sees the teacher as nec-
essary. He is energized by the new materials, and reports students exploring creatively
with PyTamaro:

> **Emil:** It does really need the teacher. My curriculum is not meant as a
> self-explanatory engine that can be given to the students and let them
> do it for 10 weeks. The experiences I make at the moment with these
> two classes are so valuable, but also energy consuming. Every lesson is a
> new adventure for me too. It's interesting, it's really exciting. I was really
> nervous in the very beginning about how PyTamaro is actually perceived.
> Is it actually something the students like? Do I need to go back and say,
> okay, it doesn't work? But I have to say it does work so far. And the
> students like it. I have some very good ones that come with stuff they
> did at home. Without any need to do with it, just because they were
> interested in a problem and they really tried to come up with a solution.
> That's great.

He was fearing that an approach perceived by some as "advanced" could not work
with his students, as many of them focus on modern or classical languages are not on
mathematics or natural science:

> **Emil:** It seems to work so far. I'm not so sure about the more challenging
> parts coming up... [when the] code pieces are getting bigger.

11.9.4.2   Students embraced the Toolbox approach

The Toolbox is intended as a mechanism for storing commonly needed functions. Not
every function students define has to be part of the Toolbox. However, the first activity
in Emil's curriculum that asks students to define their own function, also asks them to
save it to the Toolbox. Emil feels that the approach worked pretty well:

> **Emil:** We continued with the next activity [...] and at the very end of
> that activity they have to write a function for a square. And not just one,
> but several students ask me: 'shall we save that function to the Toolbox
> as well?'

Indeed, a function to draw a square is an excellent candidate for the Toolbox. Despite that, there is a risk of students conflating the ideas of defining functions and saving them to their personal library. Emil confirms that this happened, but was not a major problem:

> **Emil:** There was one student who asked whether he should save the watermelon function to the Toolbox. I said, well […] no, not really.

In general, Emil's students got acquainted well with what seems a rather smooth process for saving a function to the Toolbox:

> **Emil:** I was quite surprised actually, because it's not that easy, right? You [on the platform] wrote [some instructions as] an introductory part. That's quite clear to me at least: remove unnecessary code, stick to the definition itself, and so on. Of course I showed them how to do it with the `circle` function. We did it together. Maybe that's why it was easy. The problem with this activity is that all the code cells are connected and further down they want to draw a new circle and they don't need to write the import line [from the Toolbox] because the circle function is already imported or written on top.

11.9.4.3 Students can generally deal with nested expression

The code in one of Emil's activities used a single, deeply nested expression to create the graphics of a simple watermelon. Emil structured the starting code, which is shown in Figure 11.21, over many lines, instead of having one very long line. He reported being surprised that many of his students completed the activity relatively quickly.

```
zeige_grafik(
    ueberlagere(
        ueberlagere(
            ueberlagere(
                ueberlagere(
                    ellipse(..., 100, dunkel_gruen),
                    ellipse(..., 100, hell_gruen)
                ),
                ellipse(..., 100, dunkel_gruen)
            ),
            ellipse(..., 100, hell_gruen)
        ),
        ellipse(..., 100, dunkel_gruen)
    )
)
```

Figure 11.21. Deeply nested expression to compose a "watermelon" graphic, with the editor showing indentation guides.

However, a small incidental detail, and not nesting, was puzzling for some:

Emil: [Vertical lines.] That's what confused some of them. 'Why are there some extra vertical lines?' But apart from that, they were 'OK, the code is getting bigger and bigger', you can have it a lot smaller when you use names [for certain subexpressions].

The vertical lines of Figure 11.21 are actually just indentation guides shown by the code editor to help users quickly match indentation levels.

11.9.4.4   The neutral element for graphics is a challenge

With a more advanced class, Emil piloted an activity from the third unit of his curriculum that was supposed to teach repetition using graphics as a domain. The main problem he encountered was with using an empty graphic as the initial value for an accumulator variable:

Emil: Some of them really struggled with the idea of an empty graphic. 'What do we need it for?'. Here it would have been helpful to have a real world example with numbers, maybe that they know a bit better.

However, using an empty graphic is not a strict necessity, but a convenient neutral value. One could, for example, initialize the accumulator to the first graphic, similar to what one can do by initializing the accumulator to the first number of a list when trying to find the list's minimum value.

11.9.4.5   Faster students can explore activities on the web platform

A benefit of using the PyTamaro Web platform, according to Emil, is enabling students who are faster at completing the regular activities he created to discover and work on other, already available PyTamaro-based activities.

11.10   We synthesized findings across cases

Despite the regional and school differences pointed out in Section 11.1, all five cases focus on teachers in Switzerland and the mandatory course in informatics. It is now time to analyze the results of the five cases presented in the previous sections to identify common themes.

Case studies are not meant to provide results that will necessarily generalize to all teaching contexts. Indeed, this multiple-case study revealed an intricate set of choices made by the teachers, that are highly dependent on contingent factors in their local context.

The number of school hours devoted to informatics, the presence of more experienced colleagues, whether students may end up continuing their informatics education with different teachers, the pedagogical approaches traditionally used in a school to teach programming, the major chosen by the students, and the level of programming maturity of the teacher are all factors that affect how PyTamaro is used in practice.

The analysis of the results attempts to abstract over the individual differences to formulate broader claims. This partially satisfies the provocatively bold position of Healy [114], who vehemently protested against sociological studies that attempt to maintain infinite nuance at all costs. Complaining about the ever-growing complexity of modern social theories, Healy mocked prototypical researchers who "call for the contemplation of complexity almost for its own sake or remind everyone that things are subtler than they seem". And continues: "Theory is founded on abstraction, abstraction means throwing away detail for the sake of a bit of generality, and so things in the world are always 'more complicated than that'—for any value of 'that'." [114].

With this in mind, we illustrate the results of our abstraction over the cases, answering the research questions presented in Section 11.4.2.

11.10.1 On the choice of adopting PyTamaro

Our first research question aimed to investigate the factors that drove teachers to adopt PyTamaro. Two key aspects emerged: the need for graphics as a motivating domain, which PyTamaro offers as a novel approach to graphics, and the opportunity enabled by the retraining program to create new teaching materials.

11.10.1.1 Graphics is seen as a motivating domain, and PyTamaro as a novel approach to graphics

High school students come from a variety of different backgrounds: Ada teaches students who choose modern languages and economics as their major (Section 11.5.2.1). Barbara has a class with students who choose music, and another "elite" class of students attending a bilingual program (Section 11.6.1.2). Charles teaches a smaller course to a dozen different classes, out of which at most two are composed of students who choose a scientific major (Section 11.7.1.2). Dorothy's students study either "biology and chemistry" or arts (Section 11.8.1.2).

This degree of diversity stimulated teachers to develop a curriculum that would be appealing to their students. The perception that informatics is too close to mathematics has also been voiced as a concern (Section 11.8.2.1). In this context, graphics is seen as a domain that can elicit the interest of students who, by and large, did not choose to study computer science. The difference is striking with students who choose

a STEM major: Charles notes how the domain of mathematics would probably work well without hiccups for them, but even those students do not dislike working with graphics (Section 11.7.2.1). However, learning to program using the graphics domain can raise concerns of authenticity for a minority of the students, as Emil reports (Section 11.9.2.1).

Turtle graphics is a popular approach to teaching programming within the graphics domain. All teachers who are part of the study have experience with turtle graphics: as students themselves, as teachers, or as having colleagues who are currently teaching with it. For example, Barbara (Section 11.6.1.2) and Charles (Section 11.7.1.2) have colleagues who are teaching with turtle graphics. Turtle graphics is increasingly popular and embedded even in activities students do when they are younger, such as in middle school or in extracurricular activities that may even start from primary school. Some primary and lower-secondary schools in Switzerland include some activities involving programming using Scratch, effectively exposing students to turtle geometry.

In this sense, the choice of adopting PyTamaro is seen as a "fresh" approach that can go beyond what students have already experienced, at least partially (Section 11.9.2.1). The purported conceptual benefits of PyTamaro are exploited by the teachers, but were not mentioned as the key reasons for the adoption. Instead, teachers focused on identifying the problems with the previous approaches. Barbara and Emil noted that students struggled with figuring out the right angles in turtle geometry (Sections 11.6.3.6 and 11.9.2.1). This offers one more piece of evidence that the turtle's principle of "body syntonicity" [194] is not enough to prevent student difficulties with angles [63].

11.10.1.2   Dedicated time during training was essential to develop new teaching materials

The national retraining program was essential for all five teachers to adopt PyTamaro. The program was instrumental in two important ways.

First, in one of the courses that were part of the program, teachers were exposed to Python as a programming language and some of their assignments were based on PyTamaro. This allowed them to familiarize themselves with the library, get exposed to some plausible exercises for beginners, and assess first-hand the level of engagement the approach could generate in their students. All the five teachers had already some experience teaching programming, which helped them to compare the PyTamaro approach with the previous approaches they used. Those approaches were mostly based on materials developed by colleagues at their schools.

Second, the Swiss retraining program required teachers to independently work on a final project. The project is worth 30 ECTS (European Credit Transfer and Accumulation System), formally corresponding to 750–900 hours of work. All five teachers

were unanimous in saying that the "protected" time offered by this project was not only instrumental but essential to create the new materials.

The PyTamaro Web platform offered hundreds of activities and one curriculum aimed at beginners that covers fundamental concepts starting from zero. There is no textbook that aims to cover the entire part of programming of the mandatory computer science courses. This was seen as a significant obstacle for adoption by Barbara (Section 11.6.2). Furthermore, the number of lessons available to each teacher varies considerably: as an example, Ada's materials need to cover more than twice as much as Charles's. For these reasons, teachers felt the need to develop their own materials.

Developing materials was a significant endeavor, but Ada notes that the payoff is significant, especially for teachers like her who are still deepening their programming knowledge (Section 11.5.2.1). The need for this sense of ownership by the teacher over their materials should be taken into account when trying to "impose" tools or pedagogies onto the teachers.

While the degree of autonomy granted by the teachers to their students varies, none of the five teachers who were part of this study developed materials that are supposed to be consumed in complete autonomy by their students. This aligns with the findings of Levy and Ben-Ari [162], who note how the issue of how to keep the "centrality of the teacher" is essential to adopt a teaching innovation.

11.10.2   On the teaching materials

The second research question investigated how teachers translate the principles embodied by the PyTamaro approach into their teaching materials. We found that teachers successfully use PyTamaro to introduce most programming concepts, emphasize functions early on, and adopt the Toolbox approach. On the other hand, problem decomposition is discussed only with respect to graphics, and the transition from immutable to mutable variables is problematic.

11.10.2.1   Teachers use graphics to introduce most programming concepts

One of the key goals of PyTamaro is to enable teachers to introduce programming concepts directly using graphics, instead of presenting graphics as an additional domain to be used for further practice.

With some help, all five interviewed teachers were able to point out the major issues in the two programs we used as a minimalistic form of assessment. This provides us with a basic level of confidence in their answers when it comes to programming.

Especially in the case of teachers with very limited classroom time, such as Charles, programming concepts are introduced using the graphics domain with PyTamaro. A

first lesson introduces the idea of programming and may have students type in and execute their first chunks of Python code without PyTamaro, but this only serves as an initial overview of what it means to program. With the exception of Ada, who is currently not using an unplugged approach, the other four teachers all begin their curricula with unplugged activities using TamaroCards, the tangible notional machine presented in Chapter 6.

Given that some courses are split across two school grades, and that classes can be remixed with students ending up following the second year with a different teacher, a crucial concern for teachers in this situation is to also present the concepts using a different domain. While it is not the primary goal, showing the same concept to students in multiple domains may also help with generalizing the idea they learned and with transferring it to a different domain in the future.

All teachers use PyTamaro to introduce the concept of "custom", "personalized" functions, i.e., the definition of functions. We discuss this specific concept of function definition in the next section.

11.10.2.2   Defining functions is emphasized early on

At the level of the teachers analyzed in this case study, a key operationalization of *abstraction* is the ability to define functions.

All curricula developed by teachers introduce functions early on. This is the most significant difference expressed by the teachers regarding their curricula, compared to what they themselves used to do, or to what their colleagues do in parallel classes.

Ada developed a curriculum on the web platform dedicated to functions and presents faded examples with partially implemented functions to be completed by her students (Section 11.5.3.2). Emil's materials also cover the definition of functions before other topics that often come earlier in popular approaches for introductory programming, such as repetition with loops or conditionals with `if` statements. The same happens in Charles's notes (Section 11.7.3.7) and in Dorothy's curriculum (Section 11.8.3.10).

The idea of recognizing opportunities for abstraction by playing "a game of similarities and differences" (which we presented in Section 5.5) is still not adequately exploited by teachers to *motivate* the need for defining functions. A reference to this process only appears systematically in Barbara's slides, as she borrowed a subset of the slide we authored for a summer school with PyTamaro.

11.10.2.3   The Toolbox approach is widely adopted

The approach of using the Toolbox of Functions, described in Chapter 8, has been adopted by all teachers. The minimalism of the PyTamaro library makes it almost an

imperative to define functions that are later used to program more complex graphics.

Even Charles, who is not using the web platform (Section 11.7.3.5), translated the approach into a "local" setup with multiple files that can be used in a regular IDE.

How to implement the idea outside the web platform is not obvious, however: Dorothy admitted not having thought about that possibility (Section 11.8.3.4).

The analysis of the teaching materials also revealed a suboptimal aspect in how teachers are using the Toolbox. At times, the Toolbox is introduced together with the very first definition of a custom function. It is true that the lack of commonly needed functions such as `square` encourages defining and saving them in the Toolbox as soon as possible, but the act of defining a function and saving it to the Toolbox should not necessarily always go hand in hand. When that happens, teachers like Emil have to fix the misconception, instructing their students later on that not every function needs to be saved in the Toolbox (Section 11.9.4.2).

11.10.2.4   Decomposition is mostly discussed in the domain of graphics

Most teachers present the idea of decomposing a problem into smaller parts as a generic skill. Ada even related this to processes used in literature to analyze a text (Section 11.5.3.4).

All curricula encourage students to think about how to decompose a graphic and program each sub-graphic in isolation. This is also the case for the final projects, which usually consist of a larger graphic. When the problem (drawing a graphic) becomes bigger, the need for decomposition becomes more pressing.

However, discussions of problem decomposition outside the graphic domain remain shallow.

On the one hand, this is an inherent limitation of the PyTamaro approach, which focuses on decomposing a graphic. We argued, using the lens of programming language theory, that this graphical decomposition process directly maps to program (summarized by the motto "the structure of the graphics drives the structure of the program", Section 5.4). But there is likely still a need to observe this process in other domains for students to form appropriate generalizations.

On the other hand, this may be explained with the very limited time available to all the teachers, which prevents them from illustrating and contrasting several examples in multiple domains.

11.10.2.5   Teachers struggle to reconcile the ideas of (im)mutable variables and constants

The concept of a *variable* is ubiquitous in introductory programming. A wide body of literature discusses the difficulties novices face with the concept [287, 244]. The approach encouraged by PyTamaro emphasizes immutable variables, referred to as constants (e.g., TamaroCards such as Figure 6.7 are commonly referred to as constants). Table 11.1 aims to clarify three distinct senses for the broad concept of a variable, using terminology from mathematics and the tradition of "functional" and "imperative" programming.

Mathematics	"Functional" Programming	"Imperative" Programming
Constant	Constant	Constant
Variable	Variable	Immutable Variable
—	Mutable Variable	Variable

Table 11.1. Three distinct senses for the broad concept of *variable* as it is commonly referred to in mathematics, "functional" and "imperative" programming.

In a first sense, we have *constants* that are in some sense the only "true constants": names for values that are supposed to be universal and valid across programs. These constants are commonly found in libraries, which are indeed persistent, reusable abstractions. The `pi` constant from Python's `math` library or the `red` constant from PyTamaro refer to, respectively, the ratio between the circumference of a circle and its diameter, and the pure red color.

At a second level, we have the concept of *variables* as names bound to values in a specific region of the program during a certain program execution. The term "variable" is reasonable, because a name may assume different values (i.e., may *vary*) across different program executions (e.g., a variable `favorite_number` standing for the user's favorite number) and even within a single program execution (e.g., the parameter variable `a` of a hypothetical function `min(a, b)` that is called multiple times to compute the minimum of two numbers). This concept, as adopted in "functional" programming, matches the mathematical concept of variables. Students are used to the fact that a, b, and c in the Pythagorean equation $a^2 + b^2 = c^2$ assume different values. It is not uncommon, for example, to speak of taking the formula and plugging in some numbers to replace the variables. This is no different than the "substitution model" [3] which can be used to evaluate expressions that include variables in this sense.

At the third level, we have *variables* in the sense of named memory locations, as

emphasized by the imperative view of programming. This is also the most common way in which variables are discussed in introductory programming. When the literature speaks of novices' difficulties with variables, it is this sense of variable that is implicitly being used.

To emphasize the difference between the second and the third sense, "functional programmers" tend to explicitly use the attribute *mutable* to designate this kind of variables. Symmetrically, "imperative programmers" use variables without any modifier in the third sense, and talk about *immutable variables* when they are referring to the second sense.

Most programming languages in use today, and especially those commonly used in education, offer by default variables in the third sense when defining a name. Java, for example, requires adding the `final` modifier to explicitly designate a variable as immutable; Python does not make this possible unless one is using rather advanced and cumbersome techniques. Perhaps, it would be sensible to refer to mutable variables as *assignables*, a contraction of *assignable variables*, instead of *variables*, given the confusion the latter can cause.

As things stand, however, nearly all introductory programming classes at the school level present "variables" with the term *variable* and use the third sense above: named memory locations that can be reassigned. The five teachers in this case study took their two programming courses in the retraining program using Java, in which variables are mutable by default. Their colleagues in their schools teach using Python and also adopt this approach. The entire context surrounding them considers variables in the third sense.

However, the existing materials for PyTamaro use the term *constant* to emphasize the second sense discussed above. It is not uncommon to denote this second sense with "constants": the "roles of variables" framework also characterizes this particular use of variables in beginner programs as "constants" [226].

Given this landscape, it is perhaps unsurprising that teachers struggle to make sense of three distinct meanings.

Charles only introduces variables when they are needed, just before covering repetition with loops. His materials properly raise attention to this fact by using a memory diagram (Section 11.7.3.3). Variables contain arrows that point to objects. Assignments update these arrows.

Dorothy uses the "variable as a box" metaphor, and introduces variables rather early in the curriculum, even though they are not necessary to create PyTamaro graphics at that point (Section 11.8.3.7).

Barbara's materials stay close to the original slides we created to teach with PyTamaro and thus consistently use the term *constant* for the first part (Section 11.6.3.4). Before repetition, she also introduces mutation and uses the box metaphor for vari-

ables. In any case, she is aware of at least some of the limitations of that metaphor, such as the fact that a variable can be used multiple times in an expression.

Ada also uses the box metaphor and adopts the term *variable* from the beginning, with the exclusion of the constants defined in the PyTamaro library. Effectively, she only uses the first and the third sense of Table 11.1.

Emil's curriculum just before introducing repetition discusses how, up until that point, the term variable had been avoided deliberately. He introduces it at that moment: there is no visualization, but it is said that a variable can be taught as a container into which you place a value.

If teachers did not use loops to achieve repetition, it would be possible to stick to the second sense of "variables". The transition from the second to the third meaning demands a more elaborate conceptual model of program execution, and should be discussed explicitly and with utmost care.

11.10.3   On the student experience

Finally, the third research question explored the experience of students with PyTamaro, analyzed indirectly through the teachers' comments. We found that defining functions is challenging but doable for students, and that the limitations of PyTamaro do not make students feel excessively restricted in what they can draw.

11.10.3.1   Students are challenged by defining functions, but not overwhelmed

As highlighted in their teaching materials, all teachers introduce the definition of functions earlier than most other curricula (Section 11.10.2.2).

Ada notes how her faded examples help students practice the syntax for function definition, but the more "theoretical" aspects of the concept put off some students (Section 11.5.3.2). Emil's first experience with teaching function definition early on withstood the impact of real-world classroom use, despite the warnings of a university professor (Section 11.9.3.3).

Charles confirms that the minimalism of the PyTamaro library supports a teaching approach where functions are defined early on to create common functions to draw certain graphics. He notes, however, that students need to be constantly reminded of the opportunity to define functions: they do not internalize this automatically (Section 11.7.4.3). This is a commendable but challenging goal, as recognizing when it is worth defining a clear abstraction is a nontrivial endeavor even for experienced programmers.

The need for guidance is also emphasized by Dorothy: students can get acquainted with defining functions, but without supervision, they still tend to write large functions

without thinking about which parameters are appropriate (Section 11.8.4.1).

In summary, despite some existing challenges, the reports from the teachers do not suggest that defining functions is a topic that should be postponed. Their students, mostly 9th graders, appear capable of using this powerful concept.

11.10.3.2 Students do not deem PyTamaro too restrictive for their creativity

Programming courses often include a final project in which students have more freedom than in the prescribed exercises during the course. This project is often a moment in which students can exercise agency [96].

When programming graphics, the final project may consist of a graphic chosen by the student. However, PyTamaro's deliberate limitations can make it hard to program arbitrary graphics. Still, in three of the analyzed cases, teachers had students work on a project with satisfaction.

Ada feels that her students have enough freedom to be creative in their final projects, emphasizing a sense of empowerment that comes from being able to write code fully on their own for the first time (Section 11.5.4.3).

Barbara and a colleague of hers, in this first offering, only allowed students to pick a subset of the activities already available on the PyTamaro Web platform (Section 11.6.4.2). They plan to give students more freedom to choose in a revised version of their course, but were not dissatisfied with the first iteration.

Dorothy also has her students work on a project (Section 11.8.4.2). She scaffolded the process to nudge her students towards parameterizing their code. She reports that students were overall enthusiastic, with some voluntarily putting in additional work. To draw some graphics, she taught some students how to use transparent rectangles (effectively reintroducing a local coordinate system, see Section 5.10.2).

In the remaining two cases, projects could not be done within the first year, but there were no specific complaints caused by PyTamaro's constraints.

Emil developed some activities for faster students that are inspired by biology, but reports not having yet found the time for students to work on an independent project (Section 11.9.3.5).

Charles has only a limited time during the first year of his informatics course and is not doing a project with PyTamaro. However, he still reports that PyTamaro's approach encourages students to experiment, more than they did when using turtle graphics in his previous experience. The activities offered on the web platform also help to inspire students (Section 11.7.4.1).

Part V

Epilogue

Chapter 12

Here Is a Conclusive Look at This Thesis

This dissertation presented and investigated the PyTamaro approach from multiple angles, to substantiate the claim that the graphics domain is a suitable vehicle to teach programming in an engaging way, paying special attention to abstraction and decomposition.

12.1 The PyTamaro approach has potential, but there are challenges

To achieve this goal, in Part II we reviewed the challenge of teaching introductory programming (Chapter 2), motivated the need for an engaging approach, and highlighted abstraction and problem decomposition as fundamental programming skills. In Chapters 3 and 4 we reviewed existing libraries used in introductory programming and described pitfalls that hinder teaching abstraction and decomposition or otherwise require the use of complex language features to write even the simplest programs.

In Part III, we presented the PyTamaro approach which includes a Python library (Chapter 5), a tangible notional machine to introduce programming unplugged (Chapter 6), and a web platform (Chapter 7) with support for dedicated features such as the Toolbox of Functions (Chapter 8) and a custom documentation system (Chapter 9) that leverage the strengths of the approach to assist teaching programming with PyTamaro. We grounded our arguments in programming language theory and the existing computing education research literature.

In Part IV, we presented two empirical investigations of PyTamaro.

Chapter 10 described a randomized controlled experiment in which we compared PyTamaro with turtle graphics, one of the most popular approaches to teaching introductory programming using graphics. Student engagement was high for both groups of

learners. However, despite carefully designed mini-lessons, we could not demonstrate better transfer for the PyTamaro group to questions outside the graphics domain. Both groups performed well on simple program writing and modifying tasks. The PyTamaro group traced a PyTamaro program much better than a "comparable" but not identical turtle program.

Chapter 11 presented a multiple-case study analyzing five high school teachers in Swiss high schools who adopted PyTamaro to teach programming in the now mandatory informatics course. Teachers embraced the PyTamaro approach, using graphics as a domain to introduce most programming concepts and not just as a playful afterthought. They emphasize early on the importance of defining functions as a means of abstraction and exploit the idea of a Toolbox of Functions, both on the PyTamaro Web platform and offline. Despite the minimalism of the library, teachers report that their students had ample room for creativity in projects that used PyTamaro.

However, the case study also revealed a number of challenges. Mostly due to limited classroom time, teaching materials only discuss the idea of decomposition within the domain of graphics. Moreover, while we provide some exemplar activities with PyTamaro, high school teachers who wish to adopt the approach have to create their own curricula. This requires significant effort from teachers, who admit that it was only feasible to create new materials thanks to the dedicated time available in the context of a final project of their training program. Finally, all teachers are introducing loops to repeat computation, which requires them to explain mutable variables. Reconciling this concept, after discussing mostly immutable variables or constants with PyTamaro, remains a challenge and not all materials explain this transition adequately.

12.2 Our empirical investigations have important limitations

As pointed out in the respective chapters, our empirical investigations are subject to important limitations and suffer from a number of biases.

Most notably, we are both the authors of the approach and the investigators of its effectiveness. A positivist philosophy of science would perhaps reject altogether the case study and appreciate the controlled experiment. However, as we discussed in Section 10.3.4.1 and even empirically demonstrated in our previous research [50], an educational researcher makes a multitude of decisions even when designing an apparently objective experiment.

This thesis subscribes to pragmatism as a philosophy of science (Section 1.7) and advocates transparency as the ultimate way for the research community to judge our results.

12.3   Thanks to its flexibility, the approach is used in different contexts

Unlike many other research projects that never go beyond a prototypical phase, the PyTamaro approach is actively being used in multiple contexts. We are painfully aware that adoption does not guarantee pedagogical effectiveness, and that popularity is a bad proxy to determine what works, but the use of PyTamaro at different educational levels and by many different teachers is an encouraging indicator that PyTamaro is seen by external people as a suitable approach.

The main "deployment" of PyTamaro is currently in Swiss high schools, spanning the three main linguistic regions of the country. Five high school teachers were part of the case study, but other teachers are also using PyTamaro with materials that depend on their particular contexts and goals.

The PyTamaro approach is also implemented in a Java library that is used in a first-year programming course at the home university of the author of this dissertation. It is used in a textbook of sorts that uses graphics to bridge from Racket to Java [229], before moving on to programming with objects and mutation[1].

Lastly, PyTamaro is also used in an elective course on programming at a middle school in Tessin. We briefly reviewed this curriculum, which we co-designed with the responsible teacher, in Section 6.7. Notably, the curriculum relies heavily on unplugged activities with TamaroCards.

---

[1]Intrigued? https://luce.si.usi.ch/composition-in-java/.

Chapter 13

What Is the Future of PyTamaro?

Whether or not the previous chapters were convincing enough that PyTamaro has potential, this final one suggests some possible future directions for the approach. Some directions allude to new software tools to be developed or extended, others call for more extensive empirical investigations, others point instead to open problems in pedagogy.

13.1 More empirical studies can be conducted

Within the time frame of this research project, we could only evaluate empirically some specific aspects of the extensive PyTamaro approach. Empirical studies in education research have important differences from other studies that are more common in other fields of computer science. Once a software system has been developed, it is often straightforward to measure its performance using some metric and compare it to a benchmark. But when the system is intended to be used in education, the ultimate question is whether it is effective with students (i.e., does it bring learning gains?). When humans enter the loop, studies suddenly become "expensive". One has to find the right participants and convince them to take part in the study. If the study requires participants to be novice programmers who do not know a specific topic, those novices will undergo a learning experience and will no longer be able to serve as naïve participants.

This dissertation described two main empirical investigations. In the study presented in Chapter 10, beginner students experienced a short teaching intervention with PyTamaro in a highly controlled setting. Chapter 11, instead, focused primarily on teachers, whose teaching materials ultimately impact students.

There are many more empirical studies one could design and conduct. Each aspect of the PyTamaro approach could be investigated separately. The Toolbox of Functions,

the Judicious documentation system, and the unplugged approach with TamaroCards were all grounded in programming language theory and the literature on teaching introductory programming. However, these are only *prerequisites* and not *proofs* that these ideas will show their benefits in practice. Empirical evaluations with learners should be conducted to put our claims to the test.

The PyTamaro Web platform is currently collecting a large number of real-world programs written by students using PyTamaro. Analyzing that data could also provide a complementary, quantitative perspective on how key aspects of the PyTamaro approach, like abstraction and decomposition, manifest in student code.

Ultimately, an all-encompassing evaluation would require an extended version of the short controlled experiment presented in Chapter 10. It would need moving from the highly controlled but artificial conditions of a laboratory experiment to the messy reality of classrooms. This transition would introduce numerous confounding factors, but evidence from actual learners would provide the strongest signal to demonstrate the effectiveness of the approach.

13.2   The PyTamaro approach should still grow

The PyTamaro approach covers a meaningful part of introductory programming, but some educators may rightfully object to the lack of certain topics. This section briefly describes some next steps to extend the approach.

13.2.1   Learners should eventually write interactive programs

The "Even or odd" program, despite its limitations, has an appealing quality: it is an interactive program. Students can imagine a "user" entering a number of their choice and the program reacting to that number.

This style of interleaving I/O and the logic of the program is problematic, making it harder to design clean, independent functions. PyTamaro materials do not encourage it, and the web platform currently does not support input operations. Most PyTamaro programs are thus *batch* programs, and their "input" is provided only at the beginning (e.g., configuring constants or function arguments).

Better ways to design *interactive* programs exist. These notably include games, which can be a source of motivation for students [236]. Felleisen et al. [81] describe a programming architecture in which students provide functions that evolve the model (e.g., depending on keyboard or mouse events) and render the model into a graphic. This can be seen as a "functional", educational variation of the more elaborate Model-View-Controller architecture [152].

The PyTamaro library could be extended, or accompanied by a separate library, to enable students to create interactions. In fact, the JTamaro library[1], which adopts the same core as PyTamaro, includes additional features to create interactive Java programs. However, certain kinds of interactions, such as identifying which sub-graphic a user has clicked on, are challenging to express. How to support those effectively, in a clean but pedagogically viable style, is still an open research question. Yorgey [285] illustrates one step towards this direction, avoiding the need for students to fiddle with coordinates.

13.2.2 PyTamaro should offer more support for testing

Testing is not only central in software engineering, but ought to play an important role in introductory programming as well [282]. Tests are one way to increase our confidence in building *correct* programs.

When saving a function to the Toolbox in PyTamaro Web (Section 8.3.2), learners are asked to provide an example call of their function and execute the code to manually inspect that it produces the intended graphic. However, the PyTamaro library does not offer much support to automatically verify that the output is correct. The library currently offers only two functions to determine the width and the height of a graphic, which can be used to express very basic assertions.

We have explored different strategies for testing the equality of graphics. The problem looks simple, but providing pedagogically reasonable solutions is rather challenging. Barland et al. [18] discuss at length how equality on graphics should be defined and implemented. Equality between two graphics could be defined strictly as two graphics being constructed in the same way (i.e., identical scene trees) or when they are rendered with the same pixels.

Barland et al. [18] argue for the latter. But even if equality is defined on the graphics as rendered, it would be advantageous to offer students more feedback on where the differences are beyond saying that "some pixels at these coordinates differ". Achieving that properly is still an open research problem.

13.2.3 A graphical REPL would emphasize expressions

A Read-Eval-Print-Loop (REPL) enables students to focus on expressions at the very beginning of their programming journey. With a Python REPL, a learner can type the expression `2 ** 8` and see `256` as the result of the evaluation. There is no need to create a program and learn about the `print` function just to see a simple result.

---

[1] https://github.com/LuCEresearchlab/jtamaro

Unfortunately, nearly all REPLs are limited to displaying textual output. Two notable exceptions are the REPL within the DrRacket IDE [88] and the REPL offered by the Pyret web environment [256]. Both are capable of displaying graphical values, enabling students to begin working with graphics from day one, as they would with numbers or strings.

The PyTamaro Web platform currently only offers a "Rapid Playground": a dedicated ephemeral environment to experiment with PyTamaro programs. Offering a REPL would provide an even smoother introduction to programming, benefiting curricula that focus on expressions. The translation of a TamaroCards expression into Python could be evaluated without needing an explicit call to `show_graphic`.

13.2.4   Learners will transition beyond introductory programming in Python

Given that an "official" PyTamaro curriculum does not exist, it is hard to give a definitive answer to the question of which programming concepts are covered by the PyTamaro approach. The case study described in Chapter 11 showed how different teachers covered different ground, depending on the available time and the goals of their specific context.

Escaping the question altogether would be unwise, though. Learners will eventually transition to using more complex programming constructs and possibly entirely new languages. Prior research has investigated how learners transfer from one language to another [261]. A hypothesis that finds at least partial confirmation in empirical results is that transfer improves when the languages share large parts of the underlying semantics [168].

The transition does not necessarily have to be to a different language, however, especially when modern programming languages support many "paradigms" or "styles" of programming. Adding language features such as mutable objects demands more elaborate conceptual models for evaluating programs. This transition needs to be handled with care (see, e.g., [229] and [3, Ch. 3]). Supporting this transition after students have learned programming with PyTamaro is an exciting future direction.

Part VI

Appendices

Appendix A

Appendix to the Randomized Controlled Experiment

A.1 Pre-Survey

Participants answered a pre-survey with the following questions:

- On demographics:

    - *How old are you?* [numeric]

    - *What is your gender?* [male; female; non-binary; other; prefer not to answer]

- On prior experience:

    - *How many lines of code have you written before starting the course, across all languages except HTML and CSS?* [none; fewer than 50; fewer than 500; fewer than 5000; more]

    - *Which rounds of exercises of the course have you completed?* [subset of #1, #2, #3]

    - *Have you ever written a program that draws graphics before?* [yes; no; not sure]

- On the attitude towards programming (answers on a seven-point Likert scale):

    - *It is useful for me to know how to program.* [seven-point Likert from "not at all true" to "completely true"]

    - *Programming is boring.* [seven-point Likert]

    - *Programming is fun.* [seven-point Likert]

A.2   Teaching Intervention

This Section contains the exact text of the teaching intervention used in the study. The intervention was divided into four mini-lessons, which correspond to the four subsections below. Each lesson starts with a *common part* that was shown to both the PyTamaro and Turtle groups. The lesson then "splits in two": participants in the PyTamaro group worked through the part marked as "PyTamaro-Only", whereas participants in the Turtle group worked through the part marked as "Turtle-Only". A short recap concludes each lesson and is common to both groups.

During the intervention, the Python code were shown in a web-based environment that allowed participants to run the code and see the output.

A.2.1   Mini-Lesson 1 (of 4)

All the programs you have written so far in the CS1 course deal with text: perhaps they read some input from the user as a sequence of characters (that is, a string), they do some processing and calculations, and call the function `print` to spit out an answer that is again textual.

Over the course of the next hour, you will learn how to write programs that go beyond that and can create graphics. If that sounds scary, fear not!

Libraries   It is much quicker to reuse program code that someone has already written, instead of starting from scratch. This is why programmers constantly use so-called **libraries** of reusable program parts to accomplish various things.

You can think of a library simply as a collection of **names**. Some of those names refer to **functions** that the library provides for you to use: for instance, Python's `math` library offers a function named `sqrt` which computes the square root of a number. Other names might just refer to plain **values**: again as an example, Python's `math` library contains the name `pi` for the mathematical constant π, and thus `pi` equals approximately 3.1415.

The tiny Python program below prints an approximate value of `pi`, which comes from the `math` library. Note that you can choose to run the program — try it!

```python
from math import pi
print(pi)
```

The first line of the program above "imports" the name `pi` so that you can use it in your program. While the import is a necessary step, you will not see it again in all the programs featured in the rest of these less material. You do *not* need to worry about that: we add all the necessary imports automatically for you behind the scenes.

There are many libraries; we'll use one of them   When they need to perform a task, programmers can decide to use one of the many available libraries. Drawing graphics is no exception: there are many available libraries to draw onscreen, and they embrace different approaches.

This study explores two different libraries to draw graphics in Python. You have been randomly assigned to one of them, which you are going to use in this session. After the study is over, however, you are also welcome to look at the materials for the other library, if you are interested.

> **PyTamaro-Only Part**

Let's Draw a Rectangle   Let's dive into it and see how to use a library named **PyTamaro** to draw graphics. We start very humbly, writing a program to create a rectangle and show it on the screen. Conveniently, the PyTamaro library offers you a function named `rectangle` to create rectangles. The library also has names such as `green` for basic colors.

How do you call the `rectangle` function? Here is an illustration of the basic idea:

Figure A.1. Example call of `rectangle`

`rectangle` takes in three **parameters**, which are represented in the above illustration by the three incoming arrow-shaped "holes". The first two parameters determine the width and the height of the rectangle; the third one determines the color.

We can use numbers such as 200 and 100 for the first and the second parameter, to indicate the width and the height, and a color such as `green` for the third parameter.

How can we use this function in a Python program? Let's store its **return value**, our rectangle (represented in the image near the outgoing arrow), in a variable named `football_field`. (There's an example of this below.)

And then, just like one can pass any string such as `"Hello world"` to the function `print` to display a text on the screen, let's pass a graphic to a function named `show_graphic` to display it onscreen.

Try it! In the code below, the `show_graphic` function call contains three dots (`...`). Edit the program: replace the dots with the name of the variable where we stored the rectangle. Then run the program.

```
football_field = rectangle(200, 100, green)
show_graphic(...)
```

Have you managed to see your first graphic? Congratulations!

Let's Rotate Things   Imagine now that you are sitting right at a corner of a stadium: the football field would not look to you "horizontal", but rotated by some angle. We can try to modify our program to draw something like that. The PyTamaro library offers you a function named `rotate` that takes two parameters: an angle in degrees and a graphic. The function returns a graphic rotated counterclockwise by that angle. For example:

Figure A.2. Example call of `rotate`

In the next program below, complete the assignment in the second line by replacing the dots with a call to the function `rotate`. For `rotate`'s first parameter, write 45 (i.e., 45 degrees); for the second, write `football_field`. Do not forget to separate the two parameters with a comma.

```
football_field = rectangle(200, 100, green)
rotated_field = ...
show_graphic(rotated_field)
```

Do you see a rotated field? Awesome!

Perhaps you are wondering if we really *must* have two variables in the program above. The answer is: no. We can rewrite the solution to combine the function calls to `rectangle` and `rotate`. In terms of the illustrations above, we are plugging the outgoing arrow of `rectangle` straight into the second incoming "hole" of `rotate`.

Take another look at the program just above. We need to plug the call to `rectangle` into the place reserved for the second parameter when we are calling the function `rotate`. Do that now in the code below: copy the expression `rectangle(200, 100, green)` (the entire expression, including the closing `)`) and paste it below, replacing the three dots . . . .

```
rotated_field = rotate(45, ...)
show_graphic(rotated_field)
```

Turtle-Only Part

Let's Draw a Rectangle   Let's dive into it and see how to use a library named **turtle** to draw graphics. We start very humbly, writing a program to create a line to be shown on screen.

The turtle library is based on the following metaphor.

Imagine you are controlling a "robotic turtle" that can move on a canvas carrying a colored pen. You give commands to the turtle. When the turtle moves, it leaves a trace on the canvas, ultimately producing a drawing.

One of the commands understood by the turtle is `forward`. The function `forward` takes in one number as a **parameter**. Calling the function causes the turtle to move forward by the given amount of steps.

Note that `forward` moves the turtle in the direction it is *currently facing*. The turtle starts facing east (that is, towards the right of your screen). Here is an animation of the turtle moving `forward` with `100` steps:

$$\longrightarrow$$

Figure A.3. Animation forward [last frame — only the last frame of animations is reproduced in this article, but the full animation was visibile to the participants during the study]

We can also change the color of the pen carried by the turtle, to draw colored lines such as in the example above. The default pen color is black, but we can use the function `pencolor` to use a differently colored pen.

The `pencolor` function takes in one single parameter, a string containing the name of the desired color for the pen.

To recap, you can draw a colored line by calling `pencolor` with a string parameter such as `"green"` for the name of the color, and further calling `forward` with a numerical parameter such as `100` for how much the turtle should move forward.

Try it by yourself! Replace the three dots `...` inside the call to `pencolor` with the appropriate string to draw a green line.

```
pencolor(...)
forward(100)
```

Have you managed to see your first graphic? Congratulations!

Figure A.4. Animation forward-left [last frame]

Let's Rotate   We cannot get very far just by commanding the turtle to move forward. Luckily, there are functions that rotate the turtle — or, in other words, change the direction that the turtle faces in. Two such functions are named `left` and `right`. For example, the `left` function offered by the library has one single parameter: an angle in degrees that indicates how much the turtle should turn *left*.

We can draw a "mirrored" letter L (something like ⌐) by first moving forward as before (with the turtle moving towards the right edge of the screen), then turning the turtle left by 90 degrees, and finally moving forward again.

Here is an animation that shows the plan for our turtle:

In the third line of the following program, replace the dots with a call to the function `left` as described above.

```
pencolor("green")
forward(100)
...
forward(200)
```

Do you see something that resembles a ⌐? Awesome!

Perhaps you are wondering if we cannot draw a proper letter L. The answer is: we can. To do so, we'll make use of the `backward` function. The function behaves exactly like `forward` but moves the turtle backward (relative to the direction it is facing).

Here is the plan to draw the letter L. As always, the turtle starts facing east. We move forward by a certain amount, and then we move backward by the same amount. Then, as before, we turn left 90 degrees and move forward to complete the letter.

Now, in the code below, replace the three dots with `backward(100)` to complete the plan above.

```
pencolor("green")
forward(100)
...
left(90)
forward(200)
```

<div style="text-align:center">

Common Part (PyTamaro and Turtle)

</div>

So Far, So Good!   If everything worked, give yourself a pat on the back! In this lesson, you learned what a programming library is and how to use one to draw a very simple graphic. On to the next adventure!

A.2.2   Mini-Lesson 2 (of 4)

You now know how to draw an extremely basic shape. In this lesson, we will step up the game a bit and try to draw a house. The house is made up of a ground floor, represented by a square, on top of which sits a roof, represented by an equilateral triangle.

<div style="text-align:center">

PyTamaro-Only Part

</div>

Let's Draw the Ground Floor   You can use the `square` function to create the ground floor. It takes two parameters, the side length and the color, and returns a graphic of a square.

In the program below, replace the dots on the first line. The line should assign value to the variable `ground_floor`: the should variable holds the return value of the `square` function, when that function is called with `100` as the side length and `yellow` as the color.

```
ground_floor = ...
show_graphic(ground_floor)
```

(The `square` function works by creating a rectangle of the specified color in which width and height are the same. In fact, it internally makes use of the `rectangle` function seen in the previous lesson.)

Let's Draw the Roof   We now need to figure out how to create the roof. Luckily, the PyTamaro library offers a function named `triangle`. It takes four parameters. The first two determine the lengths of two of the triangle's sides; the third parameter determines the angle between those sides. The last parameter, as usual, describes the color.

Let's create a red equilateral triangle with a side length of 100. All of an equilateral triangle's internal angles measure 60 degrees. Therefore, we can call the `triangle` function passing in the values `100`, `100`, `60`, and `red`. Visually:

Figure A.5. Example call of `triangle`

Replace the dots in the code below with the appropriate function call.

```
roof = ...
show_graphic(roof)
```

Great! Now that we have the two individual graphics, we need a way to combine them together as we intend.

Let's Put the Pieces Together   PyTamaro caters to our needs: it offers a function `above` to place two graphics one above the other. The function `above` places the graphic received as the first parameter above the graphic specified by the second parameter. It returns a new, composed graphic.

Here is how we intend to use it:

Figure A.6. Example call of the `above` function

Complete the code below by replacing the dots in the third line with a call to the function `above`, passing in the two graphics that are stored in the `ground_floor` and the `roof` variables.

```
ground_floor = square(100, yellow)
roof = triangle(100, 100, 60, red)
house = ...
show_graphic(house)
```

Once you see the house, you can also experiment and exchange the two parameters in the call to `above`. Run the program again and observe the difference!

<div style="text-align: center; background: #0a6eba; color: white; padding: 8px; font-weight: bold;">Turtle-Only Part</div>

Let's Draw the Ground Floor   You can use the function `square` to draw the ground floor. It takes just one parameter, the side length.

It is useful to understand just how this `square` function works. It repeats this combination of commands four times: 1. it moves the turtle forward by the given side length, and 2. then it rotates the turtle 90 degrees to the right.

Remember that *at the beginning* of a program the turtle starts by facing east (that is, towards the right of the screen). Assuming that as a starting point, this is what happens when the function `square` is called: 1. The turtle moves forward drawing the top side of the square, and then it turns right. At this point, the turtle is facing south. 2. The turtle moves forward drawing the right side of the square, and then it turns right. At this point, the turtle is facing west (that is, towards the left of the screen). 3. The turtle moves forward drawing the bottom side of the square, and then it turns right. At this point, the turtle is facing north. 4. The turtle moves forward drawing the left side of the square, and then it turns right.

After all the steps, the turtle has drawn a square. It is again facing in the same direction as before the execution of the `square` function. The animation below exemplifies this process.

Figure A.7. Animation, drawing a square with turtle [last frame]

Now that you have an idea of how the `square` function works, replace the dots with a call to it, using 100 as a value for the first and only parameter.

```
pencolor("yellow")
...
```

Let's Add the Roof   We now need to figure out how to create the roof. For that, we can use a function named `triangle`, which is similar to `square`.

`triangle` commands the turtle to move forward and turn right 120 degrees *three times*. In practice, this means that the function draws the three sides of a triangle one after the other. The animation below shows what happens when the `triangle`

function is called with a side length of 100. This animation, too, assumes that the turtle faces east just before calling the function.
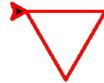
Figure A.8. Animation, drawing a triangle with turtle [last frame]

In the code below, replace the dots with a call to `triangle`, passing in the value 100 for the side length. Observe that before that command we have already added a call to `pencolor`, so that the roof is drawn in red.

```
pencolor("yellow")
square(100)
pencolor("red")
...
```

Whoops! You might have seen a drawing that was not what we intended! Don't worry: there is a simple explanation. When we call `triangle`, the first thing that happens is that the turtle moves forward, in order to draw the first side of the triangle. But this happens after the call to `square`, with the turtle (again) facing east; calling `triangle` makes it move forward — east — and then turn 120 degrees to the right, which is not what we want.

We need to rotate the turtle before calling the `triangle` function, so that the turtle well positioned to start drawing the roof simply by moving forward.

We know how to do that: we can use the `left` function to turn the turtle left by 60 degrees (the measure of an equilateral triangle's internal angle) just before drawing the triangle.

Replace the dots in the code below with the appropriate call to `left`.

```
pencolor("yellow")
square(100)
pencolor("red")
...
triangle(100)
```

<div style="text-align: center; background-color: #1a7fc4; color: white; font-weight: bold; padding: 10px;">Common Part (PyTamaro and Turtle)</div>

Wow! That took a while, but you can now feel proud: you wrote a program to draw a house!

A.2.3   Mini-Lesson 3 (of 4)

Our single house feels lonely: it is time to give it company.

Two Houses   Let's draw a duplex house, which simply means *two* single houses next to each other.

<div style="text-align: center; background-color: #1a7fc4; color: white; font-weight: bold; padding: 10px;">PyTamaro-Only Part</div>

The PyTamaro library offers a convenient `beside` function that works pretty much like the `above` function you have already used.

It takes two graphics as parameters and returns a new graphic by placing one specified as the first parameter on the left, and the one specified as the second parameter on the right.

Complete the code below by replacing the dots with a call to `beside`. As you call `beside`, you can use the variable `house` *twice*, once for each parameter (since we're placing two identical houses side by side).

```
ground_floor = square(100, yellow)
roof = triangle(100, 100, 60, red)
house = above(roof, ground_floor)

two_houses = ...
show_graphic(two_houses)
```

Got two houses? Great! However, some privacy is always welcome.

Privacy, Please!   Could we add a wall in between the houses? A narrow, black rectangle will do the job.

But how? We'd like to place *three* graphics next to each other: a house, a wall, and another house. But what we have is a *two*-parameter function `beside` that places *two* graphics next to each other.

Well, we can call our function twice: the first call can combine the left house with the wall, while the second one can combine the previous result and the right house.

Implement this idea in the code below, replacing the dots with the appropriate calls to `beside`:

```
ground_floor = square(100, yellow)
roof = triangle(100, 100, 60, red)
house = above(roof, ground_floor)

wall = rectangle(15, 187, black)

left_house_with_wall = ...
two_houses_with_wall = ...

show_graphic(two_houses_with_wall)
```

An Alternative Solution   There is a different but equally valid solution that perhaps already occurred to you.

There's no rule saying that the first step *must* involve the left house and the wall. We could first combine the wall and the right house so that they are next to each other, and then combine the left wall with the previous result!

Try to implement this variant and verify that it actually produces the same drawing.

```
ground_floor = square(100, yellow)
roof = triangle(100, 100, 60, red)
house = above(roof, ground_floor)

wall = rectangle(15, 187, black)

wall_and_right_house = ...
two_houses_with_wall = ...

show_graphic(two_houses_with_wall)
```

Turtle-Only Part

We can accomplish this with a rather simple idea. We have to: 1. Draw the first house. The turtle ends up at the top-left corner of the square. It is *not* facing east, however, because we rotated it 60 degrees left before calling `triangle`. 2. Re-position the turtle so that it will be ready to execute again the same commands as the first step. For this we need to: (a) compensate for the left turn made before calling `triangle` by turning right and (b) then move the turtle forward. 3. Execute the same commands as the first step to draw the second house.

Complete the code below with two appropriate lines that replace the dots with: 1. a call to `right` with 60 degrees as a parameter, to undo the left turn; and 2. a call to `forward` to move the turtle by the same width as one house, that is 100 steps.

```
pencolor("yellow")
square(100)
pencolor("red")
left(60)
triangle(100)
...
...
pencolor("yellow")
square(100)
pencolor("red")
left(60)
triangle(100)
```

Note something important, though! When you moved the turtle forward during the second step, it is still carrying a red pen that draws. This turns out not to be a problem in this specific program, since the turtle moves along a line that has already been drawn in red. Going over it a second time does not do any harm. But be mindful of this pitfall in general: otherwise, your drawing may have surprising and unwanted lines.

So you got two houses? Great! However, some privacy is always welcome.

Privacy, Please! Could we add a thick wall between the houses? A black square will do the job.

Modify the program below to add a wall, which requires replacing the dots with these three steps:

1. Change the color of the pen to `"black"`.

2. Draw a square of side 100.

3. Position the turtle appropriately so that it is ready to draw the second house.

Note that the third step is essential and that, in general, the order in which you give commands to the turtle matters.

```
pencolor("yellow")
square(100)
pencolor("red")
left(60)
triangle(100)
right(60)
forward(100)
...
...
...
pencolor("yellow")
square(100)
pencolor("red")
left(60)
triangle(100)
```

(Do not worry about the left border of the second house overwriting the black wall.)

Once you got the proper drawing, convince yourself that the order of the commands matters. Suppose that the plan above had step 1 and step 2 swapped (which means drawing the wall before changing the pen color). Modify the code above to reflect this change. Observe the result: what happens to the drawing?

Common Part (PyTamaro and Turtle)

You have practiced drawing slightly bigger graphics. Let's bring this one step forward with the next lesson.

A.2.4   Mini-Lesson 4 (of 4)

Let's build some bigger graphics!

In many pictures, there are *repeated* elements; a picture might have several houses, for example. That does not mean we have to duplicate a lot of code! The computer is excellent at repeating things for us.

In the CS1 course, you have already encountered one mechanism in Python to *repeat*: the `for` loop. Before getting back to graphics, let's review a simple example that might help you to refresh your knowledge.

Recap: `for` Loops   Suppose you want to write a simple program to compute the average of the five grades you obtained in the past semester. Each grade is asked to the user using the function `input`.

To compute the average, we need to divide the sum of all grades by the number of grades. How do we keep track of the sum of all grades? We can use a variable, named for example `sum_grades`. At any point during the execution of the program, the role of that variable is to track the sum of the grades seen so far.

```python
n_grades = 5

sum_grades = 0
for i in range(n_grades):
    grade = int(input())
    sum_grades = sum_grades + grade

average = sum_grades / n_grades
print("Average grade:", average)
```

The `for` loop repeats the instructions "contained" in it (the two indented lines) `n_grades` times, which here means five times. In other words, the loop does five *iterations* over the instructions; we'll use this term below.

Before the first iteration, we do not have any information about the grades, yet. We can conveniently initialize the variable `sum_grades` to `0`.

Now consider the first iteration. For example, let's say that the user first enters the grade `4`. Then, `sum_grades` will be assigned to the value `0 + 4`, which is just `4`.

Imagine that the user inputs the grade `3` at the second iteration of the loop. The value of the variable `sum_grades` will be updated to `3 + 4`, that is `7`.

This process goes on for all the specified number of iterations (`n_grades`, in the example). At the end of the last iteration, the variable `sum_grades` has *accumulated* the sum of all grades, exactly like we wanted. We can then easily compute and print the average.

Back to Graphics! Can we use a `for` loop to draw a graphic containing a repeated pattern? Sure we can!

Consider a simplified street that consists of a number of houses, all having the same appearance as the one we have drawn so far. We are looking at a densely populated neighborhood: there is no space between adjacent houses.

PyTamaro-Only Part

We can make good use of the `for` loop to *repeat* the same operation multiple times and place many graphics next to each other.

In the example presented at the beginning of this lesson, we used a variable (`sum_grades`) to *accumulate* the grades summed at each iteration. We can do the same and use a variable to accumulate the houses placed next to each other at each iteration.

Just like for grades we added *one* new grade at each iteration, we will now add *one* new house at each iteration to the ones "joined" so far.

Let's give the name `street` to the variable used to accumulate the houses. Before the loop, our street is going to be empty (of houses and indeed of anything). The first iteration will update `street` so that it contains one house. The second iteration will add one more house, so that `street` will contain two houses. The third iteration will add one more house: `street` will then contain three houses, and so on.

Which initial value should we use for `street`, before the loop? Like 0 in the example with the grades, we should use a value that works for the first iteration of the loop. Here, we need a graphic that when placed `beside` our first house, just results in that same single house.

PyTamaro has a function for this purpose: it is named `empty_graphic`. The function takes no parameters and returns an *empty graphic*. When combining an *empty graphic* with any other graphic (using `beside`, for example), the result is just the other graphic. Convenient, and a bit like zero in math!

Look closely at the code below. The first three lines create a house as we have always done so far. A variable `n_houses` contains the number of houses we want to have in our street. We initialize `street` to an empty graphic, the result of *calling* the parameterless function `empty_graphic`.

Then comes our `for` loop.

At each iteration, we need to assign to `street` a combined graphic. That graphic is the result of placing any previous houses beside one more house; in other words, we should place `street`'s earlier value beside a new house from the `house` variable.

Replace the dots below with a call to `beside`. As parameters, write the names of the two variables suggested above.

After the loop, `street` is a graphic that contains five houses next to each other, and is ready to be shown as usual with `show_graphic`.

```
ground_floor = square(100, yellow)
roof = triangle(100, 100, 60, red)
house = above(roof, ground_floor)

n_houses = 5

street = empty_graphic()
```

```
for i in range(n_houses):
    street = ...

show_graphic(street)
```

Can you see a street with five houses? Lovely!

<div align="center">

Turtle-Only Part

</div>

We can now make good use of the *for* loop we just reviewed to *repeat* the same operation multiple times and place many graphics next to each other.

We need to draw a *street* of houses. Each iteration of the for loop will just draw *one* house.

Let's recall the plan we used in the previous lesson to draw just *two* houses: 1. Draw the first house. 2. Re-position the turtle so that it will be ready to execute again the same commands as the first step. 3. Draw the second house.

This implies that at the end of each iteration we need to prepare the ground so that the next one can start properly. Concretely, it means that after drawing a house (step 1 in the plan), we always need to: (a) turn the turtle right by 60 degrees (to "undo" the left turn made before drawing the triangle), and (b) move the turtle forward by 100 steps.

Complete the code below with the two appropriate commands so that after each iteration, the turtle is positioned so that it's ready to start drawing the ground floor of the next house.

```
n_houses = 5

for i in range(n_houses):
    pencolor("yellow")
    square(100)
    pencolor("red")
    left(60)
    triangle(100)
    ...
    ...
```

Can you see a street with five houses? Lovely!

There is just one tiny inefficiency: we also reposition the turtle at the *last* iteration of the loop, even though there is no house to draw further. You can safely ignore this, given that we are not at all concerned with performance here.

As a final point, notice how using a loop helped us to avoid duplicating code, which is something that we did in the previous lesson, in which all the commands to draw a house were written twice. Experienced programmers consider code duplication a very bad thing. Think about what you would need to do if you had many houses in a graphic and decided that their roofs should be rectangles instead. It would require you to go through lots of lines and replace every occurrence of `triangle` with other code. Besides being a boring manual process, you would risk forgetting to do some replacements.

Common Part (PyTamaro and Turtle)

End of the Mini-Lessons   You have now practiced for loops a bit more and learned how they help also in programs that deal with graphics.

A.3   Post-Survey

Participants answered a post-survey with the following questions, all on a seven-point Likert scale from 1 ("not at all true") to 7 ("completely true"):

- I found the preceding lessons interesting.

- I feel that I learned about programming concepts from these lessons.

- I already knew beforehand how to do graphical programming similar to what was taught in the lessons.

- I already knew beforehand all the general programming content (variables, functions, loops, etc.) that was covered in the lessons.

- Programming with graphics is fun.

- I like programming with graphics more than the text-based programming we have done in CS1.

- I would like to learn more about programming with graphics.

A.4   Post-Test Multiple-Choice Questions

For each multiple-choice question, participants have been asked to choose the claim they believe is most accurate. Questions featured an additional "I don't know" option, to be picked only in case the participant was very unsure.

A.4.1 Question 1

"Cha Cha Cha" is the title of a song. This Python program plays with the song title and prints "Cha" three times, each one on a separate line.

```python
print("Cha")
print("Cha")
print("Cha")
```

Your friend says that it is possible to get the same output differently by introducing a variable `word`:

```python
word = "Cha"
print(word)
print(word)
print(word)
```

Is the program still working as before?

- Yes, because `word` is used only once in each instruction/line. An instruction like `print(word + word + word)` is invalid and produces an error.

- Yes, because we can use the value stored in the variable as many times as we want.

- No, only the first `print` works. To fix the second program, we would need to add `word = "Cha"` before the second and the third `print` as well.

- No, because the second program prints three times `word`.

A.4.2 Question 2

Python's math library contains a function named `sqrt`. It takes one parameter, the number to compute the square root of. Its return value is also a number, the square root of the provided number.

This program first computes the square root of 16, and then the square root of the result, which is finally printed:

```python
root_of_sixteen = sqrt(16)
final_root = sqrt(root_of_sixteen)
print(final_root)
```

Your friend says that the same result can be obtained with a shorter program:

```python
print(sqrt(sqrt(16)))
```

Is the program still working as before?

- Yes, because of the mathematical properties of the square root function. The same transformation with a function `half` that divides a number by two would not have worked.

- Yes, because it first computes the square root of 16, then computes the square root of the result, and eventually prints the final result.

- No. `sqrt(16)` works, because we are passing a number, 16, to the function. But in `sqrt(sqrt(16))` we are passing `sqrt(16)`, which is not a number but a function call.

- No, because `sqrt` is a function that takes one parameter, and the second program attempts to give the first (outermost) `sqrt` call two parameters.

- No, because we need a variable to store the result of `sqrt` before we can pass it to another function call.

A.4.3 Question 3

Imagine that a Python library contains a function named `subtract`. It takes two numbers as parameters, and returns the result of subtracting the second number from the first one.

```python
result = subtract(10, 7)
print(result)
```

Does executing the program above print 3?

- Yes, because calling `subtract` is one way to subtract a number from another. Because we are free to choose the order of parameter values, we could have also written `result = subtract(7, 10)` to get the same result.

- Yes, because we are correctly passing the numbers 10 and 7 to the function `subtract`.

- No, because functions can only have one parameter. It is therefore impossible for the library to offer a working `subtract` function with two parameters.

- No, because that is not how you should pass multiple parameters when a function requires more than one: the call should have been `subtract(10)(7)`.

A.4.4   Question 4

Imagine that a Python library contains a function named `fake_random`. It has zero parameters, and always returns the number 42 as a fake random number.

```
print(fake_random())
```

Does executing the program above print 42?

- Yes, because `fake_random()` calls the function, which will return the number 42. The result is passed to `print`. Also, the empty parentheses `()` are necessary; just `print(fake_random)` does not work.

- Yes, because `fake_random()` calls the function. Also, writing just `print(fake_random)` without the empty parentheses would have done the same, given that the function returns a constant number.

- No, because the function `fake_random` cannot possibly exist as such, as functions need to have at least one parameter.

- No, because the function `fake_random` has zero parameters, and such a parameterless function cannot return a value.

- No, because we need a variable to store the result of `fake_random` before we can pass it to `print`.

A.4.5   Question 5

Imagine you have a function named `combine` at your disposal. It takes two strings as parameters and returns a combined string. For example, `combine("hel", "lo")` returns `"hello"`.

The goal is to write a program that constructs the word `restaurant` from three pieces, then prints out the result.

One of your friends comes up with the following program:

```
first_combination = combine("re", "stau")
word = combine(first_combination, "rant")
print(word)
```

Another friend suggests this other implementation:

```
first_combination = combine("stau", "rant")
word = combine("re", first_combination)
print(word)
```

What can you say about these two programs?

- They both work, even if the word is constructed in two alternative ways.

- The first one works, but the second one does not, because "re" is added in the second line after "staurant" has been created.

- The first one works, but the second one does not, because combine is not commutative (that is, because exchanging the first parameter with the second makes a difference).

- Neither one works, as a two-parameter function cannot be defined (so that it works).

A.4.6   Question 6

You want to write a program that asks the user for five numbers, multiplies all of them together, and prints the result. Your friend suggests this skeleton, but they are unsure about what to write instead of the dots at the beginning.

```
...
for i in range(5):
    number = int(input())
    product = product * number
print(product)
```

What should the dots be replaced with?

- product = 0, because the variable product needs to be initialized to the neutral number 0 before looping.

- product = 1, because 1 is the only number that multiplied with any other number just results in the other number.

- We need to initialize product to some value, but it doesn't matter which one, because that value will in any case be replaced by the first number entered by the user during the first iteration of the loop.

- There is no number that works as an initial value for product. Other changes would need to be done to the program as well.

Appendix B

Appendix to the Case Study

B.1   Additional dedicated questions for Ada

- When using the old materials with turtle graphics, what did both you and your students understand about the magic line `from gturtle import` *?

- On the last page of the first note there is a brief mention of "Decomposition with turtle". Did you actually do decomposition with turtle graphics?

- How much were you relying on the debugger with TigerJython? Do you feel the absence of something similar for PyTamaro?

- Even before PyTamaro you felt the need of writing your own notes. Was it about the lack of materials? The desire of personalization?

- On page 7 in the third note, some snippets of code use methods. How do you explain them to students? Do you or they use the documentation for them?

- There is no appearance of TamaroCards in your materials. Do you use the cards?

- Several activities introduce the definition of functions. Why is the need of abstraction with functions not motivated with the game of "similarities and differences"?

- How are the activities on the web platform actually used in the classroom? Do students work on them on their own?

- The activity "Table of values" uses some functions that are more akin to procedures. Do students notice this? Do they have problems in understanding them?

- Why did you decide to end nearly all your activities with the learning objectives nicely divided into concepts, Python, and PyTamaro?

- You show to students how to use the `print` function with variable-length arguments at the very beginning. Are students confused by this accommodating behavior?

- When introducing functions, you establish a parallel between function definition and function use (or call). Could you use the same parallel when discussing variables (definition versus use)?

- Is repetition first introduced without PyTamaro? Why?

- Are you discussing the concept of variables versus constants, or are you only using the former as a terminology?

- Where did you learn about the "Variable as a Box" metaphor?

- Where did you learn about the "EVA" principle ("Eingabe, Verarbeitung, Ausgabe", "Input, Processing, Output")?

B.2   Additional dedicated questions for Barbara

- One slide compares "Python types" with "PyTamaro types". Do you perceive them as different?

- How are you using the TamaroCards in your teaching? What is the experience with your students?

- Are you using Thonny as the primary IDE or the PyTamaro Web platform?

- One of the grading criteria is "verschachtelungstiefe flach", "flat/shallow nesting depth". Why did you include it among your grading criteria?

- Do your students work in pairs, as suggested by one of your slides?

- The Toolbox approach is shown both in Thonny and on the PyTamaro Web platform. Do you think it is effective?

- On Slide 158. Why are you using the "variable as a box" metaphor? Why do you introduce it at that point?

B.3   Additional dedicated questions for Charles

- In part 5, you are using a memory diagram. Why do you introduce it? Why do you show it both for numbers and graphics?

- In part 5, did you deliberately try to avoid using the `range` function at first? Part 6 uses `range` in an activity with list comprehension.

- In part 5, an example uses tuples. Do you include tuples in the data structures you teach?

- What kind of exam questions do you ask to your students?

- You are using the Toolbox idea, but not the dedicated support in the web platform. How do students perceive the manual approach to the Toolbox (including managing files)? Can they see the benefits?

- How do students perceive the first unplugged activities with the TamaroCards?

- When do you stop using the cards?

- Do you also use the cards to define your own function?

- What motivated the revision of your PyTamaro materials for this school year, compared to the previous year?

B.4   Additional dedicated questions for Dorothy

- On slide 141. Why did you choose names for the variables that are almost the same as the name of the function?

- You often use types for the variables. Do you always do that? What is the reaction of the students?

- Do you think that explicitly showing common errors at the beginning helps students?

- Are you using Python IDLE as an IDE, the PyTamaro Web platform, or both?

- On an exercise that solves the compound interest example. Do students like this example better? Do they see this kind of programming differently than with PyTamaro? Can they apply what they already know?

- On the fifth question of the exam. Why did you choose to include a question about "program quality"? Did most students get it right?

- Do you think that it is helpful to have the API in German?

- Is the first activity on PyTamaro Web happening before the unplugged introduction with the TamaroCards?

- When do you stop using the TamaroCards? Why?

- On an activity about "changeable variables" on PyTamaro Web. Do you expect your students to use mutable variables also with PyTamaro?

- Do you not encourage nesting? Why? Do you believe it is hard to understand?

- In one of your activities you state: "Do not use `short`, `long`, `double` and `char`". Do your students know other programming languages?

- Why do you talk about "similarities and differences" only several activities after introducing parametrization?

- Is the Toolbox explained as something specific to PyTamaro?

B.5   Additional dedicated questions for Emil

- The second activity, after the introductory one, immediately discusses error messages. Why? Do you feel that it is effective?

- In one activity, you describe an "interpretation" activity with TamaroCards. When do you start using the cards? When do you stop?

- How do your students react to the "Principle of nesting" activity?

- How are students dealing with defining functions relatively early?

- Why is Toolbox introduced immediately when introducing function definitions for the first time?

- Do you do any activity to target decomposition, modularization, or abstraction outside the domain of graphics?

- The `for` loop is introduced by printing the numbers in a list, and then immediately after by working with graphics. Does this close presentation help students to recognize the parallel?

- Do you connect `empty_graphic` to the concept of a neutral element in mathematics?

- Are some of your activities mixing `_` as a placeholder for identifiers and `...` as a placeholder for expressions?

- What kind of exercises do you plan to ask in an exam?

References

[1]   Hal Abelson, Nat Goodman, and Lee Rudolph. *LOGO Manual*. Tech. rep. AIM-313 / LOGO Memo 7. MIT, Dec. 1974, p. 84. URL: https://dspace.mit.edu/handle/1721.1/6226.

[2]   Harold Abelson and Andrea diSessa. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. The MIT Press, June 1981. ISBN: 978-0-262-36274-0. DOI: 10.7551/mitpress/6933.001.0001.

[3]   Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, July 1996. ISBN: 978-0-262-51087-5.

[4]   Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs: JavaScript Edition*. Cambridge, MA, USA: MIT Press, May 2022. ISBN: 978-0-262-36762-2.

[5]   Shaaron Ainsworth. "DeFT: A Conceptual Framework for Considering Learning with Multiple Representations". In: *Learning and Instruction* 16 (2006), pp. 183–198.

[6]   Efthimia Aivaloglou and Felienne Hermans. "How Kids Code and How We Know: An Exploratory Study on the Scratch Repository". In: *Proceedings of the 2016 ACM Conference on International Computing Education Research*. Melbourne VIC Australia: ACM, Aug. 2016, pp. 53–61. ISBN: 978-1-4503-4449-4. DOI: 10.1145/2960310.2960325.

[7]   Reem A. Alamer, Wejdan A. Al-Doweesh, Hend S. Al-Khalifa, and Muna S. Al-Razgan. "Programming Unplugged: Bridging CS Unplugged Activities Gap for Learning Key Programming Concepts". In: *2015 Fifth International Conference on E-Learning (Econf)*. Oct. 2015, pp. 97–103. DOI: 10.1109/ECONF.2015.27.

[8]   Jeffrey R. Albrecht and Stuart A. Karabenick. "Relevance for Learning and Motivation in Education". In: *The Journal of Experimental Education* 86.1 (Jan. 2018), pp. 1–10. ISSN: 0022-0973, 1940-0683. DOI: 10.1080/00220973.2017.1380593.

[9] Carl Alphonce and Phil Ventura. "Using Graphics to Support the Teaching of Fundamental Object-Oriented Principles in CS1". In: *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. Anaheim CA USA: ACM, Oct. 2003, pp. 156–161. ISBN: 978-1-58113-751-4. DOI: 10.1145/949344.949391.

[10] Amjad Altadmri and Neil C.C. Brown. "37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data". In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. New York, NY, USA: Association for Computing Machinery, Feb. 2015, pp. 522–527. ISBN: 978-1-4503-2966-8. DOI: 10.1145/2676723.2677258.

[11] Boyd Anderson, Martin Henz, Kok-Lim Low, and Daryl Tan. "Shrinking JavaScript for CS1". In: *Proceedings of the 2021 ACM SIGPLAN International Symposium on SPLASH-E*. Chicago IL USA: ACM, Oct. 2021, pp. 87–96. ISBN: 978-1-4503-9089-7. DOI: 10.1145/3484272.3484970.

[12] Aivar Annamaa. "Introducing Thonny, a Python IDE for Learning Programming". In: *Proceedings of the 15th Koli Calling Conference on Computing Education Research - Koli Calling '15*. Koli, Finland: ACM Press, 2015, pp. 117–121. ISBN: 978-1-4503-4020-5. DOI: 10.1145/2828959.2828969.

[13] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. Van Wijngaarden, and M. Woodger. "Report on the Algorithmic Language ALGOL 60". In: *Communications of the ACM* 3.5 (May 1960). Ed. by Peter Naur, pp. 299–311. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/367236.367262.

[14] John Backus. "The History of FORTRAN I, II, and III". In: *ACM SIGPLAN Notices* 13.8 (Aug. 1978), pp. 165–180. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/960118.808380.

[15] Marini Abu Bakar, Muriati Mukhtar, and Fariza Khalid. "The Effect of Turtle Graphics Approach on Students' Motivation to Learn Programming: A Case Study in a Malaysian University". In: *International Journal of Information and Education Technology* 10.4 (2020), pp. 290–297. ISSN: 20103689. DOI: 10.18178/ijiet.2020.10.4.1378.

[16] Sebastian Baltes and Stephan Diehl. "Usage and Attribution of Stack Overflow Code Snippets in GitHub Projects". In: *Empirical Software Engineering* 24.3 (June 2019), pp. 1259–1295. ISSN: 1573-7616. DOI: 10.1007/s10664-018-9650-5.

[17] Sebastian Baltes and Christoph Treude. "Code Duplication on Stack Overflow". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. ICSE-NIER '20. New York, NY, USA: ACM, Sept. 2020, pp. 13–16. ISBN: 978-1-4503-7126-1. DOI: 10.1145/3377816.3381744.

[18] Ian Barland, Robert Bruce Findler, and Matthew Flatt. *The Design of a Functional Image Library*. 2010. URL: https://users.cs.northwestern.edu/~robby/pubs/papers/sfp2010-bff.pdf (visited on 02/07/2025).

[19] Piraye Bayman and Richard E. Mayer. "Using Conceptual Models to Teach BASIC Computer Programming." In: *Journal of Educational Psychology* 80.3 (Sept. 1988), pp. 291–298. ISSN: 1939-2176, 0022-0663. DOI: 10.1037/0022-0663.80.3.291.

[20] Theresa Beaubouef and John Mason. "Why the High Attrition Rate for Computer Science Students: Some Thoughts and Observations". In: *ACM SIGCSE Bulletin* 37.2 (June 2005), pp. 103–106. ISSN: 0097-8418. DOI: 10.1145/1083431.1083474.

[21] Tim Bell and Jan Vahrenhold. "CS Unplugged—How Is It Used, and Does It Work?" In: *Adventures Between Lower Bounds and Higher Altitudes: Essays Dedicated to Juraj Hromkovič on the Occasion of His 60th Birthday*. Ed. by Hans-Joachim Böckenhauer, Dennis Komm, and Walter Unger. Cham: Springer International Publishing, 2018, pp. 497–521. ISBN: 978-3-319-98355-4. DOI: 10.1007/978-3-319-98355-4_29.

[22] Timothy C. Bell, Ian H. Witten, and Mike Fellows. "Computer Science Unplugged: Off-line Activities and Games for All Ages". In: (1998). URL: https://classic.csunplugged.org/documents/books/english/unplugged-book-v1.pdf (visited on 07/11/2025).

[23] Mordechai Ben-Ari. "Constructivism in Computer Science Education". In: *Journal of Computers in Mathematics and Science Teaching* 20.1 (2001), pp. 45–73.

[24] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. "A Neural Probabilistic Language Model". In: *The Journal of Machine Learning Research* 3 (Mar. 2003), pp. 1137–1155. ISSN: 1532-4435.

[25] Jens Bennedsen and Michael E. Caspersen. "Failure Rates in Introductory Programming". In: *ACM SIGCSE Bulletin* 39.2 (June 2007), pp. 32–36. ISSN: 0097-8418. DOI: 10.1145/1272848.1272879.

[26]   Joey Bevilacqua, Luca Chiodini, Igor Moreno Santos, and Matthias Hauswirth. "Assessing the Understanding of Expressions: A Qualitative Study of Notional-Machine-Based Exam Questions". In: *Proceedings of the 24th Koli Calling International Conference on Computing Education Research*. Vol. 1. New York, NY, USA: Association for Computing Machinery, 2024, p. 1. DOI: 10.1145/3699538.3699554.

[27]   A.F. Blackwell. "First Steps in Programming: A Rationale for Attention Investment Models". In: *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*. Sept. 2002, pp. 2–10. DOI: 10.1109/HCC.2002.1046334.

[28]   Bootstrap. *Bootstrap :: Hour of Code*. URL: https://www.bootstrapworld.org/materials/fall2023/en-us/lessons/hoc-winter-parley/index.html (visited on 04/16/2024).

[29]   Maura Borrego, Stephanie Cutler, Michael Prince, Charles Henderson, and Jeffrey E. Froyd. "Fidelity of Implementation of Research-Based Instructional Strategies (RBIS) in Engineering Science Courses". In: *Journal of Engineering Education* 102.3 (2013), pp. 394–425. ISSN: 2168-9830. DOI: 10.1002/jee.20020.

[30]   Richard E. Boyatzis. *Transforming Qualitative Information : Thematic Analysis and Code Development*. Thousand Oaks, CA : Sage Publications, 1998. ISBN: 978-0-7619-0960-6. URL: http://archive.org/details/transformingqual0000boya.

[31]   Karen Brennan and Mitchel Resnick. *New Frameworks for Studying and Assessing the Development of Computational Thinking*. 2012. URL: http://scratched.gse.harvard.edu/ct/files/AERA2012.pdf.

[32]   Neil C. C. Brown, Amjad Altadmri, Sue Sentance, and Michael Kölling. "Blackbox, Five Years On: An Evaluation of a Large-scale Programming Data Collection Project". In: *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ICER '18. New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 196–204. ISBN: 978-1-4503-5628-2. DOI: 10.1145/3230977.3230991.

[33]   Neil C. C. Brown and Mark Guzdial. "Confidence vs Insight: Big and Rich Data in Computing Education Research". In: *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. Portland OR USA: ACM, Mar. 2024, pp. 158–164. DOI: 10.1145/3626252.3630813.

[34] Neil C. C. Brown, Pierre Weill-Tessier, Maksymilian Sekula, Alexandra-Lucia Costache, and Michael Kölling. "Novice Use of the Java Programming Language". In: *ACM Transactions on Computing Education* (July 2022). DOI: 10.1145/3551393.

[35] Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. "Blackbox: A Large Scale Repository of Novice Programmers' Activity". In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE '14. New York, NY, USA: Association for Computing Machinery, Mar. 2014, pp. 223–228. ISBN: 978-1-4503-2605-6. DOI: 10.1145/2538862.2538924.

[36] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. "Language Models Are Few-Shot Learners". In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS '20. Red Hook, NY, USA: Curran Associates Inc., Dec. 2020, pp. 1877–1901. ISBN: 978-1-7138-2954-6.

[37] Joshua Burridge and Alan Fekete. "Teaching Programming for First-Year Data Science". In: *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*. ITiCSE '22. New York, NY, USA: Association for Computing Machinery, July 2022, pp. 297–303. ISBN: 978-1-4503-9201-3. DOI: 10.1145/3502718.3524740.

[38] Donald Thomas Campbell and Julian Cecil Stanley. *Experimental and Quasi-Experimental Designs for Research*. 2. print. Boston: Houghton Mifflin Comp, 1967. ISBN: 978-0-395-30787-8.

[39] V. R. Cane and A. W. Heim. "The Effects of Repeated Retesting: III. Further Experiments and General Conclusions". In: *Quarterly Journal of Experimental Psychology* 2.4 (Dec. 1950), pp. 182–197. ISSN: 0033-555X. DOI: 10.1080/17470215008416596.

[40] Michael E. Caspersen. "Informatics as a Fundamental Discipline in General Education: The Danish Perspective". In: *Perspectives on Digital Humanism*. Ed. by Hannes Werthner, Erich Prem, Edward A. Lee, and Carlo Ghezzi. Cham: Springer International Publishing, 2022, pp. 191–200. ISBN: 978-3-030-86144-5. DOI: 10.1007/978-3-030-86144-5_26.

[41]   Michael E. Caspersen and Jens Bennedsen. "Instructional Design of a Program-ming Course: A Learning Theoretic Approach". In: *Proceedings of the Third International Workshop on Computing Education Research*. Atlanta Georgia USA: ACM, Sept. 2007, pp. 111–122. ISBN: 978-1-59593-841-1. DOI: 10.1145/1288580.1288595.

[42]   Michael E. Caspersen and Henrik Bærbak Christensen. "Here, There and Everywhere - on the Recurring Use of Turtle Graphics in CS1". In: *Proceedings of the Australasian Conference on Computing Education*. ACSE '00. New York, NY, USA: Association for Computing Machinery, Dec. 2000, pp. 34–40. ISBN: 978-1-58113-271-7. DOI: 10.1145/359369.359375.

[43]   Richard Catrambone. "The Subgoal Learning Model: Creating Better Examples so That Students Can Solve Novel Problems." In: *Journal of experimental psychology: General* 127.4 (1998), p. 355.

[44]   Walter Cazzola and Diego Mathias Olivares. "Gradually Learning Programming Supported by a Growable Programming Language". In: *IEEE Transactions on Emerging Topics in Computing* 4.3 (July 2016), pp. 404–415. ISSN: 2168-6750. DOI: 10.1109/TETC.2015.2446192.

[45]   Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. "What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–12. ISBN: 978-1-4503-6708-0. DOI: 10.1145/3313831.3376729.

[46]   Mark Chen et al. *Evaluating Large Language Models Trained on Code*. July 2021. arXiv: 2107.03374 [cs]. URL: http://arxiv.org/abs/2107.03374 (visited on 10/18/2022).

[47]   Jacqui Chetty. "Combatting the War Against Machines: An Innovative Hands-on Approach to Coding". In: *Robotics in STEM Education*. Ed. by Myint Swe Khine. Cham: Springer International Publishing, 2017, pp. 59–83. ISBN: 978-3-319-57786-9. DOI: 10.1007/978-3-319-57786-9_3.

[48]   Luca Chiodini, Joey Bevilacqua, and Matthias Hauswirth. "Surveying Upper-Secondary Teachers on Programming Misconceptions". In: *Proceedings of the 2025 ACM Conference on International Computing Education Research - Volume 1*. Vol. 1. ICER '25. New York, NY, USA: Association for Computing Machinery, Aug. 2025. ISBN: 979-8-4007-1340-8. DOI: 10.1145/3702652.3744227.

[49] Luca Chiodini, Joey Bevilacqua, and Matthias Hauswirth. "The Toolbox of Functions: Teaching Code Reuse in Schools". In: *Proceedings of the 6th European Conference on Software Engineering Education*. ECSEE '25. New York, NY, USA: Association for Computing Machinery, June 2025, pp. 185–189. ISBN: 979-8-4007-1282-1. DOI: 10.1145/3723010.3723029.

[50] Luca Chiodini and Matthias Hauswirth. "Wrong Answers for Wrong Reasons: The Risks of Ad Hoc Instruments". In: *Proceedings of the 21st Koli Calling International Conference on Computing Education Research*. Koli Calling '21. New York, NY, USA: ACM, Nov. 2021, pp. 1–11. ISBN: 978-1-4503-8488-9. DOI: 10.1145/3488042.3488045.

[51] Luca Chiodini, Matthias Hauswirth, and Andrea Gallidabino. "Conceptual Checks for Programming Teachers". In: *Technology-Enhanced Learning for a Free, Safe, and Sustainable World*. Ed. by Tinne De Laet, Roland Klemke, Carlos Alario-Hoyos, Isabel Hilliger, and Alejandro Ortega-Arranz. Vol. 12884. Cham: Springer International Publishing, 2021, pp. 399–403. ISBN: 978-3-030-86436-1. DOI: 10.1007/978-3-030-86436-1_43.

[52] Luca Chiodini, Igor Moreno Santos, Andrea Gallidabino, Anya Tafliovich, André L. Santos, and Matthias Hauswirth. "A Curated Inventory of Programming Language Misconceptions". In: *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. ITiCSE '21. New York, NY, USA: Association for Computing Machinery, June 2021, pp. 380–386. ISBN: 978-1-4503-8214-4. DOI: 10.1145/3430665.3456343.

[53] Luca Chiodini, Igor Moreno Santos, and Matthias Hauswirth. "Expressions in Java: Essential, Prevalent, Neglected?" In: *Proceedings of the 2022 ACM SIGPLAN International Symposium on SPLASH-E*. SPLASH-E 2022. New York, NY, USA: ACM, Dec. 2022, pp. 41–51. ISBN: 978-1-4503-9900-5. DOI: 10.1145/3563767.3568131.

[54] Luca Chiodini, Simone Piatti, and Matthias Hauswirth. "Judicious: API Documentation for Novices". In: *Proceedings of the 2024 ACM SIGPLAN International Symposium on SPLASH-E*. SPLASH-E 2024. New York, NY, USA: Association for Computing Machinery, 2024, pp. 1–9. ISBN: 979-8-4007-1216-6. DOI: 10.1145/3689493.3689987.

[55] Luca Chiodini, Juha Sorva, and Matthias Hauswirth. "Teaching Programming with Graphics: Pitfalls and a Solution". In: *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E*. SPLASH-E 2023. New York, NY, USA: ACM, Oct. 2023, pp. 1–12. ISBN: 979-8-4007-0390-4. DOI: 10.1145/3622780.3623644.

[56]   Luca Chiodini, Juha Sorva, Arto Hellas, Otto Seppälä, and Matthias Hauswirth. "Two Approaches for Programming Education in the Domain of Graphics: An Experiment". In: *The Art, Science, and Engineering of Programming* 10.1 (Feb. 2025), 14:1–14:48. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2025/10/14.

[57]   Alonzo Church. *The Calculi of Lambda-Conversion*. 1 6. Princeton, New Jersey, USA: Princeton University Press, 1941.

[58]   Douglas H. Clements and Julie S. Meredith. "Research on Logo: Effects and Efficacy". In: *Journal of Computing in Childhood Education* 4 (1993), pp. 263–90. ISSN: 1043-1055. (Visited on 07/21/2025).

[59]   Jacob Cohen. "A Power Primer". In: *Psychological Bulletin* 112.1 (July 1992), pp. 155–159. ISSN: 0033-2909. DOI: 10.1037//0033-2909.112.1.155.

[60]   Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences, Rev. Ed*. Statistical Power Analysis for the Behavioral Sciences, Rev. Ed. Hillsdale, NJ, USA: Lawrence Erlbaum Associates, Inc, 1977. ISBN: 978-0-12-179060-8.

[61]   Stephen Cooper, Wanda Dann, and Randy Pausch. "Alice: A 3-D Tool for Introductory Programming Concepts". In: *Journal of Computing Sciences in Colleges* 15.5 (2000), pp. 107–116. DOI: 10.5555/364132.364161.

[62]   Steve Cooper and Steve Cunningham. "Teaching Computer Science in Context". In: *ACM Inroads* 1.1 (Mar. 2010), pp. 5–8. ISSN: 2153-2184, 2153-2192. DOI: 10.1145/1721933.1721934.

[63]   P. Cope, H. Smith, and M. Simmons. "Misconceptions Concerning Rotation and Angie in LOGO". In: *Journal of Computer Assisted Learning* 8.1 (1992), pp. 16–24. ISSN: 1365-2729. DOI: 10.1111/j.1365-2729.1992.tb00381.x.

[64]   Will Crichton. *Documentation Generation as Information Visualization*. Nov. 2020. arXiv: 2011.05600 [cs]. URL: http://arxiv.org/abs/2011.05600 (visited on 05/26/2024).

[65]   Will Crichton and Shriram Krishnamurthi. "Profiling Programming Language Learning". In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA1 (Apr. 2024), pp. 29–54. ISSN: 2475-1421. DOI: 10.1145/3649812.

[66]   Paul Curzon, Jane Waite, Karl Maton, and James Donohue. "Using Semantic Waves to Analyse the Effectiveness of Unplugged Computing Activities". In: *Proceedings of the 15th Workshop on Primary and Secondary Computing Education*. Virtual Event Germany: ACM, Oct. 2020, pp. 1–10. ISBN: 978-1-4503-8759-0. DOI: 10.1145/3421590.3421606.

[67]   O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. GBR:
         Academic Press Ltd., 1972. ISBN: 978-0-12-200550-3. URL: https://dl.
         acm.org/doi/book/10.5555/1243380.

[68]   Joost de Winter and Dimitra Dodou. "Five-Point Likert Items: t Test versus
         Mann–Whitney–Wilcoxon". In: *Practical Assessment, Research and Evaluation*
         15 (Jan. 2010). DOI: 10.7275/bj1p-ts64.

[69]   Christine Dearnley. "A Reflection on the Use of Semi-Structured Interviews". In:
         Nurse Researcher 13.1 (July 2005), pp. 19–28. ISSN: 1351-5578, 2047-8992.
         DOI: 10.7748/nr2005.07.13.1.19.c5997.

[70]   Tim DeClue. "A Theory of Attrition in Computer Science Education Which Ex-
         plores the Effect of Learning Theory, Gender, and Context". In: (2009).

[71]   Douglas K. Detterman. "The Case for the Prosecution: Transfer as an Epiphe-
         nomenon". In: *Transfer on Trial: Intelligence, Cognition, and Instruction*. West-
         port, CT, US: Ablex Publishing, 1993, pp. 1–24. ISBN: 978-0-89391-825-5.

[72]   Keith Devlin. "Why Universities Require Computer Science Students to Take
         Math". In: *Communications of the ACM* 46.9 (Sept. 2003), p. 36. ISSN: 00010782.
         DOI: 10.1145/903893.903917.

[73]   J. Dewey. *Logic: The Theory of Inquiry*. Logic: The Theory of Inquiry. Oxford,
         England: Holt, 1938, pp. viii, 546.

[74]   Edsger W. Dijkstra. "Letters to the Editor: Go to Statement Considered Harm-
         ful". In: *Communications of the ACM* 11.3 (Mar. 1968), pp. 147–148. ISSN:
         0001-0782. DOI: 10.1145/362929.362947.

[75]   Edsger W. Dijkstra. "The Humble Programmer". In: *Communications of the
         ACM* 15.10 (Oct. 1972), pp. 859–866. ISSN: 0001-0782, 1557-7317. DOI:
         10.1145/355604.361591.

[76]   Ian Drosos, Philip J. Guo, and Chris Parnin. "HappyFace: Identifying and Pre-
         dicting Frustrating Obstacles for Learning Programming at Scale". In: *2017
         IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*.
         Oct. 2017, pp. 171–179. DOI: 10.1109/VLHCC.2017.8103465.

[77]   Benedict Du Boulay. "Some Difficulties of Learning to Program". In: *Journal of
         Educational Computing Research* 2.1 (Feb. 1986), pp. 57–73. ISSN: 0735-6331.
         DOI: 10.2190/3LFX-9RRF-67T8-UVK9. eprint: https://doi.org/10.
         2190/3LFX-9RRF-67T8-UVK9.

[78]   Rodrigo Duran, Juha Sorva, and Otto Seppälä. "Rules of Program Behavior".
         In: *ACM Transactions on Computing Education* 21.4 (Nov. 2021), 33:1–33:37.
         DOI: 10.1145/3469128.

[79]   Matthias Endler. *The Best Programmers I Know*. Apr. 2025. URL: https://
       web.archive.org/web/20250709084037/https://endler.dev/
       2025/best-programmers/ (visited on 07/17/2025).

[80]   Matthias Felleisen. "On the Expressive Power of Programming Languages". In:
       Science of Computer Programming 17.1 (Dec. 1991), pp. 35–75. ISSN: 0167-
       6423. DOI: 10.1016/0167-6423(91)90036-W.

[81]   Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krish-
       namurthi. "A Functional I/O System or, Fun for Freshman Kids". In: *ACM SIG-
       PLAN Notices* 44.9 (Aug. 2009), pp. 47–58. ISSN: 0362-1340. DOI: 10.1145/
       1631687.1596561.

[82]   Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishna-
       murthi. *How to Design Programs, Second Edition: An Introduction to Program-
       ming and Computing*. Cambridge, MA, USA: MIT Press, May 2018. ISBN: 978-
       0-262-34412-8.

[83]   Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishna-
       murthi. "The TeachScheme! Project: Computing and Programming for Every
       Student". In: (2003).

[84]   Matthias Felleisen and Shriram Krishnamurthi. "Why Computer Science Doesn't
       Matter". In: *Communications of the ACM* 52.7 (July 2009), pp. 37–40. ISSN:
       0001-0782, 1557-7317. DOI: 10.1145/1538788.1538803.

[85]   Robert Field. *JEP 222: Jshell: The Java Shell (Read-Eval-Print Loop)*. May 2014.
       URL: https://openjdk.org/jeps/222 (visited on 07/11/2024).

[86]   Sally Fincher, Johan Jeuring, Craig S. Miller, Peter Donaldson, Benedict du
       Boulay, Matthias Hauswirth, Arto Hellas, Felienne Hermans, Colleen Lewis,
       Andreas Mühling, Janice L. Pearce, and Andrew Petersen. "Notional Machines
       in Computing Education: The Education of Attention". In: *Proceedings of the
       Working Group Reports on Innovation and Technology in Computer Science Edu-
       cation*. ITiCSE-WGR '20. New York, NY, USA: Association for Computing Ma-
       chinery, June 2020, pp. 21–50. ISBN: 978-1-4503-8293-9. DOI: 10.1145/
       3437800.3439202.

[87]   Sally Fincher, Brad Richards, Janet Finlay, Helen Sharp, and Isobel Falconer.
       "Stories of Change: How Educators Change Their Practice". In: *2012 Frontiers
       in Education Conference Proceedings*. Oct. 2012, pp. 1–6. DOI: 10.1109/FIE.
       2012.6462317.

[88]   Robert Bruce Findler. *DrRacket: The Racket Programming Environment*. URL:
       https://mirror.racket-lang.org/docs/7.8/pdf/drracket.pdf.

[89] Sigbjorn Finne and Simon Peyton Jones. "Pictures: A Simple Structured Graphics Model". In: *Proceedings of the 1995 Glasgow Workshop on Functional Programming*. BCS Learning & Development, July 1995. DOI: 10.14236/ewic/FP1995.6.

[90] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. "The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming". In: *Australasian Computing Education Conference*. ACE '22. New York, NY, USA: Association for Computing Machinery, Feb. 2022, pp. 10–19. ISBN: 978-1-4503-9643-1. DOI: 10.1145/3511861.3511863.

[91] Kathi Fisler. "The Recurring Rainfall Problem". In: *Proceedings of the Tenth Annual Conference on International Computing Education Research*. ICER '14. New York, NY, USA: ACM, 2014, pp. 35–42. ISBN: 978-1-4503-2755-8. DOI: 10.1145/2632320.2632346.

[92] Kathi Fisler, Shriram Krishnamurthi, Benjamin S. Lerner, and Joe Gibbs Politz. *A Data-Centric Introduction to Computing*. Version 2024-09-03. [no publisher], Sept. 2024. URL: https://dcic-world.org/ (visited on 01/29/2025).

[93] Kathi Fisler, Shriram Krishnamurthi, and Preston Tunnell Wilson. "Assessing and Teaching Scope, Mutation, and Aliasing in Upper-Level Undergraduates". In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. Seattle Washington USA: ACM, Mar. 2017, pp. 213–218. ISBN: 978-1-4503-4698-6. DOI: 10.1145/3017680.3017777.

[94] Eric J. Fox. "Contextualistic Perspectives". In: *Handbook of research on educational communications and technology* 3 (2008), pp. 55–66. (Visited on 02/20/2024).

[95] Patricia Fusch, Gene Fusch, and Lawrence Ness. "Denzin's Paradigm Shift: Revisiting Triangulation in Qualitative Research". In: *Journal of Sustainable Social Change* 10.1 (Jan. 2018). ISSN: 2834-507X. DOI: 10.5590/JOSC.2018.10.1.02.

[96] Jessica Gale, Meltem Alemdar, Katherine Boice, Diley Hernández, Sunni Newton, Douglas Edwards, and Marion Usselman. "Student Agency in a High School Computer Science Course". In: *Journal for STEM Education Research* 5.2 (Aug. 2022), pp. 270–301. ISSN: 2520-8705, 2520-8713. DOI: 10.1007/s41979-022-00071-9.

[97]   Jeremy Gibbons. "How to Design Co-Programs". In: *Journal of Functional Programming* 31 (2021), e15. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796821000113.

[98]   Mary L. Gick and Keith J. Holyoak. "Schema Induction and Analogical Transfer". In: *Cognitive Psychology* 15.1 (Jan. 1983), pp. 1–38. ISSN: 0010-0285. DOI: 10.1016/0010-0285(83)90002-6.

[99]   Ken Goldman, Paul Gross, Cinda Heeren, Geoffrey Herman, Lisa Kaczmarczyk, Michael C. Loui, and Craig Zilles. "Identifying Important and Difficult Concepts in Introductory Computing Courses Using a Delphi Process". In: *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '08. New York, NY, USA: ACM, 2008, pp. 256–260. ISBN: 978-1-59593-799-5. DOI: 10.1145/1352135.1352226.

[100]   Michael H. Goldwasser and David Letscher. "A Graphics Package for the First Day and Beyond". In: *ACM SIGCSE Bulletin* 41.1 (Mar. 2009), pp. 206–210. ISSN: 0097-8418. DOI: 10.1145/1539024.1508945.

[101]   Google for Developers. *Google I/O '08 Keynote by Marissa Mayer*. June 2008. URL: https://www.youtube.com/watch?v=6x0cAzQ7PVs (visited on 07/07/2024).

[102]   Kathryn E. Gray and Matthew Flatt. "ProfessorJ: A Gradual Introduction to Java Through Language Levels". In: *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 170–177. ISBN: 978-1-58113-751-4. DOI: 10.1145/949344.949394.

[103]   Shuchi Grover and Satabdi Basu. "Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic". In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. Seattle Washington USA: ACM, Mar. 2017, pp. 267–272. ISBN: 978-1-4503-4698-6. DOI: 10.1145/3017680.3017723.

[104]   Philip J. Guo. "Non-Native English Speakers Learning Computer Programming: Barriers, Desires, and Design Opportunities". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*. Montreal QC, Canada: ACM Press, 2018, pp. 1–14. ISBN: 978-1-4503-5620-6. DOI: 10.1145/3173574.3173970.

[105]   Mark Guzdial. "A Media Computation Course for Non-Majors". In: *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*. Thessaloniki Greece: ACM, June 2003, pp. 104–108. ISBN: 978-1-58113-672-2. DOI: 10.1145/961511.961542.

[106]   Mark Guzdial. "Achieving CS for All Could Take Decades". In: *Communications of the ACM* 65.4 (Apr. 2022), pp. 6–7. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3516513.

[107]   Mark Guzdial. "Does Contextualized Computing Education Help?" In: *ACM Inroads* 1.4 (Dec. 2010), pp. 4–6. ISSN: 2153-2184, 2153-2192. DOI: 10.1145/1869746.1869747.

[108]   Mark Guzdial. "Exploring Hypotheses about Media Computation". In: *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*. San Diego San California USA: ACM, Aug. 2013, pp. 19–26. ISBN: 978-1-4503-2243-0. DOI: 10.1145/2493394.2493397.

[109]   Mark Guzdial. "From Science to Engineering". In: *Communications of the ACM* 54.2 (Feb. 2011), pp. 37–39. ISSN: 0001-0782. DOI: 10.1145/1897816.1897831.

[110]   Pontus Haglund, Filip Strömbäck, and Linda Mannila. "Understanding Students' Failure to Use Functions as a Tool for Abstraction – An Analysis of Questionnaire Responses and Lab Assignments in a CS1 Python Course". In: *Informatics in Education* 20.4 (Dec. 2021), pp. 583–614. ISSN: 1648-5831, 2335-8971. DOI: 10.15388/infedu.2021.26.

[111]   Brian Harvey. *Computer Science Logo Style: Beyond Programming*. 2nd ed. Vol. 3. Exploring with LOGO. Cambridge, MA, USA: MIT Press, Mar. 1997. ISBN: 978-0-262-58150-9.

[112]   Brian Harvey. *Computer Science Logo Style: Symbolic Computing*. 2nd ed. Vol. 1. Exploring with LOGO. Cambridge, MA, USA: MIT Press, Mar. 1997. ISBN: 978-0-262-58148-6.

[113]   Brian Harvey, Daniel D. Garcia, Tiffany Barnes, Nathaniel Titterton, Daniel Armendariz, Luke Segars, Eugene Lemon, Sean Morris, and Josh Paley. "SNAP! (Build Your Own Blocks)". In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. Denver Colorado USA: ACM, Mar. 2013, pp. 759–759. ISBN: 978-1-4503-1868-6. DOI: 10.1145/2445196.2445507.

[114]   Kieran Healy. "Fuck Nuance". In: *Sociological Theory* 35.2 (June 2017), pp. 118–127. ISSN: 0735-2751. DOI: 10.1177/0735275117709046.

[115] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. "Helium, for Learning Haskell". In: *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*. Haskell '03. New York, NY, USA: Association for Computing Machinery, Aug. 2003, pp. 62–71. ISBN: 978-1-58113-758-3. DOI: 10.1145/871895.871902.

[116] Mary Hegarty and David A. Waller. "Individual Differences in Spatial Abilities". In: *The Cambridge Handbook of Visuospatial Thinking*. New York, NY, US: Cambridge University Press, 2005, pp. 121–169. ISBN: 978-0-521-80710-4. DOI: 10.1017/CBO9780511610448.005.

[117] Peter Henderson. "Functional Geometry". In: *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*. LFP '82. New York, NY, USA: Association for Computing Machinery, Aug. 1982, pp. 179–187. ISBN: 978-0-89791-082-8. DOI: 10.1145/800068.802148.

[118] Felienne Hermans. "Hedy: A Gradual Language for Programming Education". In: *Proceedings of the 2020 ACM Conference on International Computing Education Research*. ICER '20. New York, NY, USA: Association for Computing Machinery, Aug. 2020, pp. 259–270. ISBN: 978-1-4503-7092-9. DOI: 10.1145/3372782.3406262.

[119] Felienne Hermans and Efthimia Aivaloglou. "To Scratch or Not to Scratch? A Controlled Experiment Comparing Plugged First and Unplugged First Programming Lessons". In: *Proceedings of the 12th Workshop on Primary and Secondary Computing Education*. WiPSCE '17. New York, NY, USA: Association for Computing Machinery, Nov. 2017, pp. 49–56. ISBN: 978-1-4503-5428-8. DOI: 10.1145/3137065.3137072.

[120] Rich Hickey. *Simple Made Easy*. Strange Loop Conference, Sept. 2021. URL: https://www.youtube.com/watch?v=SxdOUGdseq4 (visited on 07/17/2025).

[121] Charles Antony Richard Hoare. "Notes on Data Structuring". In: *Structured Programming*. Academic Press Ltd., 1972, pp. 83–174. (Visited on 07/08/2025).

[122] Simon Holland, Robert Griffiths, and Mark Woodman. "Avoiding Object Misconceptions". In: *Proceedings of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '97. New York, NY, USA: ACM, 1997, pp. 131–134. ISBN: 978-0-89791-889-3. DOI: 10.1145/268084.268132.

[123] Kristina Holsapple and Austin Cory Bart. "Designing Designer: The Evidence-Oriented Design Process of a Pedagogical Interactive Graphics Python Library". In: *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2022. New York, NY, USA: Association for Computing

Machinery, Feb. 2022, pp. 85–91. ISBN: 978-1-4503-9070-5. DOI: 10.1145/3478431.3499363.

[124] R. C. Holt, D. B. Wortman, D. T. Barnard, and J. R. Cordy. "SP/k: A System for Teaching Computer Programming". In: *Commun. ACM* 20.5 (May 1977), pp. 301–309. ISSN: 0001-0782. DOI: 10.1145/359581.359586.

[125] Richard C. Holt and David B. Wortman. "A Sequence of Structured Subsets of PL/I". In: *Proceedings of the Fourth SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '74. New York, NY, USA: ACM, 1974, pp. 129–132. DOI: 10.1145/800183.810456.

[126] Jason Hong. "The Use of Java as an Introductory Programming Language". In: *XRDS: Crossroads, The ACM Magazine for Students* 4.4 (May 1998), pp. 8–13. ISSN: 1528-4972, 1528-4980. DOI: 10.1145/333140.333145.

[127] Kenneth R. Howe. "Against the Quantitative-Qualitative Incompatibility Thesis or Dogmas Die Hard". In: *Educational Researcher* 17.8 (1988), pp. 10–16. ISSN: 0013-189X. DOI: 10.2307/1175845. JSTOR: 1175845.

[128] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. "Identifying and Correcting Java Programming Errors for Introductory Computer Science Students". In: *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '03. New York, NY, USA: Association for Computing Machinery, Jan. 2003, pp. 153–156. ISBN: 978-1-58113-648-7. DOI: 10.1145/611892.611956.

[129] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming Through Multimedia*. 1st ed. Cambridge University Press, Feb. 2000. ISBN: 978-0-521-64338-2. DOI: 10.1017/CBO9780511818073.

[130] Cruz Izu and Peter Dinh. "Can Novice Programmers Write C Functions?" In: *2018 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*. Wollongong, NSW: IEEE, Dec. 2018, pp. 965–970. ISBN: 978-1-5386-6522-0. DOI: 10.1109/TALE.2018.8615375.

[131] *Java® Platform, Standard Edition & Java Development Kit Version 21 API Specification*. URL: https://docs.oracle.com/en/java/javase/21/docs/api/index.html (visited on 07/03/2024).

[132] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report: ISO Pascal Standard*. Springer Science & Business Media, Dec. 2012. ISBN: 978-1-4612-4450-9.

[133] *JEP 512: Compact Source Files and Instance Main Methods*. URL: https://openjdk.org/jeps/512 (visited on 07/07/2025).

[134]   Fionnuala Johnson, Stephen McQuistin, and John O'Donnell. "Analysis of Student Misconceptions Using Python as an Introductory Programming Language". In: *Proceedings of the 4th Conference on Computing Education Practice 2020*. Durham United Kingdom: ACM, Jan. 2020, pp. 1–4. ISBN: 978-1-4503-7729-4. DOI: 10.1145/3372356.3372360.

[135]   Jeremiah W. Johnson. "Benefits and Pitfalls of Jupyter Notebooks in the Classroom". In: *Proceedings of the 21st Annual Conference on Information Technology Education*. Virtual Event USA: ACM, Oct. 2020, pp. 32–37. ISBN: 978-1-4503-7045-5. DOI: 10.1145/3368308.3415397.

[136]   R. Burke Johnson and Anthony J. Onwuegbuzie. "Mixed Methods Research: A Research Paradigm Whose Time Has Come". In: *Educational Researcher* 33.7 (Oct. 2004), pp. 14–26. ISSN: 0013-189X. DOI: 10.3102/0013189X033007014.

[137]   Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. "Advances in Dataflow Programming Languages". In: *ACM Comput. Surv.* 36.1 (Mar. 2004), pp. 1–34. ISSN: 0360-0300. DOI: 10.1145/1013208.1013209.

[138]   Cory J. Kapser and Michael W. Godfrey. ""Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software". In: *Empirical Software Engineering* 13.6 (Dec. 2008), pp. 645–692. ISSN: 1573-7616. DOI: 10.1007/s10664-008-9076-6.

[139]   Reka Kassai, Judit Futo, Zsolt Demetrovics, and Zsofia K. Takacs. "A Meta-Analysis of the Experimental Evidence on the Near- and Far-Transfer Effects Among Children's Executive Function Skills". In: *Psychological Bulletin* 145.2 (2019), pp. 165–188. ISSN: 1939-1455(Electronic),0033-2909(Print). DOI: 10.1037/bul0000180.

[140]   Cazembe Kennedy and Eileen T. Kraemer. "Qualitative Observations of Student Reasoning: Coding in the Wild". In: *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. Aberdeen Scotland Uk: ACM, July 2019, pp. 224–230. ISBN: 978-1-4503-6895-7. DOI: 10.1145/3304221.3319751.

[141]   Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2. ed., 52. print. Prentice-Hall Software Series. Upper Saddle River, NJ: Prentice-Hall PTR, 2014. ISBN: 978-0-13-110362-7.

[142]   Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. "Code Quality Issues in Student Programs". In: *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE '17. New York, NY, USA:

ACM, June 2017, pp. 110–115. ISBN: 978-1-4503-4704-4. DOI: 10.1145/3059009.3059061.

[143] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter development team. "Jupyter Notebooks – a Publishing Format for Reproducible Computational Workflows". In: *20th International Conference on Electronic Publishing (01/01/16)*. Ed. by Fernando Loizides and Birgit Scmidt. IOS Press, 2016, pp. 87–90. DOI: 10.3233/978-1-61499-649-1-87.

[144] Jonathan Knudsen. *Java 2D Graphics*. O'Reilly, 1999. ISBN: 978-1-56592-484-0.

[145] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks". In: *IEEE Transactions on Software Engineering* 32.12 (Dec. 2006), pp. 971–987. ISSN: 1939-3520. DOI: 10.1109/TSE.2006.116.

[146] Andreas P. Koenzen, Neil A. Ernst, and Margaret-Anne D. Storey. "Code Duplication and Reuse in Jupyter Notebooks". In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. New York, NY, USA: ACM, Aug. 2020, pp. 1–9. DOI: 10.1109/VL/HCC50065.2020.9127202.

[147] Tobias Kohn. "Teaching Python Programming to Novices: Addressing Misconceptions and Creating a Development Environment". PhD thesis. ETH Zurich, 2017, 165 p. DOI: 10.3929/ETHZ-A-010871088. HDL: 20.500.11850/129666.

[148] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. "The BlueJ System and Its Pedagogy". In: *Computer Science Education* 13.4 (Dec. 2003), pp. 249–268. ISSN: 0899-3408, 1744-5175. DOI: 10.1076/csed.13.4.249.17496.

[149] Herman Koppelman and Betsy van Dijk. "Teaching Abstraction in Introductory Courses". In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education - ITiCSE '10*. Bilkent, Ankara, Turkey: ACM Press, 2010, p. 174. ISBN: 978-1-60558-729-5. DOI: 10.1145/1822090.1822140.

[150] Rainer Koschke. "Survey of Research on Software Clones". In: *Duplication, Redundancy, and Similarity in Software*. Vol. 6301. Dagstuhl Seminar Proceedings (DagSemProc). Dagstuhl, Germany: Schloss Dagstuhl, 2007, pp. 1–24. DOI: 10.4230/DagSemProc.06301.13.

[151] Douglas Kramer. "API Documentation from Source Code Comments: A Case Study of Javadoc". In: *Proceedings of the 17th Annual International Conference on Computer Documentation*. New Orleans Louisiana USA: ACM, Oct. 1999, pp. 147–153. ISBN: 978-1-58113-072-0. DOI: 10.1145/318372.318577.

[152] Glenn E. Krasner and Stephen T. Pope. "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System". In: *Journal of object oriented programming* 1.3 (1988), pp. 26–49. URL: http://heaveneverywhere.com/stp/PostScript/mvc.pdf (visited on 07/20/2025).

[153] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. 3rd ed. Online: Electronic textbook, Feb. 2023. URL: https://www.plai.org/ (visited on 03/27/2024).

[154] Shriram Krishnamurthi. "Teaching Programming Languages in a Post-Linnaean Age". In: *ACM SIGPLAN Notices* 43.11 (Nov. 2008), pp. 81–83. ISSN: 0362-1340. DOI: 10.1145/1480828.1480846.

[155] Shriram Krishnamurthi and Kathi Fisler. "Programming Paradigms and Beyond". In: *The Cambridge Handbook of Computing Education Research*. Ed. by Anthony V. Robins and Sally A. Fincher. Cambridge Handbooks in Psychology. Cambridge: Cambridge University Press, 2019, pp. 377–413. ISBN: 978-1-108-49673-5. DOI: 10.1017/9781108654555.014.

[156] Amruth N. Kumar, Rajendra K. Raj, Sherif G. Aly, Monica D. Anderson, Brett A. Becker, Richard L. Blumenthal, Eric Eaton, Susan L. Epstein, Michael Goldweber, Pankaj Jalote, Douglas Lea, Michael Oudshoorn, Marcelo Pias, Susan Reiser, Christian Servin, Rahul Simha, Titus Winters, and Qiao Xiang. *Computer Science Curricula 2023*. New York, NY, USA: ACM, Jan. 2024. ISBN: 979-8-4007-1033-9. DOI: 10.1145/3664191.

[157] Steinar Kvale. *InterViews: An Introduction to Qualitative Research Interviewing*. InterViews: An Introduction to Qualitative Research Interviewing. Thousand Oaks, CA, US: Sage Publications, Inc, 1994, pp. xvii, 326. ISBN: 978-0-8039-5819-7.

[158] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. "A Systematic Review of API Evolution Literature". In: *ACM Comput. Surv.* 54.8 (Oct. 2021), 171:1–171:36. ISSN: 0360-0300. DOI: 10.1145/3470133.

[159] Kathleen J. Lehman, Julia Rose Karpicz, Veronika Rozhenkova, Jamelia Harris, and Tomoko M. Nakajima. "Growing Enrollments Require Us to Do More: Perspectives on Broadening Participation During an Undergraduate Computing Enrollment Boom". In: *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. Virtual Event USA: ACM, Mar. 2021, pp. 809–815. ISBN: 978-1-4503-8062-1. DOI: 10.1145/3408877.3432370.

[160] Teemu Lehtinen, Aleksi Lukkarinen, and Lassi Haaranen. "Students Struggle to Explain Their Own Program Code". In: *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. June 2021, pp. 206–212. DOI: 10.1145/3430665.3456322. arXiv: 2104.06710 [cs].

[161] Shing-On Leung. "A Comparison of Psychometric Properties and Normality in 4-, 5-, 6-, and 11-Point Likert Scales". In: *Journal of Social Service Research* 37.4 (July 2011), pp. 412–421. ISSN: 0148-8376. DOI: 10.1080/01488376.2011.580697.

[162] Ronit Ben-Bassat Levy and Mordechai Ben-Ari. "We Work so Hard and They Don't Use It: Acceptance of Software Tools by Teachers". In: *ACM SIGCSE Bulletin* 39.3 (June 2007), pp. 246–250. ISSN: 0097-8418. DOI: 10.1145/1269900.1268856.

[163] Suzanne Pawlan Levy. "Computer Language Usage in CS1: Survey Results". In: *ACM SIGCSE Bulletin* 27.3 (Sept. 1995), pp. 21–26. ISSN: 0097-8418. DOI: 10.1145/209849.209853.

[164] Colleen M. Lewis. "The Importance of Students' Attention to Program State: A Case Study of Debugging Behavior". In: *Proceedings of the Ninth Annual International Conference on International Computing Education Research*. ICER '12. New York, NY, USA: ACM, Sept. 2012, pp. 127–134. ISBN: 978-1-4503-1604-0. DOI: 10.1145/2361276.2361301.

[165] John Locke. *An Essay Concerning Human Understanding*. Kay & Troutman, 1847.

[166] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. "DéjàVu: A Map of Code Duplicates on GitHub". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017), 84:1–84:28. DOI: 10.1145/3133908.

[167] Kuang-Chen Lu and Shriram Krishnamurthi. "Identifying and Correcting Programming Language Behavior Misconceptions". In: *Proc. ACM Program. Lang.* 8.OOPSLA1 (Apr. 2024), 106:334–106:361. DOI: 10.1145/3649823.

[168]   Kuang-Chen Lu, Shriram Krishnamurthi, Kathi Fisler, and Ethel Tshukudu. "What Happens When Students Switch (Functional) Languages (Experience Report)". In: *Proceedings of the ACM on Programming Languages* 7.ICFP (Aug. 2023), 215:796–215:812. DOI: 10.1145/3607857.

[169]   Aleksi Lukkarinen and Juha Sorva. "Classifying the Tools of Contextualized Programming Education and Forms of Media Computation". In: *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*. Koli Finland: ACM, Nov. 2016, pp. 51–60. ISBN: 978-1-4503-4770-9. DOI: 10.1145/2999541.2999551.

[170]   Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. "Investigating the Viability of Mental Models Held by Novice Programmers". In: *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*. Covington Kentucky USA: ACM, Mar. 2007, pp. 499–503. ISBN: 978-1-59593-361-4. DOI: 10.1145/1227310.1227481.

[171]   John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. "The Scratch Programming Language and Environment". In: *ACM Transactions on Computing Education* 10.4 (Nov. 2010), pp. 1–15. ISSN: 1946-6226, 1946-6226. DOI: 10.1145/1868358.1868363.

[172]   John H. Maloney and Randall B. Smith. "Directness and Liveness in the Morphic User Interface Construction Environment". In: *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*. Pittsburgh Pennsylvania USA: ACM, Dec. 1995, pp. 21–28. ISBN: 978-0-89791-709-4. DOI: 10.1145/215585.215636.

[173]   Melissa Høegh Marcher, Ingrid Maria Christensen, Paweł Grabarczyk, Therese Graversen, and Claus Brabrand. "Computing Educational Activities Involving People Rather Than Things Appeal More to Women (CS1 Appeal Perspective)". In: *Proceedings of the 17th ACM Conference on International Computing Education Research*. Virtual Event USA: ACM, Aug. 2021, pp. 145–156. ISBN: 978-1-4503-8326-4. DOI: 10.1145/3446871.3469761.

[174]   Lauren E. Margulieux, Mark Guzdial, and Richard Catrambone. "Subgoal-Labeled Instructional Material Improves Performance and Transfer in Learning to Develop Mobile Applications". In: *Proceedings of the Ninth Annual International Conference on International Computing Education Research*. Auckland New Zealand: ACM, Sept. 2012, pp. 71–78. ISBN: 978-1-4503-1604-0. DOI: 10.1145/2361276.2361291.

[175]   Lauren E. Margulieux, Briana B. Morrison, and Adrienne Decker. "Reducing Withdrawal and Failure Rates in Introductory Programming with Subgoal Labeled Worked Examples". In: *International Journal of STEM Education* 7.1 (Dec. 2020), p. 19. ISSN: 2196-7822. DOI: 10.1186/s40594-020-00222-7.

[176]   Karl Maton. "Making Semantic Waves: A Key to Cumulative Knowledge-Building". In: *Linguistics and Education* 24 (Apr. 2013), pp. 8–22. DOI: 10.1016/j.linged.2012.11.005.

[177]   Karl Maton. "Semantic Waves: Context, Complexity and Academic Discourse". In: *Accessing Academic Discourse*. 1st. Routledge, Nov. 2019, p. 27. ISBN: 978-0-429-28072-6. DOI: 10.4324/9780429280726-3.

[178]   John McCarthy, R. Brayton, D. Edwards, P. Fox, L. Hodes, D. Luckham, K. Maling, D. Park, and S. Russell. *LISP I Programmers Manual*. Technical Report. Cambridge, MA: Artificial Intelligence Group, MIT Computation Center and Research Laboratory, Mar. 1960.

[179]   Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. "A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students". In: *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '01. New York, NY, USA: ACM, 2001, pp. 125–180. DOI: 10.1145/572133.572137.

[180]   Tanya J. McGill and Simone E. Volet. "A Conceptual Framework for Analyzing Students' Knowledge of Programming". In: *Journal of Research on Computing in Education* 29.3 (Mar. 1997), pp. 276–297. ISSN: 0888-6504. DOI: 10.1080/08886504.1997.10782199.

[181]   Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. "Habits of Programming in Scratch". In: *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*. ITiCSE '11. New York, NY, USA: ACM, June 2011, pp. 168–172. ISBN: 978-1-4503-0697-3. DOI: 10.1145/1999747.1999796.

[182]   Bartosz Milewski. *Category Theory for Programmers*. Blurb, Aug. 2019. ISBN: 978-0-464-24387-8.

[183]   Craig S. Miller and Amber Settle. "Some Trouble with Transparency: An Analysis of Student Errors with Object-oriented Python". In: *Proceedings of the 2016 ACM Conference on International Computing Education Research*. Melbourne

VIC Australia: ACM, Aug. 2016, pp. 133–141. ISBN: 978-1-4503-4449-4. DOI: 10.1145/2960310.2960327.

[184] Claudio Mirolo, Cruz Izu, Violetta Lonati, and Emanuele Scapin. "Abstraction in Computer Science Education: An Overview". In: *Informatics in Education* 20.4 (Aug. 2022), pp. 615–639. ISSN: 1648-5831, 2335-8971. DOI: 10.15388/infedu.2021.27.

[185] Igor Moreno Santos. "Sound Notional Machines". PhD thesis. Università della Svizzera italiana, 2023. URL: https://n2t.net/ark:/12658/srd1328768.

[186] Briana B. Morrison, Lauren E. Margulieux, and Mark Guzdial. "Subgoals, Context, and Worked Examples in Learning Computing Problem Solving". In: *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*. ICER '15. New York, NY, USA: Association for Computing Machinery, Aug. 2015, pp. 21–29. ISBN: 978-1-4503-3630-7. DOI: 10.1145/2787622.2787733.

[187] Bhagya Munasinghe, Tim Bell, and Anthony Robins. "Unplugged Activities as a Catalyst When Teaching Introductory Programming". In: *Journal of Pedagogical Research* 7.2 (June 2023), pp. 56–71. ISSN: 2602-3717. DOI: 10.33902/JPR.202318546.

[188] Brad A. Myers and Jeffrey Stylos. "Improving API Usability". In: *Communications of the ACM* 59.6 (May 2016), pp. 62–69. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/2896587.

[189] Mitchell J Nathan, Kenneth R Koedinger, and Martha W Alibali. "Expert Blind Spot: When Content Knowledge Eclipses Pedagogical Content Knowledge". 2001. URL: https://pact.cs.cmu.edu/pubs/2001_NathanEtAl_ICCS_EBS.pdf.

[190] Lijun Ni, Tom McKlin, and Mark Guzdial. "How Do Computing Faculty Adopt Curriculum Innovations?: The Story from Instructors". In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. Milwaukee Wisconsin USA: ACM, Mar. 2010, pp. 544–548. DOI: 10.1145/1734263.1734444.

[191] Tomohiro Nishida, Susumu Kanemune, Yukio Idosaka, Mitaro Namiki, Tim Bell, and Yasushi Kuno. "A CS Unplugged Design Pattern". In: *ACM SIGCSE Bulletin* 41.1 (Mar. 2009), pp. 231–235. ISSN: 0097-8418. DOI: 10.1145/1539024.1508951.

[192] Richard Noss. "Children's Learning of Geometrical Concepts through Logo". In: *Journal for Research in Mathematics Education* 18.5 (1987), pp. 343–362. ISSN: 0021-8251. DOI: 10.2307/749084. JSTOR: 749084.

[193] Frank G. Pagan. "Nested Sublanguages of Algol 68 for Teaching Purposes". In: *ACM SIGPLAN Notices* 15.7 and 8 (July 1980), pp. 72–81. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/947680.947687.

[194] Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. USA: Basic Books, Inc., 1980. ISBN: 978-0-465-04627-0.

[195] Seymour A. Papert. *A Computer Laboratory for Elementary Schools*. Tech. rep. AIM-246 / LOGO Memo 1. MIT, Oct. 1971. URL: https://dspace.mit.edu/handle/1721.1/5834.

[196] Seymour A. Papert and Cynthia Solomon. *Twenty Things To Do With A Computer*. Tech. rep. AIM-248 / LOGO Memo 3. MIT, June 1971. URL: https://dspace.mit.edu/handle/1721.1/5836.

[197] Jack Parkinson and Quintin Cutts. "Investigating the Relationship Between Spatial Skills and Computer Science". In: *Proceedings of the 2018 ACM Conference on International Computing Education Research*. Espoo Finland: ACM, Aug. 2018, pp. 106–114. ISBN: 978-1-4503-5628-2. DOI: 10.1145/3230977.3230990.

[198] David Lorge Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". In: *Communications of the ACM* 15 (1972), p. 1053. ISSN: 0001-0782. DOI: 10.1145/361598.361623.

[199] Erik Pasternak, Rachel Fenichel, and Andrew N. Marshall. "Tips for Creating a Block Language with Blockly". In: *2017 IEEE Blocks and Beyond Workshop (B&B)*. Oct. 2017, pp. 21–24. DOI: 10.1109/BLOCKS.2017.8120404.

[200] Michael Quinn Patton. *Qualitative Research & Evaluation Methods: Integrating Theory and Practice*. Sage publications, 2014. (Visited on 06/26/2025).

[201] Roy D. Pea. "Language-Independent Conceptual "Bugs" in Novice Programming". In: *Journal of educational computing research* 2.1 (1986), pp. 25–36. ISSN: 0735-6331. DOI: 10.2190/689T-1R2A-X4W4-29J2.

[202] Roy D. Pea. *Logo Programming and Problem Solving. [Technical Report No. 12.]* Apr. 1983. URL: https://eric.ed.gov/?id=ED319371 (visited on 02/07/2025).

[203] Roy D. Pea and D. Midian Kurland. "On the Cognitive Effects of Learning Computer Programming". In: *New Ideas in Psychology* 2.2 (Jan. 1984), pp. 137–168. ISSN: 0732-118X. DOI: 10.1016/0732-118X(84)90018-7.

[204] Charles S. Peirce. "What Pragmatism Is". In: *The Monist* 15.2 (1905), pp. 161–181. ISSN: 0026-9662. JSTOR: 27899577. URL: https://www.jstor.org/stable/27899577 (visited on 07/22/2025).

[205] *PEP 257 – Docstring Conventions | Peps.Python.Org*. URL: https://peps.python.org/pep-0257/ (visited on 07/14/2025).

[206] David N. Perkins and Gavriel Salomon. "Transfer of Learning". In: *International Encyclopedia of Education*. Vol. 2. Pergamon Press, 1992, pp. 6452–6457. ISBN: 978-0-08-041046-3.

[207] Denis Charles Phillips and Nicholas C. Burbules. *Postpositivism and Educational Research*. Postpositivism and Educational Research. Lanham, MD, US: Rowman & Littlefield, 2000. ISBN: 978-0-8476-9122-7.

[208] Luna Phipps-Costin, Michael MacLeod, Alex Vo, Tiffany Nguyen, Joe Gibbs Politz, Shriram Krishnamurthi, and Benjamin S Lerner. "Combining Interactive and Whole-Program Editing with Repartee". In: *12 Th Annual Workshop at the Intersection of PL and HCI*. Vol. 12. Boston, MA: Northeastern University, 2021, pp. 1–11.

[209] Benjamin C. Pierce. *Types and Programming Languages*. Cambridge, Mass: MIT Press, 2002. ISBN: 978-0-262-16209-8.

[210] Paul R. Pintrich. "Motivation and Classroom Learning". In: *Handbook of Psychology*. John Wiley & Sons, Ltd, 2003. Chap. 6, pp. 103–122. ISBN: 978-0-471-26438-5. DOI: 10.1002/0471264385.wei0706.

[211] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. "Python: The Full Monty". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 217–232. ISBN: 978-1-4503-2374-1. DOI: 10.1145/2509136.2509536.

[212] Joe Gibbs Politz and Mia Minnes. *Using Jshell*. URL: https://ucsd-cse8a-f18.github.io/notes/jshell/ (visited on 07/07/2025).

[213] G. Polya. *How to Solve It: A New Aspect of Mathematical Method*. Princeton University Press, 2004. ISBN: 978-0-691-11966-3.

[214] Leo Porter and Beth Simon. "Retaining Nearly One-Third More Majors with a Trio of Instructional Best Practices in CS1". In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. Denver Colorado USA: ACM, Mar. 2013, pp. 165–170. ISBN: 978-1-4503-1868-6. DOI: 10.1145/2445196.2445248.

[215]   George J. Posner, Kenneth A. Strike, Peter W. Hewson, and William A. Gertzog. "Accommodation of a Scientific Conception: Toward a Theory of Conceptual Change". In: *Science Education* 66.2 (Apr. 1982), pp. 211–227. ISSN: 0036-8326, 1098-237X. DOI: 10.1002/sce.3730660207.

[216]   Python Software Foundation. *Turtle — Turtle Graphics*. URL: https://docs.python.org/3/library/turtle.html (visited on 06/05/2024).

[217]   Yizhou Qian and James Lehman. "Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review". In: *ACM Transactions on Computing Education* 18.1 (Oct. 2017), pp. 1–24. ISSN: 19466226. DOI: 10.1145/3077618.

[218]   Anthony Ralston. "Fortran and the First Course in Computer Science". In: *ACM SIGCSE Bulletin* 3.4 (Dec. 1971), pp. 24–29. ISSN: 0097-8418. DOI: 10.1145/382214.382499.

[219]   Casey Reas and Ben Fry. "Processing: Programming for the Media Arts". In: *AI & SOCIETY* 20.4 (Sept. 2006), pp. 526–538. ISSN: 1435-5655. DOI: 10.1007/s00146-006-0050-9.

[220]   Stephen K. Reed, George W. Ernst, and Ranan Banerji. "The Role of Analogy in Transfer between Similar Problem States". In: *Cognitive Psychology* 6.3 (July 1974), pp. 436–450. ISSN: 0010-0285. DOI: 10.1016/0010-0285(74)90020-6.

[221]   Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. "Scratch: Programming for All". In: *Communications of the ACM* 52.11 (Nov. 2009), pp. 60–67. ISSN: 0001-0782. DOI: 10.1145/1592761.1592779.

[222]   Dennis M. Ritchie, Stephen C. Johnson, M. E. Lesk, and B. W. Kernighan. "The C Programming Language". In: *Bell Sys. Tech. J* 57.6 (1978), pp. 1991–2019. (Visited on 12/06/2023).

[223]   Eric Roberts. "An Overview of MiniJava". In: *ACM SIGCSE Bulletin* 33.1 (Feb. 2001), pp. 1–5. ISSN: 0097-8418. DOI: 10.1145/366413.364525.

[224]   Eric Roberts and Keith Schwarz. "A Portable Graphics Library for Introductory CS". In: *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE '13*. Canterbury, England, UK: ACM Press, 2013, p. 153. ISBN: 978-1-4503-2078-8. DOI: 10.1145/2462476.2465590.

[225]   Adam Carl Rule. "Design and Use of Computational Notebooks". PhD thesis. University of California, San Diego, 2018.

[226] J. Sajaniemi. "An Empirical Analysis of Roles of Variables in Novice-Level Procedural Programs". In: *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*. Sept. 2002, pp. 37–39. DOI: 10.1109/HCC.2002.1046340.

[227] Kate Sanders and Robert McCartney. "Threshold Concepts in Computing: Past, Present, and Future". In: *Proceedings of the 16th Koli Calling International Conference on Computing Education Research - Koli Calling '16*. Koli, Finland: ACM Press, 2016, pp. 91–100. ISBN: 978-1-4503-4770-9. DOI: 10.1145/2999541.2999546.

[228] Bianca L. Santana and Roberto A. Bittencourt. "Increasing Motivation of CS1 Non-Majors through an Approach Contextualized by Games and Media". In: *2018 IEEE Frontiers in Education Conference (FIE)*. Oct. 2018, pp. 1–9. DOI: 10.1109/FIE.2018.8659011.

[229] Igor Moreno Santos, Matthias Hauswirth, and Nathaniel Nystrom. "Experiences in Bridging from Functional to Object-Oriented Programming". In: *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E - SPLASH-E 2019*. Athens, Greece: ACM Press, 2019, pp. 36–40. ISBN: 978-1-4503-6989-3. DOI: 10.1145/3358711.3361628.

[230] Jaromir Savelka, Arav Agarwal, Marshall An, Chris Bogart, and Majd Sakr. "Thrilled by Your Progress! Large Language Models (GPT-4) No Longer Struggle to Pass Assessments in Higher Education Programming Courses". In: *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*. Vol. 1. ICER '23. New York, NY, USA: Association for Computing Machinery, Sept. 2023, pp. 78–92. ISBN: 978-1-4503-9976-0. DOI: 10.1145/3568813.3600142.

[231] Emmanuel Schanzer, Kathi Fisler, and Shriram Krishnamurthi. "Bootstrap: Going beyond Programming in after-School Computer Science". In: *SPLASH Education Symposium*. 2013.

[232] Emmanuel Schanzer, Kathi Fisler, Shriram Krishnamurthi, and Matthias Felleisen. "Transferring Skills at Solving Word Problems from Computing to Algebra Through Bootstrap". In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education - SIGCSE '15*. Kansas City, Missouri, USA: ACM Press, 2015, pp. 616–621. ISBN: 978-1-4503-2966-8. DOI: 10.1145/2676723.2677238.

[233]   Emmanuel Schanzer, Shriram Krishnamurthi, and Kathi Fisler. "Creativity, Customization, and Ownership: Game Design in Bootstrap: Algebra". In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. Baltimore Maryland USA: ACM, Feb. 2018, pp. 161–166. ISBN: 978-1-4503-5103-4. DOI: 10.1145/3159450.3159471.

[234]   Emmanuel Tanenbaum Schanzer. "Algebraic Functions, Computer Programming, and the Challenge of Transfer". PhD thesis. Harvard University, 2015.

[235]   Otto Seppälä, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. "Do We Know How Difficult the Rainfall Problem Is?" In: *Proceedings of the 15th Koli Calling Conference on Computing Education Research*. Koli Calling '15. New York, NY, USA: ACM, 2015, pp. 87–96. ISBN: 978-1-4503-4020-5. DOI: 10.1145/2828959.2828963.

[236]   Sadia Sharmin. "Creativity in CS1: A Literature Review". In: *ACM Transactions on Computing Education* 22.2 (June 2022), pp. 1–26. ISSN: 1946-6226, 1946-6226. DOI: 10.1145/3459995.

[237]   Lee S. Shulman. "Those Who Understand: Knowledge Growth in Teaching". In: *Educational Researcher* 15.2 (Feb. 1986), pp. 4–14. ISSN: 0013-189X. DOI: 10.3102/0013189X015002004.

[238]   Valerie J. Shute, Chen Sun, and Jodi Asbell-Clarke. "Demystifying Computational Thinking". In: *Educational Research Review* 22 (Nov. 2017), pp. 142–158. ISSN: 1747-938X. DOI: 10.1016/j.edurev.2017.09.003.

[239]   Robert M. Siegfried, Katherine G. Herbert-Berger, Kees Leune, and Jason P. Siegfried. "Trends Of Commonly Used Programming Languages in CS1 And CS2 Learning". In: *2021 16th International Conference on Computer Science & Education (ICCSE)*. Aug. 2021, pp. 407–412. DOI: 10.1109/ICCSE51940.2021.9569444.

[240]   Beth Simon, Päivi Kinnunen, Leo Porter, and Dov Zazkis. "Experience Report: CS1 for Majors with Media Computation". In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*. Bilkent Ankara Turkey: ACM, June 2010, pp. 214–218. DOI: 10.1145/1822090.1822151.

[241]   Robert H. Sloan and Patrick Troy. "CS 0.5: A Better Approach to Introductory Computer Science for Majors". In: *ACM SIGCSE Bulletin* 40.1 (Mar. 2008), pp. 271–275. ISSN: 0097-8418. DOI: 10.1145/1352322.1352230.

[242] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. "Cognitive Strategies and Looping Constructs: An Empirical Study". In: *Communications of the ACM* 26.11 (Nov. 1983), pp. 853–860. ISSN: 0001-0782. DOI: 10.1145/182.358436.

[243] Juha Sorva. "Misconceptions and the Beginner Programmer". In: *Computer Science Education : Perspectives on Teaching and Learning in School*. Ed. by Sue Sentance, Erik Barendsen, Nicol R. Howard, and Carsten Schulte. 1st ed. London: Bloomsbury Academic, 2023, pp. 259–274. ISBN: 978-1-350-29694-7.

[244] Juha Sorva. "Visual Program Simulation in Introductory Programming Education". PhD thesis. Espoo, Finland: Aalto University, 2012.

[245] GitHub Staff. *Octoverse: AI Leads Python to Top Language as the Number of Global Developers Surges*. Oct. 2024. URL: https://github.blog/news-insights/octoverse/octoverse-2024/ (visited on 07/07/2025).

[246] Guy L. Steele. "Growing a Language". In: *Addendum to the 1998 Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (Addendum)*. OOPSLA '98 Addendum. New York, NY, USA: Association for Computing Machinery, Jan. 1998, 0.01–A1. ISBN: 978-1-58113-286-1. DOI: 10.1145/346852.346922.

[247] Wayne Stevens, Glenford Myers, and Larry Constantine. "Structured Design". In: *IBM Systems Journal* 13.2 (1974), pp. 115–139. ISSN: 0018-8670. DOI: 10.1147/sj.132.0115.

[248] Filip Strömbäck, Pontus Haglund, Aseel Berglund, and Erik Berglund. "The Progression of Students' Ability to Work With Scope, Parameter Passing and Aliasing". In: *Proceedings of the 25th Australasian Computing Education Conference*. Melbourne VIC Australia: ACM, Jan. 2023, pp. 39–48. DOI: 10.1145/3576123.3576128.

[249] Gail M. Sullivan and Richard Feinn. "Using Effect Size—or Why the P Value Is Not Enough". In: *Journal of Graduate Medical Education* 4.3 (Sept. 2012), pp. 279–282. ISSN: 1949-8349. DOI: 10.4300/JGME-D-12-00156.1.

[250] Joshua Sunshine, James D. Herbsleb, and Jonathan Aldrich. "Structuring Documentation to Support State Search: A Laboratory Experiment about Protocol Programming". In: *ECOOP 2014 – Object-Oriented Programming*. Ed. by Richard Jones. Berlin, Heidelberg: Springer, 2014, pp. 157–181. ISBN: 978-3-662-44202-9. DOI: 10.1007/978-3-662-44202-9_7.

[251] Swiss Federal Council. *AS 2018 2669 - Verordnung über die Anerkennung von gymnasialen Maturitätsausweisen*. 2018. URL: https://www.fedlex.admin.ch/eli/oc/2018/387/de (visited on 06/05/2024).

[252] Jialiang Tan, Yu Chen, and Shuyin Jiao. *Visual Studio Code in Introductory Computer Science Course: An Experience Report*. Mar. 2023. arXiv: 2303.10174 [cs]. URL: http://arxiv.org/abs/2303.10174 (visited on 07/05/2024).

[253] Cynthia Taylor, Jaime Spacco, David P. Bunde, Zack Butler, Heather Bort, Christopher Lynnly Hovey, Francesco Maiorana, and Thomas Zeume. "Propagating the Adoption of CS Educational Innovations". In: *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. Larnaca Cyprus: ACM, July 2018, pp. 217–235. ISBN: 978-1-4503-6223-8. DOI: 10.1145/3293881.3295785.

[254] TBPN. *Amjad Masad on Replit, AI Agents, and the Death of Traditional Software*. Mar. 2025. URL: https://www.youtube.com/watch?v=TpeRtL2nsCY (visited on 07/17/2025).

[255] Allison Elliott Tew, Charles Fowler, and Mark Guzdial. "Tracking an Innovation in Introductory CS Education from a Research University to a Two-Year College". In: *ACM SIGCSE Bulletin* 37.1 (Feb. 2005), pp. 416–420. ISSN: 0097-8418. DOI: 10.1145/1047124.1047481.

[256] The Pyret Crew. *The Pyret Programming Language*. URL: http://pyret.org/ (visited on 07/03/2024).

[257] *The Python Standard Library*. URL: https://docs.python.org/3/library/index.html (visited on 07/03/2024).

[258] Bruce Thompson. *Best Practices in Quantitative Methods*. SAGE Publications, Inc., 2008. ISBN: 978-1-4129-9562-7. DOI: 10.4135/9781412995627.

[259] Nicole Trachsler. "WebTigerJython - A Browser-based Programming IDE for Education". In: (2018), 77 p. DOI: 10.3929/ETHZ-B-000338593. HDL: 20.500.11850/338593.

[260] Vicki Trowler. *Student Engagement Literature Review*. York: The Higher Education Academy, 2010.

[261] Ethel Tshukudu and Quintin Cutts. "Understanding Conceptual Transfer for Students Learning New Programming Languages". In: *Proceedings of the 2020 ACM Conference on International Computing Education Research*. ICER '20. New York, NY, USA: Association for Computing Machinery, Aug. 2020, pp. 227–237. ISBN: 978-1-4503-7092-9. DOI: 10.1145/3372782.3406270.

[262] Preston Tunnell Wilson, Kathi Fisler, and Shriram Krishnamurthi. "Evaluating the Tracing of Recursion in the Substitution Notional Machine". In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. SIGCSE '18. New York, NY, USA: ACM, 2018, pp. 1023–1028. ISBN: 978-1-4503-5103-4. DOI: 10.1145/3159450.3159479.

[263] Gias Uddin and Martin P. Robillard. "How API Documentation Fails". In: *IEEE Software* 32.4 (July 2015), pp. 68–75. ISSN: 1937-4194. DOI: 10.1109/MS.2014.80.

[264] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention Is All You Need". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., Dec. 2017, pp. 6000–6010. ISBN: 978-1-5108-6096-4.

[265] Bret Victor. *Learnable Programming*. Sept. 2012. URL: https://worrydream.com/LearnableProgramming/ (visited on 02/01/2025).

[266] A. Vihavainen, J. Helminen, and P. Ihantola. "How Novices Tackle Their First Lines of Code in an IDE: Analysis of Programming Session Traces". In: *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*. Koli Finland: ACM, Nov. 2014, pp. 109–116. ISBN: 978-1-4503-3065-7. DOI: 10.1145/2674683.2674692.

[267] *Visual Studio Code*. URL: https://code.visualstudio.com/ (visited on 07/05/2024).

[268] Paul Voigt and Axel Von Dem Bussche. *The EU General Data Protection Regulation (GDPR)*. Cham: Springer International Publishing, 2017. ISBN: 978-3-319-57958-0. DOI: 10.1007/978-3-319-57959-7.

[269] J. von Neumann. "First Draft of a Report on the EDVAC". In: *IEEE Annals of the History of Computing* 15.4 (1993), pp. 27–75. ISSN: 1934-1547. DOI: 10.1109/85.238389.

[270] Lev S. Vygotsky. *Mind in Society: The Development of Higher Psychological Processes*. Vol. 86. Harvard university press, 1978. (Visited on 07/11/2025).

[271] Christopher Watson and Frederick W.B. Li. "Failure Rates in Introductory Programming Revisited". In: *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*. ITiCSE '14. New York, NY, USA: Association for Computing Machinery, June 2014, pp. 39–44. ISBN: 978-1-4503-2833-3. DOI: 10.1145/2591708.2591749.

[272]   *WebTigerJython*. URL: https://webtigerjython.ethz.ch/ (visited on 07/03/2024).

[273]   David Weintrop, Alexandria K. Hansen, Danielle B. Harlow, and Diana Franklin. "Starting from Scratch: Outcomes of Early Computer Science Learning Experiences and Implications for What Comes Next". In: *Proceedings of the 2018 ACM Conference on International Computing Education Research*. Espoo Finland: ACM, Aug. 2018, pp. 142–150. ISBN: 978-1-4503-5628-2. DOI: 10.1145/3230977.3230988.

[274]   David Weintrop and Uri Wilensky. "Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs". In: *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*. ICER '15. New York, NY, USA: Association for Computing Machinery, Aug. 2015, pp. 101–110. ISBN: 978-1-4503-3630-7. DOI: 10.1145/2787622.2787721.

[275]   Robert S. Weiss. *Learning from Strangers: The Art and Method of Qualitative Interview Studies*. Simon and Schuster, 1995. (Visited on 06/26/2025).

[276]   Richard L. Wexelblat. *History of Programming Languages*. Academic Press, May 2014. ISBN: 978-1-4832-6616-9.

[277]   D. J. Wheeler. "The Use of Sub-Routines in Programmes". In: *Proceedings of the 1952 ACM National Meeting (Pittsburgh) on - ACM '52*. Pittsburgh, Pennsylvania: ACM Press, 1952, pp. 235–236. DOI: 10.1145/609784.609816.

[278]   Cameron Wilson. "Hour of Code—a Record Year for Computer Science". In: *ACM Inroads* 6.1 (Feb. 2015), pp. 22–22. ISSN: 2153-2184, 2153-2192. DOI: 10.1145/2723168.

[279]   Margaret Wilson. "Six Views of Embodied Cognition". In: *Psychonomic Bulletin & Review* 9.4 (Dec. 2002), pp. 625–636. ISSN: 1069-9384, 1531-5320. DOI: 10.3758/BF03196322.

[280]   Jeannette M Wing. "Computational Thinking and Thinking about Computing". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 366.1881 (Oct. 2008), pp. 3717–3725. ISSN: 1364-503X, 1471-2962. DOI: 10.1098/rsta.2008.0118.

[281]   Jeannette M. Wing. "Computational Thinking". In: *Communications of the ACM* 49.3 (Mar. 2006), pp. 33–35. ISSN: 0001-0782. DOI: 10.1145/1118178.1118215.

[282]   John Wrenn and Shriram Krishnamurthi. "Executable Examples for Program-
        ming Problem Comprehension". In: *Proceedings of the 2019 ACM Conference
        on International Computing Education Research*. ICER '19. New York, NY, USA:
        Association for Computing Machinery, July 2019, pp. 131–139. ISBN: 978-1-
        4503-6185-9. DOI: 10.1145/3291279.3339416.

[283]   Huiping Wu and Shing-On Leung. "Can Likert Scales Be Treated as Interval
        Scales?—A Simulation Study". In: *Journal of Social Service Research* 43.4 (Aug.
        2017), pp. 527–532. ISSN: 0148-8376. DOI: 10.1080/01488376.2017.
        1329775.

[284]   Robert K. Yin. *Case Study Research: Design and Methods*. Fifth edition. Los An-
        geles: SAGE, 2014. ISBN: 978-1-4522-4256-9.

[285]   Brent Yorgey. *Diagrams - Diagrams + Cairo + Gtk + Mouse Picking, Reloaded*.
        2015. URL: https://diagrams.github.io/blog/2015-04-30-GTK-
        coordinates.html (visited on 07/20/2025).

[286]   Brent A. Yorgey. "Monoids: Theme and Variations (Functional Pearl)". In: *ACM
        SIGPLAN Notices* 47.12 (Sept. 2012), pp. 105–116. ISSN: 0362-1340. DOI:
        10.1145/2430532.2364520.

[287]   Žana Žanko, Monika Mladenović, and Ivica Boljat. "Misconceptions about Vari-
        ables at the K-12 Level". In: *Education and Information Technologies* 24.2 (Mar.
        2019), pp. 1251–1268. ISSN: 1360-2357, 1573-7608. DOI: 10.1007/s10639-
        018-9824-1.