How beginner-friendly is a programming language? A short analysis based on Java and Python examples

Jean-Philippe Pellet[1,2], Amaury Dame[2], and Gabriel Parriaux[1]

[1] University of Teacher Education, Lausanne, Switzerland
{jean-philippe.pellet,gabriel.parriaux}@hepl.ch
[2] École polytechnique fédérale de Lausanne, Switzerland
amaury.dame@epfl.ch

Abstract. In this paper, we are interested in criteria to help us choose a programming language for a freshman programming course. The audience is future engineers who won't be computer scientists or IT professionals. We are therefore more interested in conveying elements of computational thinking and logic rather than full mastery of a given language. Following a rather exceptional situation where we had to give substantially the same course, in parallel, once in Java and once in Python, we relate here some syntactic and semantic aspects of the two languages which, in our experience, ease the teaching or learning of basic programming concepts. We argue that in quite a few cases, Python makes basic concepts easier to introduce because of less *syntactic noise* and less *conceptual noise*. We also propose a short list of syntax- and semantics-related desiderata for a beginner language—which neither Java nor Python completely answer.

Keywords: Choice of programming language · Computer science education · Syntax and semantics

1 Introduction

How to choose the first programming language keeps being one of the most debated questions in computer science education [11]. Lower-level languages are closer to the machine and may be able to convey a better conceptual model of how a computer works; higher-level languages let us express more concisely more complex computations, data manipulations, more general problem-solving strategies. Both can be desirable depending on the course objectives and the relevance of a language can never be assessed independently of its audience.

We teach introductory programming to college freshmen. They are future engineers, but they probably aren't future computer scientists, software engineers, or developers. Our course doesn't aim at making students proficient in the chosen programming language. It is jointly taught with another course presenting an introduction to the foundations of computing. For most students, these two courses will be the only ones fully dedicated to computing and programming. Our course duo is thus positioned as presenting the theoretical foundations

of computing, discussing the bases of algorithmic approaches, reviewing open questions, and getting acquainted with the basics of programming. The focus is set on structured approaches to problem solving emphasizing higher-level, potentially transferrable skills like modeling, abstraction, decomposition, etc.—referred to by some as *computational thinking* [15].

Many articles have been written about what a first programming language should look like, some of which we mention in Section 2. Here, owing to our experience and to the context mentioned in Section 3, we decided to explicitly put focus on the comparison between Java and Python from a syntactic and semantic point of view. Language-design decisions, be them major or minor, have a large influence on how easy certain concepts can be taught with them, and how syntactically or semantically intermingled these concepts can be. Throughout Section 4, we show and discuss short code excerpts in Java and Python taken from the course and discuss the two approaches, introducing the concepts of *syntactic noise* and of *conceptual noise*. As we go through the discussion, we propose several desiderata for a programming language used in such a beginners' course. That list can be used to determine the extent to which, from the mentioned points of view, a language can be seen as appropriate for introductory teaching of programming. We finally conclude in Section 5.

2 Related Work

A quite comprehensive survey of literature on teaching introductory programming [11] covers 4 main topics, one of which is language choice. The authors mention many references and classify them into those which (a) discuss a given language, (b) compare two or more languages, and (c) present general pedagogical criteria for an easier language choice. Among those criteria, a distinction is made between internal criteria (related to the language itself) and external criteria (industry demands, trends, availability of teaching material, etc.). Our paper can be classified as presenting internal criteria for language choice while justifying them with a comparison between two languages.

An early paper written in a similar spirit [7] lists seven "deadly sins" of introductory programming language design. The authors present internal criteria and illustrate some of them with various popular languages at the time (which, however, do not include Java or Python). This is built upon by another paper that adds the consideration of external criteria [7].

Other authors point out that generally valid criteria are hard to define as different courses have different learning objectives [6] (hence our contextualization in the last section) and that language choice remains an ever-contentious issue.

Closer to our setting, we can read about a comparison of errors made in 60 beginner programs, 30 in Java and 30 in Python [5], and find that Python programs contained on average fewer syntax errors, logic errors, and better error-handling code. The same authors further examine Python [4], concluding it is simple enough for introductory teaching in high school already.

So while the general topic of this paper has been discussed many times, the detailed syntactic and semantic elements we investigate below, to the best of our knowledge, have never been treated precisely in such a learning context.

3 Context & Initial Thoughts

In the next section, we'll present our observations and findings from the parallel Java and Python courses we gave in the setting explained in the introduction. The courses consisted in 14 weeks of a lecture of 45 minutes followed by a 90-minute practical exercise sessions. The main teaching objectives are formulated as follows: at the end of the courses, students shall be able to...

- build a conceptual model of how a computer executes code;
- use the basic control structures of a chosen programming language (conditions, loops, functions);
- choose and use appropriate data structures (sets, sequences, associative maps) to solve a given problem;
- build simple data models and process data with them.

The code excerpts in the following section which support our discussion were directly taken from the lecture slides or exercise solutions. We want to point out that the excerpts were not specially crafted to demonstrate the superior adequacy of either language. They were historically written for the Java course, and were recently migrated over to Python for a new variant of the course given to another faculty in the same institution.

Before diving into the code excerpts, we want to address a short list of preliminary objections one could make to the analysis below and which we have heard during discussions with colleagues.

— *"You can't compare Java and Python. Java is typed, safe, compiled, efficient; Python is for fast prototyping and lacks all of the above. They have totally different use cases."*

Of course, we can compare them! We must compare them, since at some point a decision has to be made. This doesn't in any way negate their different relevance. They come with different tradeoffs—which just have to be discussed from a teaching and learning perspective, independently of their merits in the industry or in other fields.

— *"I don't want to choose a single language. I could very well teach Python* and *Java. Learning to transfer skills from one language to another is important."*

In the beginning, sticking to one language is important. We know that transference cannot be expected from novices and only comes with mastery [3,1]. Plus, the question of language choice would hardly be solved—one would need to agree on *two* languages instead of just one.

— *"You say needing to care about feature X of the language is a hindrance for beginners. But I want them to have to care about X: it teaches them how to be precise, how to care about formal details, and they'll end up knowing more about the lower-level foundations and how the machine works."*

Our semester is 14 weeks and there's only so much we can do in that time. We could of course have decided that such learnings matter and are relevant. But skipping them also gives us more time to address higher-level concepts that may more easily be applicable to other languages (and, more broadly, to general computational thinking), contrary to lower-level, language-specific constructs.

— *"One should start programming with a block-based visual language, which avoids the mentioned syntactic pitfalls."*

Yes, one could. Actually, we would strongly recommend such an approach for K–12 education, for instance. But we should stay aware that not all of the syntactic pitfalls are avoided simply by using a block-based language. The issue of conceptual noise remains, and visual languages have limitations of their own.

— *"I don't want to teach in any language that is not statically typed."*

This seems to be a common gripe. Many traditionally dynamically typed languages can now include type annotations in language extensions (or derived languages)[3]. *Gradual typing* thus becomes possible and allows to fine-tune the tradeoff between (roughly out) the extra typing necessary and the additional diagnostics a static type checker can provide. It sounds hardly possible, though, for a statically typed language to relax its need for types—although modern languages have type-inference capabilities that alleviate the feeling of being too "constrained" by the need of declaring types everywhere. Whether to type or not type is a huge theme on its own that we, for lack of space but with regret, don't discuss further in this paper.

4 Analysis & Desiderata

We now present a few code excerpts[4] and comment on the differences in ease of teaching and learning that we have observed. Note that, in such a short paper, we make no claim of exhaustivity or coverage whatsoever. Also note that the arguments and desiderata presented below are to be read with the target audience in mind—non-CS, beginner students, first semester of college—and we don't claim that they automatically suit a more general context.

Excerpt 1. A slightly more complicated "hello world".

Java

```
J1 public class Demo {
J2   public static void main(String[] args) {
J3     int side = 4;
J4     int area = side * side;
J5     System.out.println(area);
J6   }
J7 }
```

Python

```
side = 4                    P1
area = side * side          P2
print(area)                 P3
```

This could come right after a "hello world" example. Its intent is to show the concepts of (a) storing a value in a variable; (b) using an arithmetic operator;

[3] MyPy with type annotations since Python 3.5, TypeScript or Flow for JavaScript, type hints since PHP 5, Typed Racket, Typed Closure, etc.

[4] The width of one unit of indentation was chosen to be 2 columns for easier layout, but was always presented to students as 4 columns.

and (c) printing something to the console, and illustrate the way to determine the "entry point"/main function of the program. To newcomers, it is already pretty dense code, as there are indeed these three new concepts to figure out conceptually and to recognize syntactically[5]. The semantics here of Java and Python is really close, but the syntactic effort is quite different. Symbols like braces and brackets are worth pointing out because they need more explaining to understand since for beginners, they are not immediately linked to a concept they know—an exception may be the parentheses in the definition and application of a mathematical function[6]. But the opening and closing braces and brackets, the dot and the semicolon are all here "parasitic" in Java, because they are needed even for this very small example to run. Moreover, no fewer than five keywords are needed, under each of which a potentially complicated concept is hiding. Usually, we tell student not to worry about them now and just to accept them—which is problematic, because from one of the very first examples, this shows that there's a lot of unknown already in the basic lines they have to write.

We call this *noise*, defined as extraneous elements that have to be typed in (and, potentially, understood), while being unrelated, or very indirectly related, to the basic concept we wish to illustrate with some code except. This noise is strongly linked to the syntax of the language. We propose to call it *syntactic noise* when it is due to simple syntactic rules of the language without a strong underlying, independent concept (e.g., the need for semicolons in Java), and to call it *conceptual noise* when additional concepts need to be explained and understood to measure the implications of the syntax being used (e.g., the use of `static` in Java).

We see that the Python version, on the contrary, is terse and corresponds line by line to the three new concepts with no syntactic or conceptual noise. (We avoid the issue of typing, rapidly mentioned in Section 3.)

> **Desideratum 1.** The first steps in a language (e.g., console print, variable assignment) should be painless and minimize both the syntactic noise and the conceptual noise.

Excerpt 2. Creating an associative data structure containing 3 key–value pairs.

Java

```
J1 Map<Integer, String> numbers = new HashMap<>();
J2 numbers.put(1, "one");
J3 numbers.put(2, "two");
J4 numbers.put(10, "ten");
```

Python

```
numbers = {
  1: "one",
  2: "two",
  10: "ten"
}
```
P1 P2 P3 P4 P5

This examples demonstrates the definition of a simple map (the preferred term in Python being "dictionary") containing three entries. If it is meant to

[5] There are other, less directly apparent concepts like scoping and memory management that can be discussed with this example and that we opted to leave out for now with beginners.

[6] For consistency with the syntax used for stack-allocated objets with custom constructors, C++ goes the other way and allows any variable to be initialized with a parenthesis-based syntax resembling function application.

solely illustrate the possibility of such a data structure, then the Python code does just that, without conceptual noise. The Java code as shown here needs the `new` keyword, the `Integer` object type (to beginners, subtly different from the basic type `int`), type inference with the diamond notation, and repeated method calls on the constructed objects. Syntactically viewed, Python here has a more declarative approach, whereas Java's approach is imperative and requires a more complex mental model of what is going on, which must then be explained. Conversely, Python has special syntax here: the braces, the colon and the comma all have a specific meaning in this context, which must be learned. We argue that the cognitive effort of it, however, is lesser than the one of learning the bits of the object-oriented concepts needed to grasp the essence of the Java code.

Note that, since Java 9, one could write: `Map.of(1, "one", 2, "two", 10, "ten")` (with up to 10 key–value pairs—yielding here an immutable map), wrapping lines as needed to achieve a style closer to the Python style. The syntax, however, denotes nothing else than a regular function application—looking in the middle of the argument list provides no immediate way for the reader to distinguish a key from a value without scanning further left or right, and even less if the keys and values have the same type.

This example here begs the question: as we introduce a new concept that is materialized in a language, when is new syntax desirable? On the one hand, reusing previously introduced syntax and concepts seems elegant: new ideas can be expressed in terms of simpler, basic constructs, and keeping the core of a language small and expressive enough to cover most needs sounds elegant. On the other hand, one can argue that a sufficiently different concept warrants new syntax. Having a specialized syntax dedicated to a concept makes it immediately recognizable in the code. Provided the use of that concept is sufficiently widespread, a dedicated syntax becomes actually an easier way to embody and convey the new concept, rather than showing how existing syntax can be "semantically overloaded" to support this new concept as well. Reusing existing syntax puts the emphasis on *how* a certain mechanism is implemented in the language, whereas special syntax better denotes the *intent* of the programmer[7].

> **Desideratum 2.** Broadly used concepts sufficiently different from other concepts should have dedicated syntax (and not rely on existing syntactic constructs).

Basic data structures, we argue, fit these requirements. Data structures are often mentioned right next to algorithms as a constituents of programs, right up to the very name of classic textbooks [16]. To us, it seems reasonable to define such initial data with a syntax that looks declarative rather than imperative—just like defining the structure of graphical user interface is more conveniently tacked declaratively (e.g., React, Vue.js, SwiftUI, etc.). A syntax denoting primarily a function call is maybe not best used to represent associative data. In that spirit,

[7] As a side note: this is one of the main arguments for having a dedicated syntax for implicit parameters in Scala 3, whereas they were written as regular parameter lists in previous versions. The mechanism was found embodying sufficiently different concepts (like typeclasses) and was found to be clearer with dedicated syntax [10].

having specialized syntax to denote, for instance, a linear data structure (an array, list, or set), and an associative data structure (map) thus seems worthwhile. In this regard, Python has special syntax for list, set and (as shown above) dictionary literals; Java has special syntax for array literals only. Today, Java arrays are regarded as a low-level data structure, even being called "deprecated" in favor of generic dynamic lists [9]. The interest of special syntax for them seems clearly smaller than it would be for more dynamic and flexible data structures.

We also note that many operations mutating simple linear and associative data structures can be done via subscripting in Python. Subscripting exists in Java but only for arrays and only via `int` indices. In Python, lists for instance support subscripting via the indication of a single index like in Java, but also of slices of indices[8]. Slice-based subscripting supports assignment as well, thus enabling higher-level multiple modifications in a single statement. A syntactic advantage is that the assignment operator `=` used more widely in various circumstances where it denotes a mutation of the data structure on its left hand side. This is consistent with a simple variable assignment known from some of the very first code examples.

Excerpt 3. Extracting repeated code sections in methods/functions.

Java

```
J1  public class Friends {
J2    public static void main(String[] args) {
J3      Map<String, Set<String>> friendships =
J4          new HashMap<>();
J5      addFriends(friendships, "A", "B");
J6      addFriends(friendships, "A", "C");
J7      addFriends(friendships, "D", "C");
J8      System.out.println(friendships);
J9    }
J10   public static void addFriends(
J11         Map<String, Set<String>> friendships,
J12         String name1, String name2) {
J13     addOneWay(friendships, name1, name2);
J14     addOneWay(friendships, name2, name1);
J15   }
J16   public static void addOneWay(
J17         Map<String, Set<String>> friendships,
J18         String name1, String name2) {
J19     if (friendships.containsKey(name1)) {
J20       friendships.get(name1).add(name2);
J21     } else {
J22       Set<String> newSet = new HashSet<>();
J23       newSet.add(name2);
J24       friendships.put(name1, newSet);
J25     }
J26   }
J27 }
```

Python

```
friendships = {}                                    P1

def add_friends(name1, name2):                      P2

  def add_one_way(name1, name2):                    P3
    if name1 in friendships:                        P4
      friendships[name1].add(name2)                 P5
    else:                                           P6
      friendships[name1] = {name2}                  P7

  add_one_way(name1, name2)                         P8
  add_one_way(name2, name1)                         P9

add_friends("A", "B")                               P10
add_friends("A", "C")                               P11
add_friends("D", "C")                               P12

print(friendships)                                  P13
```

This excerpt demonstrate the effort to factor out repeated code. Here, we are trying to populate a map linking names to sets of the names of their friends, symmetrically (i.e., if `"A"` is in the set of friends of `"B"`, then the converse must be true as well). We are trying to isolate two procedures (here, methods in Java and

[8] For instance, `mylist[1:4]` yields a new Python list with elements 1 (incl.) to 4 (excl.). Custom classes can also support subscripting with arbitrary subscript types.

functions in Python): (a) a high-level procedure linking two new friends (lines J10–J15 and P2–P9) which, for its implementation, calls twice (b) a lower-level procedure adding an element to a set in a map (lines J16–J26 and P3–P7), creating the set as required[9].

One immediately evident difference is the block syntax: Java uses a brace-based syntax heavily based on C's whereas Python uses the colon character and significant indentation to determine block start and end. Much has been written about it already [13,4] regarding ease of reading, writing, editing, including caveats with copy-pasting, so we'll focus on other aspects. Another key difference is the ability, in Python, to define named functions much more flexibly. Java's method as always top-level members of a class whereas Python functions can be defined in any code block and have access to symbols of the enclosing lexical scopes (simplifying a bit). Python's `add_friends` and `add_one_way` can refer to `friendships` without it being passed along as an argument as in Java. There would be other ways to avoid the extra parameter in Java: declaring it as a field. But this comes with its own conceptual noise: if it's a static field, one has to either explain its semantics or ask students to accept that it works as such. If it's a non-static field, the code in `main` needs to create a new instance. In both cases, the question arises of where to initialize the field (inline or in a method) and of initialization order, which may not follow the order of the lines in the file any more. Furthermore, in Java, both method definitions and field definitions may begin with as many as the same two keywords plus a type information (e.g., `public static int`), which can be confusing to the untrained eye. In Python, from the first keyword on, `def`, it is unambiguously clear that a function is being defined and no mix up with a variable (or attribute) is possible[10]. Also note that the possibility to define nested functions avoids polluting the outer namespace with symbols that are not used later anyway.

Finding repeated patterns in data and similar patterns in code is an essential part of computational thinking. Giving names to subprograms performing a given task is equally an essential means of abstraction, as it allows the creation of higher-level constructs based on lower-level ones. We believe it should be made as easy as possible in a language used for beginners.

Desideratum 3. Functions should be syntactically easily definable and quickly recognizable so as to allow convenient code factorization.

[9] Usage of a `defaultdict` was intentionally avoided here in Python as it requires passing a function as argument.

[10] This says nothing of how, for instance, fields and methods should be *accessed*—Eiffel's uniform access principe [8] provides an nice abstraction over a class's implementation details and can be achieved in Python with *properties* (though not in Java).

Excerpt 4. Creating a simple compound data type with basic inheritance.

Java

```java
public class Rectangle extends BoardElement {
  public double width;
  public double height;
  public Rectangle(Point center,
         double width, double height) {
    super(center);
    this.width = width;
    this.height = height;
  }
  @Override public String toString() {
    return "Rectangle(c=" + center + ", w=" +
        width + ", h=" + height + ")";
  }
}

Rectangle r = new Rectangle(
    new Point(10.0, 10.0), 5.0, 2.0);
```

Python

```python
class Rectangle(BoardElement):
  def __init__(self, center,
               width, height):
    BoardElement.__init__(self, center)
    self.width = width
    self.height = height

  def __repr__(self):
    return (
      f"Rectangle(c={self.center}, "
      f"w={self.width}, h={self.height})"
    )


r = Rectangle(
      Point(10.0, 10.0), 5.0, 2.0)
```

This example serves the discussion of two concepts: the definition of a custom compound data type, and the basic introduction of class inheritance. They are treated separately in the text below. (Note that this Java code is not idiomatic: fields are typically private in Java with a public getter or setter as appropriate. In Python, attributes are technically all public and, by convention, are prefixed with one or two underscores to signal their non-publicness.)

The ability to define and use compound data types (i.e., a type constituted from several basic types of the language and possibly other compound types) is one of the first concrete steps of data modeling in a programming course. Students get taught that certain values that belong together in order to model a given entity should also "live" and "travel" together in code. In Java and Python, the most obvious way to achieve that is by declaring new classes[11].

Both the Java and the Python ways to declare a new class are relatively verbose and contain seemingly redundant elements. Both languages have a new keyword for it, `class`. In languages supporting class-based object orientation, classes are such a central concept that contesting the relevance of that keyword cannot be justified. The way to treat fields, constructors and constructor parameters, however, is different. Java requires the explicit declaration of all fields, whereas in Python, conventions dictate that the fields (referred to as attributes) be assigned in the `__init__` "constructor method". Owing to its dynamic nature, Python actually allows new attributes to be defined at later points of the lifetime of a created instance with no prior declaration, whereas Java forbids it. Python is consistent here with its treatment of attributes and its treatment of, say, local variables: no declaration is needed. But in the perspective of basic data modeling, the essence of the definition of a compound type is the identification of what subparts it is made of. Java field definitions makes it clear, but Python makes it

[11] In Python, one could also used `namedtuples` for cases where methods are not needed. The data structures built this way, however, are immutable. Our take was not to skip discussions over mutable data structures, an essential side of imperative programming language.

(at least partly) implicit. On the other hand, Java's syntax seems overly verbose for the very common use case where all fields obtain their initial values from the constructor: the identifier `width` appears no fewer than 4 times, compared to 3 in Python, owing to the nonexistent declarations.

It is notably difficult to explain to students the need for a simple constructor as shown in the example above. Especially the use of the same identifier as parameter name and as field, as is idiomatic in both languages, must be explained carefully. In Java, the use of the "`this.`" prefix also often is a source of confusion: it has to appear on lines J7 and J8, but is optional on lines J11 and J12 to refer to the fields. In general, referring to the current object is difficult to understand [12] and confuses beginners even more if it can be done in a non-systematic way. Of course, we as instructors could make it a rule to always use it even when optional. But the fact that students are bound to find code online that doesn't and, most probably, to end up writing code of their own that works without this prefix forces us to explain it anyway. Python here has another approach: the `self` parameter is always mentioned explicitly in the parameter list of methods, and only when prefixed by "`self.`" can attributes be referenced (also on lines P9 and P10). The always needed prefix improves on the syntactic consistency: no matter the context, class members (attributes and methods) are always prefixed by some expression evaluating to an instance, contrary to Java. It is thus always immediately apparent whether the code is accessing a local variable or a field, or calling a function or a method. But the mismatch in the number of parenthesized elements at definition site and at call site (where explicitly passing `self` mustn't be done) is confusing and requires additional explanations. Worse is the allowed coexistence of a way to call methods as functions, bypassing virtual dispatch, as shown on line P4, where `self` must be explicitly passed[12].

Desideratum 4. Compound data types should be definable in a syntactically light way and should clearly allow the identification of their fields.

Both languages suffer from the fact that even after the above definition, there is no out-of-the-box mechanism for checking for the equivalence of two instances; i.e., if `r1` and `r2` are two `Rectangle`s constructed with the same arguments, they will not be considered equivalent neither in Java (via `r1.equals(r2)`) nor in Python (via `r1 == r2`—more on that syntax below). Additional methods have to be implemented for that—even if, in most cases, their implementation is systematic and repetitive. The same goes for the generation of hash values for such instances: the default implementation is based on reference equality and not instance equivalence and will probably not act as intuitively expected when inserted into hashing containers such as sets and maps[13].

[12] Here, one could have written `super().__init__(center)` instead. But the fact remains: calling the constructor method of the superclass happens by using `__init__` name, in contradiction with the general guideline that "you shouldn't access it yourself if it has double underscores".

[13] At this stage, one cannot help but point to, e.g., Scala's way of defining such a compound type, where the definition of the constructor and the fields coincide

Excerpt 5. Manipulating data structures representing numerical data.

Java

```
J1  Vector3 v1 = new Vector3(1, 2, 3);
J2  Vector3 v2 = new Vector3(4, 5, 6);
J3  Vector3 v3 = v1.plus(v2).div(2);

J4  // Given a class similar to this (with a proper
J5  // toString() and equals() omitted here for brevity):
J6  public class Vector3 {
J7    public final double x, y, z;
J8    public Vector3(double x, double y, double z) {
J9      this.x = x; this.y = y; this.z = z;
J10   }
J11   public Vector3 plus(Vector3 v) {
J12     return new Vector3(x + v.x, y + v.y, z + v.z);
J13   }
J14   public Vector3 div(double d) {
J15     return new Vector3(x / d, y / d, z / d);
J16   }
J17 }
```

Python

```
P1  from numpy import array

P2  v1 = array([1, 2, 3])
P3  v2 = array([3, 2, 1])
P4  v3 = (v1 + v2) / 2
```

Lines J3 and P3 both compute in a 3D space, the vector $\boldsymbol{v_3} = \frac{\boldsymbol{v_1}+\boldsymbol{v_2}}{2}$. Java's impossibility to use arithmetic operators on custom types makes it impossible not to use method calls to accomplish this. But Python makes it more legible and writable by allowing operators (here in conjunction with a type defined in the `numpy` library). Allowing operators has the added benefits of reproducing the precedence rules we know from algebra, whereas, to the untrained eye, distinguishing, e.g., `v1.plus(v2).div(2)` from `v1.plus(v2.div(2))` is difficult and may even be counterintuitive.

Note that the class definition in Java only serves to demonstrate how "arithmetic methods" are achieved, not to artificially lengthen the Java code. Conceptually, one must understand that, in order for chained method calls (as on line J3) to work, one needs to return an instance of the class, too. This in turns begs more questions that are avoided by Python's syntactic possibilities.

Another more difficult point of Java is equivalence checking: `==` denotes an identity check rather than an equivalence check, which must explicitly be done by means of the `equals()` method. It even gets trickier, because students rapidly get used to using `==` for all basic types and even for instances of `String`—and, more often than not, it actually works, since Java interns all strings literals compiled together. It only bites them later. Python's `==`, on the other hand, calls the method `__eq__` on its left operand, thus allowing custom behavior via standard syntax[14]. The same arguments can be made for the `<`, `<=`, `>`, and `>=` operators on instances of data types which are ordered.

Desideratum 5. Common arithmetic, equality, and comparison operators should work on compound data when relevant to ensure syntactic consistency with basic types.

syntactically, and code for equality, hash value, and string representation is generated automatically (see https://docs.scala-lang.org/tour/case-classes.html).

[14] Even if we all know that `==` is all but standard for beginners and that the confusion with `=` is extremely common.

Having stated these few desiderata, we regret not having evaluated them according to students' performance on tests or exams. This analysis emerged after the course was completed, at a point where all tests were already completed and turned out not to allow a proper isolation and evaluation of the aspects above. They are, however, based on our experience as teachers and are also stated in a way such that they are independent of this precise Java–Python comparison, being in principle applicable to any potential first langage for beginners.

5 Conclusion & Outlook

Forty years ago already, researches wrote that "programmers tend to favor their first language to the extent of applying the style or structures of this language when programming in other languages" [14]. As teachers, we had better give serious thinking about the choice of the first language: especially when teaching to beginners clear embodiment of general concepts rather than of language-specific intricacies. The same author, however, writes that "the effects of poor teaching are an order of magnitude more significant than the effects of a poor first language"—so, of course, we know that wisely choosing the first language is not the end of the story. Recently, ten quick tips for teaching programming were proposed [1], each of which is worth considering.

Based on our observations above, we have decided to move away from Java in favor of Python because many concepts were found to be more easily explainable without extraneous noise—while also noting that Python does not completely answers our desiderata. Another observation is that Python seems to be a choice more and more common at high school for students taking computing classes, such that the skillfulness at the start of the semester in Python may differ greatly among students. Choosing a radically different language (e.g., a functional language like Scheme or Haskell) was also mentioned as a way to start from scratch for everyone with high probability [2].

References

1. Brown, N.C.C., Wilson, G.: Ten quick tips for teaching programming. PLOS Computational Biology **14**(4) (2018)
2. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: The structure and interpretation of the computer science curriculum. Journal of Functional Programming **14**(4), 365–378 (2004)
3. Gick, M.L., Holyoak, K.J.: The cognitive basis of knowledge transfer. In: Cormier, S.M., Hagman, J.D. (eds.) Transfer of learning: Contemporary research and applications. Academic Press (1987)
4. Grandell, L., Peltomäki, M., Back, R.J., Salakoski, T.: Why complicate things?: introducing programming in high school using python. In: Proceedings of the 8th Australasian Conference on Computing Education-Volume 52. pp. 71–80. Australian Computer Society, Inc. (2006)
5. Mannila, L., Peltomäki, M., Salakoski, T.: What about a simple language? analyzing the difficulties in learning to program. Computer Science Education **16**(3), 211–227 (2006)

6. McIver, L.: Evaluating languages and environments for novice programmers. In: PPIG. p. 10 (2002)
7. McIver, L., Conway, D.: Seven deadly sins of introductory programming language design. In: Proceedings 1996 International Conference Software Engineering: Education and Practice. pp. 309–316. IEEE (1996)
8. Meyer, B.: Object-oriented Software Construction. Prentice-Hall (1988)
9. Naftalin, M., Walder, P.: Java Generics and Collections. O'Reilly (2007)
10. Odersky, M.: Some mistakes we made when designing implicits (and some things we got right). Talk given at Typelevel Summit 2019 in Lausanne (2019)
11. Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., Paterson, J.: A survey of literature on the teaching of introductory programming. ACM SIGCSE Bulletin **39**(4), 204–223 (2007)
12. Ragonis, N., Shmallo, R.: A diagnostic tool for assessing students' perceptions and misconceptions regards the current object "this". In: Pozdniakov, S.N., Dagienė, V. (eds.) Informatics in Schools. Fundamentals of Computer Science and Software Engineering. pp. 84–100. Springer International Publishing, Cham (2018)
13. Sanner, M.F.: Python: a programming language for software integration and development. J Mol Graph Model **17**(1), 57–61 (1999)
14. Wexelblat, R.L.: The consequences of one's first programming language. In: Proceedings of the 3rd ACM SIGSMALL symposium. pp. 52–55. ACM Press, New York, New York, USA (1980)
15. Wing, J.M.: Computational Thinking. Communications of the ACM, Viewpoint **49**(3), 33–35 (2006)
16. Wirth, N.: Algorithms + Data Structures = Programs. Prentice Hall (1976)