Introduction to data-science with Python

Nicolas Jouvin

M2 data science Évry

2022

Outline

What this course is not about

Python ?


What this course is not about

Data science ?

Data science ?

Singularity

What this course is not about

Data science ?

Singularity

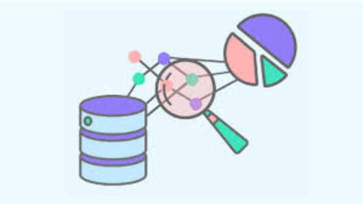
Shitty visualization

What this course is not about

Data science ?

Singularity

Shitty visualization



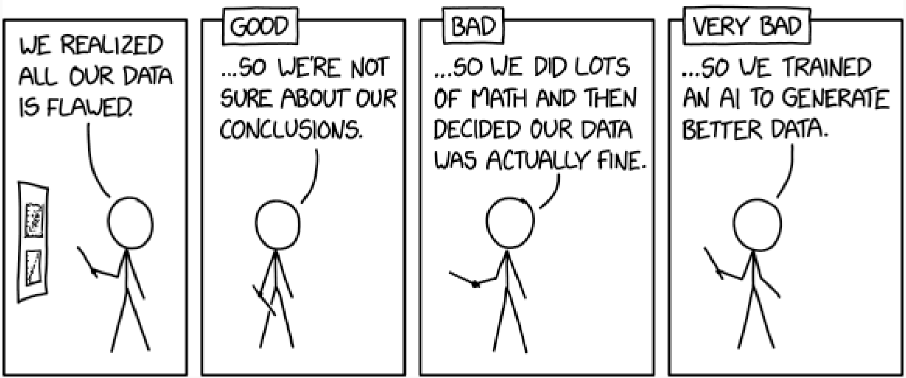
WTF Google Image ?!

```python
import numpy as np

def predict(X, y):
    """
    Documentation
    """
    # Do something clever

res = predict(X, y)
res.plot()
```

Definition

Machine learning = statistics + scientific programming.

XKCD[1] always has the final words...



[1] https://xkcd.com/

Organization

Course dates

Teaching material On https://github.com/nicolasJouvin/introduction_python

4 sessions, 3h each. Each session is a mix of slides + practical session (+ breaks)

Important Form groups if you need

- 2 student/group max ideally
- one machine per group (minimum)
- heterogeneous levels in Python (collaboration)

The dates are on the master's agenda

Evaluation Final exam **on machine** Check the course's website for news about this.

Practicalities

Software requirement

Mandatory For next session every machine should at least have

- Python > 3.8 : if possible, use the Anaconda distribution
- A dedicated **Python environment**
  1. `$pip install -r requirement.txt` executed
  2. `$jupyter notebook` running in your default browser

Advised

- Python IDE: VSCode or PyCharm Community (free)
- Git (not covered during this course)

Who is familiar with Python programming ?
(not necessarily Machine Learning)

Who is familiar with Python programming ?
(not necessarily Machine Learning)

Today's program Python syntax, data structure and types

Basics of Python

Python is...

High-level (like R or Matlab), *i.e.* not a compiled language (C/C++)
Question: good or bad ?

Python is...

High-level (like R or Matlab), *i.e.* not a compiled language (C/C++)
Question: good or bad ? **it depends**

- Pros: easy to learn, faster to deploy
- Cons: naive implementations can lead to slow computations

Python is...

High-level (like R or Matlab), *i.e.* not a compiled language (C/C++)
Question: good or bad ? **it depends**

- Pros: easy to learn, faster to deploy
- Cons: naive implementations can lead to slow computations

Versatile not dedicated to statistics/machine learning but many scientific libraries

- Numpy (matrix)
- Pandas (data manipulation)
- Matplotlib (plotting/visualization)
- Scikit-learn, tensorflow, pytorch, etc.

Python is...

High-level (like R or Matlab), *i.e.* not a compiled language (C/C++)
Question: good or bad ? **it depends**

- Pros: easy to learn, faster to deploy
- Cons: naive implementations can lead to slow computations

Versatile not dedicated to statistics/machine learning but many scientific libraries

- Numpy (matrix)
- Pandas (data manipulation)
- Matplotlib (plotting/visualization)
- Scikit-learn, tensorflow, pytorch, etc.

Trendy nowadays

Our first Hello World !

Open some text editor (VSCode/PyCharm are better)

Save a new file as `helloworld.py` with the following lines

```python
# This is a comment
print('Hello World !')
```

Open

- Windows Conda bash
- Linux or MacOS bash

Type `$python helloworld.py`

The Python interpreter

No magic Python is installed somewhere on your machine

```
$python --version
```

You can even have co-existing versions of Python (environments)

- Pros: flexibility on versions, manage clean project
- Cons: source of errors! ⟶ always know which `python` you are using!

The Python interpreter

No magic Python is installed somewhere on your machine

```
$python --version
```

You can even have co-existing versions of Python (environments)

- Pros: flexibility on versions, manage clean project
- Cons: source of errors! $\longrightarrow$ always know which `python` you are using!

Homework

Create an environment named "M2Evry" with Python 3.9

Hint: follow this tutorial

Interactive mode

In this course we will alternate between three ways of using Python

1. **Command line** (Bash `$python -options my_script.py`)

Interactive mode

In this course we will alternate between three ways of using Python

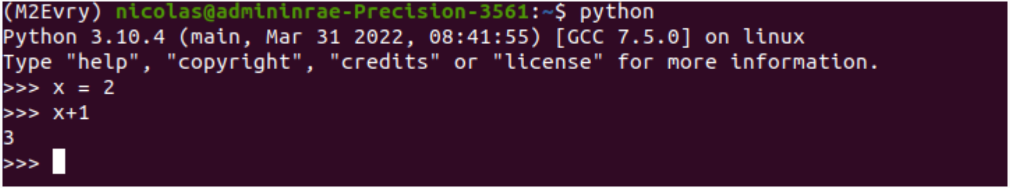
1. **Command line** (Bash `$python -options my_script.py`)
2. **Python Shell**

```
(M2Evry) nicolas@admininrae-Precision-3561:~$ python
Python 3.10.4 (main, Mar 31 2022, 08:41:55) [GCC 7.5.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 2
>>> x+1
3
>>> 
```

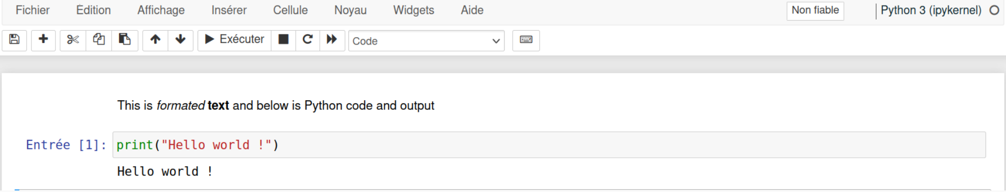
Interactive mode

In this course we will alternate between three ways of using Python

1. **Command line** (Bash `$python -options my_script.py`)
2. **Python Shell**



3. **Jupyter notebooks**: mix of formated text and Python code (∼ Rmarkdown `.Rmd`)



Frequent types and data structure

Objects, types and variables

Objects

- Python manipulates **objects**
- each object has a **type**
  - specify the possible values
  - specify the possible operations
- example
  - `1` is an `int`
  - `1.3` is a `float`
  - `"abcd"` is a `str`
  - `False` is a `bool`

Variables

- objects can be named
- a variable is a name for an object
- Affectation = setting/binding a name

  ```
  variable = object
  ```

- Names are replaced by the object in expressions

  ```
  x = 2
  2 * x
  ```

Difference between `int` and `float`

```
x = 2
print(type(x))
x_f = 2.0
print(type(x_f))
```

Maths

Difference between `int` and `float`

```
x = 2
print(type(x))
x_f = 2.0
print(type(x_f))
```

Python knows how to add, multiply, exponentiate

```
y = 3
print(x)
print(x+y)
print(x*y)
print(x ** y)
```

Maths

Difference between `int` and `float`

```
x = 2
print(type(x))
x_f = 2.0
print(type(x_f))
```

Python knows how to add, multiply, exponentiate

```
y = 3
print(x)
print(x+y)
print(x*y)
print(x ** y)
```

Question: what is the expected type of `2.0 * 2` ?

Booleans

Three keywords: `True`, `False` and `not`

```
x = True
y = not x
print(x, y)
print(int(x), int(y))
```

`If... Else` statements

```
if x:
    print("I'm True")
else:
    print("I'm False")
```

Logical conditions

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

Output is a float

Exercise

Write a program that prints the maximum of two floats a and b

Lists

Lists are collection of objects with different types $\longrightarrow$ **ubiquitous** in Python

```
l = [1, 2.3, 'a']
print(l)
print(type(l)) # a list
print(len(l))  # number of elts
print(l[0])    # first elt
```

Indexation starts at 0 !!

```
# Some really un-pythonic way to browse a list...
for i in range(len(l)):
    print(i, l[i])
```

Lists (cont'd)

- `[content]` is the litteral value, and `[]` is the empty list
- `n=len(l)` is the length, indexes vary in $0, \ldots, n-1$
- `l[-i]` access the i-th element starting from the **end**
- Lists are iterable objects

Lists (cont'd)

- `[content]` is the litteral value, and `[]` is the empty list
- `n=len(l)` is the length, indexes vary in $0, \dots, n-1$
- `l[-i]` access the i-th element starting from the **end**
- Lists are iterable objects

```python
# only run through elements
for element in l:
    print(element)
```

```python
# keep the index information
for idx, elt in enumerate(l):
    print(idx, elt)
```

Lists (cont'd)

- `[content]` is the litteral value, and `[]` is the empty list
- `n=len(l)` is the length, indexes vary in $0, \dots, n-1$
- `l[-i]` access the i-th element starting from the **end**
- Lists are iterable objects

```
# only run through elements
for element in l:
    print(element)
```

```
# keep the index information
for idx, elt in enumerate(l):
    print(idx, elt)
```

Exercise

Let `l = [-1, 3, -2.3, 7.6, 0.6]`. Write a program creating a list with

1. the non-negative values of `l`
2. their indexes

Hint: Use the `append()` method $\longrightarrow$ `<list>.append(<something>)`

Lists (cont'd 2)

Lists are objects[2] in Python $\longrightarrow$ several *methods* exists

- `append(x)` create a slot at the end of the list and add `x`.
- `insert(i, x)` insert `x` at position `i`
- `pop(i)` delete the i-th element and returns it.
- `count(x)` number of occurences of `x` in the list.
- `reverse()` reverse the order of elements.
- `extend(l2)` add `l2` in the end of the calling list
- And more...

[2] Python use a dot to link objects and methods: `object.method()`

Lists (cont'd 2)

Lists are objects[2] in Python $\longrightarrow$ several *methods* exists

- `append(x)` create a slot at the end of the list and add `x`.
- `insert(i, x)` insert `x` at position `i`
- `pop(i)` delete the i-th element and returns it.
- `count(x)` number of occurences of `x` in the list.
- `reverse()` reverse the order of elements.
- `extend(l2)` add `l2` in the end of the calling list
- And more...

Exercise

Reverse the order of element of `l` without using `l.reverse()`

[2] Python use a dot to link objects and methods: `object.method()`

Slices

Some Python objects, including lists, can be indexed with `[]` and sliced with `:`

- numbering always starts at 0
- negative ordering $\leftrightarrow$ reverse ordering
- `l[i:j]`
  1. from `i` to `j-1`
  2. missing `i` means a start at 0
  3. missing `j` means an end at `len(l)-1`

```
l[:3]        # compare to l[0:3]
l[2:len(l)] # compare to l[2:len(l)]
l[2:4]
```

Comprehension

List comprehension is the combination of `for` loop with `list` syntax

- great way to make your code more *Pythonic*
- apply to other data structure such as dictionaries (cf. later)

Comprehension

List comprehension is the combination of `for` loop with `list` syntax

- great way to make your code more *Pythonic*
- apply to other data structure such as dictionaries (cf. later)

General syntax: `[expression for variable in iterable]`

The three following lines are thus equivalent

```
l = [-2, -1, 0, 1, 2]
l = [elt for elt in range(-2, 3)]
l = list(range(-2, 3))

pos_l = [x for x in l if x > 0] # you can even add conditions !
```

Comprehension

List comprehension is the combination of `for` loop with `list` syntax

- great way to make your code more *Pythonic*
- apply to other data structure such as dictionaries (cf. later)

General syntax: `[expression for variable in iterable]`

The three following lines are thus equivalent

```
l = [-2, -1, 0, 1, 2]
l = [elt for elt in range(-2, 3)]
l = list(range(-2, 3))

pos_l = [x for x in l if x > 0] # you can even add conditions !
```

Exercise

Use list comprehension to return the index of positive elements in `l`

Hint: `enumerate` returns an iterable

Nested comprehension

We can chain **for** inside list comprehension $\longleftrightarrow$ nested **for** loops

Example:

```
l = [i + j for i in range(3) for j in [2,5,10]]
```

is equivalent to

```
l = []
for i in range(3):
    for j in [2, 5, 10]:
        l.append(i+j)
```

Nested comprehension

We can chain **for** inside list comprehension $\longleftrightarrow$ nested **for** loops

Example:

```python
l = [i + j for i in range(3) for j in [2,5,10]]
```

is equivalent to

```python
l = []
for i in range(3):
    for j in [2, 5, 10]:
        l.append(i+j)
```

Exercise

Rewrite as nested loops

- `[x - y for x in range(4) for y in range(x + 2) if x != y]`
- `[[x / 2 for x in range(y)] for y in range(2, 5)]`

Be careful: references & copies

In Python `variable = object`, remember ?

Be careful: references & copies

In Python `variable = object`, remember ?

However, Python manipulates `object` by reference

- The content of the variable is the **adress** of the object in memory
- It's like an ID or a phone number!

Be careful: references & copies

In Python `variable = object`, remember ?

However, Python manipulates `object` by reference

- The content of the variable is the **adress** of the object in memory
- It's like an ID or a phone number!

Warning: it's a VERY frequent type of mistakes

```
l1 = [1, 2] # l1 stores the adress the list containing [1, 2]
l2 = l1     # This adress is copied in l2
l2[0] = 8   # The list at the adress of l2 is modified
print(l1)   # Hence, l1 is modified as well
```

Workaround: use `l2 = l1.copy()` to create a new list

Strings

Litteral: `s = "A String"` or `p = "another string"` / Type is `str`(string)
Strings are *iterable* and behave like lists

```
for char in s:
    print(char)
```

```
print(len(s))
print(s[0])
```

```
s + s
3 * s
```

Lots of methods for `str`

- `upper()` upper cases whole string: `'Help!'.upper()`
- `lower()` upper cases whole string: `'Help!'.lower()`
- Many more...

Comprehension `[char for char in s+p if char.isupper()]`

Exercise

Create a sub-string only only from alphabet letter of `s="No57$9i74s0:!y"`
Hint: Use `''.join(list)` and `.isalpha()`

Dictionaries

Dictionaries are basically lists indexed by a `key`

```
d = {'mykey': 1, 'foo': 2.3, 'bar': 'a'} # dict
```

Try `d[0]`, what happens ?

Dictionaries

Dictionaries are basically lists indexed by a `key`

```
d = {'mykey': 1, 'foo': 2.3, 'bar': 'a'} # dict
```

Try `d[0]`, what happens ?

Important: dictionaries are indexed **keys** not integers: `d['bar']`

Built-in methods returning an `iterable`

- Keys: `d.keys()`
- Values: `d.values()`
- Both: `d.items()`

Exercise

Write a program that prints the keys and values of a dictionary.

Dict comprehension

```
for key, val in d.items():
    print(d[key] == val)
```

General syntax: `{key:expression for variable in iterable}`

The two following lines are thus equivalent

```
d = {'1':0, '2':1, '3':2}
d = {str(i+1):i for i in range(3)}

d2 = {key:val for key, val in d.items() if val >=1} # conditions
```

```
for key, val in d.items():
    print(d[key] == val)
```

General syntax: `{key:expression for variable in iterable}`

The two following lines are thus equivalent

```
d = {'1':0, '2':1, '3':2}
d = {str(i+1):i for i in range(3)}

d2 = {key:val for key, val in d.items() if val >=1} # conditions
```

Exercise

Let `d = {str(i+1):i/2 for i in range(10)}`.

Use list comprehension to create a new dict with only integer values of `d`
Hint: Use `isinstance(val, int)` to test if `val` is an `int`

Tuples

Tuples are non-modifiable lists

```
t = (1, 2.3, 'a') # tuple
```

Try `t[0] = 2`, what happens ?

Important: tuples are **immutable**, they cannot me modified.

Tuples

Tuples are non-modifiable lists

```
t = (1, 2.3, 'a') # tuple
```

Try `t[0] = 2`, what happens ?

Important: tuples are **immutable**, they cannot me modified.

Packing & unpacking: the Pythonic way

```
t = 1, 2.3, 'a' # packing
print(t)
```

```
x, y, z = t             # unpacking
x, y, z = 1, 2.3, 'a' # together
```

Tuples

Tuples are non-modifiable lists

```
t = (1, 2.3, 'a') # tuple
```

Try `t[0] = 2`, what happens ?

Important: tuples are **immutable**, they cannot me modified.

Packing & unpacking: the Pythonic way

```
t = 1, 2.3, 'a' # packing
print(t)
```

```
x, y, z = t              # unpacking
x, y, z = 1, 2.3, 'a' # together
```

Quizz: we have already seen unpacking, can you guess when ?

Tuples

Tuples are non-modifiable lists

```
t = (1, 2.3, 'a') # tuple
```

Try `t[0] = 2`, what happens ?

Important: tuples are **immutable**, they cannot me modified.

Packing & unpacking: the Pythonic way

```
t = 1, 2.3, 'a' # packing
print(t)
```

```
x, y, z = t              # unpacking
x, y, z = 1, 2.3, 'a' # together
```

Quizz: we have already seen unpacking, can you guess when ?

`for i, val in enumerate(l):` returns an iterable on tuples `(idx, value)`

Each iteration there is an unpacking: `i, val = idx, value`

Again: references & copy

Same warning as for lists applies for `dict`

Again: references & copy

Same warning as for lists applies for `dict`

```
d = {'a':2, 'b':'astring'}
d2 = d
d2['a'] = 'modified!'
print(d)
```

Workaround: use `d.copy()`

Functions

A first example

A function allows

- to manipulate objects with more than operators `+`, `*`, etc.
- to organize our program in small, simple blocks
- to reduce code repetition

Example: parity of an integer (modulo operator in python %)

```
def is_even(x):
    val = False
    if x%2==0:
        val = True
    return val
is_even(2)  # returns True
is_even(3)  # returns False
```

Formal structure & call

```
def my_function(argt_1, ...,  argt_n):
    """
    This is a Python docstring: a long comment used to
    document functions.
    """
    instructions_1
    ...
    instructions_m

    return something

val = my_function(argt_1= &,..., argt_n= &) # a function call
```

Important keywords

- **def**: initiate a function definition
- **return**: ends the function and define its *value*

Scope (1)

Python code is organized in blocks an sub-blocks, defining *scope*

```python
def func():
    # this is the function body, z only exists here (locally)
    z = 2
    print(x + z)

x = 2 #     x exists globally
func()
```

In this example there are two types of scope

- **global** scope: contains all the names defined in the top level
  Hence, `x` is available everywhere in the script, even if not passed as an argument.
- **local** scope: contains the names defined inside the function only
  Hence, `z` is only available inside `func()` block. Try `print(z)` outside of `func()` ?

Scope (2)

Variables can be redefined in the local scope.

$\implies$ we can use the same name for different variable in *global* and *local* scopes.

```python
def func(x):
    z = 2
    print(x + z)

x = 2
func(3)
print(x)
```

Scope (2)

Variables can be redefined in the local scope.

$\Longrightarrow$ we can use the same name for different variable in *global* and *local* scopes.

Unrolling what happens here

```python
def func(x):
    z = 2
    print(x + z)

x = 2
func(3)
print(x)
```

Scope (2)

Variables can be redefined in the local scope.

$\Longrightarrow$ we can use the same name for different variable in *global* and *local* scopes.

Unrolling what happens here

1. Variables `x` and `func` are created in the global scope

```
def func(x):
    z = 2
    print(x + z)

x = 2
func(3)
print(x)
```

Scope (2)

Variables can be redefined in the local scope.

$\Longrightarrow$ we can use the same name for different variable in *global* and *local* scopes.

```
def func(x):
    z = 2
    print(x + z)

x = 2
func(3)
print(x)
```

Unrolling what happens here

1. Variables `x` and `func` are created in the global scope
2. The call `func(3)` is read as `func(x=3)`

Scope (2)

Variables can be redefined in the local scope.

$\Longrightarrow$ we can use the same name for different variable in *global* and *local* scopes.

```python
def func(x):
    z = 2
    print(x + z)

x = 2
func(3)
print(x)
```

Unrolling what happens here

1. Variables `x` and `func` are created in the global scope
2. The call `func(3)` is read as `func(x=3)`
3. The formal parameter `x` become a variable in the local scope
4. The local `x` wins over the global one

Scope (2)

Variables can be redefined in the local scope.

$\Longrightarrow$ we can use the same name for different variable in *global* and *local* scopes.

```python
def func(x):
    z = 2
    print(x + z)

x = 2
func(3)
print(x)
```

Unrolling what happens here

1. Variables `x` and `func` are created in the global scope
2. The call `func(3)` is read as `func(x=3)`
3. The formal parameter `x` become a variable in the local scope
4. The local `x` wins over the global one
5. The body of `func` is executed with `x = 3`

Scope (2)

Variables can be redefined in the local scope.

$\Longrightarrow$ we can use the same name for different variable in *global* and *local* scopes.

```
def func(x):
    z = 2
    print(x + z)

x = 2
func(3)
print(x)
```

Unrolling what happens here

1. Variables `x` and `func` are created in the global scope
2. The call `func(3)` is read as `func(x=3)`
3. The formal parameter `x` become a variable in the local scope
4. The local `x` wins over the global one
5. The body of `func` is executed with `x = 3`
6. We get outside the function, local scope is erased.

Scope (2)

Variables can be redefined in the local scope.

$\Longrightarrow$ we can use the same name for different variable in *global* and *local* scopes.

```python
def func(x):
    z = 2
    print(x + z)

x = 2
func(3)
print(x)
```

Unrolling what happens here

1. Variables `x` and `func` are created in the global scope
2. The call `func(3)` is read as `func(x=3)`
3. The formal parameter `x` become a variable in the local scope
4. The local `x` wins over the global one
5. The body of `func` is executed with `x = 3`
6. We get outside the function, local scope is erased.
7. `print(x)` uses the global `x` since it is the only one existing.

Function value

The return any value can be

- Nothing: `return None` or no return
- Numerical, string, list
- Another function
- Basically any *object*

```
def exponentiate(n):
    def power(x):
        return x**n
    return power
x = 2
power_2 = exponentiate(2)
power_2(x)
[exponentiate(n)(x) for n in range(10)]
```

Lambda functions

Python onliner for short, simple functions $\longrightarrow$ avoid using **def**

General syntax: `lambda p_1, p_2, ...: expression`

The created function has no name (anonymous)

```
f = lambda x: x**2
f(2)
f(3)
```

$\iff$

```
def anonymous_func(x):
    return x**2
f = anonymous_func
f(2)
f(3)
```

Lambda functions

Python onliner for short, simple functions $\longrightarrow$ avoid using **def**

General syntax: `lambda p_1, p_2, ...: expression`

The created function has no name (anonymous)

```
f = lambda x: x**2
f(2)
f(3)
```

$\Longleftrightarrow$

```
def anonymous_func(x):
    return x**2
f = anonymous_func
f(2)
f(3)
```

Exercise

Rewrite the `exponentiate` function from previous slide using `lambda`

Default argument & function calls option

Some argument may be set to default values $\longrightarrow$ ease the use

Used everywhere in built-in functions, *e.g.* `list.sort(reverse=True)`

Syntax:

```
def f(a, b=2):
    return a + b

f(1)
f(1,3)
```

Default argument & function calls option

Some argument may be set to default values $\longrightarrow$ ease the use

Used everywhere in built-in functions, *e.g.* `list.sort(reverse=True)`

Syntax:

```
def f(a, b=2):
    return a + b

f(1)
f(1,3)
```

Possible ways of calling:

1. *Positional* arguments: `f(1, 3)` $\longrightarrow$ Python assign according to *order*
2. *Keywords* arguments: `f(b=3, a=1)` $\longrightarrow$ Python assign by their *names*
3. Positional then named: `f(1, b=3)`

Named then positional is not possible, *e.g.* `f(a=1, 2)` returns an error !

Function as argument

Functions are `objects`, *e.g.* `f = lambda x: x+1` and `print(type(f))`
Thus, `g=f` and `g(1)` returns 2
Why ? Same reason as for list, etc. `f` (and `g`) only store the reference toward the function

Ok... But why is it interesting ? Well, because `f` can be passed as an argument to another function

```
def filter_negative(f, l):
    return [x for x in l if f(x) < 0]

filter_negative(f, range(-4, 5))
```

The `functools` module implements some utilities based on this mechanism
`from functools import map, filter`

- `map(f, ite)` - Apply `f` to every item of `iter` and return a list of the results.
- `filter(f, ite)` - Construct a list from elements of `ite` for which `f` returns true.

Some exercises

Exercise 1 (easy)

Try to re-implement the function `map(f, ite)`

Exercise 2 (intermediary)

Let `l` be a list, sort `l` according to the values of the **square** of its elements.
E.g.: if `l=list(range(-2, 3))`, the program should return `[0, -1, 1, -2, 2]`
Hint: Use the key argument of `l.sort(key=f)` for some function `f` you should write.

Exercise 3 (difficult)

Search on the web for `reduce()` of the `functools` module
Using only two lines, write `sum_int(n)` that computes the sum of the n first integers $\sum_{i=0}^{n} i$ **without** using the formula $n(n+1)/2$

Objects

In-place modification