Learn how to talk parseltongue

Workshop Python3

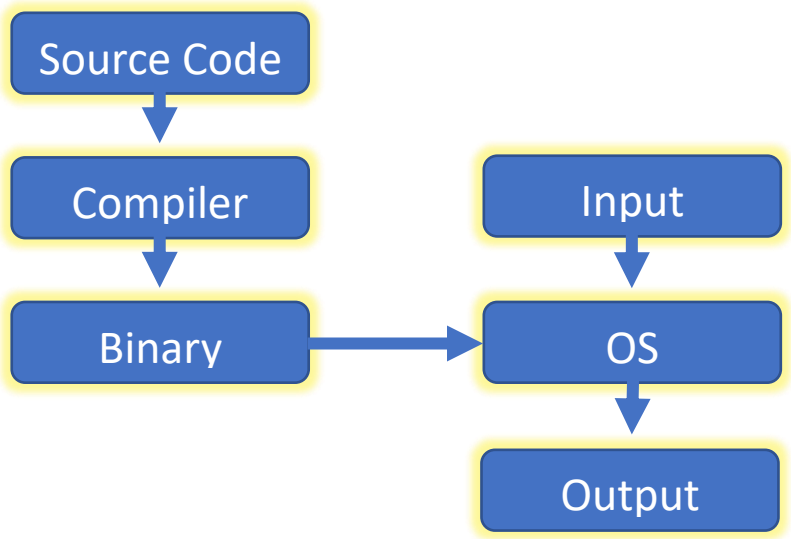
Corentin COUTRET-ROZET

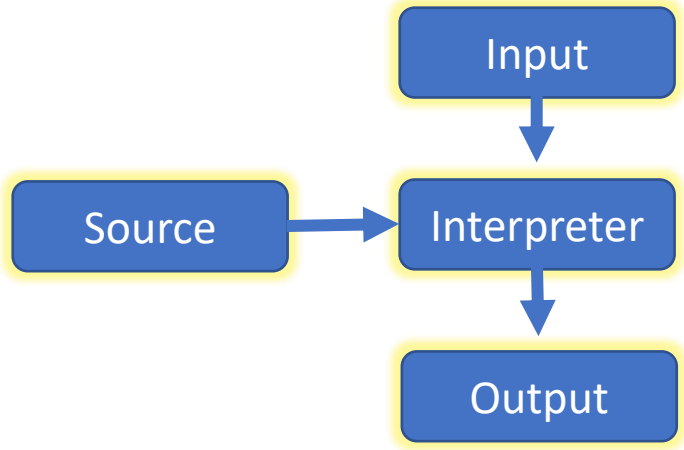
Contents

An interpreted language

Compiled languages

In that kind of language, source code is firstly compiled with what is called a **compiler** to create a **binary**, only readable from machines. At this point, the OS (**Operating system**) of machines can use the binary to compute output from a given input.





Interpreted languages

Interpreted language can directly compute an **output** from the **source code** using what is called an **interpreter**. "Interpretation" of the code is processed *step-by-step*: the interpreter executes line of codes one after another.

Main differences

Using an interpreted language, the source code is directly usable, while using a compiled a language, you will have to compile the binary, which is not always an easy thing to do. Otherwise, the binary of compiled language is directly read and executed by the machine, which lead to a stronger and a fastest execution.

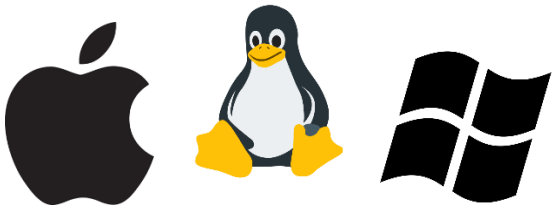
A multi-tools language

Multi-platform

In computing, **cross-platform software** (also **multi-platform software**) is computer software that is implemented on multiple computing platforms. Cross-platform software may be divided into two types:

- One requires **individual building** or compilation for each platform that it supports
- The other one can be **directly run on any platform** without special preparation, e.g., software written in an interpreted language or pre-compiled portable bytecode for which the interpreters or run-time packages are common or standard components of all platforms.

As an example, Python is a cross-platform language than can run on Microsoft Windows, Linux, and macOS.



➢ Source : https://en.wikipedia.org/wiki/Cross-platform_software

Multi-paradigm

A paradigm, as a simple definition in the scientific context, would be the set of rules accepted and internalized as norms by the scientific community. As a language, a paradigm is so, the set of rules, syntaxes, usage, that permit to classify languages. Python together two of all the paradigms, which are the Object-Oriented paradigm and the imperative paradigm.

Object-oriented programming (OOP) is based on the concept of "**objects**", which can contain data, in the form of fields (often known as **attributes** or *properties*), and code, in the form of procedures (often known as **methods**). In OOP, computer programs are designed by making them out of objects that interact with one another. OOP languages are diverse, but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.

➢ Source : https://en.wikipedia.org/wiki/Object-oriented_programming

What is about **imperative programming**. It uses statements that change a program's state. In much the same way that the imperative mood in natural languages expresses commands, an **imperative program consists of commands for the computer to perform**. Imperative programming focuses on describing *how* a program operates.

➢ Source : https://en.wikipedia.org/wiki/Imperative_programming

1st step with Python

Variables

Here are some interesting things to know about how Python manage its variables:

- Implicit types: no need to declare the type while creating a variable

```
a = 0
b = "Hello world!"
c = [a,]
```

- Easy conversion: python is made to allow easy conversions from a variable's type to another

```
a = 123
b = str("123")  # b == "123"
c = list(a)     # c == ['1', '2', '3']
```

- None and Boolean: as most of languages

```
a = None
b = True
c = False
```

- String and formatting

```
a = 1
b = 12.3
c = "Let's play!"
d = "Player {}: score = {:.2f}\n{}".format(a, b, c)
```

- Tuple: an immutable sequence of Python objects

```
a = 1
b = "str"
c = (a, b)
```

- List: as most of languages

```
a = 1
b = "str"
c = [a, b,]
d = [a, b, c,]
```

- Dictionary: a powerful tool that take a value and a corresponding key

```
a = {1: 1, 2: "two", "third_key": 3}
# a[2] == "two"
# a["third_key"] == 3
```

To note: pointers are implicit in Python

Functions and control flow

Here an example on how to declare a **function**.

```
def NOM(PARAM1, PARAM2):
        # CONTENT
        return RESULT

NOM(VALUE1, VALUE2)
```

Pro-tips: It is allowed to declare function insides functions and can be really useful

Here an example on **control flow**.

```
if A and (B or C):
        # CONTENT
elif not D or D is None:
        # CONTENT
elif E in ('-h', '--help'):
        #CONTENT
else:
        # CONTENT
```

Here an example on **loops** syntax.

```
for i in range(len(tab)):
        print(tab[i])

for element in tab:
        print(element)
```

```
i = 0
size = len(tab)
while i < size:
        print(tab[i])
        i += 1
```

Structures and classes

```
class Plop:
        alpha = 1
        beta = "plop"
        gamma = {"a": "Hey", "b": None}

p = Plop()
p.alpha += 3

print(Plop.gamma)
```

As a reminder:

- `alpla` is called an **attribute**
- `function` is called a **method** of the Plop class

```
class Plop:
        def __init__(self, value):
                self._value = value

        def public(self):
                self.__private()

        def __private(self):
                print(self._value)

class Plip(Plop):
        def __init__(self, value):
                Plop.__init__(self, value)

p = Plip()
p.public()
```

Try and exceptions

Here is an example of how to use **try** catches and **exceptions**.

```
try:
        # ACTION
except EXCEPTION_POSSIBLE:
        # ACTION SI KO
else:
        # ACTION SI OK
# SUITE

raise NOM_EXCEPTION
```

There is actually a lot of **exceptions** and it's possible to create your own using the **raise** built in.

- https://docs.python.org/3/library/exceptions.html

Built-ins

Built-ins are the real power of Python. As a reminder, built-ins are functions already built in python library. There is in Python3 a long list of built-ins. Knowledge come with experience…

- https://docs.python.org/3/library/functions.html

Some other links…

- Tutorial: https://www.tutorialspoint.com/python/
- Coding Style: https://pypi.org/project/pep8/