Bachelor 1 - Python

Mathieu Fourré, Paul Lege

Bachelor Cyber EPITA

Python

Have you aready used Python ?

What

Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation. [@Wikipedia]

Caracteristics

- Interpreted
- High-level
- Dynamically typed
- Garbage-collected
- Object-Oriented

Questions ?

Who

- Guido van Rossum (born 31 January 1956)
- Dutch programmer, creator of the Python programming language
- “benevolent dictator for life” (BDFL) until 2018

photo

When

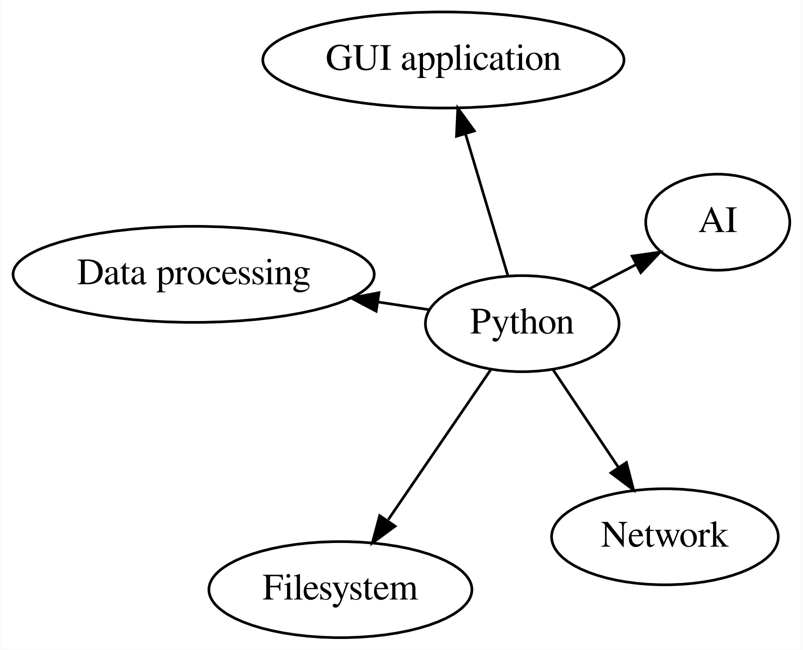
- 1991: Python 0.9.0 (first release)
- 2000: Python 2.0
- 2008: Python 3.0
- 2020: Python 2.7.18 (last release of Python 2)
- 24-08-2023: Python 3.11.5

Why

Why should you care about learning Python ?

- Easy syntax
- Fast iteration/prototyping
- Rich standard libray
- Richer community of library (PyPi)

Best tool to automate tasks and test ideas. [@me]


GUI application
AI
Data processing
Python
Network
Filesystem

Questions ?

Python basics

Running Python

```
$ python

$ python3

$ python3.11
```

Running Python

```
Python 3.11.3 (main, Jun  5 2023, 09:32:32) [GCC 13.1.1 20230429] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hello world")
hello world
```

Interactive mode

REPL

Read–eval–print loop

```
>>> a = 5
>>> print(a)
5
```

Programming basics

Variable

Name associated to some data.

```
# this is the first comment
foo = 5 # second comment
bar = "this is a string"
baz = "#not a comment because in quotes"
```

Type

Every data has a `type`.

Python has many built-in types:

- int (`0, -12, 50`)
- float (`2.0, 0.0, 50.3`)
- string (`"abc", "", "hello !"`)
- boolean (`True, False`)
- ...

```
>>> a = 10
>>> type(a)
<class 'int'>
>>> a = "hello"
>>> type(a)
<class 'str'>
```

Warning

Data has a `type`, however variables don't. You can store any type of data in a variable. This is not the case in many programming langages.

Questions ?

Operations

```
>>> a = 1
>>> b = 3
>>> a + b
4
```

```
+:   add
-:   substract
/:   divide (returns a float)
//:  floor divide (returns an int)
*:   multiply
%:   modulus
and: logical and
or:  logical or
```

Comparaison

`==:`	`equality`
`!=:`	`not-equality`
`>:`	`superior`
`<:`	`inferior`
`>=:`	`superior-equal`
`<=:`	`inferior-equal`

Python as a calulator

You can use the REPL as a calculator

```
>>> width = 10
>>> height = 2 * 10
>>> width * height
200
```

Errors

Your first error.

```
>>> n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Control structures

If

```
>>> age = 21
>>> if age >= 18:
...     print("is an adult")
...
is an adult
```

Else

```
>>> if False:
...     print("never printed")
... else:
...     print("printed")
...
printed
```

Elif

```
>>> if False:
...     print("never printed")
... elif True:
...     print("printed")
... else:
...     print("not printed")
...
printed
```

Tips

You can use as many `elif` as you want.

Examples

```
age=21
if age < 13:
    print("child")
elif age < 20:
    print("teen")
else:
    print("old")
```

Functions

Functions

In order to easily reuse your code, you will use `functions`.

Definition

To define a new function, you need to use the keyword `def`.

example.py

```
def my_func(num: int) -> int:
    num = num * 2 + 1
    return num
```

This function is named `my_func`, takes one argument `num` and returns `num * 2 + 1`.

Calling

To use this function in your code, you will then write:

func_call.py

```
my_func(5)
```

To see, the result, you can then print it.

example.py

```
result = my_func(5)
print(result)
```

This will print

```
42sh$ ./example.py
11
```

Loops

Why

What if you want to display all the number from 1 to 100 included ?

While

Execute the statement `while` the condition is `True`.

example.py

```
i = 1
while i <= 100:
    print(i)
    i = i+1 # i += 1
```

```
42sh$ ./example.py
1
2
3
[...]
99
100
```

Questions ?

More advanced types

Many values

Sometimes, you need to store multiple values in a single variable, or return multiple values.

Python has many types containing multiple values.

Tuple

Tuple

- Imutable
- Fixed number of element

To create a `tuple`, separate values or variable with a comma `,`. Usually, it is surrounded by parenthesis.

example.py

```
msg = "Hello"
t = 1, 2
v = msg, True, 1.5
print(t)
print(v)
print(type(v))
```

```
42sh$ ./example.py
(1, 2)
('Hello', True, 1.5)
<class 'tuple'>
```

Access elements of a tuple

To access a value of a tuple, you can use the `[]`.

example.py

```python
t = (1, "hello")
print(t)
print(t[0]) # /!\ index starts at zero
print(t[1])
```

```
42sh$ ./example.py
(1, 'hello')
1
hello
```

Multiple return value

You may want to write a function, that returns multiple values. This is a good use case of tuples.

example.py

```python
import math
from typing import Tuple

def sqrt_and_sqare(num: int) -> Tuple[float, int]:
    return math.sqrt(num), num**2

print(sqrt_and_sqare(4))
```

Multiple return value

You may want to write a function, that returns multiple values. This is a good use case of tuples.

example.py

```python
import math
from typing import Tuple

def sqrt_and_sqare(num: int) -> Tuple[float, int]:
    return math.sqrt(num), num**2

print(sqrt_and_sqare(4))
```

```
42sh$ ./example.py
(2.0, 16)
```

Questions ?

Tuple limitations

- Tuples are very useful but limited in what they can do.

Tuple limitations

- Tuples are very useful but limited in what they can do.
- You cannot modify what in in a tuple

Tuple limitations

- Tuples are very useful but limited in what they can do.
- You cannot modify what in in a tuple
- You cannot add/remove an element in a tuple

Python REPL

```
>>> t = (1, 3)
>>> t[1] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Lists

Lists

- Variable number of elements
- You can modify an elements
- You can add/remove elements

List example

The syntax to create a list is `[]`. Elements are then separated with commas (`,`).

example.py

```python
l = [1, 2, "hello", 5.6]
```

Access and modification

To access a value, the syntax is the same as the tuple.

example.py

```python
l = [1, 2, "hello", 5.6]
print(l[1])
l[2] = "world"
print(l)
```

```
42sh$ ./example.py
2
[1, 2, 'world', 5.6]
```

Methods

Methods

In Python, a list is an `object`. Objects may have `methods`. A method is a special function associated to the object, which usually modifies the object.

To call a method use `<object>.<method>(<arguments>)`

Python REPL

```
>>> l = []
>>> l.append(5) # here, the method is called `append`
>>> print(l)
[5]
```

Append() / Pop()

To find the list of methodes defined for lists, you can read the official documentation. link_to_doc

Here are some of them.

```
l = []
l.append(5) # add 5 at the end of the list
l.pop() # remove last element and returns it
l.insert(i, 7) # insert 7 at the index i
l.remove(5) # search 5 and remove it from the list
```

Questions ?

For loop

For

You have seen how `while` works. There is another way to do a loop: `for`.

`for` does not work in the same way. It will run for each element in the iterable given (ex: `tuple`, `list` and more !).

Example For

example.py

```python
l = [1, 2, "hello", 5.6]

for x in l:
    print(x)
```

```
42sh$ ./example.py
1
2
hello
5.6
```

Example For (Tuple)

example.py

```
l = (1, 2, "hello", 5.6)

for x in l:
    print(x)
```

```
42sh$ ./example.py
1
2
hello
5.6
```

Example application

We want to sum all elements of a list.

example.py

```python
l = [1, 2, 3, 4, 5]
res = 0

for x in l:
    res = res + x

print(res)
```

```
42sh$ ./example.py
15
```

Questions ?

Common pattern

In programming, you will often want to do something `n` times.

Until now, you have done:

example.py

```python
i = 0

while i < n:
    # actual work
    i = i + 1
```

Range

Range

In python you can use the function `range`.

example.py

```python
for i in range(n):
    # actual work
```

Range example

example.py

```python
for x in range(4):
    print(x)
```

```
42sh$ ./example.py
0
1
2
3
```

Range options

By default, `range` goes from `0` to `n` with step of `1`. However you can change this by adding arguments.

Documentation

```
# range(end)
range(5)        # 0 1 2 3 4
# range(start, end)
range(1, 5)     # 1 2 3 4
# range(start, end, step)
range(1, 5, 2)  # 1 3
range(5, 0, -1) # 5 4 3 2 1
```

Range warning

Warning !

`range(...)` IS NOT A LIST. You cannot do assignement:
`range(5)[2] = 4` is an error.

You may think of it as a special tuple.

Python REPL

```
>>> r = range(5)
>>> r[2] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'range' object does not support item assignment
```

Recursivity

Why ?

Some algorithms are way simpler to implement that way.

What ?

To understand recursivity, you have to understand recursivity.

Example

A recusive function is a function calling itself.

example.py

```python
def my_rec_fun(n: int) -> int:
    if n <= 0: # stoping condition
        return 0
    return n + my_rec_fun(n-1) # recusive call

print(my_rec_fun(5))
```

Example

A recusive function is a function calling itself.

example.py

```
def my_rec_fun(n: int) -> int:
    if n <= 0: # stoping condition
        return 0
    return n + my_rec_fun(n-1) # recusive call

print(my_rec_fun(5))
```

```
42sh$ ./example.py
15
```

Explanation

- `my_rec_fun(5)`
- `5 + my_rec_fun(5-1)`
- `5 + 4 + my_rec_fun(4-1)`
- `5 + 4 + 3 + my_rec_fun(3-1)`
- `5 + 4 + 3 + 2 + my_rec_fun(2-1)`
- `5 + 4 + 3 + 2 + 1 + my_rec_fun(1-1)`
- `5 + 4 + 3 + 2 + 1 + 0`
- `15`

Questions ?

Dictionnary

Lists are cool but. . .

The only way to access a value in a `list` is by knowing its index.

Sometimes, this is clunky.

Example

You have to store the birthdays of a list of personne.

```
bob -> 01/01/2001
jake -> 02/02/2003
ben -> 17/03/2005
```

You want to efficiently get the birthday when given a name.

Naive implementation

A naive way of doing this would be:

example.py

```
def find_bday(name: str):
    bdays = [
        ("bob", "01/01/2001"),
        ("jake", "02/02/2003"),
        ("ben", "17/03/2005"),
    ]
    for x in bdays:
        if x[0] == name:
            return x[1]
    return "not found"
```

The solution

This situation, where you have data associated with a `key`, is common.

This often implemented with a `hashmap` in Python, this type is `dict`.

You can create it with `{}`:

example.py

```
{"bob": "01/01/2001", "jake": "02/02/2003",
 "ben": "17/03/2005"}
```

Access a value

A dictionnary is a set of key-value pairs.

```
{"key": "value", "key2": "value2"}
```

To get the values associated with a key, you use the [] syntax.

Python REPL

```
>>> d = {"bob": "01/01/2001", "jake": "02/02/2003",
 "ben": "17/03/2005"}
>>> print(d["bob"])

01/01/2001
```

Types

In a dictionnary, values can be of any type and key must be `immutable`.

For example:

- `dict[int, str]`
- `dict[str, list]`
- `dict[tuple, dict]`
- `dict[bool, tuple]`

Can not be:

- `dict[list, str]` **# list is not immutable, doesn't work**

Classes

You remember `methods` ?

You remember `methods` ?

What if you could create **your own methods** !!!

. . . on your own object !!!

What is a class

Objects are a way to bundle together data and actions that can be performed on the data. They allow programmers to abstract concepts and provide a conventient way to represent real-world entities.

In order to define an object, we need to write a class. You can view it as a blueprint that defines what variables objects of this class contain (we call them attributes), and what functions they have (we call them methods).

Example

class.py

```python
class Person:
    def __init__(self, name: str):
        self.name = name

    def say_hi(self):
        print("Hello my name is", self.name)

    def rename(self, new_name: str):
        self.name = new_name
```

Methods and attribute

This class has one attribute called `name` and three methods: `say_hi` and `rename` which are regular methods and `__init__`, the constructor.

Tips

The constructor is a special method that is called when creating a new object from the class. Its role is to set up initial values for the attributes

Notice that the methods take at the very least `self` in their parameters, which refers to the `object` being constructed or manipulated. Both the constructor (`__init__`) and `rename` accept an additional argument and set the value of the attribute name to it.

Usage

Once the class is defined, we can **instantiate** `Person` objects.

instanciate.py

```
>>> dupond = Person("Dupond")
>>> dupont = Person("Dupont")
>>> dupond.say_hi()
Hello my name is Dupond
>>> dupont.say_hi()
Hello my name is Dupont
>>> dupond.rename("Dupong")
>>> dupond.say_hi()
Hello my name is Dupong
```

Explaination

Both `dupond` and `dupont` are objets created from the `Person` class. They are called instances of `Person`. We call the methods `say_hi` and `rename` on these objects. Note that the output of `say_hi` indeed depends on the value of the attribute `name`.