Teaching Programming with Graphics: Pitfalls and a Solution

Luca Chiodini
luca.chiodini@usi.ch
Software Institute, Università della Svizzera italiana
Lugano, Switzerland

Juha Sorva
juha.sorva@aalto.fi
Department of Computer Science, Aalto University
Espoo, Finland

Matthias Hauswirth
matthias.hauswirth@usi.ch
Software Institute, Università della Svizzera italiana
Lugano, Switzerland

Abstract

Many introductory programming courses employ graphics libraries that promote engagement and enable fun visuals. However, student excitement over graphical outputs is not a guarantee of conceptual understanding of programming, and graphics may even distract from intended learning outcomes. Our contribution is twofold. First, we analyze a selection of existing graphics libraries designed for novice programmers. We consider how these libraries foster clean decomposition, direct students' attention to key content, and manage complexity; we find shortcomings in these respects. These shortcomings involve the libraries' support for global coordinates and external graphics, as well as their rich APIs; we argue that these features, although powerful, are also potential pitfalls in student learning. Second, we present the design of a new graphics library, PyTamaro, which avoids the pitfalls with a minimalist design that eschews coordinates; we also outline a pedagogical approach that builds on PyTamaro's strengths and deliberate limitations. We briefly discuss PyTamaro's trade-offs in comparison to coordinate-based libraries. The work reported here paves the way for future empirical evaluations of PyTamaro and associated teaching practices.

CCS Concepts: • **Social and professional topics → Computer science education**.

Keywords: education, graphics, library, visual, novices, programming, decomposition

ACM Reference Format:
Luca Chiodini, Juha Sorva, and Matthias Hauswirth. 2023. Teaching Programming with Graphics: Pitfalls and a Solution. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E (SPLASH-E '23), October 25, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3622780.3623644

SPLASH-E '23, October 25, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0390-4/23/10…$15.00
https://doi.org/10.1145/3622780.3623644

1 Introduction

"Why should computers in schools be confined to computing the sum of the squares of the first twenty odd numbers?" complained Papert and Solomon back in 1971 [39]. As a remedy, they proposed that students program a 'turtle' that can draw onscreen. Turtle graphics are a famous example of a broader phenomenon: teachers seek to engage beginner programmers with graphics. Besides turtle graphics, many software tools have been built for this purpose, including programmable sprites in block-based environments, software libraries with graphical outputs, beginner-friendly GUI toolkits, and so on (see, e.g., [19, 33]).

Despite the spread of such approaches over the last decades, many textbooks and courses still follow a very traditional curriculum [44] that introduces the syntax and semantics of a programming language through programs that produce text akin to "Hello, world!" or perform basic numerical calculations. Such programs have been criticized by many as "tedious and dull" [19]. Nevertheless, many educators stick with them, perhaps because they perceive more interesting exercises as being too difficult for novices.

For many students, graphics are more engaging than text and numbers—and engagement has been long sought by educators, as it is robustly correlated with improved learning outcomes [48]. However, the complexity of a graphics library may diminish the educational impact of graphical programming or even cause novices to fail. For a graphics library to be a realistic choice for teaching complete beginners, it needs to have *Manageable Complexity*. This is the first of three *guiding goals* that we strive towards in this article.

For some purposes, such as getting children intrigued about computing in general, engagement may be almost an end to itself. On the other hand, graphical approaches to programming education are sometimes criticized as overly playful and not focused enough on the core of computing; engagement alone is not enough for learning key computing concepts and programming skills. Moreover, there is evidence, for instance, that students may engage with the graphical affordances of Scratch in ways that are exciting to them but unproductive for programming skill development (e.g., [17, 50]). Ideally, beginner programmers would

engage enthusiastically with content that is aligned with intended learning outcomes. This kind of desirable engagement, which minimizes the time when learners' direct their attention to superficial concerns, is the second of our guiding goals. We dub it *Meaningful Engagement.*

What qualifies as a key computing concept is subject to debate, but there is little dispute over the importance of programmers learning how to decompose problems. Decomposition is an essential skill that enables humans to solve non-trivial problems; alongside the related skill of abstraction, decomposition is also central to "computational thinking" [51]. As Brennan and Resnick state, *"abstracting and modularizing, which we characterize as building something large by putting together collections of smaller parts, is an important practice for all design and problem solving"* [6].

However, and as the Software Engineering community realized a long time ago, not every way to divide a problem into subproblems—a big 'system' into 'subsystems' or 'modules'—is equally good. In 1974, Stevens et al. stressed the paramount importance of structured design: *"the fewer and simpler the connections between modules, the easier it is to understand each module"* [47]. Around the same time, Parnas explained how one should modularize a system so that modules "reveal as little as possible about their inner workings" [40]. Our third guiding goal in this article is that students get to practice *Clean Problem Decomposition* early on. We argue that learners at various levels of the educational system may reap the benefits of learning decomposition skills.

Putting together all of the above, we can summarize our contribution as follows. We explore the design space of graphics libraries for introductory programming courses that employ a textual programming language, with a particular emphasis on our guiding goals: *Clean Problem Decomposition*, *Meaningful Engagement*, and *Manageable Complexity*. Section 2 below provides an overview of existing graphics libraries. In Section 3, we explore those existing solutions to answer our first research question:

RQ1 *Given the above guiding goals, what pitfalls are there in using existing graphics libraries for introductory programming education?*

Multiple pitfalls are revealed, so we ask how to avoid them:

RQ2 *How can a graphics library be designed to support Clean Problem Decomposition, Meaningful Engagement, and Manageable Complexity?*

Section 4 presents our answer to this question, a graphics library named PyTamaro; we describe PyTamaro's design rationale as well as a teaching approach that builds on the library's strengths and that we have anecdotal experience of. Section 5 covers some limitations of our work. Finally, Section 6 considers future empirical evaluations of PyTamaro and concludes the article.

2 Related Work: Graphics for Education

Many software libraries have been developed to support graphics in introductory programming classes. We will characterize this design space by reviewing three 'families' of libraries. We will restrict our focus on 2D graphics, on textual languages, and on tools designed with education in mind; already within this scope, there is radical variation in designs. GUI libraries, game engines, and blocks-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 present purposes.

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 1. The 'house' consists of a single 'floor,' represented by a square, atop which sits the 'roof,' an equilateral triangle.

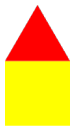

Figure 1. Abelson and diSessa's [1] house, colorized.

2.1 Graphics on a Canvas with Coordinates

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 [29], `acm.graphics`, and the Portable Graphics Library derived from the latter [43],

Listing 1 draws a house with `cs1graphics` [16] 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.

```
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 1. Drawing a house with the `cs1graphics` library.

Not all libraries in this family require calling methods on objects. For example, Designer [24] creates images with plain function calls and modifies their state with subscripts, as in `floor['x'] = 50`.

2.2 Turtle Graphics

Turtle graphics was introduced by Papert [38] as a simple way to draw with computers as early as in elementary school. It is based on a metaphor 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 in Logo, turtle graphics have 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 2.

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

Listing 2. Drawing a house with Python's `turtle` module.

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.

2.3 Graphics as Values

A different approach was proposed in 1982 by Henderson [22], who introduced a purely functional way to describe pictures. Finne and Peyton Jones [14] later proposed the idea of *composing* pictures from primitives using what they called *combinators*. Some of these principles have been adopted in the textbook *How to Design Programs* [11]: an "image teachpack" [4] accompanies the book and is available as a Racket library. Felleisen and Krishnamurthi [12] discuss how the teachpack enables students to construct algebraic expressions that "consume and compute pictorial values."

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 functional programming. In Listing 3, we build the house with Pyret, whose syntax is similar to Python's.

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

Listing 3. Drawing a house with Pyret's `image` module.

3 Pitfalls in Existing Libraries

In this section, we answer our first research question: what pitfalls are there in using existing graphics libraries in the light of our guiding goals (Clean Problem Decomposition, Meaningful Engagement, and Manageable Complexity)?

The results of our analysis are summarized in Table 1 and explained below. In the subsections that follow, we discuss one guiding goal at a time, identifying pitfalls related to that goal. For each pitfall identified, we argue why its presence in a graphics library for novices can hinder reaching the intended goal. Our analysis draws on the research literature on computing education and is complemented by anecdotal evidence from teaching introductory programming.

3.1 Clean Problem Decomposition

Problem decomposition is at the core of "computational thinking" [51]. Decomposition—along the corresponding *composition* of sub-solutions to solve an overall problem—has been considered "the essence of programming" [36]. However, a look a student programs reveals that achieving proper decomposition is quite difficult. For example, Scratch projects often exhibit decomposition into substructures that are not coherent [35]. These findings also extend to code written by novices in text-based programming languages [28].

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

Table 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`	**Graphics as Values** Pyret's `image`
Clean Problem Decomposition	Global coordinates	✗	(✗)	
	Turtle's state		✗	
	Local coordinates		✗	✗
	Scaling	✗		✗
Meaningful Engagement	External graphics	✗		✗
	Rich API	(✗)	(✗)	✗
Manageable Complexity	Extra lang. features	✗		
	Mutability	✗		

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.

Learners thus need to be helped to learn this difficult skill. They need opportunities to practice decomposition; we argue that it is a good idea to put learners in situations where the benefits of clean decomposition are clear and it is difficult or even impossible to exploit 'ugly hacks.'

What pitfalls are there in existing graphics libraries when teaching how to cleanly decompose a problem? We find four: *global coordinates*, *global state*, *local coordinates*, and *scaling*.

As we discuss each pitfall in turn below, we again resort to the 'house' from Figure 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.

3.1.1 Global coordinates break independence. Consider Listing 1, which draws the house on a canvas. At first glance, it may seem that the floor and the roof are constructed quite independently from each other before being separately added to the canvas. There is one caveat, however: the graphics are positioned using offsets in a global coordinate system.

Coordinates such as `(50, 137)` implicitly depend on the origin of the plane, a globally shared 'zero' that acts as a reference point. The issue becomes evident when one needs to change a feature in a subproblem. 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 breaks this promise, a pitfall shared by all the libraries in the family that adopts global coordinates (Section 2.1).

3.1.2 Turtle state also breaks independence. One of the original goals of "turtle geometry" [38] 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 2 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," as Abelson and diSessa advocated [1]. Things are not so simple, however, as shown by the extra command on Line 4 of Listing 2: `left(60)`. To 'connect' the subproblems—to compose the graphics—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" [1]) 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. Global state violates the independence that we strive for through decomposition.

Harvey [21] 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 (ibid.) partial patch for this issue is to have a higher-order procedure reset the turtle's heading. 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.

3.1.3 Local coordinates are prone to misuse. Listing 3 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. Overlaying enables a huge class of more interesting graphics to be created.

One might rightfully object that 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 [12].

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

The issue may also extend to teaching materials. Listing 4 exhibits an example[1] taken from an Hour of Code activity by

[1]Step 11 of https://www.bootstrapworld.org/materials/fall2023/en-us/lessons/hoc-winter-parley/index.html.

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

3.1.4 Scalable graphics may interfere with abstraction.

Devlin argues that *"computing is all about constructing, manipulating, and reasoning about abstractions"* [9]. 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 [20].

Graphical programs offer many opportunities for abstraction. 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 Section 2, a change in this single aspect necessitates multiple changes to code: Listing 1 needs seven edits, Listings 2 and 3 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 practised; this is the sort of 'hack' that we would like to prevent.

3.2 Meaningful Engagement

3.2.1 External graphics may lower motivation.

When a student creates their first graphical program from scratch, their sense of empowerment is often palpable: it's 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 doesn't 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 [24] even offers an `emoji` function. There are lots of fun graphics out there, and letting students select custom images has been shown to contribute to their sense of ownership over the resulting program [45]. This therefore sounds like a great feature (and it can be, in the right context).

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. If the goal is to use graphics to teach decomposition and related concepts—as it is for us—then external images are distracting and potentially demotivating.

3.2.2 Rich APIs shift the emphasis from programming to libraries and 'spoil' opportunities for learning.

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" [2]. 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 in a course adds something to what students need to learn. How much is added may vary 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 documentation.

Consider Pyret's `image` module. Just for the purpose of composing images, there are eleven functions, seven of which place an image on top of another: `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. But if they find canned solutions for these problems in a library, there is little motivation 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.

3.3 Manageable Complexity

3.3.1 At the very beginning, some language features are unnecessary and distracting. Beginner programmers often resort to "bricolage": extensive trial-and-error whose "*manifestation ... is endless debugging: try it and see what happens*" [5]. While experimentation is certainly valuable, ineffective experimentation is common and leads to frustration and poor learning outcomes. Even when a student writes a program that produces the correct output, there is no guarantee of understanding—this has been recently demonstrated in multiple studies (e.g., [27, 32]). As teachers, our aim is to ensure that learners understand the source code they write. This means that we 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 1 features instantiations with and without parameters, method invocations, and lists. Introducing all these constructs "from the first day" [16], 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, however, that this approach is incompatible with the goal of learners understanding the concepts in the code they write [37]. 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., [2, 16]). But at the very beginning of an introductory course, object-oriented language features add complexity. To understand the third line of Listing 1, for example, students need to understand function invocations *and* how functions (methods) relate to the objects on which they are invoked.

3.3.2 Mutability makes it harder to reason about programs. Mutable state is often introduced early, even though it breaks referential transparency and demands a more complex mental model for reasoning about programs [49]. There is extensive research (e.g., [8, 23, 34, 42, 46]) on misconceptions that novices have about assignments, both with primitive values and references [34], and to (mutable) objects in general [23].

Some graphics libraries, too, model images as objects with a mutable state. Consider Listing 1 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` and `floor2.moveTo(50, 237)`, only to be puzzled about the unexpected result. As illustrated, mutable state is closely associated with *aliasing*, another concept that is fundamental but not easy: difficulties abound even among upper-level undergraduates [15].

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

4 PyTamaro's Design and a Teaching Approach

Our second research question seeks a design that avoids the pitfalls identified above. In this section, we sketch an answer by introducing **PyTamaro**, a minimalist library for Python, publicly available as open source at https://github.com/LuCEresearchlab/pytamaro. We illustrate the key aspects of PyTamaro's design gradually, alongside a teaching approach that builds on the library's strengths, leveraging the experience that we have accumulated over the last couple of years while using the library in high schools and teacher-training courses.

4.1 An Initial Example

Listing 5 below shows how to draw a simple graphic with PyTamaro, resorting one last time to the house example of Figure 1. Unlike our previous code listings, Listing 5 is a standalone Python program, complete with an import statement and a function call to display the resulting graphic.

```
from anon 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 5. Drawing a house with PyTamaro.

All the programming language constructs used in Listing 5 *can be 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 also used when importing and calling a function from the standard library to compute the square root of a number and print the result—a common example in introductory programming without graphics.

PyTamaro belongs to the family from Section 2.3: it treats graphics as values. Therefore, when students reason about graphical programs, they can rely on the same mental model that they use for expressions that operate on numbers [12]. This design choice avoids three previously identified pitfalls: there is *no global coordinate system* and *no stateful turtle*, and all *graphics are immutable*.

4.2 Defining Abstractions Early

Listing 5 looks similar to the earlier Listing 3, and indeed the two share many key aspects. The keen eye also notices a subtle difference: PyTamaro does not seem to have a function for a square, such a primitive shape! This is a deliberate design choice so as to *avoid a rich API* (Section 3.2.2). 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 [7]. We argue that novices also need early opportunities to *define* their own abstractions [31] and indeed should be placed in situations that veritably *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 6.

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

Listing 6. A function to create a square with PyTamaro.

Implementing such a function is challenging for many beginners [25], 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 a lot of throw-away code that we do not bother packaging into cleanly defined functions. Instead, students are encouraged and supported to build their own 'toolbox,' enriching it with functions they implemented and deem useful. Over time, they create increasingly interesting graphics that are nevertheless entirely based on their very own code.

An auxiliary benefit of custom functions is that they help gradually introduce type annotations (which are optional in Python and PyTamaro). 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 7. We have multiple reasons for introducing type annotations to beginners: one is that types guide the design of programs [10]; another is that types help catch errors early. IDEs with a static type checker for Python (e.g., Thonny [3] for education) show warnings when types do not match, giving students rapid 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 [30]; types make the distinction explicit and checkable.

4.3 Visual Problem Decomposition

PyTamaro's design does not expose any coordinate system and does not maintain state; this supports clean decomposition. Below, we discuss other aspects in PyTamaro's design

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

Listing 7. A function to create an equilateral triangle, with type annotations.

that further help students engage with decomposition despite the library's limitations—or even because of them.

4.3.1 Basic visual decomposition. Graphics in PyTamaro make decomposition *visually* apparent and relatively easy to intuit. This can be leveraged in teaching. Consider the emblem of the International Red Cross at the top of Figure 2. 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 also *meaningfully engages* 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. We must repeatedly break down our problem into smaller ones until we reach elementary ones: a powerful *recursive* process for problem-solving. Visualizing the entire (de)composition as a tree can help to reify the (de)composition process, and the visualization can support explanations and discussions in class.

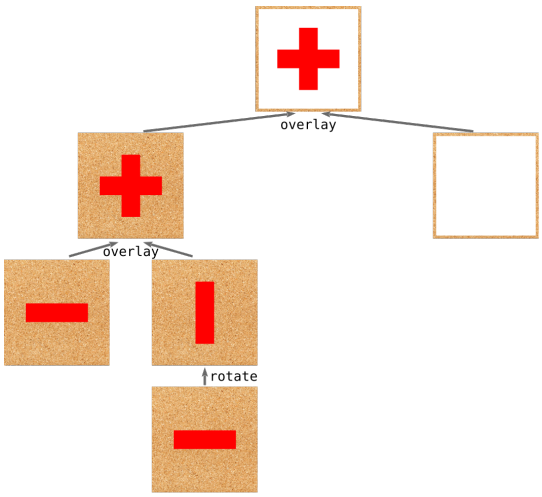

Figure 2. Hierarchical (de)composition of the IRC emblem. The bars' "cork board" background indicates transparency.

This hierarchical composition of independent subproblems 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.

4.3.2 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 `+` is an operator that 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 have not been an equally valid way to compose the emblem in Figure 2.

Now consider the Italian flag shown twice at the top of Figure 3. 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. As shown in Figure 3, there are multiple 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.

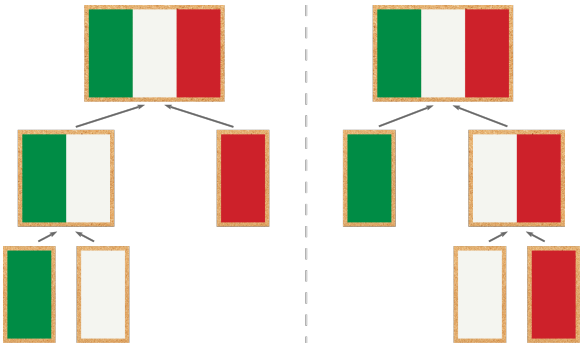

Figure 3. Two different but equally valid ways to compose the Italian flag, exploiting the associativity of `beside`.

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.

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.

4.3.3 More challenging 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 3.1.3)?

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' [13] with a cork board, a pin, and paper cutouts as described below (cf. [16]). 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 4 illustrates the (de)composition of a graphic using pins.

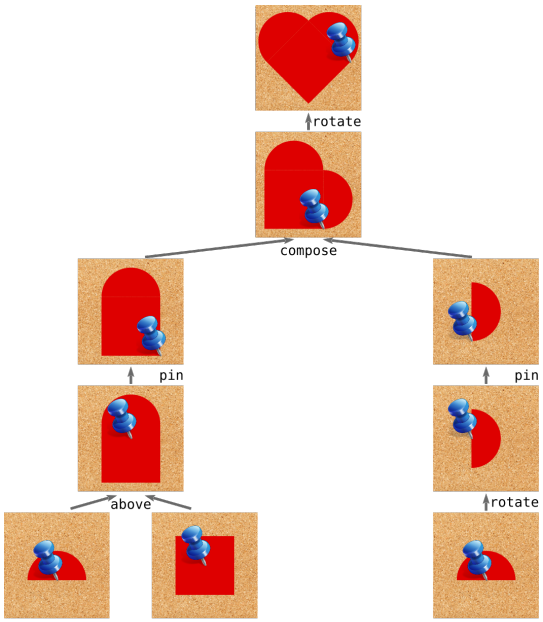

Figure 4. Drawing a heart with `pin` and `compose`. The tree leaves correspond to PyTamaro primitives.

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`).

Once learners familiarize themselves with this way of composing, they can draw many more challenging and interesting graphics. The right-hand side of Figure 5 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 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.

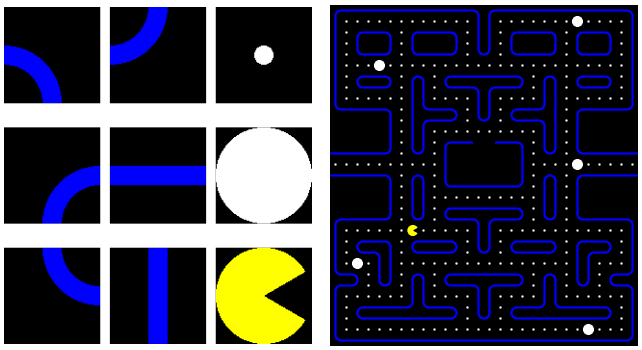

Figure 5. A Pac-Man maze (right) created out of tiles (left), which are in turn composed from PyTamaro's primitives.

4.3.4 Meaningful graphics.

Teaching programming with PyTamaro does not need to be limited to flags or dry geometric shapes. Students can 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 6. 'Warming stripes' created with PyTamaro.

Figure 6 shows the famous 'warming stripes' visualization of temperature anomalies over time. There are several questions involved in creating such a graphic; here we only explore an important one related to PyTamaro's design.

The essence of Figure 6 is a sequence of colored rectangles side by side. Placing many graphics in a row is indeed a sub-task that occurs frequently. It is worthwhile to define a general function to solve the problem once and add it to one's 'toolbox' (like `square` from Section 4.2).

Listing 8 shows one possible solution, which also handles the corner cases where the list is empty or contains just one graphic[2]. 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*.

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

Listing 8. A general function to place many graphics next to each other.

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 [52]. 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" [52]. PyTamaro's design gives a taste of that elegance and power to novice programmers.

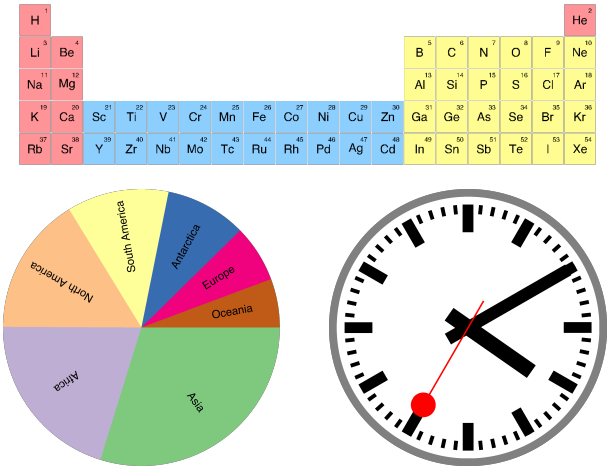

Figure 7. Three examples of meaningful graphics created with PyTamaro: the periodic table of chemical elements (top, first five periods only), a pie chart of area by continent (bottom left), and the Swiss railway clock (bottom right).

Figure 7 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

[2]PyTamaro does not require assignment statements. The solution can also be implemented using recursion or `reduce`.

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.

4.4 Localizing for Human and Programming Languages

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.

First, not all introductory courses use Python[3]. The minimalism of PyTamaro's design reduces the cost of porting the library to new programming languages. In fact, we have already implemented a Java library with the exact same design, and it is being used in a first-year university course.

Second, students' learning of programming should not be hampered by their native language. Research has demonstrated that the prevalence of English can be a barrier for non-native speakers [18]. This is especially true now: students around the world start programming at increasingly young ages and are not necessarily comfortable reading and writing English.

PyTamaro attends to this. The documentation is available in multiple languages, and the entire API is localized. For example, an Italian-speaking student can make a triangle with `triangolo`, whose parameters, types, and error messages are localized; a German-speaker can use `dreieck`.

5 Limitations

There are various limitations to the work presented here.

Although our guiding goals—Clean Problem Decomposition, Meaningful Engagement, and Manageable Complexity—are amply supported by the research literature, they are certainly not the only considerations when designing a pedagogy. We make no claims about their superiority over other legitimate goals in graphics-based programming education.

In Section 2, we explored this design space in order to identify a few families of libraries, but we did not conduct a systematic, comprehensive review of all graphics libraries for education. Moreover, our analysis of pitfalls in Section 3 is primarily based on only three libraries that we consider representative of the different families. We argue that certain design choices constitute pitfalls, drawing in part on the extant research literature and prior empirical work. However, we do not have direct evidence of how frequently each pitfall translates to problems for students in practice.

The design of any library for beginners involves major trade-offs, and this is also true for PyTamaro. PyTamaro is designed with certain guiding goals in mind and emphasizes those; the choices we made in its design are not optimal or even viable for every introductory programming context. For example, some teaching approaches revolve around games or simulations that take place in evolving 2D worlds; PyTamaro is not designed for this. Placing externally loaded graphics at arbitrary positions in a coordinate system is a powerful feature that can be misused but does support reasonable pedagogical goals other than the ones we have adopted here.

A major limitation of our work is that PyTamaro's ability to avoid the design pitfalls was analyzed from a theoretical point of view only. Although our discussion is consistent with our impressions from classrooms, those are only anecdotes. PyTamaro still lacks a rigorous empirical evaluation to show that it is engaging for learners and useful for teaching abstraction, decomposition, and more. Even less have we shown that (de)composition skills learned through PyTamaro transfer to non-graphical domains, which is of course the ultimate goal.

6 Conclusion and Future Work

In this article, we have explored the design space of educational graphics libraries in search of a design that is ideal for teaching core programming skills to complete beginners, with an emphasis on problem decomposition and abstraction. Our contribution is twofold.

First, we have overviewed existing libraries, grouping them into three families, and showed how some otherwise perfectly legitimate design decisions might actually hinder beginners from reaching certain important learning goals.

Second, we have presented the design rationale of a new graphics library, PyTamaro, which avoids the pitfalls identified in existing libraries with a less-is-more design: fewer, carefully chosen features may mean more learning. Our discussion of PyTamaro illustrates how the library may be fruitfully used in teaching to engage students with problem decomposition and other powerful ideas in computing.

PyTamaro and its Java counterpart have already been used in several courses whose target audiences range from high-school students to future high-school teachers to computer science undergraduates. Anecdotally, the library has been a success, but its effectiveness has not been evaluated with rigor. The present article informs such future research: empirical studies should explore the pitfalls that we have identified and evaluate the claimed strengths of our design.

Acknowledgments

This work was partially funded by the Swiss National Science Foundation project 200021_184689.

[3] PyTamaro has been originally implemented in Python to serve the needs of our specific contexts. Despite its complexity (see [26, 37, 41]), Python is currently a popular introductory language.

References

[1] Harold Abelson and Andrea diSessa. 1981. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics.* MIT Press, Cambridge, MA, USA.

[2] Carl Alphonce and Phil Ventura. 2003. 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.* ACM, Anaheim CA USA, 156–161. https://doi.org/10.1145/949344.949391

[3] Aivar Annamaa. 2015. Introducing Thonny, a Python IDE for Learning Programming. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research - Koli Calling '15.* ACM Press, Koli, Finland, 117–121. https://doi.org/10.1145/2828959.2828969

[4] Ian Barland, Robert Bruce Findler, and Matthew Flatt. 2010. The Design of a Functional Image Library. In *Workshop on Scheme and Functional Programming (SFP).*

[5] Mordechai Ben-Ari. 2001. Constructivism in Computer Science Education. *Journal of Computers in Mathematics and Science Teaching* 20, 1 (2001), 45–73.

[6] Karen Brennan and Mitchel Resnick. 2012. New Frameworks for Studying and Assessing the Development of Computational Thinking. In *Proceedings of the 2012 Annual Meeting of the American Educational Research Association, Vancouver, Canada*, Vol. 1. 25.

[7] Michael E. Caspersen and Jens Bennedsen. 2007. Instructional Design of a Programming Course: A Learning Theoretic Approach. In *Proceedings of the Third International Workshop on Computing Education Research.* ACM, Atlanta Georgia USA, 111–122. https://doi.org/10.1145/1288580.1288595

[8] Luca Chiodini, Igor Moreno Santos, Andrea Gallidabino, Anya Tafliovich, André L. Santos, and Matthias Hauswirth. 2021. 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).* Association for Computing Machinery, New York, NY, USA, 380–386. https://doi.org/10.1145/3430665.3456343

[9] Keith Devlin. 2003. Why Universities Require Computer Science Students to Take Math. *Commun. ACM* 46, 9 (Sept. 2003), 36. https://doi.org/10.1145/903893.903917

[10] Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Science of Computer Programming* 17, 1 (Dec. 1991), 35–75. https://doi.org/10.1016/0167-6423(91)90036-W

[11] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs, Second Edition: An Introduction to Programming and Computing.* MIT Press.

[12] Matthias Felleisen and Shriram Krishnamurthi. 2009. Why Computer Science Doesn't Matter. *Commun. ACM* 52, 7 (July 2009), 37–40. https://doi.org/10.1145/1538788.1538803

[13] 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. 2020. Notional Machines in Computing Education: The Education of Attention. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR '20).* Association for Computing Machinery, New York, NY, USA, 21–50. https://doi.org/10.1145/3437800.3439202

[14] Sigbjorn Finne and Simon Peyton Jones. 1995. Pictures: A Simple Structured Graphics Model. In *Proceedings of the 1995 Glasgow Workshop on Functional Programming.* https://doi.org/10.14236/ewic/FP1995.6

[15] Kathi Fisler, Shriram Krishnamurthi, and Preston Tunnell Wilson. 2017. Assessing and Teaching Scope, Mutation, and Aliasing in Upper-Level Undergraduates. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education.* ACM, Seattle Washington USA, 213–218. https://doi.org/10.1145/3017680.3017777

[16] Michael H. Goldwasser and David Letscher. 2009. A Graphics Package for the First Day and Beyond. *ACM SIGCSE Bulletin* 41, 1 (March 2009), 206–210. https://doi.org/10.1145/1539024.1508945

[17] Shuchi Grover and Satabdi Basu. 2017. 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 (SIGCSE '17).* Association for Computing Machinery, New York, NY, USA, 267–272. https://doi.org/10.1145/3017680.3017723

[18] Philip J. Guo. 2018. 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.* ACM Press, Montreal QC, Canada, 1–14. https://doi.org/10.1145/3173574.3173970

[19] Mark Guzdial. 2003. A Media Computation Course for Non-Majors. In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education.* ACM, Thessaloniki Greece, 104–108. https://doi.org/10.1145/961511.961542

[20] Pontus Haglund, Filip Strömbäck, and Linda Mannila. 2021. Understanding Students' Failure to Use Functions as a Tool for Abstraction – An Analysis of Questionnaire Responses and Lab Assignments in a CS1 Python Course. *Informatics in Education* 20, 4 (Dec. 2021), 583–614. https://doi.org/10.15388/infedu.2021.26

[21] Brian Harvey. 1997. *Computer Science Logo Style: Beyond Programming* (second ed.). Exploring with LOGO, Vol. 3. MIT Press, Cambridge, MA, USA.

[22] Peter Henderson. 1982. Functional Geometry. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming (LFP '82).* Association for Computing Machinery, New York, NY, USA, 179–187. https://doi.org/10.1145/800068.802148

[23] Simon Holland, Robert Griffiths, and Mark Woodman. 1997. Avoiding Object Misconceptions. In *Proceedings of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '97).* ACM, New York, NY, USA, 131–134. https://doi.org/10.1145/268084.268132

[24] Kristina Holsapple and Austin Cory Bart. 2022. 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).* Association for Computing Machinery, New York, NY, USA, 85–91. https://doi.org/10.1145/3478431.3499363

[25] Cruz Izu and Peter Dinh. 2018. Can Novice Programmers Write C Functions?. In *2018 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE).* IEEE, Wollongong, NSW, 965–970. https://doi.org/10.1109/TALE.2018.8615375

[26] Fionnuala Johnson, Stephen McQuistin, and John O'Donnell. 2020. Analysis of Student Misconceptions Using Python as an Introductory Programming Language. In *Proceedings of the 4th Conference on Computing Education Practice 2020.* ACM, Durham United Kingdom, 1–4. https://doi.org/10.1145/3372356.3372360

[27] Cazembe Kennedy and Eileen T. Kraemer. 2019. Qualitative Observations of Student Reasoning: Coding in the Wild. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education.* ACM, Aberdeen Scotland Uk, 224–230. https://doi.org/10.1145/3304221.3319751

[28] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code Quality Issues in Student Programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17).* Association for Computing Machinery, New York, NY, USA, 110–115. https://doi.org/10.1145/3059009.3059061

[29] Jonathan Knudsen. 1999. *Java 2D Graphics.* O'Reilly.

[30] Tobias Kohn. 2017. *Teaching Python Programming to Novices: Addressing Misconceptions and Creating a Development Environment.* Ph. D. Dissertation. ETH Zurich. https://doi.org/10.3929/ETHZ-A-010871088

[31] Herman Koppelman and Betsy van Dijk. 2010. Teaching Abstraction in Introductory Courses. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education - ITiCSE '10*. ACM Press, Bilkent, Ankara, Turkey, 174. https://doi.org/10.1145/1822090.1822140

[32] Teemu Lehtinen, Aleksi Lukkarinen, and Lassi Haaranen. 2021. 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*. 206–212. https://doi.org/10.1145/3430665.3456322 arXiv:2104.06710 [cs]

[33] Aleksi Lukkarinen and Juha Sorva. 2016. 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*. ACM, Koli Finland, 51–60. https://doi.org/10.1145/2999541.2999551

[34] Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. 2007. Investigating the Viability of Mental Models Held by Novice Programmers. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*. ACM, Covington Kentucky USA, 499–503. https://doi.org/10.1145/1227310.1227481

[35] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2011. Habits of Programming in Scratch. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education (ITiCSE '11)*. Association for Computing Machinery, New York, NY, USA, 168–172. https://doi.org/10.1145/1999747.1999796

[36] Bartosz Milewski. 2018. *Category Theory for Programmers*. Blurb.

[37] Craig S. Miller and Amber Settle. 2016. 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*. ACM, Melbourne VIC Australia, 133–141. https://doi.org/10.1145/2960310.2960327

[38] Seymour A. Papert. 1971. *A Computer Laboratory for Elementary Schools*. Technical Report AIM-246 / LOGO Memo 1. MIT.

[39] Seymour A. Papert and Cynthia Solomon. 1971. *Twenty Things To Do With A Computer*. Technical Report AIM-248 / LOGO Memo 3. MIT.

[40] D. L. Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. https://doi.org/10.1145/361598.361623

[41] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: The Full Monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 217–232. https://doi.org/10.1145/2509136.2509536

[42] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Transactions on Computing Education* 18, 1 (Oct. 2017), 1–24. https://doi.org/10.1145/3077618

[43] Eric Roberts and Keith Schwarz. 2013. A Portable Graphics Library for Introductory CS. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE '13*. ACM Press, Canterbury, England, UK, 153. https://doi.org/10.1145/2462476.2465590

[44] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education* 13, 2 (June 2003), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

[45] Emmanuel Schanzer, Shriram Krishnamurthi, and Kathi Fisler. 2018. Creativity, Customization, and Ownership: Game Design in Bootstrap: Algebra. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, Baltimore Maryland USA, 161–166. https://doi.org/10.1145/3159450.3159471

[46] Juha Sorva. 2023. Misconceptions and the Beginner Programmer. In *Computer Science Education : Perspectives on Teaching and Learning in School* (second ed.), Sue Sentance, Erik Barendsen, Nicol R. Howard, and Carsten Schulte (Eds.). Bloomsbury Academic, London, 259–273.

[47] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. 1974. Structured Design. *IBM Systems Journal* 13, 2 (1974), 115–139.

[48] Vicki Trowler. 2010. *Student Engagement Literature Review*. The Higher Education Academy, York.

[49] Preston Tunnell Wilson, Kathi Fisler, and Shriram Krishnamurthi. 2018. Evaluating the Tracing of Recursion in the Substitution Notional Machine. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, USA, 1023–1028. https://doi.org/10.1145/3159450.3159479

[50] David Weintrop, Alexandria K. Hansen, Danielle B. Harlow, and Diana Franklin. 2018. 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*. ACM, Espoo Finland, 142–150. https://doi.org/10.1145/3230977.3230988

[51] Jeannette M. Wing. 2006. Computational Thinking. *Commun. ACM* 49, 3 (March 2006), 33–35. https://doi.org/10.1145/1118178.1118215

[52] Brent A. Yorgey. 2012. Monoids: Theme and Variations (Functional Pearl). *ACM SIGPLAN Notices* 47, 12 (Sept. 2012), 105–116. https://doi.org/10.1145/2430532.2364520

Received 2023-07-27; accepted 2023-08-24