Programming languages

Author(s)
Silvio Peroni – silvio.peroni@unibo.it
Department of Classical Philology and Italian Studies, University of Bologna, Bologna, Italy

Keywords
Grace Hopper; Python; Test-driven development

Copyright notice
This work is licensed under a Creative Commons Attribution 4.0 International License. You are free to share (i.e. copy and redistribute the material in any medium or format) and adapt (e.g. remix, transform, and build upon the material) for any purpose, even commercially, under the following terms: attribution, i.e. you must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use. The licensor cannot revoke these freedoms as long as you follow the license terms.

Abstract

These lecture notes provide a general introduction to programming languages and then focus on a particular language: Python. The historic hero introduced in these notes is Grace Hopper, who was the first programmer of the Harvard Mark I computer and was responsible for the development of some of the first programming languages.

Historic hero: Grace Hopper

Grace Brewster Murray Hopper (depicted in Figure 1) was a computer scientist and the first programmer of the Harvard Mark I, i.e. a general purpose electromechanical computer that was used during the Second World War and that was fully-inspired by Babbage's Analytical Engine. She was firmly convinced of the need of having machine-independent programming languages that brought her in the development of *COBOL*, one of the first high-level programming languages, which is still used today for some applications.

COBOL (i.e. the *common business-oriented language*) is a programming language designed for business use that brings a quite extensive use of English terms for describing the operations of a program. The idea of adopting, for the very first time, English for commands made the programming language a bit more verbose but also more readable and even self-documenting. Just for making an example, in today's languages if we want to compare if the value assigned to a variable `x` is greater than the one assigned to another variable `y` we should use `x > y`. In COBOL, the same comparison is done with the following instruction: `x IS GREATER THAN y`.

Figure 1. Portrait of Grace Hopper. Picture by James S. Davis, source: https://en.wikipedia.org/wiki/File:Commodore_Grace_M._Hopper,_USN_(covered).jpg.

A brief history of programming languages

After the Second World War, several programming languages have been developed according to several design principles and intended usage in terms of the computational problems to be solved. While all of them, in principle, make possible to develop solutions for any solvable computational problem, some of them are more suited for a specific domain than others. For instance, COBOL has been developed for business applications, while FORTRAN was designed to deal with scientific computing.

While an extensive analysis of all the programming languages is out of the scope of the topics of these lectures, it is worth mentioning, at least graphically, a timeline of their evolution, shown in Figure 2. As highlighted in the timeline, we are going to introduce and use a particular programming language in this course, i.e. Python, in particular according to its third version released in 2006.

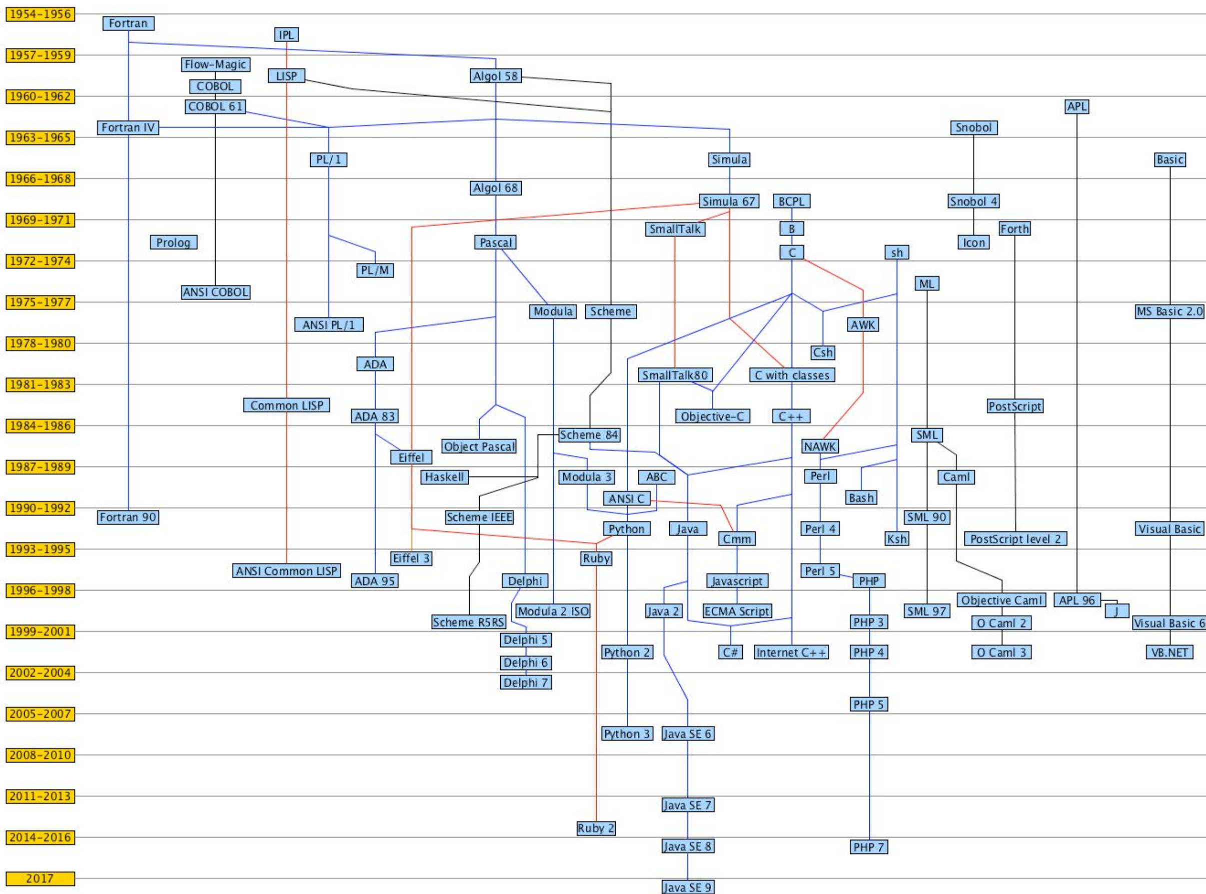

Figure 2. A graphic timeline summary of some of the main programming languages from 1954 to 2017. The different line colour is used only for readability reasons, and it does not have any particular meaning.

Python

Python is an high-level programming language for general-purpose programming, which is currently one of the most used languages for programming in the Web, for Data Science and Natural Language Processing tasks. The good thing about Python is that it is one of the simple languages for starting to study how to program and create software.

In this course, we will use Python in its latest version, i.e. Python 3. Luckily, there are a lot of resources freely available online for learning this language from scratch, such as:

- the introductory book *Dive into Python 3* [Pilgrim, 2009];
- the official documentation of the language;
- an online platform for playing with Python 3 without installing any software on your computer;
- an interactive online course for learning Python from scratch;

- an other book entirely dedicated to problem solving and algorithms developed in Python [Miller and Ranum, 2011];
- a digital book which contains an introduction to Python for the Humanities.

The goal of today's lecture is to develop our first algorithm in Python. The algorithm we develop is the very first one we have introduced in the second lecture of this course about algorithms, that can be described informally by the following natural language text: taking in input three different *strings*, i.e. two words and a bibliographic entry of a published paper, return *2* if both the words are contained in the bibliographic entry, *1* if only one of the words is contained in the bibliographic entry, and *0* otherwise.

First incomplete version, in Python

In Python, each algorithm is defined by creating a new *function* by means of the keyword `def` (which stands for *define*) followed by the name of the algorithm and a comma-separated list of input parameters between round brackets, e.g. `def contains_word(first_word, second_word, bib_entry)`. This definition is then followed by `:` and all the instructions of the algorithm must be specified in the following lines, as an indented block (preferably using 4 spaces). This is illustrated in Listing 1. It is worth mentioning that the name of any function, as well as all its parameters, cannot contain space characters and must always start with a letter – e.g. `this_is_my_parameter` is correct, while `1_parameter` is not.

```
def contains_word(first_word, second_word, bib_entry):
    ...
    ...
    ...
```

Listing 1. The definition of an algorithm, with its input parameter, and some dots identifying where to put the instruction of such algorithm – one per line, indented of 4 space characters.

In this first version of the algorithm, we would like to introduce only some basic constructs of Python. To this end, we provide only a partial solution in this subsection, which will be finalised in the following subsections, following the same strategy used in the previous lecture entitled "Algorithms". In particular, we want just to say that if the first input word is contained in the bibliographic entry, then the number *1* is returned, otherwise *0* is returned. This partial version of the algorithm is introduced in Listing 2.

```
def contains_word(first_word, second_word, bib_entry):
    if first_word in bib_entry:
        return 1
    else:
        return 0
```

Listing 2. An incomplete version of the algorithm, that is used to introduce some basic constructs of Python.

In this partial version, there are already specified some important constructs of Python. The first one is the *if-else* conditional block. This kind of block allows one to execute a particular instruction if a condition is true (the `if` statement), while an alternative set of instructions is executed instead if the condition specified is false (the `else` statement). The `else` clause can be avoided if no alternative set of instructions is needed. The instructions to execute in one case or the other are specified in indented sub-blocks (again 4 additional spaces). As already introduced in Listing 2, every time we have to introduce a new block of instructions, we need to use `:` after the statement of interest, as shown in Listing 3.

```
if <condition>:
    ...
    ...
else:
    ...
    ...
```

Listing 3. The generic structure of an *if-else* conditional block.

The condition specified in the `if` statement shown in Listing 2 allows one to check if a certain string is contained in another one by means of the command in. In particular, `<string1> in <string2>` would be true if the value *<string1>* is contained in *<string2>*, where a *string* is a particular type of value that records a sequence of characters, and it is usually defined by using the quotes. For instance, `"Peroni"`, `"Osborne"`, and `"Peroni, S., Osborne, F., Di Iorio, A., Nuzzolese, A. G., Poggi, F., Vitali, F., Motta, E. (2017). Research Articles in Simplified HTML: a Web-first format for HTML-based scholarly articles. PeerJ Computer Science 3: e132. e2513. DOI: https://doi.org/10.7717/peerj-cs.132"` are all strings. Note that *<string1>* and *<string2>* are just placeholders for strings: we can use directly strings, e.g. `"Peroni" in "Peroni beer"`, or variables referring to strings, as shown in Listing 2 – where a *variable* is a symbolic name that contains some information referred to as a value (e.g. first_word). For instance, any input value is, in fact, a particular kind of variable. As defined previously, all the input parameters of the algorithm are expected to refer to strings.

The last construct of the partial algorithm introduced in this sub-section is the `return` statement. It is defined by specifying the token `return` followed by the value (or the variable containing a value) that must be returned. The execution of a `return` statement finishes the whole execution of an algorithm – thus, all the instructions that follow that statement are not processed anymore. In the example in Listing 2, two different numbers are returned, depending on which branch of the *if-else* block is actually executed. In particular, the algorithm returns *1* if the condition of the `if` statement is true, while it returns *0* otherwise. In Python, any number is defined by writing it down as it is – e.g. `42` and `-42` for positive/negative integers, `1.625` and `-1.625` for positive/negative decimals. Note that strings and numbers are distinct kinds of objects – e.g. the string `"42"` and the number `42` (without the quotes) **are not** defining the same value at all.

Complex boolean statements

The original text of the algorithm, introduced at the beginning of Section "Python", explicitly mentions the simultaneous truth of two conditions for returning *2*: return `2` if both the words are contained in the bibliographic entry. Of course, this can be handled by means of a hierarchy of *if-else* blocks, as shown in Listing 4.

```
if first_word in bib_entry:
    if second_word in bib_entry:
        return 2
    else:
        return 1
else:
    if second_word in bib_entry:
        if first_word in bib_entry:
            return 2
        else:
            return 1
    else:
        return 0
```

Listing 4. A hierarchy of *if-else* blocks for describing the three possible return values of the algorithm.

However, the readability of the previous example is rather difficult, since it repeats several times the same conditions, even if they have been specified in a different order. Thus, Python makes available some operations for assessing compositions of multiple *boolean values*, and for deriving boolean values from number and string comparisons. A boolean type (or, simply, *boolean*, named after George Boole, who was a great logician of the 19th century) can be assigned to one out of two distinct and disjoint values, *True* and *False*. For instance, the condition `first_word in bib_entry` returns a particular boolean: *True* if the word is indeed contained in the bibliographic entry, *False* otherwise. In algorithms (and in any programming language), boolean values are used for organising the execution flow of conditional blocks.

Sometimes it is useful to combine somehow two distinct boolean values in order to simplify the organisation of the conditional blocks. This can be done by using specific operators that apply to one (`<operator> <B1>`) or two boolean values (`<B1> <operator> <B2>`), and return a new boolean value. These operators are called *logical not* (`not` in Python, which applies to one boolean value only), *logical and* (`and`, between two boolean values), and *logical or* (`or`, between two boolean values). They are *logical* operators since all of them derive from the logic Boole proposed in his works on Boolean algebra. Their use is summarised in Table 1, where the *truth table* of the application of such operators is shown. In particular, given two input boolean

values, *B1* and *B2*, the table shows the result of all their possible combinations according to the specific operator. Thus, for instance, in the example in Listing 4, we could return *2* if both the strings are contained in the bibliographic reference, expressing this constraint in one condition only, i.e. `first_word in bib_entry and second_word in bib_entry`.

B1	B2	not B1	B1 and B2	B1 or B2
True	True	False	True	True
True	False	False	False	True
False	True	True	False	True
False	False	True	False	False

Table 1. The truth table of all the boolean operations.

Round brackets can be used for grouping boolean operations, e.g. `(True and False) or False` applies the `and` operation first, and the result is used as the first value of the `or` operation – given *False* as result. In case no brackets are used, the application order is the following: first, all the `not` operation are applied, then all the `and` operations, and finally the remaining `or` operations – for instance, `True and not False or False` returns *True* since it is interpreted as `(True and (not False)) or False`.

In addition to the aforementioned boolean operations, it is also possible to use string comparisons for obtaining boolean values. Table 2 shows all the comparisons that one can apply on two strings, i.e. <S1> <operator> <S2>. In this case, the operators are those typically used numerical comparison, i.e.:

- `<`, less than;
- `<=`, less than or equal to;
- `>`, greater than;
- `>=` greater than or equal to;
- `==`, equal to;
- `!=`, different from;
- `in`, included in;
- `not in`, not included in.

S1	S2	S1 < S2	S1 <= S2	S1 > S2	S1 >= S2	S1 == S2	S1 != S2	S1 in S2	S1 not in S2
`"Alice"`	`"Bob"`	True	True	False	False	False	True	False	True
`"Alice"`	`"Alice"`	False	True	False	True	True	False	True	False

Table 2. The truth table of all string comparisons.

In the case of strings, a string *S1* is *less than* another string *S2* if the former one precedes the latter one according to a pure alphabetic order. Of course, the alphabetic order is used also for assessing when a string is *greater than* another one.

Note that similar operators (excluding in) can be used also for comparing numbers, as shown in Table 3. In this cases, the common mathematical numeric comparisons hold.

N1	N2	N1 < N2	N1 <= N2	N1 > N2	N1 >= N2	N1 == N2	N1 != N2
3	4	True	True	False	False	False	True
4	4	False	True	False	True	True	False

Table 3. The truth table of all the arithmetic comparisons.

Thus, we can reuse these boolean operations in order to rewrite the *if-else* blocks shown in Listing 4 in a more readable way. The result is shown in Listing 5.

```
if first_word in bib_entry and second_word in bib_entry:
    return 2
else:
    if first_word in bib_entry or second_word in bib_entry:
        return 1
    else:
        return 0
```

Listing 5. A hierarchy of *if-else* blocks shown in Listing 4 rewritten according to the boolean operations presented in this section.

Conditional statements with multiple branches

While in the previous subsections we have improved the readability of the *if-else* blocks, Python allows us to do even better. First of all, in the two if statements in Listing 5, we ask to evaluate the same sub-conditions (i.e. `first_word in bib_entry and second_word in bib_entry`) twice. This can be easily avoided by defining new variables. A variable are defined by specifying its name (without spaces), followed by an = and the value to associate to it, i.e. `<variable_name> = <variable_value>`. The value can be specified directly (e.g. a number) or indirectly by using other existing variables, or even complex operations.

In our example, we could create two variables, called `contains_first_word` and `contains_second_word`, assigned to the boolean returned by the aforementioned string comparisons, i.e. `first_word in bib_entry` and `second_word in bib_entry` respectively. In that way, we can reuse such variables in the two if statements, as shown in Listing 6.

```
if contains_first_word and contains_second_word:
    return 2
else:
    if contains_first_word or contains_second_word:
       return 1
   else:
       return 0
```
Listing 6. The *if-else* blocks introduced in Listing 5 where the conditions in the if statements are specified by means of two variables.

In addition to that, we can improve even further the readability of the code by collapsing occurrences of else statements when these contain an if statement as their first instruction. In this case, both the *else-if* pair can be safely replaced by an elif (i.e. *else if*) statement, which specifies the same condition used in the if statement. Thus, the code in Listing 6 can be rewritten as shown in Listing 7.

```
if contains_first_word and contains_second_word:
    return 2
elif contains_first_word or contains_second_word:
    return 1
else:
    return 0
```
Listing 7. The *if-else* blocks introduced in Listing 6 collapsed my means of an elif statement.

Final algorithm

In this lecture we have seen some initial constructs that Python makes available for developing an algorithm, in particular: algorithm definition with input parameters, variables, conditional statements (i.e. `if`, `elif`, and `else`), string, numeric, and boolean values, as well as boolean operations and string and numeric comparisons. All these constructs enabled us to define our algorithm, which is finally introduced in Listing 8.

```
def contains_word(first_word, second_word, bib_entry):
    contains_first_word = first_word in bib_entry
    contains_second_word = second_word in bib_entry

    if contains_first_word and contains_second_word:
        return 2
    elif contains_first_word or contains_second_word:
        return 1
    else:
        return 0
```
Listing 8. The final algorithm developed.

It is worth mentioning that the algorithm proposed originally in the lecture notes entitled "Algorithms" as a flowchart does not map with the one presented in Listing 8. This has been done on purpose, so as to explicitly show that it is entirely possible to develop two different algorithms for addressing the same computational problem.

As a final note, and in addition to use the Python interpreter installed in your machine (in any), several Web applications have been developed for testing your Python code and to show which kinds of objects it creates when running. One of these tools, i.e. Python Tutor, is very helpful for people that are approaching Python for the very first time, since it allows one to see what happens as the (electronic) computer runs each line of code.

Test-driven development

There are different development strategies that can be adopted when one wants to understand whether the piece of software he/she has developed is correct or not – i.e. if it is returning the expected result. One of the most used and effective methods used by programmers is called *Test-Driven Development* (or *TDD*) [Beck, 2003], summarised in Figure 3.

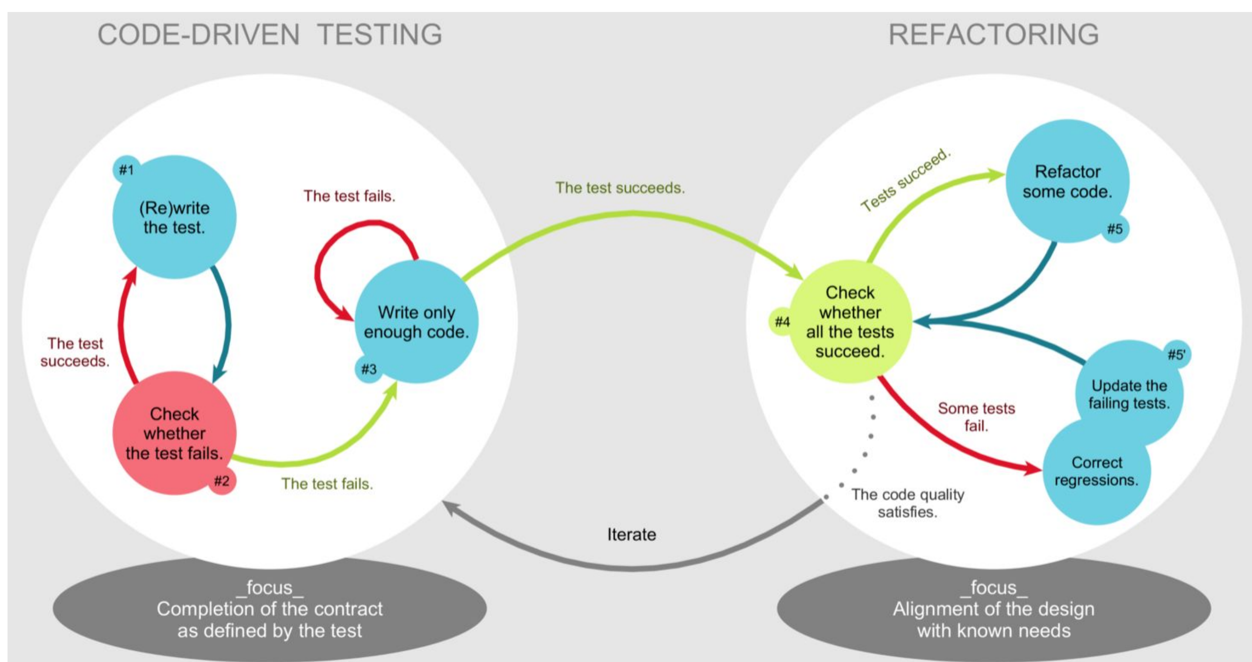

Figure 3. A diagram summarising the steps of the test-driven development approach. Picture by Xarawn, source: https://en.wikipedia.org/wiki/File:TDD_Global_Lifecycle.png.

In practice, when one has a computational problem to solve and he/she needs to develop a piece of software to address it, the first thing to develop is a test so as to check if the software that eventually will be developed behaves correctly (i.e. returns the correct result) or not.

Usually, thus, such test is actually software which must be developed to test the correctness of another software.

Writing a test before to start to developed the software allows one to focus on the problem one has to solve and on the requirements of the software since the very beginning. This approach can be used even when one decides to extend an existing software – where the idea is first to develop the test for assessing the correctness of such new extension, and second to write the extension and check if the extended software passes the new test.

Summarising, the main steps of the test-driven development process are:

1. Write a new test – once understood the computational problem to solve and the related requirements, a new test is written and then added to a collection of previously developed tests.
2. Run the all the tests – all the tests available in the aforementioned collection are run, so as to check that the new test fails, i.e. there is no code available that addresses the particular computational problem described by such test. It is worth mentioning that in the first iteration of the test-driven development the test fails since no code has been already developed at all.
3. Write the new code – in this step, a new piece of code is developed so as to pass the test just added in the collection.
4. Run again all the tests – in this passage, one checks if the addition of such new code developed to address the new test has not break the other features already developed, and tested by all the other tests available in the collection (in any). In case any test fail, then the new code must be corrected until all the tests are passed successfully.
5. Refactor the code – after several iteration of the process, the code grows naturally, and it may be necessary to refactor it so as to clean the code as more as possible, so as to guarantee its readability and maintainability in the long term. As a suggestion, every refactoring action should be checked by re-run all the tests available, so as to be sure that a modification to the code does not break its correctness as well.

```
def test_contains_word(first_word, second_word, bib_entry, expected):
    result = contains_word(first_word, second_word, bib_entry)
    if expected == result:
        return True
    else:
        return False
```

Listing 9. The test function developed for testing the `contains_word` code, introduced in Listing 8. The source code of this listing is available at XX.

Following this approach to the development is very useful also when one has to implement a particular algorithm in Python, so as to check its correctness according to different kinds of input that can be used to run the algorithm itself. A plausible test to check the correctness of the

algorithm introduced in this lecture notes is presented in Listing 9. It is possible to use such tests function in order to test the `contains_word` code with different kinds of input values and related expected results. For instance, Listing 10 shows the test code, the code of the algorithm presented in this lecture notes, and some checks done by running the test code (and thus the algorithm itself) with different input values. The result of the various checks are printed on screen by using the Python function `print()`.

```
def test_contains_word(first_word, second_word, bib_entry, expected):
    result = contains_word(first_word, second_word, bib_entry)
    if expected == result:
        return True
    else:
        return False

def contains_word(first_word, second_word, bib_entry):
    contains_first_word = first_word in bib_entry
    contains_second_word = second_word in bib_entry

    if contains_first_word and contains_second_word:
        return 2
    elif contains_first_word or contains_second_word:
        return 1
    else:
        return 0

print(test_contains_word("a", "b", "abcd", 2))
print(test_contains_word("a", "b", "acde", 1))
print(test_contains_word("a", "b", "cdef", 0))
```

Listing 10. The test code, the algorithm implementation in Python, and three distinct run of the test with different configurations and expected results. The source code of this listing is available as part of the material of the course.

While the proposed development approach could seem banal at a first sight, it is actually adopted regularly by programmers so as to think carefully about the requirements of a certain code to develop and to avoid the introduction of bugs.

In the context of this course, the suggestion is to adopt the test-driven development approach systematically when implementing algorithms in Python, since it is a very useful tool for checking the correctness of the outcomes of an algorithm. To this end, the introduction of all the algorithms in the following lectures will be anticipated by the presentation of the related test, by following the particular template shown in Listing 11, where all the words between angular brackets should be replaced by the appropriate names. It is worth mentioning that, at the very beginning, all the instruction of the algorithm will be substituted by the instruction `return`, so as

to say that the algorithm is actually returning nothing and, thus, to allow all the new tests to fail, as prescribed by the second step of the test-driven development process, introduced above.

```
def test_<algorithm>(<algorithm input params>, expected):
    result = <algorithm>(<algorithm input params>)
    if result == expected:
        return True
    else:
        return False


def <algorithm>(<algorithm input params>):
    return


print(test_<algorithm>(<algorithm input params 1>, <expected_1>))
print(test_<algorithm>(<algorithm input params 2>, <expected_2>))
…
```

Listing 11. The template that will be used for presenting all the algorithms introduced in this course, accompanied by its tests.

Exercises

1. What is the boolean value of `not (not True or False and True) or False`?
2. What is the boolean value of `"spam" not in "spa span sparql" and not ("egg" > "span")`?
3. Following the template in Listing 11, write in Python the algorithm proposed originally in Figure 4 of the lecture notes entitled "Algorithms" as a flowchart, and accompany such code with the related test function and some executions with different input values.

Acknowledgments

I would like to thank some students of the edition of the Computational Thinking and Programming course (a) to have suggested Miller and Ranum's book [Miller and Ranum, 2011] about Python, problem solving and algorithms, which has been added to the list of material suggested for learning Python of this lecture (Sebnem Kabadayi), (b) to have proposed the use of Python Tutor as an application for making the execution of a Python code clear to a novice (Bruno Santini), and (c) to have corrected a mistake in the order of application of boolean operations in Python (Alessandra Foschi). I would also like to thank Paolo Ciancarini, professor in Software Engineering at the Department of Computer Science and Engineering, University of Bologna, to have suggested the adoption of the test-driven development as a teaching tool for the students of the course.

References

Beck, K. (2003). Test-Driven Development by Example. Addison Wesley. ISBN: 978-0321146533, freely available at https://www.eecs.yorku.ca/course_archive/2003-04/W/3311/sectionM/case_studies/money/KentBeck_TDD_byexample.pdf

Miller, B. N., Ranum, D. L. (2011). Problem Solving with Algorithms and Data Structures using Python. ISBN: 978-1590282571. Freely available at https://runestone.academy/runestone/static/pythonds/index.html (last visited 2 December 2017)

Pilgrim, M. (2009). Dive into Python 3. ISBN: 978-1430224150. Freely available at http://www.diveintopython3.net (last visited 2 December 2017)