KI - Künstliche Intelligenz (2022) 36:17–33
https://doi.org/10.1007/s13218-021-00748-0

TECHNICAL CONTRIBUTION

Simplifying Programming for Non-technical Students: A Hermeneutic Approach

Andrea Valente[1] · **Emanuela Marchetti**[2]

Received: 28 May 2021 / Accepted: 7 December 2021 / Published online: 17 January 2022

Abstract

This paper investigates the simplification of programming for non-technical university students. Typical simplification strategies are outlined, and according to our findings CT courses for non-technical students typically address learners from different faculties, providing generic and basic knowledge, not specifically related to their major. In this study, we propose instead a hermeneutic approach to simplify programming, in which we aim at clarifying the problem-solving aspect of programming, addressing computational problems that are specific to their studies and leveraging on learners' preunderstanding of the digital media they have experienced as users. The practical counterpart of our theoretical approach is a minimalistic Python multimedia library, called Medialib, that we designed to enable university students with a non-technical profile to create visual media and games with short and readable code. We discuss the use of Medialib in two empirical case studies: a collaboration with the university of Kyushu in Fukuoka, Japan, and a coding module for Media Studies students at the University of Southern Denmark. Furthermore, we use Notional Machines to attempt a comparison of the simplicity of learning tools for programming, and to ground our claim that Medialib is "simpler" for learners than other popular approaches. The main contribution is a hermeneutic approach to the simplification of programming for specific contexts that combines the hermeneutic spiral and notional machines. The approach is supported by a tool, the Medialib library; the two case studies provide examples of how the approach and tool can be deployed in beginners in CT courses.

Keywords Computational Thinking · Learn programming · Simplification · Hermeneutic · Notional Machines

1 Introduction

Programming is hard to learn, hence in the field of Computational Thinking (CT for short) various pedagogical approaches and tools have been proposed to simplify access to programming for pupils at different educational levels. This paper investigates what it means to simplify programming for non-technical university students, discussing typical strategies embodied in different programming tools, both from literature and from reflections on two empirical case studies: an online study, in collaboration with the university of Kyushu in Fukuoka (Japan), and a 4-weeks coding module addressing Media Studies students, in Odense, at the University of Southern Denmark. The two studies involved the deployment and testing of Medialib, a minimalistic Python multimedia library designed to enable university students with a non-technical profile to create visual media and games with simple and readable code, using given or self-created assets.

In order to discuss the complexity that learners have to face when learning programming via different tools and approaches, we will employ Notional Machines [2, 5], NOMs for short: NOMs will allow us to work concretely with the otherwise vague idea of simplification. The design principles for the Medialib derive from our comparison of various NOMs, that help express the complexity of other, very popular approaches, such as learning programming via Python and Pygame or Pygame Zero.

This study is grounded on a theoretical framework combining the concepts of hermeneutic spiral and of NOMs, which we see as a concretization of the hermeneutic spiral, translated into the domain of learning to program in a

✉ Andrea Valente
anva@mmmi.sdu.dk

1 Maersk Mc-Kinney Moller Institute, Game Development and Learning Technology, University of Southern Denmark (SDU), Odense, Denmark

2 Media, Department for the Study of Culture, University of Southern Denmark (SDU), Odense, Denmark

specific language. From a hermeneutic perspective we see the Medialib as a mediating language between the learners and actual programming languages that are used in programming practice, specifically Python. In this sense, the Medialib is seen as a concretization of our approach, embodying both hermeneutic pedagogy and NOMs, with the goal of facilitating our learners to access the hermeneutic spiral in their learning of programming in Python.

The following sections present related work and our theoretical framework (Sects. 2 and 3); Sect. 4 introduces the concept of NOM and shows how it can be used to simplify CT, and programming in particular, by providing a guide to the design of simpler libraries for beginners, such as our Medialib. Section 5 discusses two use cases where the Medialib was used with different groups of learners; reflections on these two experiences are framed with respect to our hermeneutic approach. Section 6 concludes the paper.

2 Related Work

2.1 CT and Programming

Teaching and learning programming has received increasing attention, especially within the field of Computational Thinking, which is defined as abilities typically associated with programmers such as "solving problems, designing systems, and understanding human behavior" ([37] p. 33). In fact, CT can be defined as an interdisciplinary set of skills and knowledge from fields such as engineering and computer science, design, business, and social sciences among others [32]. The goal of CT studies is to find effective approaches to provide young people with knowledge and skills that can enable them to access the global job market and act as citizens, aware of their rights and duties, in increasingly digitized societies [14]. Although it has been stated that programming does not equal CT [32, 37], programming is still perceived as a central skill in CT, and a main concern for CT studies. Programming is seen as a complex practice, aimed at making software and it is segmented into: analysis and design, which is the process of analysis of the problem to solve through the making of software, coding, which is intended as the actual process of writing the code with the selected programming language, and testing of software according to usability principles. Programming and specifically coding involves also practices like debugging and refactoring, which deal with identifying and fixing various issues in a program, and techniques to safely restructuring existing code. CT studies have explored pedagogical approaches and tools aimed at simplifying programming, to make it more accessible and officially introduce it in schools.

2.2 How is Programming Simplified

In this study we adopted Python as a beginner-friendly language, given its popularity in the learning community as a good, entry-level and scalable programming language. In a previous paper [34], together with a colleague from Fukuoka, we performed a review of typical programming materials used in beginners' programming courses. These materials, which include books, lecture slides and typical exercises, were selected based on our collective teaching experience with beginners programming courses, the fact that most introductory materials we have used or know from university-level courses are based on the Python language, and that these books appear to be used in both Denmark and Japan. We analyzed and compared the structure of beginners' books (such as [27, 31, 36], with [19] being an outlier), of online courses [18, 26, 28], and the way popular libraries are presented (like Pygame Zero for instance [22]). Most of these materials are organized in a traditional, bottom-up fashion:

– Variables and primitive types;
– Control flow (conditionals and loops);
– Data-structures, such as lists, arrays, and perhaps objects;
– Files, followed eventually by more advanced topics

We call this "typical course structure", or TCS. We found that it is typical for these topics to be grounded in somewhat generic, simplified mathematical or logical problems. Textbooks and online courses often do not offer a coherent narrative in which the learners are confronted with problems meaningful to them. Furthermore, concepts are typically introduced in a specific order mainly because of their importance in understanding concepts that will follow, a sort of *internal logic*. Moreover, in our analysis beginners textbooks tend to focus too much on formal definitions and terminology, instead of helping the learners to build a solid, practical understanding of coding as a craft. As examples of these problems, comparing [27] with [19] we noticed that [27] introduces functions before loops, and lists much after. [19] instead follows a spiral approach: a quick intro to variables and basic data types, then sequences (i.e. strings and lists) and dictionaries; after that conditionals and loops are also shortly introduced (e.g. using loops with lists), followed by a simple definition of functions. Then [19] introduces the Pygame library, followed by some more iteration on topics like conditional, loops and data structures, but this time with games in focus. In our experience, the approach adopted in [19] works better for beginners in technical faculties, and we consider it promising also for students from non-technical areas.

From our analysis we also found out that most materials adopt one or a few of a short list of strategies to simplify programming:

– Removing theoretical explanations requiring further knowledge;
– Providing block-coding tools, to enable younger programmers to compose their code from provided command-blocks, without having to memorize instructions' syntax;
– Providing practical exercises to apply knowledge acquired through the topics;
– Promote creative engagement with coding through development of simple games, or some form of interactive programs

Practical exercises are often framed to enable the creation of simple games in specifically designed systems or programming environments, like Scratch, Python with Pygame or Pygame Zero, P5 (a reimplementation of Processing in JavaScript). Development of simplified games (or parts of games) is seen mainly as a motivational resource, leveraging the learners' personal interests. But while games and multimedia seem to be regarded as motivating and rewarding elements for learners, most materials still follow closely the TCS, and even books targeted at primary school learners tend to introduce graphics and multimedia in the second half of the text (as is the case in books for children of the series created by Carol Vorderman, e.g. [35], which have been translated in multiple languages). On the other hand, technically-framed exercises usually involve the solution of elementary numeric problems, through manipulation of numbers, strings, or simple data structures like lists or arrays. In this sense, simplification of programming appears to be approached from a quantitative, *reductionist* perspective, cutting down and reducing the complexity of computational problems presented to learners. Moreover, in our experience, CT courses for non-technical students typically address learners from different faculties, providing generic knowledge, non-specifically related to their major.

2.3 Notional Machines and Simplicity

In this paper we are interested in a different meaning of *complexity* and *simplification*. Following current research [5, 6, 29] and [2], we adopt the idea of Notional Machines, or NOMs, and use them to reason about the complexity (or more precisely about the simplicity) of programming and programming learning. NOMs were first introduced by DuBoulay in the 1980s [5], and are based on two main ideas: (1) that learners need a model to reason about computation, but also (2) that the model does not need to be complete or highly complex form the very beginning, and instead it would make more pedagogical sense to proceed with multiple models, in a spiral or incremental fashion. Interestingly, DuBoulay wrote that *"A Notional Machine is the best lie we tell students [about how the machine works]"*, and in [5, 6] NOMs are presented as:

> [...] artifacts intentionally designed to serve the pedagogical purpose of representing and explaining the behavior of a computational system. A notional machine uses terminology and abstraction levels aimed at a particular audience to support their practices in a particular context. It is often a simplification and can be communicated in different formats.

An example of a NOM for Java is presented in [2], and its advantages for learners discussed. Since we are interested in simplifying programming for beginners, we could ask the question: *when is a pedagogical approach to programming simpler than another?*, or more practically: ***simple with respect to what?***. The strategy presented in [6] allows to compare the *Cognitive Complexity of Computer Programs*, CCCP for short; in this remarkable work, the authors considered "a number of short programs as case studies", then they applied their Cognitive Complexity method to "illustrate why one program or construct is more complex than another, to identify dependencies between constructs that a novice programmer needs to learn and to contrast the complexity of different strategies for program composition". We cannot use their method directly, because what we want to compare are not programs, but approaches to teaching programming to beginners. Also, although we are inspired by [6], we prefer to consider NOMs as ways to mentally and manually run code, to make sense of programs, and not in relation to algorithm animation as the authors of that paper do. Our approach (detailed in Sect. 4) proposes instead to take a few examples of programming tasks, solve them in the simplest, shortest and most readable way using 2 different programming environments (for example Python with Pygame, and Python with our Medialib library). Then using a similar method as CCCP, analyze and list all concepts (and their dependencies) needed to create a minimal NOM that complete beginners could use to manually execute the code. And once we have these two minimal NOMs, we can compare them, and conclude which one is **simpler**, i.e. which NOM is defined with fewer and least complex concepts. The idea that NOMs can be used to compare more than programs, but also entire programming languages is supported by papers like [25], where the authors discuss their comparison of Python and Scratch via the complexity of their NOMs, and conclude that Scratch has surprisingly much hidden complexity, and a rather different expressive power than more common programming languages such as Python.

3 Hermeneutic Grounding

The design of our Medialib is grounded on a hermeneutic approach to learning, which focuses on describing the process of understanding from the perspective of the learners. The term hermeneutics derives from the Greek "hermeneutikos", which means *meaning to interpret*. Hermeneutics is mainly concerned with the understanding and interpretation of texts [1, 10], intended as a process of decoding and sense making of a given text from the subjective perspective of the learners. In our project, we aim at enabling non-technical students to approach programming as a form of problem-solving, leveraging the construction of technological artefacts through scientific inquiry, as well as design, algorithmic thinking, and coding. In our view, coding deals with the practice of writing code in a programming language and through a specialized editor (as discussed in [17]). In this respect, we consider code as a text, written in an artificial language to solve specific problems through a computer, that the learners need to understand and master, to gain core competences in CT, to make sense of how digital technologies work in general, and be able to modify and write new code from scratch in relation to different software applications. According to Gadamer and Tomkins [9, 33], understanding emerges through a critical dialogue between the reader and the text, highlighting that the learners will understand the text interpreting it from their individual perspective, in relation to their sociocultural backgrounds and what the text actually means to them (as also in [30]). Taking inspiration from Gadamer [9], our inquiry focuses on the simplification of programming practice, to facilitate the learners in actively engaging with code, starting from an interpretive process with given code and then moving towards editing, and finally creating new code from scratch: the use-modify-create progression [17]. Being text interpretation strictly intertwined with "the cultural and discursive setting, in which—and from which—it emerges" ([33], p. 4), we have created an innovative multimedia library for Python, the Medialib, that simplifies operations such as visualizing images and creating interactive elements. Medialib supports learners in their interpretation of code, by providing powerful and atomic commands that are cognitively simple to execute; and this enables learners to focus on problem-solving and semantics. This is a strategy in contrast with the idea of simplifying the syntax of a language through block coding, as it is commonly done in current programming environments for beginners, like Scratch or Blockly. Our library achieves this by hiding part of the complexity of media programming, and by offering a coding style that reduces the number of concepts needed by beginners to understand simple programs; as the students progress, we can gradually expand on what has been hidden, while introducing incrementally more complex problems to solve.

Taking these considerations into account, the pedagogical model behind the development of the Medialib represents an application of the hermeneutic circle or spiral, concretized through the concept of NOM (as discussed in Sect. 2, and in [2, 5, 6]). According to Gadamer [9] and Heidegger [12], text understanding and interpretation take place through a circle, which represents a metaphor of the process through which learners engage in meaning making [30]. There is no unique or exhaustive definition of the hermeneutic circle, however, it is a simple but powerful symbol, representing how learning and understanding are not linear but iterative processes. The hermeneutic circle captures in fact the initial difficulty of the learners to access new knowledge through a tentative and conflictual dynamic, as they move from their initial knowledge to embrace new, technical knowledge from a specific field [33]. Following the hermeneutic circle, understanding is a relational and referential process, through which we make connections by means of comparison, contrast or juxtaposition between the whole and the parts of the bits of knowledge we are trying to make sense of, and also through a comparison with our previous knowledge. According to Schleiermacher ([24], pp. 24, [33]), a text can be understood by connecting the whole to the parts that compose it, decoding how they depend on each other to form the text. In our understanding, this principle can be applied to the understanding of code as a text, as learners of programming have to make sense of code, establishing connections between the different syntactic units such as: technical terms, instructions and constructs, and data structures. A classical example of meaning that is distributed across code is the declaration of a variable and its uses. To know what the value of a variable is at a given point in the execution of a program, a learner might have to correlate many lines of code, possibly spread across the program. Moreover, to move further and edit or write new code, learners must have understood what is the meaning of the different words in the code in relation to the whole. However, differently from a literary text, code has a more practical aspect as it has to return a result, or express a certain behavior, when run on a computer. In this sense, programming learners have the possibility to alter and run code to test its correctness, and of their expectations circa its behavior. Hence, the practices of running and debugging code, acquire cognitive meaning from a hermeneutic perspective, providing learners an objective test bench for their understanding, through the computer, as a form of dialogue.

As in the process of understanding a literary text in natural language, the process of understanding code in an artificial language necessitates to leverage the learners' preunderstanding, which is defined as a prejudice or preconception about any new knowledge we are going to acquire. According to Heidegger and Gadamer, a preunderstanding is a necessary precondition for understanding to take place (see also [33]). Heidegger [12] and Gadamer [9] argue that any textual understanding is the result of a personal interpretation, dependent on the previous experiences and sociocultural background of the readers. Heidegger argues that:

> *an interpretation is never a presuppositionless apprehending of something presented to us. If, when one is engaged in a particular concrete kind of interpretation, in the sense of exact textual interpretation, one likes to appeal to what "stands there" , then one finds that what "stands there" in the first instance is nothing other than the obvious un-discussed assumption of the person who does the interpreting*

(from [12], pp. 191–192). Similarly, Gadamer describes the hermeneutic circle as a process centered on"the anticipation of meaning in which the whole is envisaged becomes actual understanding when the parts that are determined by the whole themselves also determine this whole" ([9], p. 291). Building on these insights, understanding becomes a dialogue between the text and the learners, in which the learners gain meaning from the text reassembling the whole and parts, in relation to their previous knowledge. Moreover, so defined the process of understanding has a temporal aspect, which frames the passage from the initial preunderstanding to the achievement of actual understanding. This temporal aspect of the understanding process is emphasized by Gadamer, who argues that as the learners engage in the circle they gain new knowledge [9]. The hermeneutic circle can therefore be reformulated into a spiral, which illustrates how learners do not simply move in a circle between whole and parts while engaging with the understanding process, but instead they acquire new knowledge and shift from general preunderstanding to more specialized, deeper knowledge [13, 21]. During this process, learners might have to negotiate between their preunderstanding and the new knowledge they are encountering. This might happen through a harmonious integration or through forms of negotiation and conflicts. The idea of growing knowledge spirally, from simpler but grounded approximations, to more complex and correct models is also found in NOMs [5, 29], possibly because of their pedagogical nature. By moving through the circle or spiral, the learners are expected to reach a Fusion of Horizons between their preunderstanding and the new targeted knowledge [9, 12]. The notion of Fusion of Horizons is a model on the essence of understanding and learning, rooted in an interpretation process, through which learners gain ownership on the text they are facing, making it their own from the perspective of their previous knowledge and values.

Moving towards programming and the specific challenges that emerge with learning problem-solving through coding, we see preunderstanding consisting of resources that enable learners to approach the new subject from their individual intuition. In this way we distinguish:

– Cultural preunderstanding, in which students can build on their own intuition regarding their experience with digital media in their free time;
– Sensorial preunderstanding, in which simple manipulation of visuals and sounds are seen as enabling the students to understand their code, yet avoiding complicated math.

Cultural preunderstanding corresponds with the traditional notion of preunderstanding in hermeneutics, constituted by previous knowledge and experiences of the learners. On the other hand, when tackling CT problem-solving through algorithmic thinking, it might be hard for the learners to even evaluate their solution to a problem, when the problem itself might be complex and possibly require knowledge from mathematics. Therefore, when we teach programming with Medialib, we formulate "visual" exercises in which the correct outcome is visually different from an incorrect one; in this way our students can easily be sure whether their solutions are correct, just by looking at the results of their solutions. Typical tasks could be moving an image on the screen or drawing a bar chart from a known data set: in this way we combine their cultural and sensorial understanding, and ground the activities of testing and debugging in the preunderstanding of the learners. We designed our courses in this way also to gain the extra advantage of avoiding classical math-based problems and focus instead on problem-solving.

4 Python NOMs and Medialib's Design

Following the approach of [5] we have defined a NOM for Python, but instead of using only natural language rules, as done in [5], we present a more visual representation of our NOM's rules.

In Fig. 1 we define a minimal NOM for a core imperative fragment of Python, called NOM_1. To show how NOM_1 works, we can consider a simple Python program, called **program 1**, that could be part of an introductory course (see Fig. 2).

Using the rules of NOM_1 (as in Fig. 1) it is possible to read program 1 and correctly execute it. The "sequence" rule tells us that the three lines of code in the program have to be executed one after the other; the first line is an input instruction, and according to NOM_1 its execution creates a box in the memory, labeled "name" , and the program will wait for the user to type his or her name, and the resulting string will be inserted in the "name" box as its value. The second line is an assignment of a string expression, and according

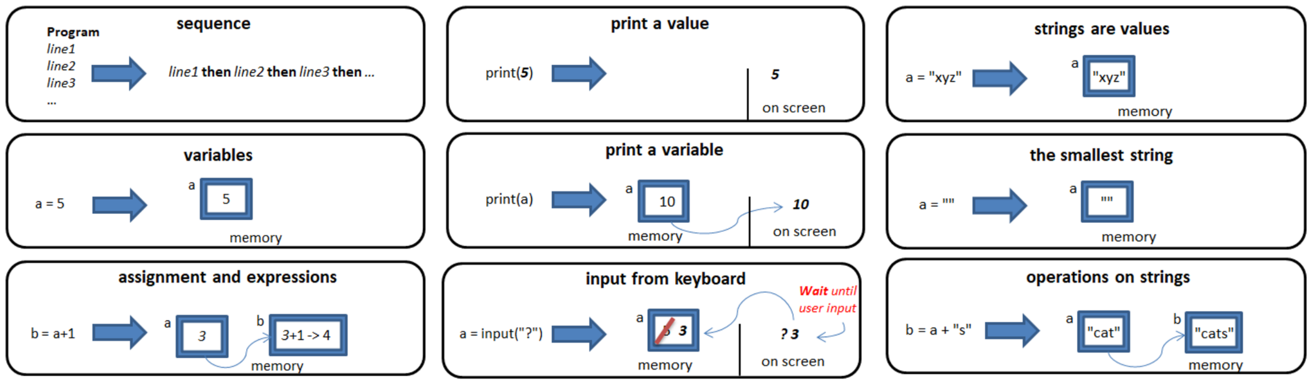
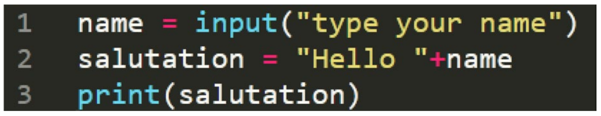


Fig. 1 The rules of NOM_1



```
name = input("type your name")
salutation = "Hello "+name
print(salutation)
```

Fig. 2 Program 1, possibly one of the first examples shown to beginners

to the "operations on strings" rule in NOM_1 its execution will create a box labeled "salutation" in the memory of the machine, and set its value to the concatenation of the string "Hello" and whatever name the user typed and it is currently stored in the box labeled "name" . Finally, the third line is a print instruction, and printing a variable has the effect of looking up the value in box "salutation" which is a string in this case, and output that value to the screen.

NOM_1 is good enough to describe the behavior of simple programs like this, and with the addition of a few more mathematical and string operators (and possibly conditionals and loops) it could be used by beginners to understand a substantial portion of imperative Python programs. We are interested in particular in flat programs, i.e. programs without function definitions, because functions require a more complex execution model, including for instance the execution stack and the scope of variables, and therefore a more complex NOM. Finally, NOM_1 has a blocking input instruction. In our experience this is a straightforward enough concept beginners to learn; moreover, from our analysis of programming textbooks in Sect. 2.2 and in [34], we know that sequential execution and blocking on inputs appear to be the normal way to introduce simple, input-compute-output programs to beginners.

4.1 Extending Python with Multimedia and Comparing NOMs

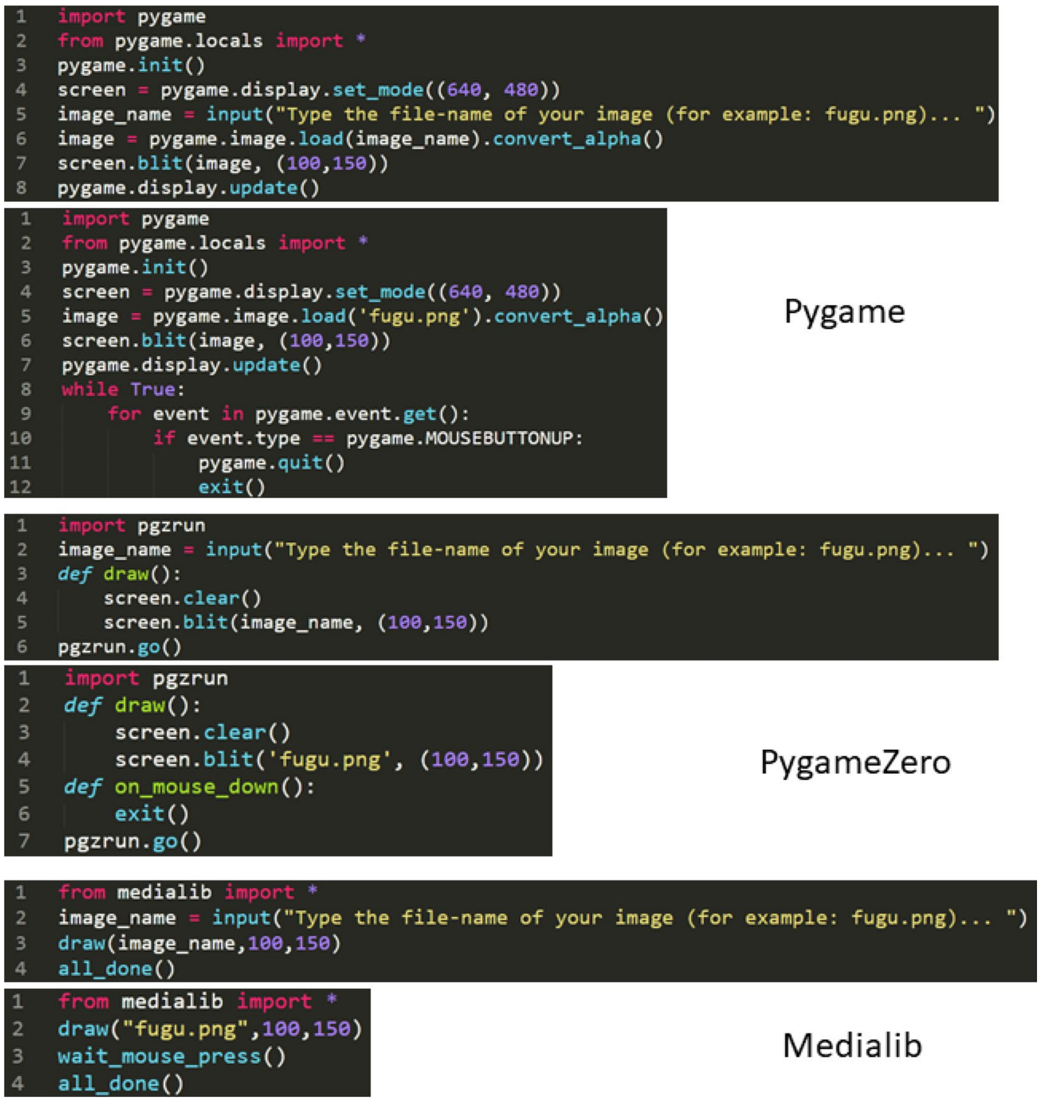
How can we define a NOM for multimedia programs in Python that is at a similar level of complexity as NOM_1? We want to follow [6] in our approach to compare complexity in relation to NOMs, therefore we propose to investigate this question by first defining two simple tasks involving images. An implementation of each task will then be shown, using the Pygame and the Pygame Zero Python libraries. The code for these implementations will be as short and readable as possible. Two NOMs, NOM_{pg} and NOM_{pgz} will then be defined based on the programming styles and assumptions implicit in the code for each implementation. We will then present solutions for the two tasks written using our Medialib library [34], and a NOM will also be constructed for the "Python with Medialib" language, and we will call it NOM_{ml}. We will argue that comparing NOM_{pg} and NOM_{pgz} with NOM_{ml}, the latter results the simplest, hence programs written with Python and the Medialib are cognitively simpler to understand than using the other two widely adopted libraries. The two multimedia tasks are defined as follows:

1. A program that asks the user the name of a picture, and shows it on the screen
2. A program that displays a picture, then waits for the user to click a mouse button, and terminates

The Pygame solutions to the two tasks could look like the first two solutions in Fig. 3. Those two programs are simplified versions of the examples in [22] chapter 3 and chapter 6, and are arguably the shortest and clearest possible. The Pygame Zero solutions (PGZ for short) are the next two listed in Fig. 3. And the last two listings in Fig. 3 are coded using our Medialib. Notice that programs written using Medialib should always terminate with a call to the *all_done()* function, which has the effect of closing the graphic window.

4.2 Three NOMs: NOM_{pg}, NOM_{pgz} and NOM_{ml}

An analysis of the Pygame solutions in Fig. 3 shows that a beginner programmer would need to understand quite a number of complex concepts. In particular for solution 1:



```
1  import pygame
2  from pygame.locals import *
3  pygame.init()
4  screen = pygame.display.set_mode((640, 480))
5  image_name = input("Type the file-name of your image (for example: fugu.png)... ")
6  image = pygame.image.load(image_name).convert_alpha()
7  screen.blit(image, (100,150))
8  pygame.display.update()
```

```
1  import pygame
2  from pygame.locals import *
3  pygame.init()
4  screen = pygame.display.set_mode((640, 480))
5  image = pygame.image.load('fugu.png').convert_alpha()
6  screen.blit(image, (100,150))
7  pygame.display.update()
8  while True:
9      for event in pygame.event.get():
10         if event.type == pygame.MOUSEBUTTONUP:
11             pygame.quit()
12             exit()
```

```
1  import pgzrun
2  image_name = input("Type the file-name of your image (for example: fugu.png)... ")
3  def draw():
4      screen.clear()
5      screen.blit(image_name, (100,150))
6  pgzrun.go()
```

```
1  import pgzrun
2  def draw():
3      screen.clear()
4      screen.blit('fugu.png', (100,150))
5  def on_mouse_down():
6      exit()
7  pgzrun.go()
```

```
1  from medialib import *
2  image_name = input("Type the file-name of your image (for example: fugu.png)... ")
3  draw(image_name,100,150)
4  all_done()
```

```
1  from medialib import *
2  draw("fugu.png",100,150)
3  wait_mouse_press()
4  all_done()
```

Fig. 3 All Python listings. From the top: solutions to tasks 1 and 2 using Pygame, then using PygameZero, and Medialib

- The concept of importing Python modules, that there are at least 2 ways to import, which in turn require some ideas about namespaces and scope of variables across modules;
- Dot notation and *dot paths* (i.e. multiple dot notations used in the same expression, as visible in line 4) require to know how functions can be accessed across modules, but also that in some cases the dot notation has to do with objects and not modules;
- Concepts like tuples, objects and classes are unavoidable even with such simple Pygame examples;
- Drawing images on screen requires a minimal understanding of coordinates, but here it also requires knowing about technical concepts like Surface objects and buffering, and that drawing images in Pygame is not really drawing pixels on screen, and that images will be visible on screen only after having updated the screen buffer (another Surface object);
- Even rather low-level concepts like image transparency and alpha channel figure explicitly in the code, and would require some explanation and possibly be included in the Pygame's NOM.

The Pygame solution of task 2 in Fig. 3 is similar to solution 1, but it involves more concepts such as: events and polling, lists, as well as loops, nested loops, and *endless looping with breaking*. This program waits by repeatedly polling events from the event queue, and when a mouse click is detected by the endless while loop in line 8, the program terminates, effectively breaking out of the loop. The use of endless loops with breaking is typical of certain coding styles, inspired by similar practices used in C and derived languages, and in this case we kept it because it was in many of the examples in book [19]; however, we find it to be a coding style that supports bad habits and invites non-algoritmic ways of thinking. The lines from 8 to 12 require the NOM to address events and how they work; learners will also need to know about queues, polling and busy-waiting. Since multiple events might occur at the same time, e.g. typing a key on the keyboard and simultaneously clicking the mouse button, Pygame requires a nested for loop, in line 9, to visit all the elements of the event queue. Line 10 detects the mouse button being released, and triggers lines 11 and 12, which close the Pygame window and terminate the program. The dot notation used in *event.type* reveals that the variable *event* is in fact structured and has attributes (i.e. it is an object). Finally, **MOUSEBUTTONUP** is a *constant* that exists in the module pygame, so NOM_{pg} has to provide a rather comprehensive explanation of how external modules work. The NOM for Python with Pygame would then have to include all or most of the concepts listed above, and be quite complex. This is perhaps not surprising when considering that Pygame is meant as a library to efficiently code 2D games, and not simplified with novice programmers in mind.

On the other hand, the PGZ library was created specifically to support learning, however, the solutions with PGZ still require the programmer to understand quite a number of complex concepts such as:

- Function definitions, functional programming concepts like callbacks, and consequently scope rules;
- Data-types like tuples and quite a few built-in objects;
- Concurrency, given that the PGZ code goes beyond sequential programming

Solution 1 for PGZ starts with a function definition (in line 3); however, this is a special function that must be named *draw*, and PGZ will automatically call this function repeatedly, multiple times per second. Therefore, NOM_{pgz} has to include function definitions and calls, and possibly the idea of callbacks. Talking about functions makes the code non-flat and potentially requires the introduction of local and global variables, scope and execution stack. This program uses the variable *screen* (in line 4 and 5s) that according to [22] it's an object that represents the screen, and it is one of many *build-in objects* that form the API of PGZ. Since PGZ is based on Pygame, the screen object is a Pygame Surface, and in fact PGZ inherits from Pygame the need to add objects, classes and dot paths to its NOM. Line 6 starts the *Pygame Zero runtime*, and this is the last line in most examples on the official PGZ web page [22]. Interestingly, this program cannot be understood as sequential instructions; moreover, the *draw* function is never called explicitly, and yet it will execute multiple times. NOM_{pgz} will also have to include a description of the hidden *update-draw loop* at the core of PGZ. Solution 2 is very similar to solution 1, but in lines 5 and 6 there is another function with a mandatory name, *on_mouse_down()*, which is automatically executed every time a mouse button is pressed. The dovetailing between the *draw()* and *on_mouse_down()* functions is not evident from the code, and could require a concurrent or asynchronous execution model to be added to NOM_{pgz}, since event-based programming cannot be described by a simple sequential execution model. At this stage we can see that the NOMs for Pygame and PGZ are similar in complexity. It might seem that NOM_{pgz} could be simpler with respect to its handling of user input, thanks to the change from polling to event-based. Unfortunately this change also requires adding rules and a model for asynchronous programming (or concurrency).

Medialib is also based on Pygame, but aspires at reducing the number and complexity of the concepts needed in its NOM, i.e. NOM_{ml}. Looking at the two solutions implemented with Medialib in Fig. 3 we find the following:

- The code only uses primitive data types, i.e. numbers and strings;
- Drawing images is done without explicitly using dot notation, objects or classes. The complexity of loading, storing and drawing images on screen is hidden, so to appear as atomic instructions to the learner;
- User input processing is done via a mix of explicitly blocking instructions and simplified manual polling (when non-blocking solutions are needed);
- Sequential thinking is enough to understand (and mentally execute) these programs;
- The need to discuss details of Python's importing mechanism is minimized by only importing the entire Medialib library as the first line of most code examples, and by never using dot paths.

Medialib solution 1 starts by importing Medialib in such a way that effectively pollutes the global namespace: this might be considered a bad practice for a Python library to be used by professionals, but it is perfectly in line with our didactic purpose. Medialib then automatically starts Pygame and opens a graphic window. Line 3 draws the image by name, at the position *100,150*, using only strings and numbers. To hide the complexity of loading, storing and drawing the image on screen, the Medialib keeps a table of already loaded+ images. This results in a slightly uneven performance of the drawing function, however, we designed Medialib with the idea that performance is an advanced concept in programming and belongs with algorithm complexity analysis, not in the beginners' very first NOM. Moreover, considering the small-to-medium size of the images typically used in beginners examples, the variation in performance is imperceptible. Once the image is loaded, Medialib *blits* it on screen and automatically updates the screen surface. Transparency is also automatically managed. Medialib is designed to degrade gracefully, so in case of errors in loading an image a small white rectangle is drawn where the image should be, no exceptions are thrown and the program continues its execution. This allows learners to visually tell if their programs work as intended, and supports sense making in testing and debugging (as discussed in the previous section). All operations in Medialib are designed to look atomic to the programmer, including the *draw()* function, so that the learner can be sure that her sequential model of execution is always respected: the next instruction, in line 4, will be executed only when the image is completely drawn on the screen. Solution 2 only adds one line to the code of solution 1: a call to the *wait_mouse_press()* function, which blocks the program until the user clicks the mouse button. What appears as a blocking, atomic instruction to the programmer, is in fact just a *wrapper* for a busy-wait that polls Pygame events, but with the advantage of keeping the code completely sequential and the NOM minimal.

To summarize, NOM_{ml} is rather similar to our initial NOM for Python with numbers and strings, i.e. NOM_1. NOM_{ml} only needs rules for sequential execution, blocking operations and primitive data-types; moreover, Medialib adds only a few instructions to the imperative Python fragment defined by NOM_1, to work with images, audio and user input. And of course NOM_{ml} needs to be extended with conditionals, booleans, loops and possibly lists, as they are needed in more complex examples we used in our courses based on Medialib (see next section). But with our approach we can avoid, or at least postpone, the discussion of object-oriented concepts, event-handling and function definitions, scope of variables, and still write short, working and readable multimedia programs. NOM_{ml} can effectively be used as the start of the spiral of incrementally more complex NOMs.

4.3 Medialib Commands and Programming Style

The main inspiration for the Medialib design came from Processing [8], a language aimed at enabling designers to rapidly prototype visual and multimedia programs. Processing is not implemented as a standalone programming language, but as a simplification of Java: the result is an Embedded Domain Specific Language (or EDSL for short, as discussed in [7]) for Java, which offers a rather elegant programming style and is quite different and simpler than its host language. We see a strong connection between the idea of an EDSL and the concept of a NOM, therefore we decided to follow the same approach with Medialib, and only implement a minimal set of commands designed to cover the typical tasks that we needed to discuss in our introductory programming courses. These few commands are meant to appear to the beginner programmer as build-in instructions, and Medialib only uses primitive types and avoids complex parameters in the commands parameters, such as Python lists, tuples or objects. Table 1 shows all the commands of the latest version of Medialib, grouped in four categories.

Medialib has commands for input such as keystrokes and mouse clicks or mouse positions, and they exist in two versions: a blocking and a reading version. The programmer can have her program block and wait for any keystroke, or decide to retain control and use a reading command to check whether there is a key pressed at a specific point in the execution. This second solution allows implementing more responsive programs, in which things can happen even if the user has not provided any input (typical of interactive programs or games). Providing both versions of input commands helps maintain a simple and intuitive sequential interpretation of programs' execution. Given that most

Table 1 Complete table of commands in the Medialib library

Category	Definition	Effect
Graphics	clear()	Clears the drawing window
	draw(img_file_name,x,y)	Loads and draws image
	rect(x,y,w,h)	Draws a rectangle
	save_screen(file_name)	Saves screenshot to file
Graphical text	set_font(file_name)	Sets new current font from font-file
	text(message,x,y,font_size)	Writes message at position
	get_text_rect(message,x,y,font_size)	Returns width and height of text bounding box
Input	wait_key_press()	Blocks and returns the pressed key
	is_key_press(key_name)	Non-blocking keyboard input
	wait_mouse_press()	Blocks until mouse clicked
	is_mouse_press()	Non-blocking mouse input
	get_mouse_pos()	Returns mouse coordinates
Others	wait(secs)	Blocking pause
	play(sound_file_name)	Loads and plays audio file
	all_done()	Closes the drawing window
	point_inside_rect(px,py, x,y,w,h)	Tests if point (px,py) is inside rectangle (x,y,w,h)
	distance(x1,y1, x2,y2)	Returns Euclidean distance between 2 points
	a_to_b(a,b,t_percent)	Linear interpolation

programs written in beginner's courses are short and shallow (with respect to cyclomatic complexity), we adopted a specific programming style in the examples we provide to the learners: we write flat code (i.e. avoid user-defined functions) to keep a single, global scope, and we avoid complex data structures in favour of multiple simple-typed variables. Functions can easily be introduced as code-reuse devices, in later, more refined iterations of the NOM.

5 Case Studies

We developed the Medialib tool as an exemplar of our understanding of how programming could be simplified for non-technical university students (as discussed in [34]). Our multimedia library was developed through a loose participatory design method [3], as it addressed specifically the need of our colleague from Kyushu University (in Fukuoka, Japan), Jingyun Wang, who had much experience in conducting a Python course for non-technical students, and asked us for help to cope with recurring issues. This collaboration between us and Jingyun became the first case study for the Medialib; she was in charge of a general introductory to programming course at KU, where bachelor exchange students from many different educations, mostly non-technical ones, get introduced to programming and Python. During the course, our colleague sent us requests to improve specific features or fix bugs, so that new versions of the Medialib were released as the KU course ran. Our second case study was a course held in Odense, Denmark, at the University of Southern Denmark (SDU) by one of us. The students at SDU were in the first year of their master, and their course was an introductory course in Data Science called Digital Methodologies, which included elements of CT and enabled us to introduce basic programming in Python with the Medialib, focusing on basic competences in Data Science. The Medialib was further improved to solve minor usability issues and introduce new features needed in the Data Science course, in particular there was a need to add basic support for fonts and graphical text, in connection to the creation of interactive infographics. Considering both case studies, the Medialib was developed and tested in a series of four iterations, based on data gathered through the feedback of our colleague Jingyun, her KU students, and the SDU students.

Because of Covid-19 restrictions, both courses ran online through video conference, email, and local Content Management Systems for the distribution of course material and students' assignments. This caused additional issues with the quality of the courses, hindering the teachers' ability to closely follow the students' technical issues and learning difficulties. Both case studies were organized as inductive, qualitative, research through design investigations [38], and given the circumstances, we adopted a netnographic approach to our study [4, 16], collecting data through: video conference and note taking during online classes, analysis of the students' assignment during the class, questionnaires, and a series of final video interviews conducted with a small focus group from the students at KU. The students of both courses were presented with the same questionnaire, with minor changes in relation to the course structure; the questionnaire included multiple choice questions, Likert scale and open-ended questions, in which the students had for instance to write about their major, list the software they use and how often. The collected data were analyzed through an interpretive thematic analysis of the students responses, regarding what they found easy to understand or challenging in the use of the Medialib, the tasks we gave them, as well as new design requirements for the improvement of the library.

Our plan is to continue to develop and test our theoretical approach and our Medialib library, to gain a deeper understanding on the matter of simplifying programming for non-technical students and to improve the Medialib tools for future employment in universities and high schools.

5.1 Introductory Programming Course in Fukuoka

The Medialib was developed and tested for the first time during a generic, introductory CT course at KU, targeting exchange students enrolled in various non-technical educations. The class was composed of 22 students, from different Asian countries and enrolled in their first year of their master studies. The course was centered on the Python language and the book "Think Python" and it ran for 15 lectures, each lasting 90 minutes, through the spring semester 2019. Data was gathered through the teacher's observations of classes, note taking, questionnaires and a series of final interviews with a subgroup of the students. The questionnaire was proposed twice, during the course and at the end, to compare responses and to evaluate how the students' perception of the course and the Medialib tool changed over time. The final interviews were conducted with 16 students in a semi-structured form [4], starting from an initial set of five questions on their experience of the Medialib and of the course structure, they were also asked to comment on specific answers they gave to the final questionnaire in order to gain more details on their experience (details on the study can be found in [34]). The course was mandatory and aimed at providing exchange students with basic programming skills in Python and the students perceived it as a way to gain technical skills that could improve their CV, and might be useful in their future studies. According to data from the questionnaire, the students came from different educations, mostly related to biology and business, and only one student had taken programming courses before. The students were proficient in the use of different software systems beside the Office Suite, a few of them had tried

HTML, software packages for audio-video editing and statistics. All the students also said that they did not feel confident in their mathematical knowledge; moreover, 35% said that they struggled with mathematics and preferred avoiding it. In general, the students participated actively in the lectures and in doing the assignments, demonstrating engagement and asking for help or clarifications on several occasions. The original structure of the course, from previous years, was based on the topics covered in introductory programming courses for technical students, covering simple algorithms for mathematical problem-solving very much in line with the TCS outlined in Sect. 2. Even in the restructured instance of the course that we studied, the first lectures still revolved around presenting Python without the Medialib, and the teacher assigned classic introductory Computer Science tasks to the students. The main challenge of this course was its *generalist nature*: programming is a vast practice, addressing a variety of problems and fields of inquiry, each requiring specific set of skills. For instance, programming to create a web page or to scrape data online represent two rather distant and different practices, requiring specific languages, data structures, algorithms, and possibly different NOMs (as discussed in Sect. 4). When restructuring the KU course materials, our goal was to concretize programming to make it more accessible to the students with largely different backgrounds, therefore, we adopted multimedia as a **generic** and **known** field on which the students were expected to be able to leverage their experience of multimedia as users [34]. As a result the course materials focused around composition of and interaction with images and audio, through algorithmic thinking, leveraging student cultural and sensorial preunderstanding, to enable the them to make sense of their code without having to recall or understand too many mathematical notions.

Results from the study suggest that the students were able to quickly make sense of their code and engage in debugging, which was turned into a self-questioning practice. The students argued in the questionnaires and interviews that they started to spontaneously wonder “what could I do next?” with their code, conducting trial and error experiments. During the interviews, all the students said that they found it easier to do their exercise with the Medialib, than with the initial Python exercises without our library. A female student from biology said: “It is easier, it requires to write less, so I can better think of the problem!”. A male student from business said:“It is of course important to be able to use the language per se, but when I moved to the Medialib I could better understand the problem, I think I can move to the language with a clearer understanding of [...] how it works!” During the interviews, the students were asked about the exercise that they found most appealing or interesting, and all of them agreed on the last one in the course, which required to make a game, combining various materials encountered during the whole course. Another girl from biology said that:“[I liked] The game! It put all the lectures into perspective!”, a boy and a girl from Agriculture said that:“[the game exercise] made everything more clear,[...] how the different parts of the language work together!”. Several students commented that it was exciting or interesting to see how games are made. Moreover, during the interviews three students said that they felt “proud” when the code worked, especially when mistakes were made and they were able to correct them on their own. A critical point was raised by three students in biology and one from business, regarding how and if they will use what they learned in their future studies, but that it was a nice experience, and they might want to learn more about programming.

Our choice of multimedia as the main domain for the course and our Medialib was dictated by our hermeneutic approach: we tried to compensate for the lack of a common focus among the students, picking media as a familiar domain that could enable them to access the hermeneutic spiral, leveraging their preunderstanding of digital media as users [9, 33]. And according to our findings the students were in fact able to connect elements of their code to specific features and behaviors, leading them to make sense of their code, and providing a sense of achievement.

5.2 Digital Methodologies Course in Odense

During the spring semester of 2021, the Medialib was deployed and tested with a class of 24 students, enrolled in their first year of the master programme in Media Studies at SDU in Denmark. The test took place during the *Digital Methodologies* course, which targets digital competences in the field of Data Science aimed at conducting inquiries in media sociology. The course covered topics such as: research design, netnography, online interviews and surveys, content and thematic analysis, use of software to scrape data from the Internet, and visual representation of data through diagrams and infographics.

This course provided a less challenging context to test our approach and the Medialib with respect to the KU course, as it had already a clear focus on Data Science and the students had a similar background: they were all Danish and humanists, twenty from the BA in Media Studies, two from the BA in multimedia design, and other two from English Literature. In this way, it was relatively easy to find a grounding to tailor the course so that it would concretely support their education, providing practical skills in Python that they could use later in their master projects and in their professional life as communication and media professionals. The course ran for twelve, three-hour lectures, with one weekly lecture. The course was divided in three modules of 4 lectures each:

- Module 1: an introductory module on research design for Data Science, qualitative and quantitative methods, including some initial exercises in netnography and thematic analysis with the tool *Nvivo*;
- Module 2: a programming module, covering basic coding in Python, graphics via the Medialib, and Data Science via a few standard Python libraries and data structures;
- Module 3: thematic, content and network analysis.

As part of the course, the students were also expected to conduct an inquiry on a topic of their choice, from the domain of Media Sociology, applying relevant methods and software tools presented during the course. Their inquiry would be discussed in a report following the structure of a research paper circa 20 pages in length; in fact, this course is considered essential in preparing the students to tackle the methodological part of their master thesis project and to provide an opportunity to practice scientific writing. The main challenge for us was to organize the course, since we could use only one third of the course (i.e. four lectures) for teaching basic programming in Python, introduce graphics programming as well as scraping and cover some of the most common data formats for Data Science, like CSV files. Therefore, we decided to cover the following topics in the programming module: basic imperative programming, algorithmic data visualization, interactive visualizations that can react to key-presses or mouse clicks, automatic data analysis with simple statistics computed on mock or offline data sets, and web-scraping through API. As in the KU course, on the last lecture of the programming module the students were given a recapitulating exercise, which they had to personalize and deliver together with their final report.

Because of the lockdown in Denmark, the course was conducted online: the students were given the lecture material through the SDU Content Management System and we had class through video conference. We gathered data by taking notes during the lectures, and the students were sent an online questionnaire in the end of the course. The questionnaire was the same used to gather data on the Japanese case study, with a few alterations according to the framework of the new course. Furthermore, the Media students got the questionnaire only once, while KU students got it twice, as the Media module was too short to allow for multiple data gathering on the students experience with the Medialib. Regarding the students' experience with software, all the students used professional software package beside the Office Suite, mostly statistical software and the video editing tools from the Adobe Suite, as film shooting and editing is a core part of the Media Studies programme. However, none of them had ever tried programming before and their expectations for the course were rather vague, such as discovering a new knowledge field and trying new specialized software (according to our questionnaire).

During the course the students engaged actively with the material, a few issues were encountered by the students who used Mac laptops. Some instructions were given also through the chat, which we consider as a summarized log of the classes. A student needed specific guidance to open the terminal and locate Python on her computer to check that it was the right version, she said with pride: "Oh, I have never done it before!". Being the course online, it was difficult to make sure that all the students were active, we could only be sure when they asked for help and this happened on several occasions during each lecture. Other issues emerged with Mac laptops, for instance one of the initial code examples was supposed to show a pop-up window with images of cupcakes, but it kept freezing for Mac users; a student asked: *"Anyone with a MacBook having issues with the pop-up freezing?"*. Other students said they had the same problem and another student proposed a solution:"Cmd+space search for terminal [...] check top of screen [...] options is on top of screen in Mac". Accidents like these were made more complicated by the absence of physical presence, as we could not act on the students' computers, and neither of us has a Mac, so it was helpful that the students found ways to help each other even online. A few students were eager to show that they were in control and as soon as they finished their task they would write in the chat: "Done" or even "Done :)". A female student also wrote: "It works well :D". The students habitually added an emoji to communicate their sense of accomplishment, a behavior that we encouraged adding emojis to our own replies in the video-lecture chat. In this way we aimed at reinforcing an informal and reassuring atmosphere in the class, so that the students could feel free to discuss their difficulties with the exercises and technical issues without fearing negative judgement on our side. The lack of physical presence, prevented us from walking among the desks of the students to find out about any issues, so we tried to use the chat to create a more *discreet space* for students with issues to come forward without feeling too exposed. Some students for instance would write to us in private in the chat, and in some occasions we managed to talk in the breakout room or during breaks. In one instance a group had issues with an exercise and wrote privately in the chat; we remained connected after class and we were able to solve the issue together. However, in other occasions the students would just come forward presenting an issue, openly discussing eventual mistakes. In this sense, the video-lecture chat provided protection for shy students, who could opt to contact us in private.

During the course we received positive feedback from the students, who pointed out how the course was different from the typical courses in humanities, which are centered on reading and analyzing written texts. At the end of the second programming class a female student commented in the chat: "It has been cool. I really enjoyed it :) [...] It's nice to have some practical stuff. It differentiates from what we're used to :D". The students showed particular interest for the coding exercises, which turned into forms of self-questioning practice, and triggered trial-and-error experiments when things did not work as expected. The students showed or expressed pride when they could solve their issues on their own, spontaneously engaging in debugging practice. A group of female students wrote us an email asking for help and to meet online before the class; they wrote again a few hours later, saying that they managed to make their code work:"I just managed to make it work :-) [...] It just teases you now and then ;-)".

In conclusion, even from our preliminary analysis of the data in this second case study, the course seems to have enabled the students to engage with coding in a gradual way, as we leveraged Data Science as a topic to enable them to enter the hermeneutic spiral with respect to programming.

6 Discussion

The main contribution of this study is theoretical: a pedagogical approach in simplifying programming which combines hermeneutics and NOMs. the approach is embodied by the Medialib library, a design exemplar (in line with [38]). The two case studies described in the previous section show how our approach can be used to restructure introductory programming courses, taking advantage of our library. The two studies carry distinctive differences that provided us with specific and complementary insights.

6.1 Different Structures

As already mentioned (see previous section) the courses in the two case studies had a different structure, as the first one was a general purpose introductory course to programming in Python, addressing non-technical exchange students and lasting for an entire semester, while the other was a 4-weeks programming module, conducted within a course in Data Science for 1st semester master students in Media Studies. In the first course, we lacked a common ground among the students' interests and cultural background, which could provide a concretizing foundation to the course, hence we found in multimedia a common ground to provide a gradual access to programming practice, in the terms of the hermeneutic spiral [13]. As recommended in hermeneutical pedagogy [30, 33] we took advantage of the students' cultural preunderstanding and their sensorial preunderstanding. In fact, the focus on multimedia leveraged the students' cultural preunderstanding, as we counted on their personal experience with images, audio and video as users of digital technologies. The sensorial preunderstanding was central to the design of the Medialib and the exercises we developed. Most exercises (in particular the early ones) focused on placing, composing or moving images on screen, or playing audio file when certain user inputs are detected. In this way, the students we able to form simple and intuitive expectations of what their programs should do, in terms of seen or hearing media, and that allowed them to be very effective in debugging their code. Multimedia proved a rich enough source of CT tasks and exercises, that we could avoid traditional mathematical and logical problems, yet still have learners engage with programming features like conditionals, loops, data-structures and debugging.

On the other hand, the second case study came already with a common ground among the students, who were all Media Students and had to learn about Data Science. In this case, we had since the start a clearer selection of topics to cover, such as infographics and algorithmic data visualization, user interaction, and scraping via WebAPIs. However, we had less time at our disposal as the programming module only lasted 4 weeks, and these students had no prior programming experience. We decided to structure the programming module as shorter, compact version of the hermeneutic spiral, that instead developed throughout the entire course in the first case study. Therefore, we used our Medialib to both introduce them to Python and programming in general, and to show them how to visualize data with short, readable programs. From the first programming lecture, the students were provided with premade code to be run and later edited, to alter code's the behavior. In some of these pre-made examples simple bar charts would be drawn on screen: data was used to change the width of strips of colored rectangles. These programs generate infographics by drawing a series of diagrams, made of rectangles, showing different grouping of the data and relations among them. In hermeneutic terms, the diagrams that these code examples generate represented wholes composed of parts [9, 33] whose properties are based on a data set. The students could easily link alterations in size, color and placing of the rectangles to specific lines of the code; furthermore, they could see where to edit the code to change its behavior and the resulting diagrams. In other exercises, the students had to work on web scraping: they were provided with code that could extract data from a web site, and save the data into a CSV file, to be later analyzed. Thanks to the pre-made examples, the students were able to start coding from the very beginning of the programming module, leveraging their preunderstanding of Data Science, introduced to them during their BA and in the first module of the course. Moreover, focusing on how simple data sets could be algorithmically visualized through diagrams and infographics, the students could use their visual and intuitive understanding of the data to verify the behavior of the code was as expected.

6.2 Findings

Comparing the two case studies, we found that it is easier to approach programming for non-technical students within the context of a specific course than within a generic course, simply aiming at introducing programming. In our experience, it is necessary to first consider the question: "programming what?". Hence, we suggest proceeding by finding an application field (i.e. a domain) for the programming, which can act as a common ground for the students. Generic courses are generally shaped on introductory courses for technical students, and implicitly on the TCS we outlined in Sect. 2. However, those students must develop a deep knowledge of programming, therefore, they need to get a solid theoretical basis, which will become useful while exploring different forms of programming on their path of becoming professional programmers. On the contrary, non-technical students are in the process of developing other professional profiles, in this respect programming for them will be a tool to be used in solving specific problems, delimited within specialized areas of their fields, and often in collaboration with technical professionals, as part of working teams. Therefore, providing non-technical students with generic introductory knowledge in programming might be less useful than providing basics knowledge of programming targeting specific fields, and result in a longer unproductive phase, when the generic knowledge cannot be applied to any field-relevant problems. Grounding programming on a specific field has the added benefit to suggest to the students how and in which area of their future studies programming might be useful, hence, elicit motivation to learn. In hermeneutic terms, the field in which the course is grounded will also provide a resource for preunderstanding, leveraging what the students have started to learn in the parts of the course that are more familiar to them and also in other courses in their programme, hence smoothing their path towards a fusion of horizon between their more familiar areas of knowledge and programming.

During our observations we also noticed that students felt a sense of accomplishment when they were able to *read the code*, i.e. understanding which instructions did what. In this way, they started spontaneously to zoom in and out of the code, and back and forth between the code and its (visual) behavior, acquiring a broader perspective on the whole of the code, accessing the hermeneutic spiral on their own. Through this dynamic, the students engaged in forms of self-regulated inquiry, setting questions to themselves on how they could alter the code to change its result. In this sense, we find that sensorial preunderstanding and employing multimedia in introductory programming courses for non-technical students is a precondition to enable the students to spontaneously engage in exploratory inquiries on their code. This is in line with current research on CT, already starting from Wing [37], which proposes to concretize CT learning activities through designerly inquiries. However, we find that there is a need to further understand how non-technical students experience coding and come to develop an understanding of their code as a complex text, shifting from its whole and its part, and from the text to its behavior.

6.3 Future Support and AI

On a more technical level, working with Python and specific libraries can be challenging for teachers. While tools exist that can visualize and help explain how code executes (brilliant examples are Python Tutor [11], Jeliot [20] and BlueJ [2]), and are sometimes based on explicit NOMs, many of these tools work only for the core language they support, recognizing possibly a few of the main standard libraries, but they cannot cope with external libraries. So, a teacher interested in showing to her students the execution of a Python program that uses Pygame Zero, cannot in fact rely on execution visualization tools. We see a need for better integration of pedagogical libraries in existing IDEs; for example, next-generation IDEs could provide ways to declare, recognize NOMs and external libraries, and support execution visualization modules that can be customized and abstracted with respect to given NOMs.

Moreover, based on our experience in designing and teaching programming courses for beginners, as well as from other studies we are conducting in the orchestration of hybrid classrooms [15],we know that programming teachers using specific libraries with beginners (such as PGZ or our Medialib) typically need support in 2 areas:

- The generation of multiple variations of an exercise, all with similar complexity and characteristics;
- Support to validate submitted solutions.

The first problem can benefit from AI-supported example-based generation of multiple programming exercises with similar characteristics; the teacher could present a code example, possibly annotated to better express certain stylistic choices or constraints to consider when mutating the code. The example could then be mutated using refactoring-like operators, and following smart heuristics to ensure the resulting variations are still meaningful and in the same complexity class of the original code. Even if tools exist that are capable of similar code mutations, we are not aware of systems that can take in consideration external libraries, and that could work for instance with Python and our Medialib. The second problem is a long-standing one: real-time, semi-automatic validation of submitted solutions to programming tasks, given one or a few valid solutions as reference. Also here we expect that AI could help. Unit testing could offer a good initial metaphor: it is written by a programmer, in the form of assertions or test suites, but it can be checked automatically and eventually produce a descriptive summary of the problems and fail points within the code. A more automatic, AI-enhanced version of a unit testing system could be able to take a set of submitted solutions and an official solution, suitably annotated by the teacher, and test-check each submitted solution against the actual behavior of the teacher's solution. Interactive programs are usually the most complex to compare in this way, therefore we propose that annotations should refer to expectations circa the user inputs, and possibly constraints on the use of memory resources or types of data in the code.

Finally, our approach in the creation of the Medialib could be regarded as a high-level recipe to create domain-specific libraries for beginners, that can be fruitfully applied to teaching AI itself in beginners' courses. According to our approach the first step would be an analysis of the typical, most commonly adopted teaching material and textbooks, their structures and types of exercises. A minimal NOM could then be built starting from a small but powerful enough fragment of the Python language, and extended to include a minimal number of central concepts needed to express the typical material and exercises. From our experience we would suggest avoiding event-based mechanisms and OOP in the initial NOM, and focus instead on a few powerful (and possibly modular) imperative commands that can be given a clear meaning, i.e. that allow the NOM to explain their semantics in a straightforward way. Data structures should also be reduced to a minimum in favor of build-in types. Libraries that attempt to simplify AI exist, and a good example is *simpleai* (official web page *https://simpleai.readthedocs.io*) which is based on material from the classic Russel and Norvig AI book [23]. The documentation and examples show that the focuses of simpleai are code readability, reusability and modularity of AI algorithms, and to work with a library that is

[...] made with a more "pythonic" approach [...]

Therefore, it is natural that the library designers leverage on classes and inheritance. However, this approach is contrary to our hermeneutic spiral approach and would require a rather complex "minimal NOM" to be defined. Considering these problems, we regard a library like *ml5.js* to be perhaps a better starting point for creating a simplified, beginner's friendly AI library in Python. The ml5.js library (see https://ml5js.org/) is implemented to work on the web, with the *p5js* programming language, which in turn is related to the Processing language, and it builds on top of the powerful and modern *TensorFlow.js* machine learning library. The main challenge in creating a Medialib-style version of ml5.js in Python would be in finding ways to reduce the conceptual complexity of machine learning and TensorFlow's entities and operations; fitting such powerful and complex ideas in a minimal NOM, usable by beginners, would require working together with machine learning experts and deconstruct textbooks and exercises together.

7 Conclusion

A central concern to the field of CT deals with how to simplify programming, to make it accessible to individuals without a technical background. Programming is generally regarded as a complex professional practice, aimed at the making of software and framed within a design process. Although it has been stated that CT is not only programming (Wing [37]), yet programming remains the main challenge in the design of CT pedagogical approaches and tools. In our study, we explored how programming can be simplified for non-technical university students and we propose a double contribution:

1. A knowledge contribution represented by our new pedagogical approach in simplifying programming, grounded on hermeneutics and NOMs;
2. The Medialib library, which represents a design exemplar of our approach [38].

Our approach to simplify programming was to find a theoretical lens that could describe the learning path of beginner programmers in terms of knowledge distance. In our case, that lens was hermeneutic (and the hermeneutic spiral), and we proceeded by combining it with a more operational counterpart, notional machines (or NOMs). NOMs can be used to define the cognitive complexity of algorithms [6], but here instead we needed a way to define and assess the complexity (or by opposition, the simplicity) of pedagogical approaches to CT. We analyzed typical code examples used in textbooks and video-courses and at the various libraries for beginners (in particular for Python). We then developed our own library, called Medialib, based on the assumption that the first NOM presented to beginners programmers should be as simple and *small* as possible, in terms of number of concepts and their interconnections. To investigate *how simple* our library was, we defined a NOM for a minimal imperative fragment of Python. This fragment is powerful enough that beginners can use it to mentally execute *flat*, imperative programs of a complexity comparable to the typical examples used in beginners' textbooks and online material. We then proceeded to define possible initial NOMs for different, popular approaches, and in this way we could establish that the Medialib has indeed a rather small NOM, possibly smaller than most other approaches. In fact, the Medialib's NOM is not much more complex than the NOM for our minimal Python fragment.

Medialib simplifies programming also by using multimedia as the main domain to introduce CT: this idea spawns directly from the hermeneutic spiral and the concepts of cultural and sensorial preunderstanding. The early Medialib programs that we present to the learners are about visualizing images, and that provides them with an intuitive grounding to explore their code and its resulting outcome, specifically linking specific instructions to characteristics of the visualization such as: sizes, placing, colors, repetition of elements. We argue that the Medialib enables non-technical students to build, from the very beginning, an intuitive understanding of their code, leveraging sensorial and cultural preunderstanding of digital media.

Our Medialib was created and tested as part of the restructuring of two different programming courses for non-technical university students. Data from those case studies suggests that simplifying programming is most effective if a specific application field, i.e. a domain, can be found, that can act as a common ground for the students. This is in line with the hermeneutic spiral, since it is a way to leverage on the learners' cultural preunderstanding: in one of our case studies the domain was multimedia programming itself, while in the other it was Data Science.

Finally, we found that the hermeneutic spiral and notional machines proved to be a very compatible and productive combination, which allowed us to approach code as a *particular form of text* aimed at problem-solving, and provided us with an interpretive perspective, respectful of the needs of non-technical students, who need to approach programming and code from their own perspective. We believe that the approach presented in this paper can be used to create "simplified" libraries also for other domains, for example machine learning. We also identify the need for better programming environments for non-technical students, that could support incremental NOMs and offer a smoother learning curve. We see AI playing an important role in making such environments effective, by supporting learners and teachers in areas like intelligent semi-automatic assessment, and example-based generation of tasks of comparable complexity.

Acknowledgements We would like to thank Jingyun Wang (from Durham University, UK) for providing the initial problem that brought us to create the Medialib, and for supporting its testing and deployment. We would also like to thank Marco Ragni and Nina Bonderup Dohn (from SDU, Denmark) for the very productive and stimulating discussions that helped frame and improve this paper.

References

1. Balzer W, Eleftheriadis A, Kurzawe D (2018) Digital humanities and hermeneutics. Philos Inquiry 42(3/4):103–119
2. Berry M, Kölling M (2016) Novis: a notional machine implementation for teaching introductory programming. In: International conference on learning and teaching in computing and engineering, LaTICE 2016, Mumbai, March 31–April 3, 2016, pp. 54–59. IEEE Computer Society . https://doi.org/10.1109/LaTiCE.2016.5
3. Björgvinsson E, Ehn P, Hillgren PA (2010) Participatory design and "democratizing innovation". In: Proceedings of the 11th Biennial participatory design conference, pp 41–50
4. Drotner K, Iversen SM (2017) Digitale metoder: at skabe, analysere og dele data. Samfundslitteratur
5. Duran R (2019) Blog post "notional machines" . https://compedonline.school.blog/2019/07/26/notional-machines
6. Duran R, Sorva J, Leite S (2018) Towards an analysis of program complexity from a cognitive perspective. In: Proceedings of the 2018 ACM conference on international computing education research, ICER '18, pp 21–30. Association for Computing Machinery, New York. https://doi.org/10.1145/3230977.3230986
7. Fowler M (2010) Domain-specific languages. Addison-Wesley, Upper Saddle River
8. Fry B, Reas C (2021) The processing language, official website . https://processing.org/
9. Gadamer HG (1989) Truth and method (J. Weinsheimer & DG Marshall, Trans.). New York: Continuum
10. Grondin 2017 (2017) Gadamer's interest for legal hermeneutics. Law's hermeneutics: other investigations. Routledge, Oxford, pp 48–62
11. Guo PJ (2013) Online python tutor: embeddable web-based program visualization for CS education. In: Proceeding of the 44th ACM technical symposium on Computer science education, pp 579–584
12. Heidegger M (1962) Being and time (J. Macquarrie & E. Robinson, Trans.)
13. Horban O, Maletska M (2019) Basic hermeneutic approaches to interpretation of videogames. Skhid 163(5):5–12
14. Iordache C, Mariën I, Baelden D (2017) Developing digital skills and competences: a quick-scan analysis of 13 digital literacy models. Ital J Sociol Educ 9(1):6–30
15. Jakobsen M, Nyborg M, Valente A (2021) Towards a new tool for individualized content delivery in classrooms. In: Learning and Collaboration Technologies (HCII 2021). Springery
16. Kozinets RV (2015) Netnography. Int Encycl Digital Commun Soc 39:1–8
17. Kristensen K, Marchetti E, Valente A (2021) The global challenge of designing e-learning tools for computational thinking: a comparison between east asia and scandinavia. In: e Lecture Notes in Computer Science (LNCS). Springer, Germany . http://2021.hci.international/
18. Malan DJ (2019) Cs50 2019-lecture 0-computational thinking, scratch . https://www.youtube.com/watch?v=jjqgP9dpD1k
19. McGugan W (2007) Beginning game development with python and pygame: from novice to professional (beginning from novice to professional). Apress, New York
20. Moreno A, Myller N, Sutinen E, Ben-Ari M (2004) Visualizing programs with jeliot 3. In: Proceedings of the working conference on advanced visual interfaces, pp 373–376
21. Piotrowski M, Neuwirth M (2020) Prospects for computational hermeneutics. In: Atti del IX Convegno Annuale AIUCD
22. Pope D (2021) Pygame zero—official webpage . https://pygame-zero.readthedocs.io/en/stable/introduction.html
23. Russell SJ, Norvig P (2003) Artificial intelligence: a modern approach. Pearson Education . http://portal.acm.org/citation.cfm?id=773294
24. Schleiermacher F (1998) Hermeneutics and criticism and other writings. Cambridge University Press, Cambridge
25. Seppälä O, Duran R, Becker B, Denny P, Barik T, Ball T, Velázquez-Iturbide Á, Sorva J (2019) Notional machines for scratch and python. In: Dagstuhl Seminar 19281, pp. 18–19

26. Severance CR (2021) Online course—programming for everybody (getting started with python) . https://www.coursera.org/learn/python?specialization=python
27. Severance CR, Blumenberg S, Hauser E (2016) Python for everybody: exploring data in python 3. CreateSpace Independent Publishing Platform, North Charleston
28. Shiffman D (2021) Online video course—the coding train . https://www.youtube.com/watch?v=yPWkPOfnGsw
29. Sorva J (2013) Notional machines and introductory programming education. ACM Trans Comput Educ 13(2):8. https://doi.org/10.1145/2483710.2483713
30. Sotirou P (1993) Articulating a hermeneutic pedagogy: the philosophy of interpretation. J Adv Compos 13(2):365–380
31. Sweigart A (2016) Invent your own computer games with python, 4th Edition-free online book. No Starch Press. https://inventwithpython.com/invent4thed/
32. Tedre M, Denning PJ (2016) The long quest for computational thinking. In: Proceedings of the 16th Koli calling international conference on computing education research, pp 120–129
33. Tomkins L, Eatough V (2018) Hermeneutics: interpretation, understanding and sense-making. SAGE handbook of qualitative business and management research methods pp. 185–200
34. Valente A, Marchetti E, Wang J (2020) Design of an educational multimedia library to teach python to non-technical university students. In: P. Zaphiris, A. Ioannou (eds.) Proceedings of the 9th International Congress on Advanced Applied Informatics (IIAI-AAI), pp. 169–175. IEEE. https://doi.org/10.1109/IIAI-AAI50415.2020.00041
35. Vorderman C (2017) Computer coding python projects for kids: a step-by-step visual guide. Computer coding. Dorling Kindersley Limited . https://www.dk.com/uk/book/9780241286869-computer-coding-python-projects-for-kids/
36. Vorderman C (2018) Computer coding python games for kids. Dorling Kindersley Limited, London
37. Wing J (2017) Computational thinking's influence on research and education for all. Ital J Educ Technol 25(2):7–14
38. Zimmerman J, Forlizzi J (2014) Research through design in HCI. In: Ways of knowing in HCI, pp. 167–189. Springer