PYTHON REVIEW

Made by:
Di Grandi Daniele
Master in COSC:
CPLD course (2020-21)
Student number:
7035616

1. History of the language [1]

Python was conceived towards the end of the 1980s and its implementation started in December 1989 by Guido van Rossum in the Netherlands with the objective of replacing the programming language ABC to interface with the Amoeba operating system.
The continuous central role of Guido van Rossum in the decision-making process of Python has been maintained since nowadays, and this fact let him gain the title of *Benevolent Dictator for Life (BDFL)* although he stopped as leader on July 2018. The name of the language came from a popular BBC TV show *Monty Python's Flying Circus,* of which van Rossum is an admirer.

The first version of Python (0.9.0), published on February 1991 to `alt.sources`, already had classes with inheritance, exception handling, functions and the core datatypes of `list, dict, str` and others.
This initial release also had a module system borrowed from the programming language Modula-3 in order to better separate the functionality of a program into interchangeable modules.

In January 1994, version 1.0 was released. The most important features introduced in this version were the functional programming tools such as `lambda, map, filter` and `reduce`. Later on, other minor versions were released, including features dealing with keyword arguments, complex numbers and data hiding.

In October 2000, version 2.0 was released. This version took from functional programming languages such as SETL and Haskell the concept of list comprehensions and implemented it in Python, with a syntax very similar to Haskell's, apart from Haskell's preference for punctuation characters and Python's preference for alphabetic keywords. In the next versions, a number of features were added, such as: garbage collection, support of nested scope, unification of Python's types (types written in C) and classes (types written in Python) into one hierarchy (making Python's object model purely and consistently object oriented), the Boolean `True` and `False` constants and the "with" statement. In December 2008, version 3.0 was released. The philosophy with which it was developed was: "reduce feature duplication by removing old ways of doing things" and as a state of that, this version broke the backward compatibility: Python 2 code does not run unmodified on Python 3. The major changes were: the `print` function was not more a statement but a built-in function, the `raw_input` function was renamed to only `input`, which behaves the same with the input returned as a string instead of being evaluated as an expression, the unification of `str/unicode` types and the introduction of the `bytes` type and the removal of backward compatibility features.

The latest version released (in current time: January 2021) is the 3.9.1 version, released on October 2020.

2. Python style

The language core philosophy is summarized in the document the *"Zen of Python"* [2] written by the software engineer Tim Peters, which is a collection of 19 aphorisms that have the scope to guide the programmer writing good computer programs and these principles are listed as follows:

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.

- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one - and preferably only one - obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea - let's do more of those!

Fun fact: this list is included as an Easter Egg in the Python interpreter, and it will be visible by entering `import this`.

One of the reasons of Python's popularity has to be searched in its modularity nature, in fact, rather than having all the functionality built into its core, Python was designed to be extensible and, because of that, it can be easily used to add programmable interfaces to existing applications.
A common word used in the Python community is: *pythonic,* to describe something that is related to the Python style. Saying that a code is *pythonic* corresponds to saying that it well uses Python idioms, that it conforms to Python's minimalist philosophy and puts emphasis on readability. On the contrary, code that is difficult to understand or that is in contrast with just mentioned principles is called *unpythonic*. [3] An example of *pythonic* code versus *unpythonic* code to swap 2 elements:

UNPYTHONIC:

```
a = 1
b = 2

tmp = b
b = a
a = tmp
```

Image nr. 1

PYTHONIC:

```
a = 1
b = 2

a, b = b, a
```

Image nr. 2

This short but expressive example makes visible the philosophy of readability and minimalism that is built-in as a core part in the syntax of Python: what the *pythonic* program does, it's immediately clear even for a beginner user of Python.

3. Technical analysis

Overview

Python is a high-level multi-paradigm programming language, in fact it fully supports object-oriented programming as well as structured, imperative, procedural programming, and, as already said, many of its features support also functional programming and aspect-oriented programming. Other paradigms are supported via extensions, thanks to the modularity nature that characterizes Python. [3]
The language uses dynamic typing, a combination of reference counting and a cycle-detecting garbage collector for memory management [4]. It also features dynamic name resolution (late binding), which binds method and variable names while the program is in execution. [3]

A Python program is constructed from code blocks, where a block is a piece of Python program text that is executed as a unit. The following are some examples of blocks: a module, a function body and a class definition. Python also has an execution frame, in which a code block is executed. A frame keeps track of some administrative information (used for debugging) and also decides where and how the execution of a program continues, after a code block's execution has ended. [5]

In Python, names refer to objects. Names are introduced by name binding operations. There are several constructs that bind names, such as import statements, class and function definitions, for-loop header, etc.
If a name is bound in a block, it is a local variable of that block, unless declared as nonlocal or global. If a name is bound at the module level, it is a global variable. If a variable is used in a code block but not defined there, it is called *free variable*.
A *scope* defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block. If the definition occurs in a function block, the scope extends to any blocks contained within the defining one, unless a contained block introduces a different binding for the name. When a name is used in a code block, it is resolved using the nearest enclosing scope and the set of all such scopes visible to a code block is called the block's *environment*. Though it is possible that a name is not found, in that case a `NameError` exception is raised. Another case is when the current scope is a function scope and the name refers to a local variable that has not yet been bound to a value at the point where the name is used. In this case, an `UnboundLocalError` exception is raised, which is a subclass of the `NameError` exception.
Python allows name binding operations to occur anywhere within a code block. Local variables of a code block can be determined by scanning the entire text of that block for name binding operations. Moreover, name resolution of free variables doesn't occur at compile time but does occur at runtime. [5]

```
i = 10
def f():
    print(i)
i = 42
f()
```

Image nr. 3

As a simple example, the way name resolution works means that this code will output 42.

Syntax and Semantics

As already said, Python was designed to be readable, and in order to pursue this objective it uses English keywords frequently, while other languages use punctuation. The syntax of Python was developed having in mind the mantra: "there should be one - and preferably only one - obvious way to do it" in order to keep it simple and consistent. Moreover, like many other languages, Python is described by an informal manual and a number of implementations, examples and plain text. No systematic, formal descriptions of its semantics are available, although there was an attempt in the literature to write an operational semantics for a subset language of Python, called *minpy*. [6]
In this master thesis, the author described the state transitions of an abstract machine for the *minpy* language in a formal manner, describing in literate Haskell the rules that cause the abstract machine to change state, where each state is described by an heap (mapping of addresses to values), an environment, a control stack and an instruction, which can be an expression, statement, block that is being executed, the result of the previously executed instruction or an unwind instruction. It is important that the semantics of Python are deterministic, thus, there is one, and only one, rule for each machine state, which are discerned by pattern matches in the rules.
One of the reasons why it could be important to formalize the semantics of a language is that it improves the understanding of that language to its finest details, resulting in a simplification of

reasoning about programs at an abstract level, especially with operational semantics, using an abstract machine for the state, it will closely resemble an interpreter for that language.

3.1 Indentation [7]

From ABC, Python borrows this feature to use whitespaces to delimit control-flow blocks, instead of using punctuation or keywords. The standard rule for indentation is to use 4 spaces. This feature can be implemented in the lexical analysis phase of the language development, where increasing the indentation results in the lexer outputting an INDENT token and decreasing the indentation results in the lexer outputting a DEDENT token. This method corresponds to use curly brackets in other programming languages, such as C.

Consider different versions of a function: a C version and two Python versions, one correct and one wrong because of an indentation error which causes an endless recursion since `foo(x – 1)` is always called:

Image nr. 4

```c
void foo(int x)
{
    if (x == 0) {
        bar();
        baz();
    } else {
        qux(x);
        foo(x - 1);
    }
}
```

C version

Image nr. 5

```python
def foo(x):
    if x == 0:
        bar()
        baz()
    else:
        qux(x)
        foo(x - 1)
```

Python correct version

Image nr. 6

```python
def foo(x):
    if x == 0:
        bar()
        baz()
    else:
        qux(x)
    foo(x - 1)
```

Python wrong version

While the program written in C could not have this type of errors because have distinct markers that distinguish between different blocks, and deleting a block-end marker in C would lead to a compiler error, in Python, especially in this case, not even an editor with automatic indentation could prevent the erroneous behavior of this Python code, resulting in a syntax more readable but less robust.

3.2 Operators [8]

Regarding the arithmetic expressions, Python includes the following operators: `+, -, *, /, //` (floor division), `%` (modulus) and `**` (exponentiations) with their classic mathematical precedence rules. Python also includes the usual comparison operators: `<, >, <=, >=, ==, !=`. These operators can be used to compare plenty of values: numbers, strings, sequences, and mappings can all be compared. For expressions without side effects, `a < b < c` is equivalent to `a < b and b < c`. However, there is a substantial difference when expressions have side effects: `a < f(x) < b` will evaluate `f(x)` exactly once, whereas `a < f(x) and f(x) < b` will evaluate it twice if the value of `a` is less than `f(x)` and once otherwise. As for the Boolean operators `and` and `or`, the minimal evaluation is used: the second argument is executed or evaluated only if the first argument does not suffice to determine the value of the expression.

3.3 Keywords [9]

Python has the following 35 reserved keywords that cannot be used as ordinary identifiers, since they are locked to another scope:

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

Image nr. 7

3.4 Objects [10]

Python is mainly an object-oriented language, where objects are Python's abstraction for data. In a Python program, all the data are represented by an object or by relations between different objects. Every object has an identity, a type and a value.

Identity: Once it has been created, it never changes, it could be imagined as the object's address in memory. The keyword `is` is used to compare the identity of two objects.
Type: The type of an object defines the operations that the object supports as well as the possible values that the object can assume.
Value: The value is simply the value that the object has, and it can be mutable (e.g. lists and dictionaries) or immutable (e.g. numbers, strings and tuples), depending on its type.

When an object is deleted, instead of being destroyed it becomes unreachable and it may be garbage-collected.

3.5 Interesting Classes [10]

Python has several built-in types of classes that offer support for the creation of container types.
- The first class is a `list` class, used to create lists, which are ordered and mutable sequences of items of arbitrary types. In fact, Python allows to append elements with different types in the same list, while in other languages it is forbidden. Elements of a list are wrapped in squared brackets.
- The second class is a `tuple` class, used to create tuples, which are immutable sequences of items of arbitrary types. The difference between tuples and lists is in fact that while lists are mutable, tuples are immutable. Elements of a tuple are wrapped in round brackets.
- The third class is a `set` class, used to create sets, which are mutable containers of hashable items of arbitrary types, with no duplicates. In a set, the items are not ordered, though they support iteration over the items, as well as intersection and union operations. Elements of a set are wrapped in curly brackets, though an empty set is not created by typing: `{}` (which creates an empty dictionary) but are created by typing: `set()`.
- The last interesting class that characterizes Python, is the `dict` class, used to create dictionaries, which are mutable mappings tying keys and corresponding values. The syntax of a dictionary is particular and worth to be mentioned, and it is: `{key: value}`.
Thus, for example, a dictionary could be:

```
a_dictionary = {"key 1": "value 1", 2: 3, 4: []}
```

That is, both numeric and literal values could be used as keys.

3.6 Types

As basic (thus not composite) interesting types, Python supports the following [10][11]:

`int`
Represents integer numbers. From Python 3.0, there is effectively no limit to how long an integer value can be. Of course, it is constrained by the amount of memory that the system has. Variables declared to be `int` type are immutable, it is only possible to assign them a new value.

`float`
Designates a floating-point number. These values are specified with a decimal point. `float` values are represented as 64-bit "double-precision" values, according to the *IEEE 754* [12] standard. In that case, the maximum value a floating-point number can have is approximately 1.8×10^{308}. Python will indicate a number greater than that with the string `inf`, while the closest a nonzero number can be to zero is approximately 5.0×10^{-324}. Anything closer to zero than that is effectively zero. Variables declared to be `float` are immutable in Python.

`complex`
Python does also support the complex type, in order to represent a complex number, composed by a real and an imaginary part.

`str`
In order to handle textual data, the `str` type was created. Strings are immutable sequences of Unicode code points and they are wrapped between quotation marks (either 1, 2 or 3 quotation marks). A string can contain as many characters as the machine's memory permits. Strings also have some special characters in order to specify a desired behavior from the program, such as `\n` for a new line and `\t` for a tab character.

`bool`
Objects of Boolean type may have one of two values, `True` or `False`, which are keywords, however, unlike many other Python keywords, `True` and `False` are Python expressions. The "truthiness" of an object of Boolean type is self-evident: Boolean objects that are equal to `True` are truthy (true), and those equal to `False` are falsy (false). But non-Boolean objects can be evaluated in Boolean context as well and determined to be true or false.

Python is a dynamically typed language. This means that the Python interpreter does type checking only as code runs, and that the type of a variable is allowed to change over its lifetime. The following example demonstrates that Python has dynamic typing:

```
>>> if False:
...     1 + "two"  # This line never runs, so no TypeError is raised
... else:
...     1 + 2
...
3
```

Image nr. 8

Since that the `if` branch will never be executed, its type will never be checked and, as a result of which, no error was raised.

Let's consider another example:

```
>>> 1 + "two"  # Now this is type checked, and a TypeError is raised
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Image nr. 9

Since it is the same expression as before, but now is type checked, a type error was raised.

Moreover, a variable can change its type:

Image nr. 10

```
>>> thing = "Hello"
>>> type(thing)
<class 'str'>

>>> thing = 28.1
>>> type(thing)
<class 'float'>
```

In this example, the variable `thing` has changed its type, passing from a `str` type to a `float` type.

These examples show in a simple way that Python is a dynamically typed language, hence different from languages like C and Java that are statically typed, where the type checks are performed at compile time, thus, without running the program.

This means that in Python, one is free to bind names (variables) to different objects with a different type. So long as only operations valid for the type are performed, the interpreter doesn't care what type they actually are.

Python uses a concept called *Duck Typing* [13][14] synthetized in the phrase: "If it walks like a duck and it quacks like a duck, then it must be a duck". *Duck typing* is a concept related to dynamic typing, where the type or the class of an object is less important than the methods it defines. Using duck typing, Python does not check types at all. Instead, Python checks for the presence of a given method or attribute. By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution.

An example can be seen here:

Image nr. 11

```
class Duck:
    def fly(self):
        print("Duck flying")

class Sparrow:
    def fly(self):
        print("Sparrow flying")

class Whale:
    def swim(self):
        print("Whale swimming")

for animal in Duck(), Sparrow(), Whale():
    animal.fly()
```

This code will produce the following output:

Image nr. 12

```
Duck flying
Sparrow flying
AttributeError: 'Whale' object has no attribute 'fly'
```

Meaning that the *Duck Typing* system allows to use any objects in any context up until it is used in a way that it does not support.

Despite being dynamically typed, Python is strongly typed [15] since the interpreter keeps track of all variable types. It's also very dynamic as it rarely uses what it knows to limit variable usage, and, as said before, if a performed operation is valid for a type, there will be no problems, though, somewhere there is the need to keep track of the variable types in order to check whether the operation is valid, and that's the reason why Python is strongly typed. For example, attempting to add numbers to strings will fail in Python (as Image nr. 9 shows), because it is not a well-defined operation. This kind of problems are easier to debug because the exception is raised precisely at the point where the error occurs, rather than at some other place.

Although Python will ever remain a dynamically typed language [16], *PEP 484* introduced type hints, which make it possible to also do static type checking of Python code. Unlike how types work in most other statically typed languages, type hints by themselves don't cause Python to enforce types. As the name says, type hints just suggest types. This implementation (always possible because of the modularity nature of Python) is found in the method *Mypy* [17], created by Jukka Lehtosalo which was then, following a suggestion by Guido van Rossum [18], rewritten to use annotations instead. Today *Mypy* is a static type checker for regular Python code.
Moreover, also some editors like PyCharm support type hints for checking types of objects [19].

3.7 Type conversions [20]

While writing Python programs, one may need to convert the used data types. For example, the sum operator can be applied to two operands, one of which has type `int` and the other `float`. Since this operation would not be allowed as the sum operator is differently defined based on the data type considered, without loose of generality, the operator of type `int` must be converted to type `float` before performing the sum.
There are two types of type conversion in Python: implicit and explicit conversions.

Implicit conversion

Implicit conversion is the type of conversion that is performed automatically by Python.
Implicit conversion applies in examples as above: if an `int` and a `float` types are summed, the `int` type is converted automatically into a `float` type before evaluating the expression, as the result of the operation should be a `float` type, since `int` is a subtype of `float`.
An example of that could be as follows:

```
result = 8.5 + 7 // 3 – 2.5
```

Since Python performs the operations accordingly to the standard order of precedence, the operator `//` has the highest precedence, so it is processed first: 7 and 3 are both of `int` type and the operator `//` is the integer division, thus, the result is the value 2 of type `int`. Addition and subtraction have the same level of precedence, so they are processed left-to-right. Hence, 2 is converted to the floating-point number 2.0 in order to be added to 8.5, resulting in 10.5. Finally, 10.5 – 2.5 is performed, resulting in the `float` type value 8.0, which is assigned to the variable result.
Moreover, if a single number is passed in the `print` function, it will be converted automatically to the type `str`, but, as already said, if an addition between an `int` type and a `str` type is performed, an error will occur.

Explicit conversion (or *casting*)

Operations that will result in a loss of precision after the conversion, must be manually (or explicitly) converted so that Python knows that we are aware of that loss.
An example of that, could be as follows:

```
result = int(3.141592)
```

The variable result will have the integer value of 3, because the conversion operation from `float` to `int` is, by default, rounded down, thus, it will discard the fractional part (unless declared differently using some functions from the `math` library).
In order to perform the addition of a string with a number, the string (or the number, based on what the final objective is) must be explicitly converted to the other type.

However, one must be careful when using implicit conversion, since Python allows to convert type like `int` or `str` to the `bool` type, possibly leading to a wrong condition in `if` statements.
An example could be as follows:

```
my_number = 3

if my_number:
    print("My number is non-zero!")
```

Image nr. 13

Since non-zero numbers are considered to be `True`, the `if` statement will be executed in this case. However, when using strings, one should be very careful, because the empty string is treated as `False`, but any other string is `True`, even `"0"` and `"False"`.

3.8 Functions and Closures

In Python, functions are first-class objects, which are defined using a `def` statement.
A function definition is an executable statement which defines a function object. The execution binds the function name in the current local namespace to a function object, which contains a reference to the current global namespace as the global namespace to be used when the function is called. It is important to remember that a function definition does not execute the function body, which will be executed only when the function gets called. Functions accept parameters, and some parameters could be passed as default for a certain function. When this happens, the function is said to have "default parameter values". For a parameter with a default value, the corresponding argument may be omitted from a call, in which case the parameter's default value is substituted. Unlike other languages, functions in Python could return more than one value simultaneously. A function call always assigns values to all parameters mentioned in the parameter list, either from position arguments, from keyword arguments, or from default values. [21]

Python deals with Closures with the `func_closure` (in Python < 3.x) or `__closure__` (in Python > 3.x) function attributes which they have the job to save the free variables. In fact, a function that does not use free variables, doesn't form a closure, though that does not mean that a function has not a `__closure__` (or a `func_closure`) attribute: every function has this attribute, but it doesn't save any content if there are no free variables. Moreover, a closure occurs when a function has access to a local variable from an enclosing scope that has finished its execution.
Consider the following code as an example:

```
def make_printer(msg):
    def printer():
        print msg
    return printer

printer = make_printer('Foo!')
printer()
```

Image nr. 14

When `make_printer` is called, a new frame is put on the stack and it contains the compiled code for the `printer` function (as a constant) and the value of `msg` (as a local). Then, it creates and returns the function. Because the function `printer` references to the `msg` variable, it is kept alive after the `make_printer` function has returned, obtaining as output: "`Foo!`"

By looking at the `func_closure` attribute of the function `printer` (which is present, as confirmed by the function `dir`):

Image nr. 15

```
>>> 'func_closure' in dir(printer)
True
>>> printer.func_closure
(<cell at 0x108154c90: str object at 0x108151de0>,)
```

Some interesting information are displayed, such as that this attribute returns a tuple of cell objects which contains some details about the variables defined in the enclosing scope.
The first element in the `func_closure` could be `None` (Python keyword to express the `null` concept) or a tuple of cells that contains bindings for the function's free variables. Let's see the attributes of that tuple object:

```
>>> dir(printer.func_closure[0])
['__class__', '__cmp__', '__delattr__', '__doc__', '__format__', '__getattribute__',
 '__hash__', '__init__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__',  '__sizeof__', '__str__', '__subclasshook__', 'cell_contents']
```

Image nr. 16

In this output, one of the attributes is `cell_contents`.

It is possible to see what `cell_contents` stores:

Image nr. 17

```
>>> printer.func_closure[0].cell_contents
'Foo!'
>>> type(printer.func_closure[0].cell_contents)
<type 'str'>
```

Thus, when the function `printer` is called, it accesses the value stored inside `cell_contents` and this is the reason why the output of the whole program is "`Foo!`". [22]

4. Advantages and Disadvantages

Of course, Python has its own pros and cons too.

Talking about the major *pros*:

Easy to use, learn and read

The syntax very similar to English and the philosophy behind this language contribute to make really simple both to learn and read Python code. Lots of programmers (including myself) started to approach the Computer Science world by learning this simple although really powerful language, and this is certainly a plus, since it persuades more people to get into the IT world.
Moreover, the same operation often requires in Python relatively fewer number of lines of code respect to other programming languages such as C, C++ or Java.

Libraries and Community

Having a modular nature, Python provides large libraries that include areas like string operations, Internet, web service tools, operating system interfaces and protocols, machine learning, data science, games, optimization, etc.
In fact, although the standard library is vast, in order to perform some more specific tasks, external libraries could be imported really easily with the Python package manage (pip).
New libraries and already built functions pop up every day on GitHub and in the Python community, making really simple to look up for some information or already written code to perform a certain task, saving lot of time to the programmer which result in a productivity boost.
Because of its versatility and the presence of several libraries, Python has become, for example, the most widely used programming language in the field of machine learning. [23]

Interpreted language

Python, being an interpreted language, can execute the code directly, one line after the other. Moreover, if there is any error, then rather than continuing with further execution, it instead reports back the error that occurred.

Easily portable

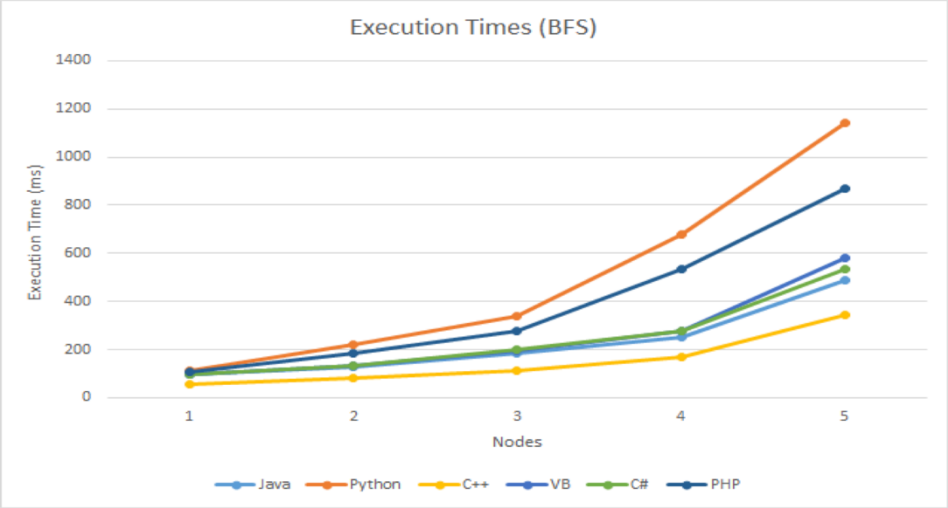
In most languages, one needs to make changes in the code to run a given program on distinct platforms. However, that is not the case with Python programming. In Python, the code could be written only once, and it will run on almost any platform, adapting the 'write once, run anywhere' motto from Java. However, the programmer needs to make sure that he/she does not involve any feature that is system-dependent.

While the major *cons* are:

Low speed

As already said, Python is a dynamically typed and interpreted language, but this means that the code is executed line-by-line, further leading to its slow execution. Python's dynamic nature is mainly the reason for its low speed since there is a requirement for some extra work during the execution process. This is one of the reasons why Python is not used when speed is a significant aspect of a given program.

Image nr. 18



Alomari et al. [24] have done a comparative study on 6 different programming languages on the same algorithm, testing different algorithms as well. This graph shows the performances in execution time on the BFS algorithm (Breadth-first search). It can be seen that Python requires always more time respect to the other languages, although this is not always true in other algorithms tested in the paper.

Inefficient memory management

To offer some simplicity to programmers, Python needs to make some trade-offs. This language uses a huge amount of memory, far beyond the efficient memory management that is performed by a language such as C, though, this makes C more complicated to read and learn, against the principles that characterize Python. This could become a problem when it comes to develop applications that requires memory optimization.

However, Python allows to easily bypass the obstacle of pure performances: it is in fact relatively simple to write an extension in C or C++ and then use it within Python, thus exploiting the high speed of a compiled language only in the parts in which it actually serves and instead leveraging the power and versatility of Python for all the rest of the software. [25][26] Of course, it is an improvement, but its performances remain lower than directly programming in those languages.

Weak in programming for mobile devices

Developers usually use Python for server-side or AI programming, rather than using it for mobile applications or client-side programming. This is because Python has slow processing power and, as just said, is hardly memory efficient when compared to other programming languages.

Runtime errors

Python's dynamic feature allows it to change a variable's data type at any time. A variable that once held an integer value, may hold a string value in the future. This may lead to runtime errors. This also means that there would be no way to guarantee that a particular piece of code would run successfully for all the different data-types scenarios simply because it had run successfully with one type. Hence, developers must perform several rounds of testing for any application developed, resulting in more time loss during this phase, compared to other programming languages.

5. Uses of Python

Thanks to its strengths, Python has been chosen to develop loads of commercial and industrial software, although nowadays it is mainly utilized in the field of machine learning, data science, web development and server maintenance. Examples of business uses, are the ERPs “Odoo” and “Tryton” [27]. Some famous applications written in Python are “Dropbox”, “OpenStack”, “OpenLP”, “Calibre” and others [28]. “Instagram” backend is written in Python [29], “NASA” uses Python to implement a CAD/CAE/PDM repository [30], “Reddit” was completely rewritten in Python in 2005 [31] and also “YouTube” uses Python “to produce maintainable features in record times, with a minimum of developers” [32]. In the field of videogames, “Battlefield 2” and “The Sims 4” uses Python [28]. To see a more complete list of software written in Python, refer to [28].

6. Conclusion

Although Python is not the best language to choose if performances are the main requirements for a program, its simplicity, modularity nature and the frequently updates that always introduce new features in order to keep Python up to date with the recent papers provided by the research community (especially in the AI field) have contributed to make it one of the most popular languages of the last years [33]. Moreover, the importance that neural networks and data science are covering in this historical period, but above all they will in the future, will contribute to spreading it even more, increasing the research possibilities in the more general field of artificial intelligence.

7. Bibliography

[1] URL: https://en.wikipedia.org/wiki/History_of_Python
[2] URL: https://www.python.org/dev/peps/pep-0020/
[3] URL: https://en.wikipedia.org/wiki/Python_(programming_language)
[4] URL: https://docs.python.org/3/extending/extending.html#reference-counts
[5] URL: https://docs.python.org/3/reference/executionmodel.html
[6] Smeding, Gideon. (2009). An executable operational semantics for Python. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.624.7392&rep=rep1&type=pdf
[7] URL: https://docs.python.org/3/reference/lexical_analysis.html#indentation
[8] URL: https://docs.python.org/3/reference/lexical_analysis.html#operators
[9] URL: https://docs.python.org/3/reference/lexical_analysis.html#keywords
[10] URL: https://docs.python.org/3/reference/datamodel.html#objects-values-and-types
[11] URL: https://docs.python.org/3/library/stdtypes.html
[12] URL: https://en.wikipedia.org/wiki/IEEE_754-2008_revision
[13] URL: https://en.wikipedia.org/wiki/Duck_typing
[14] URL: https://docs.python.org/3/glossary.html#term-duck-typing
[15] URL:https://wiki.python.org/moin/Why%20is%20Python%20a%20dynamic%20language%20and%20also%20a%20strongly%20typed%20language
[16] URL: https://www.python.org/dev/peps/pep-0484/#non-goals
[17] URL: http://mypy-lang.org
[18] URL: https://twitter.com/gvanrossum/status/700741601966985216
[19] URL: https://www.jetbrains.com/help/pycharm/type-hinting-in-product.html
[20] URL: https://python-textbok.readthedocs.io/en/1.0/Variables_and_Scope.html
[21] URL: https://docs.python.org/3/reference/compound_stmts.html#function-definitions
[22] URL: https://stackoverflow.com/questions/4020419/why-arent-python-nested-functions-called-closures
[23] URL: https://in.springboard.com/blog/best-language-for-machine-learning/
[24] Alomari, Zakaria & Halimi, Oualid & Sivaprasad, Kaushik & Pandit, Chitrang. (2015). Comparative Studies of Six Programming Languages. URL: https://arxiv.org/pdf/1504.00693.pdf
[25] URL: https://docs.python.org/3/extending/extending.html
[26] URL: https://realpython.com/python-bindings-overview/
[27] URL: https://www.python.org/about/apps/
[28] URL: https://en.wikipedia.org/wiki/List_of_Python_software
[29] URL: https://www.fastcompany.com/3047642/do-the-simple-thing-first-the-engineering-behind-instagram
[30] URL: https://code.nasa.gov/?q=python
[31] URL: https://redditblog.com/2005/12/05/on-lisp/
[32] URL: https://www.python.org/about/quotes/
[33] YouTube video. URL: https://www.youtube.com/watch?v=Og847HVwRSI