



Python pour le traitement et l'analyse de données financières

Course outline



- Chapter 1 : Algorithms
- Chapter 2 : Flow control instructions
- Chapter 3 : Built-in types
- Chapter 4 : Functions
- Chapter 5 : Exception handling
- Chapter 6 : Object Oriented Programming
- Chapter 7 : Functional programming
- Chapter 8 : File and Socket IO
- Chapter 9 : Intro to NumPy and Pandas
- Chapter 10 : Concurrency

Semester Grade

- Midterm (MT)
 - Multiple choice
 - Small programming assignment
- Final Exam (FE)
 - Multiple choice
 - Small programming assignment
- Exams Grade (EG) = $\max(0,5 * MT + 0,5 * FE, FE)$
- Final project (FP)
- Semester grade = $0,6 * EG + 0,4 * FP$

Final Project Idea 1



- Implement a simple market maker
- Listen to real time orderbook and trade data
- Calibrate spread around a mid price
- Backtest model
- Test model on real time data

Final Project Idea 2



- Find a simple correlated signal
- Time series trend + signal decomposition
- Use linear regression to find correlation of returns with a carefully selected feature
- Find optimal lag
- Backtest signal on historical price data
- Display results

Python installation



- Go to <https://www.anaconda.com/download>
- Download the version matching your OS (Windows, Mac, or Linux) and install it

Why Anaconda ?



In case of problems



**Google (and chaptgpt) are your
« friends »**

Python is one of the most well-documented
programming languages on the net 😊.



Let's dive into it !

Course outline



- Chapter 1 : Algorithms
- Chapter 2 : Flow control instructions
- Chapter 3 : Built-in types
- Chapter 4 : Functions
- Chapter 5 : Exception handling
- Chapter 6 : Object Oriented Programming
- Chapter 7 : Functional programming
- Chapter 8 : File and Socket IO
- Chapter 9 : Intro to NumPy and Pandas
- Chapter 10 : Concurrency

Definition



An **algorithm** is a finite and unambiguous list of instructions or operations to allow problem solving or to get a result.

Variables



A variable is like a box with a label. To get the data inside the box, we use the label to find it.

Variables



```
x = False
```

```
y = 1
```

```
z = 2.5
```

```
s = "I am a string"
```

x, y, z, and s are variables, you can use their names to access the value store in each of them

Arrays



An array is like a chest of drawers with 'n' drawers indexed from 0 to n - 1

`a = [1, 2, 3]`

the variable "a" is an array with 3 elements.

The elements of "a" can be accessed using indexes, for example:

the first element of a is `a[0]`, the second `a[1]` and the third `a[2]`

IF... THEN ... ELSE



If *test* is True

Then do

instructions1...

Else do

instructions2...

End If

IF... THEN ... ELSE



x = "input from keyboard"

If $x < 5$

Then do

$y = x + 5$


Else do

$y = x - 5$

End If

print y

WHILE



While *test* is True Do
 instructions...
End While

WHILE



Compute the following sum:

$$S = 1 + 2 + 3 + \dots + N$$

FOR i FROM 1 TO N



```
For i From 1 To N Do  
    instructions...  
End For
```


FOR i FROM 1 TO N



Compute the factorial:

$$N! = 1 * 2 * 3 * \dots * N$$

FOR EACH element IN



For Each *value* In *Sequence*, Do
 instructions...
End For

FOR EACH element IN



Exercise:

`seq = [1, 2, 3, 4, .., 10]`

Compute the sum of the square of the elements in seq...

FUNCTION



A function is a sequence of instructions.

The result depends on arguments.

For example the function $f(x) = x^2 + 3x - 5$ depends on the argument x

FUNCTION



Function f(x):

Return $x*x + 3 * x - 5$

End Function

For x From 0 to 100 with a step 0.1 do

print 'f(, x, ') =', f(x)

End for

Course outline



- Chapter 1 : Algorithms
- Chapter 2 : Flow control instructions
- Chapter 3 : Built-in types
- Chapter 4 : Functions
- Chapter 5 : Exception handling
- Chapter 6 : Object Oriented Programming
- Chapter 7 : Functional programming
- Chapter 8 : File and Socket IO
- Chapter 9 : Intro to NumPy and Pandas
- Chapter 10 : Concurrency

Variable



A variable is like a box with a label 'name'. Then we use the label to find the box and get the value

A variable contains 3 information: a name, a type and a value

We can destroy/remove a variable in using : `del variable_name`

Keywords



Python 3 reserved keywords:

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

These words are reserved for Python, You are not authorized to use them for a variable or a function name.

Comments



Comments in Python start by the character `#` and continue until the end of the line

Comments are not interpreted and are used to add information for developers about the aim of a portion of code

```
# this is the first comment  
spam = 1 # and this is the second  
        # ... And then the third!
```

```
text = "# This is not a comment because it's embedded inside a  
string."
```

Blocks and indents



To identify instructions sequence, Python is using of « significant indent ».

This syntax highlights a block of instructions and increased readability of source codes.

A compound statement is 2 things :

- Header line ended by **colons** “:”
- And an indented block of instructions

```
if n < 0 :  
    print("Negative number.")
```

Operators

When there are more than one operator in an expression, the order in which operators are evaluated depends on *priority rules*. In Python, the rules are the same than in math

Symbol	Name
{}	Dictionary
()	Argument
[]	Indexation operator
.	Attribute
**	Power
~	Bitwise invert

Symbole	Nom
<<	Left shift
>>	Right shift
&	Bitwise and
^	Bitwise exclusive or
	Bitwise or

Operators



Symbol	Name
==	Is equal
!=	Is not equal
is	Similar than
in	In/inside
not	Logical not
and	Logical and
or	Logical or
lambda	Lambda function

Symbol	Name
+	Positive
-	<u>negative</u>
*	Multiply
/	Divide
//	<u>Euclidian divide</u>
%	Modulo
+	Add
-	Minus
<	Strictly lower
>	Strictly upper
<=	Lower or equal
>=	Upper or equal

Enter data

Most of scripts in text mode need to enter data from keyboard, the best method consists in calling internal function **input()**.

User entered characters with the keyboard and ended the sequence by *<Enter>*.

The function `input()` returns a string with the text.

The string can be stored in a variable.

Exemple :

```
surname = input("Enter your surname : ")  
print("Hello,", surname)
```


IF ... THEN ... ELSE ...

We can use the « if » statement alone, with « else » or with « elif » statement ('elif' correspond to 'else if').

We are authorized to write $a \leq x \leq b$

```
if a > 5:  
... a = a + 1  
else:  
... a = a - 1  
...
```

```
if a > 5:  
... a = a + 1  
elif a == 5:  
... a = a + 1000  
else:  
... a = a - 1  
...
```

WHILE/FOR loops

To iterate from 0 to 100 we can write:

```
i = 0
while i < 100:
...   print("Once more")
...   i = i + 1
...
```

```
for i in range(0,100):
...   print("Once more")
...
```

To stop a loop immediately we can use the keyword **break**.

To jump to the next iteration, use **continue**

FOR loop

```
v = "Bonjour"
for c in v:
... print( c )
...
B
o
n
j
o
u
r
```

range



```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

Text file reading



```
# print lines of a file
f = open("file.txt") # defaulting to "r" mode
for line in f :
    print(line)
f.close()
```

Text file is a sequence of lines

Walrus operator (python version ≥ 3.8)

```
inputs = list()
current = input("Write something: ")
while current != "quit":
    inputs.append(current)
    current = input("Write something: ")
```

Can be simplify in:

```
inputs = list()
while (current := input("Write something: ")) != "quit":
    inputs.append(current)
```

Course outline



- Chapter 1 : Algorithms
- Chapter 2 : Flow control instructions
- Chapter 3 : **Built-in types**
- Chapter 4 : Functions
- Chapter 5 : Exception handling
- Chapter 6 : Object Oriented Programming
- Chapter 7 : Functional programming
- Chapter 8 : File and Socket IO
- Chapter 9 : Intro to NumPy and Pandas
- Chapter 10 : Concurrency

Types in Python

- A **type** is an information about what is contained inside the variable. It indicates the rules to modify that information to the *python interpreter*
- Python has a dynamic typing : the type is not defined explicitly, type is attached to the data
- For example: `pi=3.14`
 - Type of the variable is not defined
 - But, it's implicit that it's a floating-point value
 - `pi` is a float
 - At first initialization, a variable is, most of the time, set with a constant

Mutable/immutable

- An immutable variable **can't be modified**
- A writable operation on that kind of type force the creation of a new data, even if the data is temporary.
- For example, `x+=3` which consists in adding 3 to the values contained in x
 - Create a new variable with value `x+3`
 - Copy the result under variable `x`.
- Lots of immutable types exist in Python like string and tuple

Type NoneType (nothing)

- NoneType is for a variable which contains nothing
- Variable type is NoneType and its value is None
- Example :
 - `s = None`
 - `print(s)` # print None
- Some functions use this way to return always a value in the case where computation is not possible
- There are other solutions to do:
 - Raise an exception
 - Return a default value
 - Return None
- There is no best choice, it depends on the convention/contract

Boolean type : bool

- Booleans are result of logical operations and there are 2 different values : **True** or **False**
- Boolean operators are:
 - and
 - or
 - not

```
x = 4 < 5
```

```
print(x)          # affiche True
```

```
print(not x)      # affiche False
```

Numerics



- 3 types of numerics are available in python :
 - Floating-point values: float
 - Integers: int
 - Complex number: complex
- Functions int & float convert a data or a string into an integer (sometimes rounded) or in floating-point value.

```
x = int(3.5)
```

```
y = float(3)
```

```
z = int("3")
```

Numerics

```
i = int("2.0")           # raise an exception!  
i = int( float ("3.5") ) # is valid!
```

Integers

- Sign change: $+x$, $-x$
- Addition, subtract, product: $x+y$, $x-y$, $x*y$
- Division to float: x/y
- Euclidian division: $q = x//y$
- Remainder of euclidian division: $r = x\%y$
- Euclidian division: $q, r = \text{divmod}(x, y)$
- Convert: $\text{int}(x)$
- Absolute value: $y = \text{abs}(x)$
- Power: $\text{pow}(x, y)$, $x**y$

Floating-point value

- Sign change: `+x`, `-x`
- Addition, subtract, product: `x+y`, `x-y`, `x*y`
- Division: `x/y`
- Convert: `float(x)`
- Power: `pow(x, y)`, `x**y`
- Get a fraction: `(3.5).as_integer_ratio()`
- Is a real integer: `(-2.0).is_integer()`

Strings



- A string is like an array of characters in Python. Operators and functions are similar
- **str** function converts a number, an array, or an object in printable string
- **len** function returns the string length (the number of characters inside)

Utilisation des chaînes de caractères



```
"hello"+"world"
```

```
"hello"*3
```

```
"hello"[0]
```

```
"hello"[-1]
```

```
"hello"[1:4]
```

```
len("hello")
```

```
"hello" < "jello"
```

```
"e" in "hello"
```

Format... to print



- Since Python 1.x:
we can build strings in using « format string » (which contain %s, %d, etc..) with the % operator
 - %s is to say : it's a "string" (text)
 - %d is a "digit" (integer)
 - %f is a "floating-point"
- Since Python 2.7:
format() method allow a fine management of string creation. For example, we will use it before calling print()
- Since Python 3.6, we have the F-string

Format... to print

```
print("{} {} and {}".format("zero", "one", "two"))
```

zero, one and two

Old style	New style
'%s %s' % ('un', 'deux')	'{} {}'.format('un', 'deux')
'%d %d' % (1, 2)	'{} {}'.format(1, 2)

Format... to print



```
'{1} {0}'.format('one', 'two')
```

prints :

two one

This authorizes positional reference and we can repeat some terms

```
'{1} {0} {1}'.format('one', 'two')
```

prints :

two one two

Format... to print

- Named parameters:

```
a, b = 5, 3
```

```
print("The story of {c} and {d}".format(c=a+b, d=a-b))
```

```
The story of 8 and 2
```

- Index inside list:

```
stock = ['paper', 'envelope', 'shirt', 'ink']
```

```
print("We have {0[3]} and {0[0]} in our inventory\n".format(stock))
```

```
prints:
```

```
We have ink and paper in our inventory
```

Format... to print

- Alignment:
- By default, values are converted in using only the needed spaces to display the contain. However we can choose a specific output size.
- For example :

```
'{:>10}'.format('test')
```

displays :

test

Format... to print

- Fill with specific character:

```
'{: _<10}'.format('test')
```

```
'test_____'
```

- Center:

```
'{: ^10}'.format('test')
```

```
'   test   '
```

Format... to print

- Truncate the output :

```
'{: .5}'.format('xylophone')  
'xylop'
```

- Fill & truncate in one step :

```
'{: _>10.5}'.format('xylophone')  
'_____xylop'
```


Format... to print

- Floating-point format:

```
'{:f}'.format(3.141592653589793)
```

```
'3.141593'
```

- Integers:

```
'{:04d}'.format(42)
```

```
0042
```

```
'{:05d}'.format(42)
```

```
00042
```

```
'{:+d}'.format(42)
```

```
+42
```

Containers



- For Python, a container is an object in which we can store other objects, as a string, the lists, the sets or the dictionaries
- Several containers (also named collections) are deeply integrated in the language

Lists

- A list is like a collection of objects between brackets and separated by commas.
- Some samples:

```
x = [4,5]           # list of 2 integers
x = ["un",1,"deux",2] # several types mixed into one
x = [3,]            # list with only one element
x = [ ]             # empty list
x = [0] * 3          # list with repeated sequence
x = list ()          # empty list
y = x[0]              # get the first element
y = x[-1]             # get the last element
```

Lists



- Lists are mutable objects, so we can modify its elements/data
- List operations are the same than for tuples in adding some functionalities due to update properties
- We can insert data, remove element(s), or order them.
- La syntaxe des opérations sur les listes est similaire à celle des opérations qui s'appliquent sur les chaînes de caractères

Lists : operations



x in l	True if x is inside l
l + t	Concat l and t
l * n	Concat n times l
len(l)	Return the number of elements of l
min(l)	Returns the smallest value inside l. Types of values have to be comparable
max(l)	Return the greatest values of l
sum(l)	Sum of all elements
del l [i : j]	Remove éléments between index i and j excluded. Equivalent to <code>l [i:j] = []</code> .

Lists : operations



list (x)	Convert x into a list if it's possible
l.count (x)	Return the occurrence count of element x.
<u>l.index (x)</u>	Return the index of first occurrence of x in list l.
l.append (x)	Append x at list end. if x is a list, this function add list x as an element.
l.extend (k)	Append all k éléments inside the list l.
l.insert(i,x)	Insert x at position i into list l.
l.remove (x)	Remove first occurrence of x in list l. if no occurrence was found, an exception will be raised.

Lists : operations



<code>l.pop (i)</code>	Returns the element <code>l[i]</code> and remove it from the list. <code>i</code> is not mandatory, by default, the last element is targeted.
<code>l.reverse ()</code>	Reverse in-place the list data.
<code>l.sort ([key=None, reverse=False])</code>	Sort the list from smallest to greatest value. 'key' parameter allows to define the comparison key. If reverse is True, the sort is in reverse order.

Lists

- Create an array of 10 float initialised to 0:
 - `a = [0] * 10`
 - `print(a)`
- Create an array with the square values from 0 to 10 included:
 - `a = [i*i for i in range(11)]`
 - `print(a)`
- Create an array with n+1 first integers (0, 1, ..., n):
 - `a = list(range(n+1))`
 - `print(a)`

Lists



```
a = [[]] * 5
```

```
a[0].append( "a" )
```

```
print( a )
```

Lists

```
a = [[]] * 5
```

```
a[0].append( "a" )
```

```
print( a )
```

prints:

```
[['a'], [], [], [], []]
```

Lists



```
a = [ [] for i in range(5) ]  
a[ 0 ].append( "a" )  
print( a )
```

Lists

```
a = [ [] for i in range(5) ]  
a[ 0 ].append( "a" )  
print( a )
```

prints:

```
[['a'], [], [], [], []]
```

Tuples

- Tuples are immutable lists.
- A tuple was defined like a list but using parenthesis instead of brackets.

```
x = (4,5)                # tuple of 2 integers
x = ("one",1, "two",2)  # different types, order is important
x = (3,)                # tuple of one element
x = (3)                  # it's an integer
```

Tuples : operations



<code>x in s</code>	True if x is in s
<code>x not in s</code>	True if x is not in s
<code>s + t</code>	Concat s with t
<code>s * n</code>	Concat n times s one after others
<code>s[i]</code>	Return the i-th element of s (0 based index)
<code>s[i:j]</code>	Return a tuple with elements of s from index i to j excluded

Tuples : operations

<code>s[i:j:k]</code>	Return a tuple with element of s with indices from i to j excluded with a step of k : i, i+k, i+2k, i+3k,
<code>len(s)</code>	Return the number of elements of s
<code>min(s)</code>	Return the smallest element of s, éléments have to be comparable
<code>max(s)</code>	Return the greatest element of s
<code>sum(s)</code>	Return the sum of all elements

Tuples : operations

- As tuple are immutable, it's not possible to modify elements.
- Sample :

```
x = (3,2)
```

```
x[0] = 1
```

TypeError: 'tuple' object does not support item assignment

- However, we can write : `x = (4,5)`
- How ? Last operation modified the reference to the object and not the object itself (which is immutable)

Tuples : operations

- So, we can write `x = (1,2)` to modify `x[0]` : we create a new tuple
- Similar way to do :

```
x = (3,2)
```

```
x = (1,) + x[1:2]  # create a new tuple with  
                  # x part which is unchanged
```

- `a, b = b, a`
exchange a and b values

Set



- A set is an unordered object collection not ***with unique elements***.
- It's like a mathematical set with equivalent operations.
- A *frozenset* is an immutable set
- `set()` returns an empty list

Set

- Create :

```
s = {1,2,3}
```

```
s = set ([1,2,3,2])
```

```
s = {1.0, "Hello", (1, 2, 3)}
```

```
s = set()          # empty set
```

Set : operations



add()	add an element
clear()	Remove all elements
copy()	Return a copy of the set
<u>difference()</u>	Return the difference of two or more sets as a new set
<u>difference_update()</u>	Remove all elements of another set from this set
discard()	Remove an element from a set if it is a member. If the element is not a member, do nothing.
intersection()	Return the intersection of two sets as a new set

Set : operations



<code>intersection()</code>	Return the intersection of two sets as a new set
<code>isdisjoint()</code>	Return True if two sets have a null intersection.
<code>issubset()</code>	Report whether another set contains this set
<code>issuperset()</code>	Report whether this set contains another set
<code>pop()</code>	Remove and return an arbitrary set element. Raises <code>KeyError</code> if the set is empty.
<code>remove()</code>	Remove an element from a set; it must be a member. If the element is not a member, raise a <code>KeyError</code> .

Set : operations



<code>symmetric_difference()</code>	Return the symmetric difference of two sets as a new set
<code>symmetric_difference_update()</code>	Update a set with the symmetric difference of itself and another
<code>union()</code>	Return the union of sets as a new set
<code>update()</code>	Update a set with the union of itself and others

Dictionaries

- Search by key is optimized.
- For example, the chemical symbol associate to the name:

```
symbols = {  
    "H": "hydrogen",  
    "He": "helium",  
    "Li": "lithium",  
    "C": "carbon",  
    "O": "oxygen",  
    "N": "nitrogen"  
}
```

Dictionaries

- Creation :

```
x = { "key1": "value1", "key2": "value2" }  
y = { }           # crée un dictionnaire vide  
z = dict()        # crée un dictionnaire vide
```

```
symbols = {  
    "H": "hydrogen",  
    "He": "helium",  
    "Li": "lithium",  
    "C": "carbon",  
    "O": "oxygen",  
    "N": "nitrogen"  
}
```


Dictionaries

```
symbols["C"]
```

```
'carbon'
```

```
"O" in symbols, "U" in symbols
```

```
(True, False)
```

```
"oxygen" in symbols
```

```
False
```

```
symbols["P"]
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
KeyError: 'P'
```

```
symbols.get("P", "unknown")
```

```
'unknown'
```

```
symbols.get("C", "unknown")
```

```
'carbon'
```

Dictionaries

```
list(symbols.keys())
['C', 'H', 'O', 'N', 'Li', 'He']
list( symbols.values())
['carbon', 'hydrogen', 'oxygen', 'nitrogen', 'lithium', 'helium']
symbols.update( {"P": "phosphorous", "S": "sulfur"} )
list(symbols.items())
[('C', 'carbon'), ('H', 'hydrogen'), ('O', 'oxygen'), ('N', 'nitrogen'),
('P', 'phosphorous'), ('S', 'sulfur'), ('Li', 'lithium'), ('He', 'helium')]
del symbols['C']
symbols
{'H': 'hydrogen', 'O': 'oxygen', 'N': 'nitrogen', 'Li': 'lithium', 'He':
'helium'}
```

Dictionaries : operations

x in d	True if x is a key of d
x not in d	True if x is not a key of d
d[i]	Return the value associated to the key i
len(d)	Return the number of associations stored in d
min(d)	Return the smallest key
max(d)	Return the greatest key
del d [i]	Remove the couple of data associated to key i
list (d)	Return the list of keys
dict (x)	Convert x en in dictionary if it's possible,

Dictionaries : operations



d.copy ()	Return a copy of d
d.items ()	Return an iterator of (key value) inside the dictionary.
d.keys ()	Iterator on keys
d.values ()	Iterator on values
d.get (k[,x])	If k is a key of d, return d[k] else returns the value x.
d.clear ()	Empties the dictionary.

Dictionaries : operations



<code>d.update(d2)</code>	Data contained in d2 are added in d.
<code>d.setdefault(k[,x])</code>	Insert key with a value of default if key is not in the dictionary. Return the value for key if key is in the dictionary, else default
<code>d.pop(k[,d])</code>	remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised.

File management

- By default python use text file. For binary file, use the option b.
- Open options mode are :

```
f1 = open("file_1", "r", encoding='utf8') # "r" reading mode
f2 = open("file_2", "w", encoding='utf8') # "w" writing mode
f3 = open("file_3", "a", encoding='utf8') # "a" append mode
```
- Optional parameter encoding allows to convert the text on disk
- Frequently used encoding 'utf8', but you can use other types: 'latin1', 'ascii'. . .
- Don't forget to close the file : file.close()

Sequential writing

```
f= open("C://Users/Moez/Desktop/file.txt", "w")
s = 'text1\n'
f.write(s) # write the strings in f
l = ['a', 'b', 'c']
f.writelines(l) # write the strings in the list l in f in adding
                 # carriage return after each string
f.close()
f2 = open("C://Users/Moez/Desktop/file.txt", "w")
print("text2", file=f2) # redirect output into file
f2.close()
```

```
f = open("file.txt", "r")  
s = f.read()          # all the file --> string  
                        # DANGER: memory consuming  
s = f.read(3)         # read max 3 bytes --> string  
s = f.readline()      # read a line--> string  
s = f.readlines()     # read all the file --> list of strings  
                        # DANGER: memory consuming  
  
f.close()
```


Advanced lists functions



- La fonction **enumerate()**
- La fonction **zip()**
- Le mot-clé **with**

enumerate()



- Objective is to access the index and the correspond value at the same time

- A wrong way to do:

```
for indice in range(len(liste)):
    print("liste[%d] = %s" % (indice, liste[indice]))
```

- Pythonic way:

```
for indice, valeur in enumerate(liste):
    print("liste[%d] = %s" % (indice, valeur))
```

zip()

- Functional languages provide methods to iterate through several sequences in the same time to be more efficient.
- Naïve solution is:

```
for i in range(min(len(liste1), len(liste2))):  
    e1, e2 = liste1[i], liste2[i]  
    # action on e1 and e2
```

- Pythonic way:

```
for e1, e2 in zip(liste1, liste2):  
    # Actions en utilisant e1 et e2
```

- Exemple simple, zip renvoie une liste de tuples:

```
list( zip(['a', 'b', 'c'], ['d', 'e', 'f', 'g']) )  
[('a', 'd'), ('b', 'e'), ('c', 'f')]
```

Keyword with

Keyword ***with*** allow code simplification. For example, with a file, we can guarantee that the file will be closed when exiting the with block, even in case of an exception:

```
with open('fichier', 'w') as f:  
    for line in f:  
        print( line )
```

Course outline



- Chapter 1 : Algorithms
- Chapter 2 : Flow control instructions
- Chapter 3 : Built-in types
- Chapter 4 : **Functions**
- Chapter 5 : Exception handling
- Chapter 6 : Object Oriented Programming
- Chapter 7 : Functional programming
- Chapter 8 : File and Socket IO
- Chapter 9 : Intro to NumPy and Pandas
- Chapter 10 : Concurrency

FUNCTIONS



lambdas

recursion

décorateur

FUNCTIONS: SCOPE



Function variables have 2 scopes: **local** and **global**

- local: what is inside the function (in Python that means what is indented inside the function definition)
- global: as its name indicates, this scope refers to all variables defined outside the function, for example a global variable defined in a module is part of the global scope and can be modified inside a function using the keyword “global”

FUNCTIONS: LOCAL SCOPE EXAMPLE



```
def f():
```

```
    x = 1
```

```
    return x
```

```
print(x) # this will return an error
```

- x is part of the function f local scope
- x cannot be accessed from outside the function

FUNCTIONS: GLOBAL SCOPE EXAMPLE



```
x = 1
```

```
def f():
```

```
    print(x)
```

- x is part of the global scope, it can be accessed inside f
- x cannot be modified by f if the keyword “global” is not used (see on the right)

```
x = 1
```

```
def g():
```

```
    global x
```

```
    x += 1
```

- x is part of the global scope, it can be accessed inside g
- x can be modified by because the keyword “global” was used

FUNCTIONS: PARAMETERS



Functions can take one parameters as inputs:

```
def f(x):
```

```
    print(x)
```

```
>>> f(2)
```

```
2
```

- x is a parameter

Functions can take multiple parameters as inputs:

```
def f(x, y, z):
```

```
    print(x, y, z)
```

```
>>> f(1, 2, 34)
```

```
1 2 34
```

- x, y, and z are parameters

FUNCTIONS: PARAMETERS - *args

Functions can take an arbitrary number of parameters:

```
def f(*args):  
    # here Python sees args as a tuple  
    for el in args:  
        print(el)
```

```
>>> f(1,2,"some argument", "hey")
```

```
1
```

```
2
```

```
some argument
```

```
hey
```

FUNCTIONS: PARAMETERS - `**kwargs`



Functions can take an arbitrary number of named parameters, `kwargs` is shorthand for “keyword arguments”, but you can call it whatever you’d like:

```
def f(**kwargs):  
    # here Python sees kwargs as a dict  
    print(kwargs)
```

```
>>> f(a=32, blabla="hello", b=45)  
{“a”: 32, “blabla”: “hello”, “b”: 45}
```

FUNCTIONS PARAMETERS: MIXUP



You can mix up the way you enter parameters at your convenience, but the order must always follow

- 1) regular parameters
- 2) `*args`
- 3) `**kwargs`

For example:

```
def f(a, *args, **kwargs):  
    print(a)  
    print([i for i in args])  
    print(kwargs)  
  
c f(34, 25, 25, 64, b="hey", c=56789)  
  
34  
[25, 25, 64]  
{'b': 'hey', 'c': 56789}
```

FUNCTIONS: IMMUTABLE PARAMETERS



An immutable parameter will not be altered by the function

You need to return it and re-assign if you want the new value

```
x = 1 # x is an int so it is immutable
```

```
def f(x):
```

```
    x += 1
```

```
f(x)
```

```
print(x) # this will output 1, not 2
```

```
x = 1 # x is an int so it is immutable
```

```
def f(x):
```

```
    x += 1
```

```
    return x
```

```
x = f(x)
```

```
print(x) # this will output 2
```

FUNCTIONS: MUTABLE PARAMETERS



A mutable parameter will be altered by the function

You don't need to return it (although you might want to, it's fine) you want the new value

/!\ Be careful when passing around mutable types as your code may alter them in unexpected ways if you don't pay attention, for example when passing around lists defined globally as a function parameter

```
l = [1, 2, 3]
```

```
def f(x):
```

```
    x[2] = 34
```

```
>>> f(l)
```

```
>>> print(l)
```

```
[1, 2, 34]
```

FUNCTIONS: LAMBDA EXPRESSIONS



Lambda expressions can be used to define functions in a more compact (functional) way

```
def f(x):
```

```
    return x * x
```

Can be re-written as

```
f = lambda x: x*x
```


FUNCTIONS: LAMBDA EXPRESSIONS



Given the following list of lists `l = [[1, 34], [2, 2]]`, let's say I want to sort it in ascending order with respect to the second position (at index 1)

```
>>> sorted(l, key=lambda x: x[1], reverse=False)
```

```
[[2, 2], [1, 34]]
```

FUNCTIONS: DECORATORS



A decorator, as its name indicates, decorate a function with additional code to be executed “around” the function

Let’s say I have a function f and I want to print additional information around it

```
def my_decorator(f):  
    def inner():  
        print("about to execute the function f")  
        out = f()  
        print("f was executed, exiting the decorator now")  
        return out  
    return inner  
  
@my_decorator  
def f():  
    return 1  
  
>>> f()  
hey  
done  
1
```

FUNCTIONS: RECURSION



Functions can be called recursively, for example if I were to reverse the string “hello, world !” using recursion:

```
def reverse(s):  
    if s == "": # this “base case” is NECESSARY for your recursion to stop  
        return ""  
    return reverse(s[1:]) + s[0]
```

In more compact form (more “pythonic” way):

```
def reverse(s):  
    return reverse(s[1:]) + s[0] if s else ""
```

EXCEPTIONS



Exceptions are used to handle certain behavior that would cause your code to crash

Often, not all the time, logic statements are not enough and you will likely require the use exceptions

EXCEPTIONS



`Exceptions` catches all exceptions. It can be used but good practice is often to use the correct exception for a given error.

Catching everything with `Exceptions` means your program never crashes, even when you might want it to !

Exceptions are used using the `try-except` statements, or a more complete

`try-except-finally` (or an even more complete, ask if interested) `try-except-else-finally` statement

EXCEPTIONS: CATCHING



For example, you may want to catch a `ZeroDivisionError` like so:

```
def divide(n, d):  
    try:  
        return n / d  
    except ZeroDivisionError:  
        print("You cannot divide by zero!")  
        print("My program didn't crash, yeah !")
```

```
>>> divide(10, 0)
```

You cannot divide by zero!

My program didn't crash, yeah !

EXCEPTIONS: FINALLY CLAUSE



For example, you may want to catch a `ZeroDivisionError` like so:

```
def divide(n, d):  
    try:  
        return n / d  
    except ZeroDivisionError:  
        print("You cannot divide by zero!")  
    finally:  
        print("THIS PART ALWAYS EXECUTE ... ALWAYS !")
```

```
>>> divide(10, 0)
```

You cannot divide by zero!

THIS PART ALWAYS EXECUTE ... ALWAYS !

EXCEPTIONS: WHICH ONE ?



Checkout the official [Python language documentation](#) online for an extensive list of all implemented Exceptions at your disposal