Python pour le traitement et l'analyse de données financières

Course outline

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 ?

jupyter
SciPy
SPYDER
NumPy
DASK
bokeh
scikit learn
pandas
$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$
TensorFlow
Numba
Panel
matplotlib
HoloViews
PYTORCH
CONDA

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** : Modules
- **Chapter 8** : Functional programming
- **Chapter 9** : File and Socket IO
- **Chapter 10** : Intro to NumPy and Pandas
- **Chapter 11** : 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:

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

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 * … * 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** : Modules
- **Chapter 8** : Functional programming
- **Chapter 9** : File and Socket IO
- **Chapter 10** : Intro to NumPy and Pandas
- **Chapter 11** : Concurrency

Variable

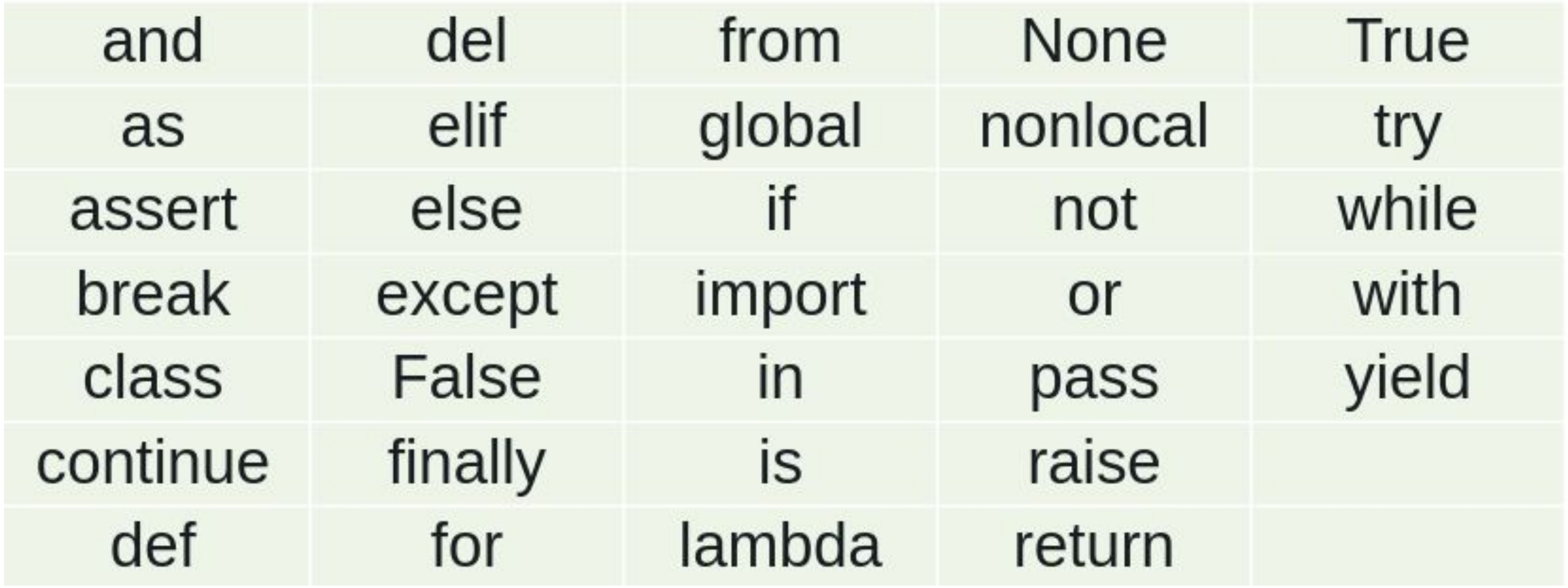
A variable is like a box with a label ‘name’. Then we use the label to find the box and get the value

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.

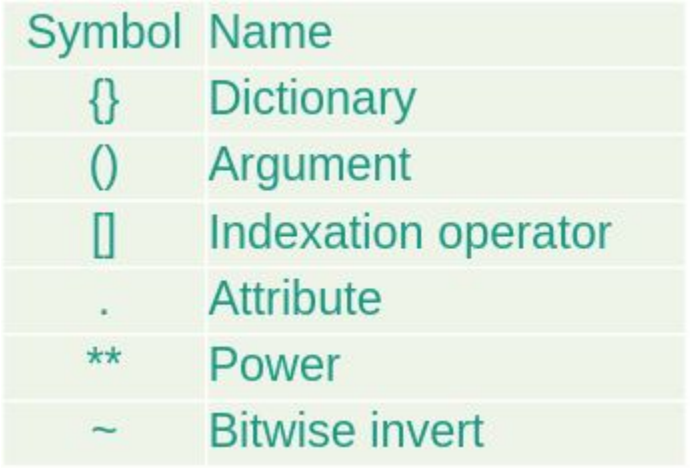
A compound statement is 2 things :

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

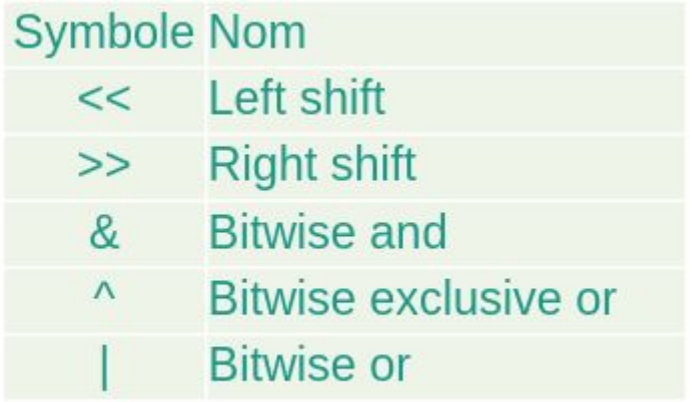
```
if n < 0 :
    print("Negative number.")
```

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

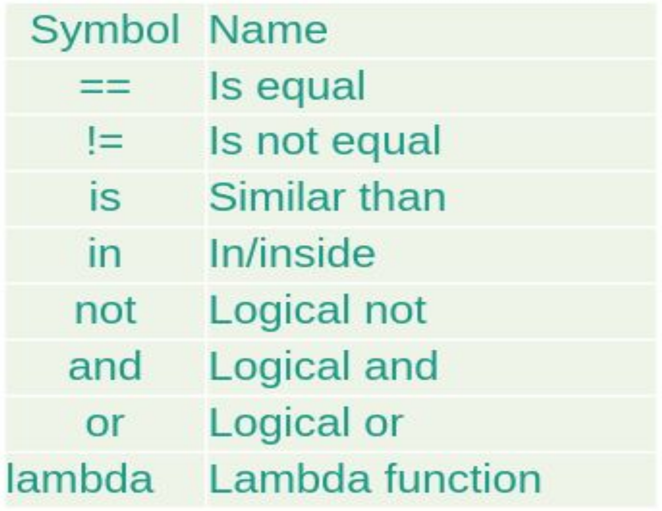


Symbol	Name
{}	Dictionary
()	Argument
[]	Indexation operator
.	Attribute
**	Power
~	Bitwise invert

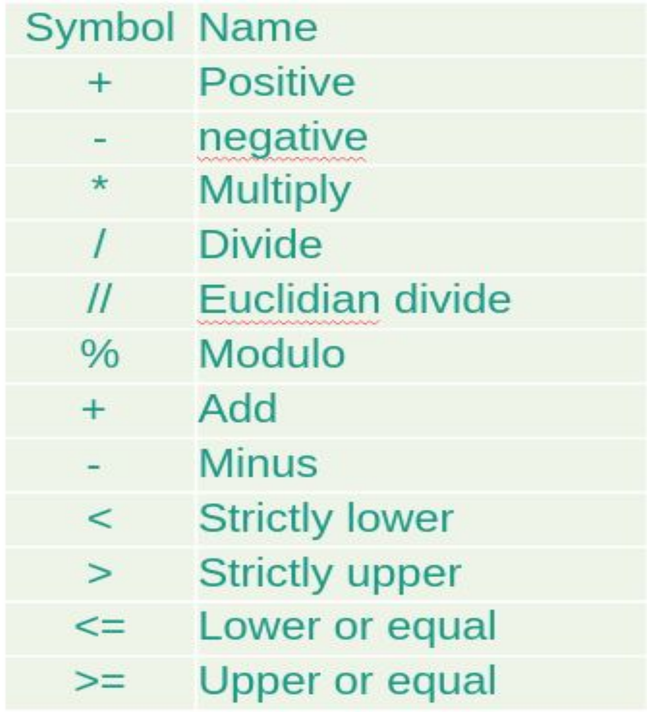


Symbole	Nom
<<	Left shift
>>	Right shift
&	Bitwise and
^	Bitwise exclusive or
|	Bitwise or

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
-	negative
*	Multiply
/	Divide
//	Euclidian divide
%	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.

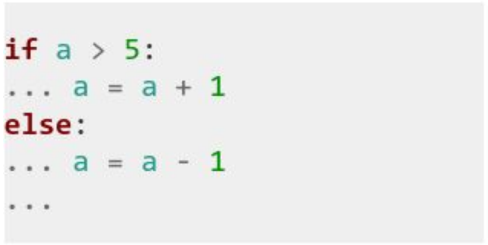
Exemple :

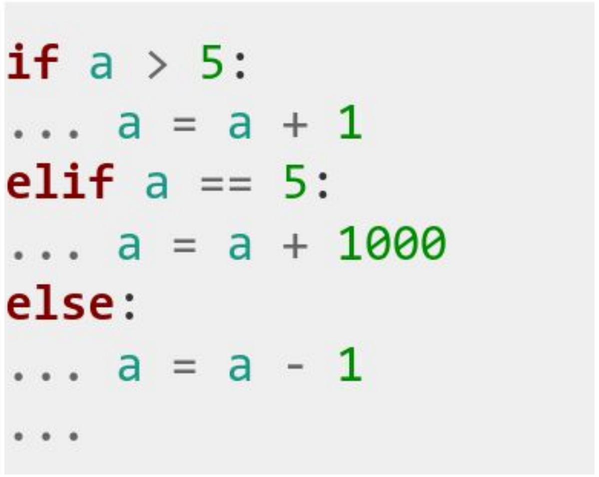
```
surname = input("Enter your surname : ")
print("Hello,", surname)
```

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 <= x <= 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")
...
```

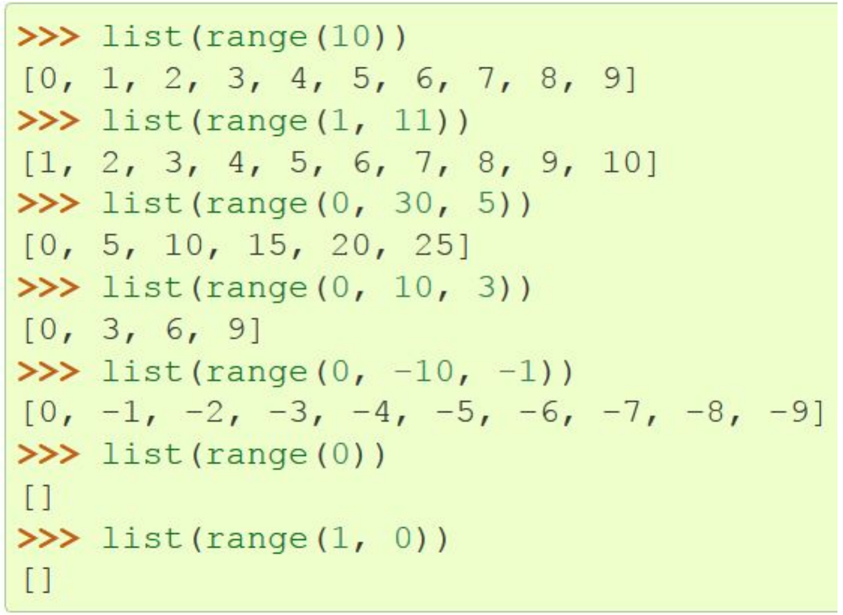
To stop a loop immediately we can use the keyword break.

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

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
- Exemple :
  - `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, substract, 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 = divmod(x, y)
- Convert: int(x)
- Absolute value: y = abs(x)
- Power: pow(x, y), x**y

Floating-point value

- Sign change: +x, -x
- Addition, substract, 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

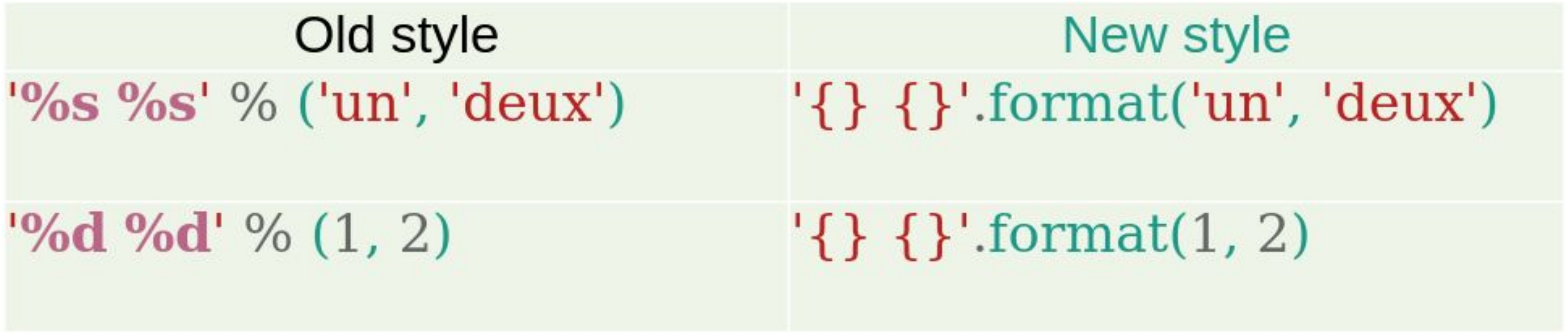
- Since Python 1.x:
  we can build strings in using « format string » (which contain %s, %d, etc..) with the % operator
  - %s is to say : it’s a "string" (text)
  - %d is a "digit" (integer)
  - %f is a "floating-point"

- 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

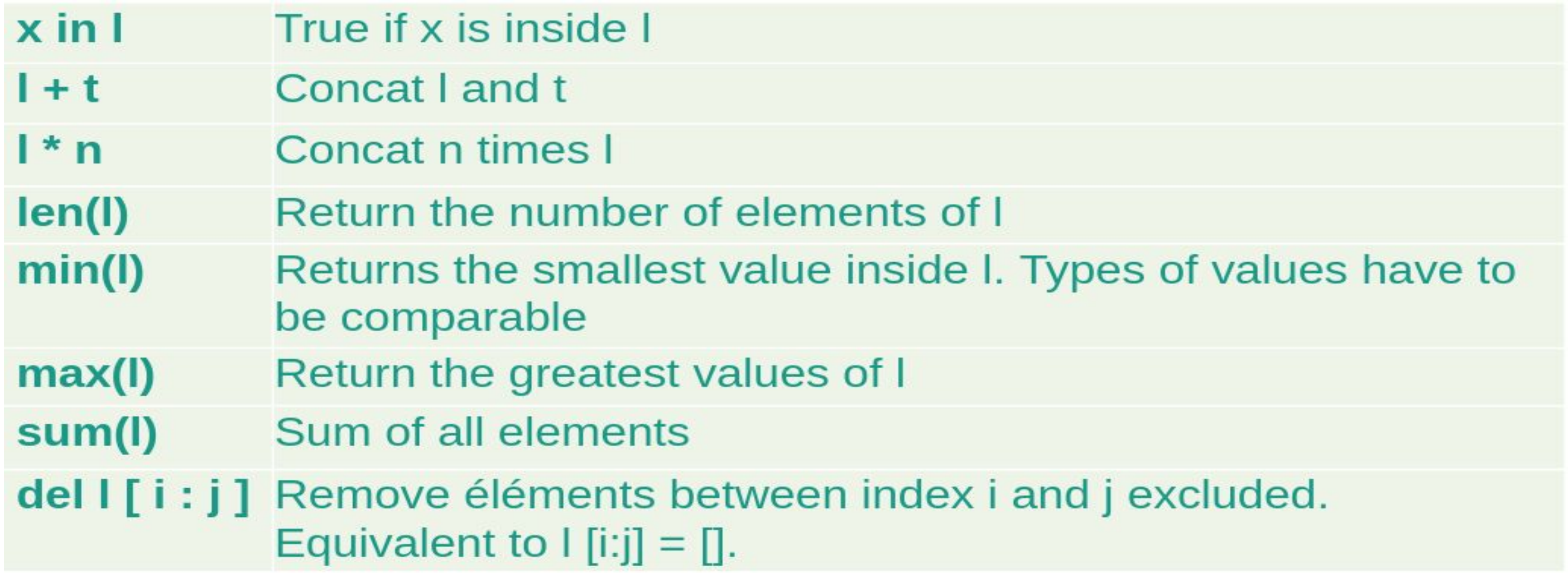
- A list is like a collection of objects between brackets and separated by commas.
- Some samples:

```
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

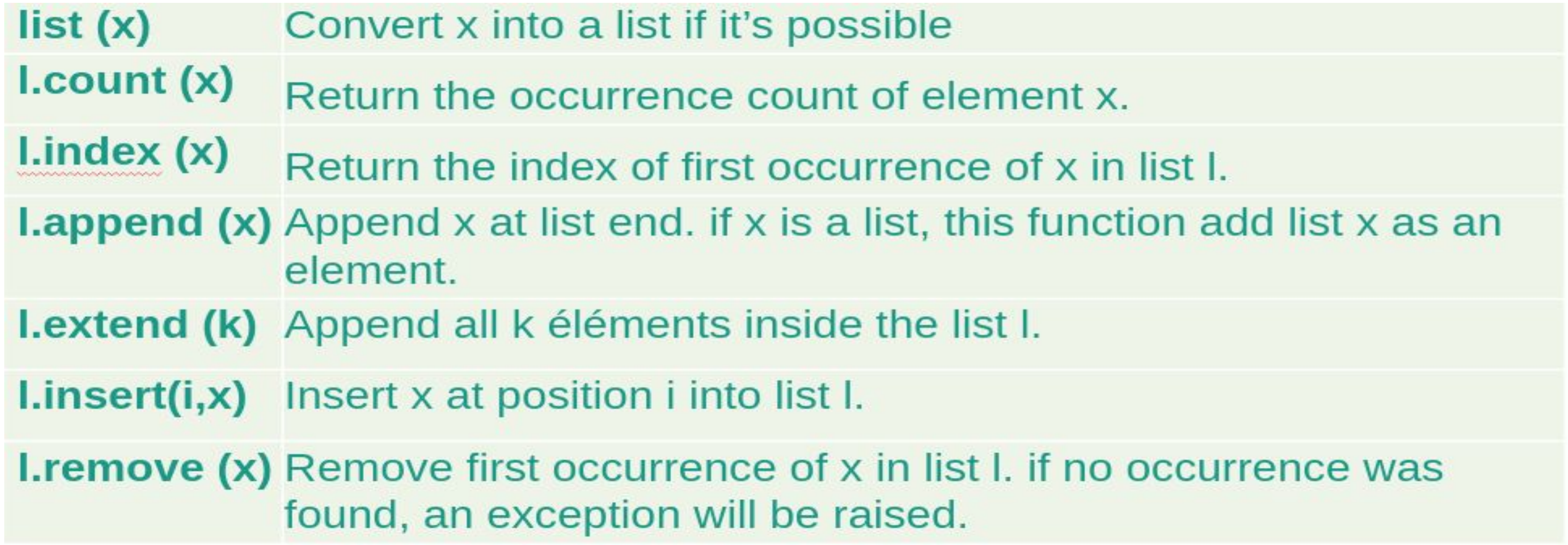
- Lists are mutable objects, so we can modify its elements/data
- List operations are the same than for tuples in adding some functionalities due to update properties
- We can insert data, remove element(s), or order them.
- La syntaxe des opérations sur les listes est similaire à celle des opérations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

Lists : operations



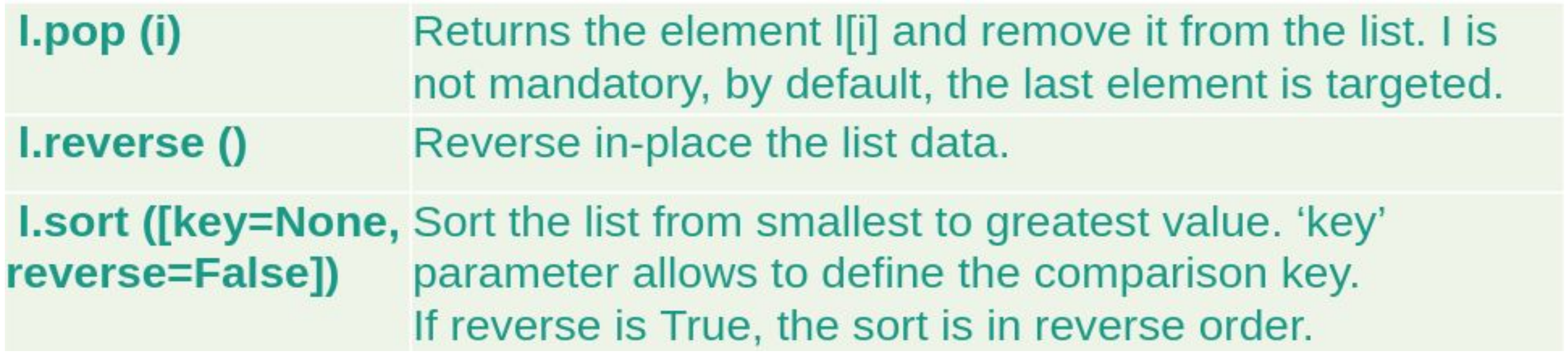
x in l	True if x is inside l
l + t	Concat l and t
l * n	Concat n times l
len(l)	Return the number of elements of l
min(l)	Returns the smallest value inside l. Types of values have to be comparable
max(l)	Return the greatest values of l
sum(l)	Sum of all elements
del l [i : j]	Remove éléments between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

Lists : operations



list (x)	Convert x into a list if it’s possible
l.count (x)	Return the occurrence count of element x.
l.index (x)	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



l.pop (i)	Returns the element l[i] and remove it from the list. I is not mandatory, by default, the last element is targeted.
l.reverse ()	Reverse in-place the list data.
l.sort ([key=None, reverse=False])	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'], ['a'], ['a'], ['a'], ['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:

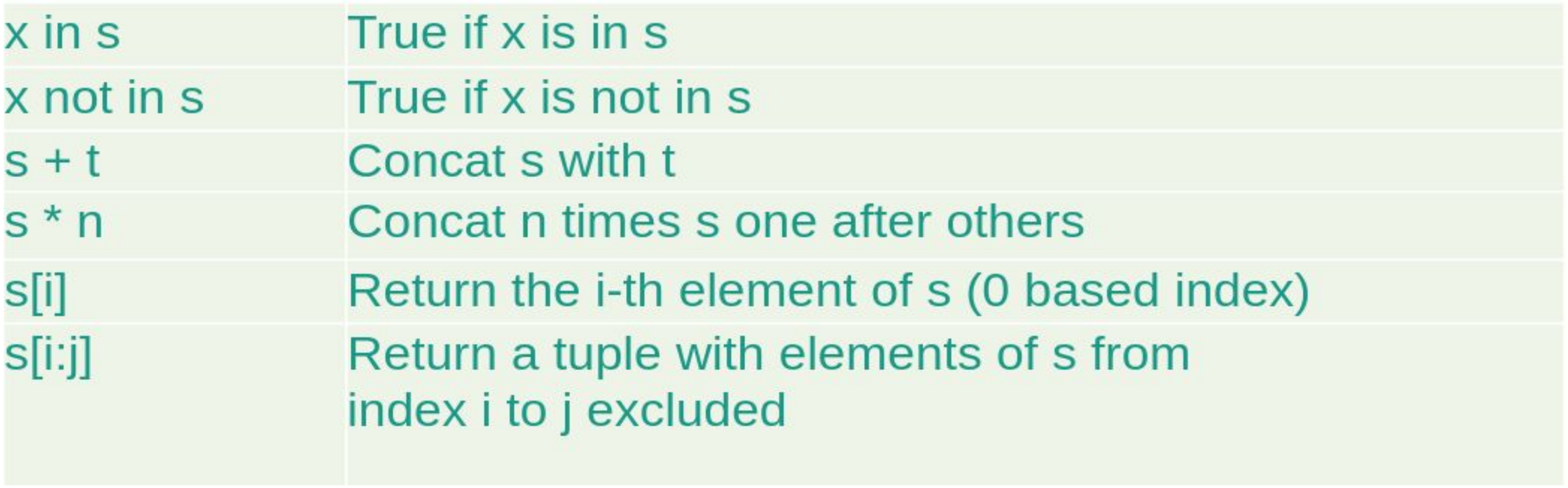
```
        [['a'], [], [], [], []]
```

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



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

Tuples : operations

s[i:j:k]	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, …..
len(s)	Return the number of elements of s
min(s)	Return the smallest element of s, éléments have to be comparable
max(s)	Return the greatest element of s
sum(s)	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 itselft (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

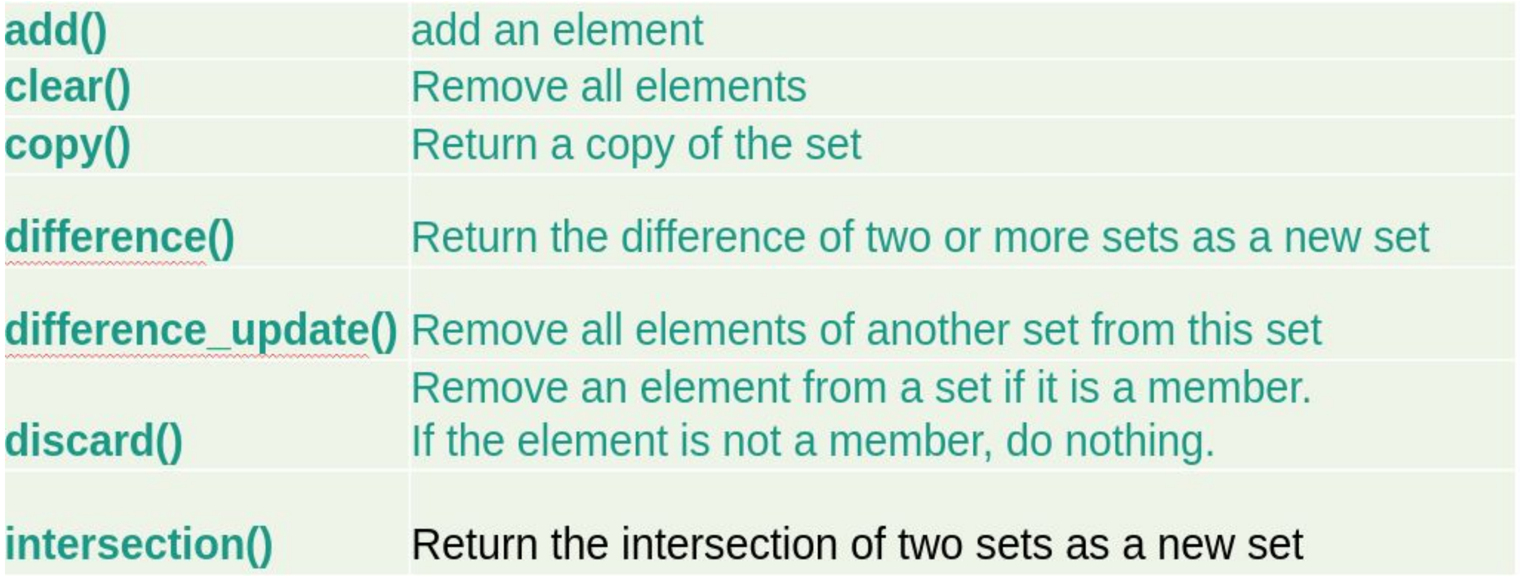
- A *set* is an unordered object collection nut ***with unique elements***.
- It's like a mathematical set with equivalent operations.
- A *frozenset* is an immutable set
- set() returns an empty list

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
difference()	Return the difference of two or more sets as a new set
difference_update()	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

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

Set : operations

symmetric_difference()	Return the symmetric difference of two sets as a new set
symmetric_difference_update()	Update a set with the symmetric difference of itself and another
union()	Return the union of sets as a new set
update()	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'
```

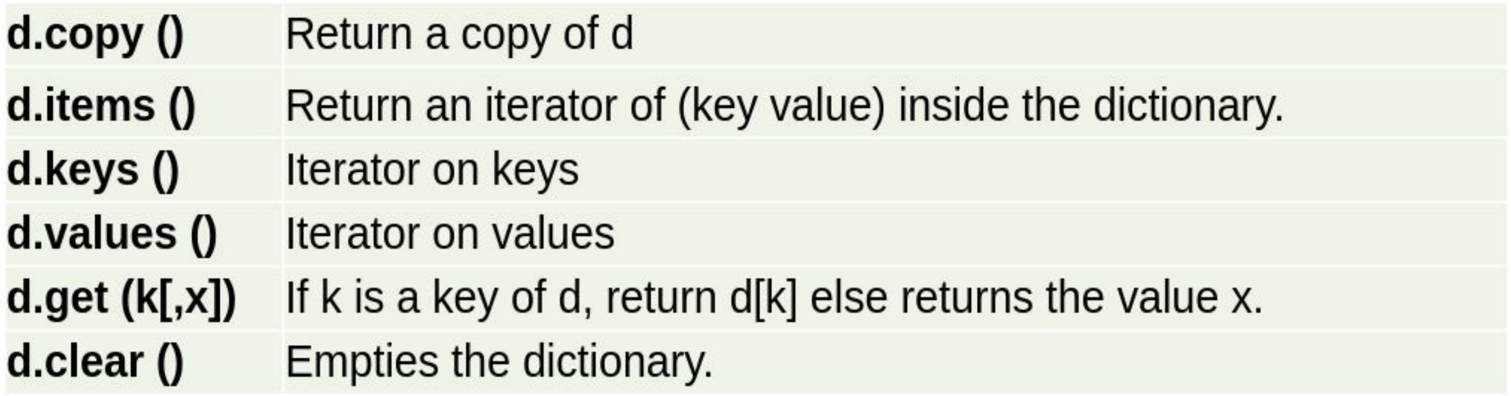
Dictionaries

```
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

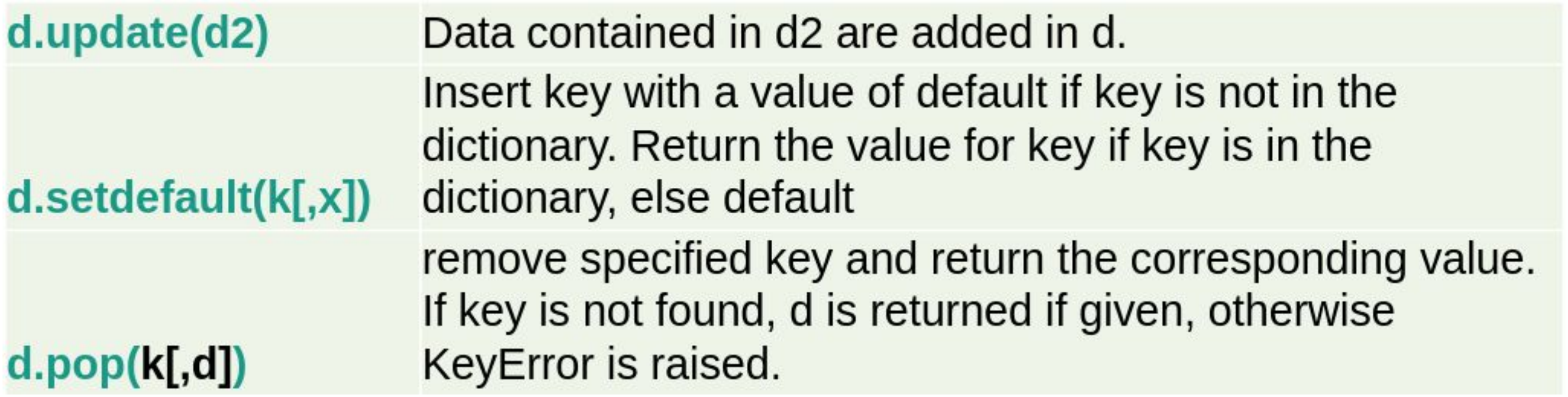
x in d	True if x is a key of d
x not in d	True if x is not a key of d
d[i]	Return the value associated to the key i
len(d)	Return the number of associations stored in d
min(d)	Return the smallest key
max(d)	Return the greatest key
del d [i]	Remove the couple of data associated to key i
list (d)	Return 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



d.update(d2)	Data contained in d2 are added in d.
d.setdefault(k[,x])	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
d.pop(k[,d])	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()
```

Reading

```
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

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 “”
```

Course outline

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

Course outline

OBJECT ORIENTED PROGRAMMING (OOP)

- OOP is the most widely used programming paradigm
- You can model and abstract away lots of things using OOP
- In Python, everything is an object ! Even a simple integer variable
- An object essentially holds two things: **data** and **methods** to run computations on that data
    - your class’ data is usually called attributes

OOP: Built-in **dir** Function

x = 2

x is an integer (int), which is an object in Python world.

In order to access its methods, you can use the built-in function dir

OOP: Example with int

```
>>> print(dir(x))
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__',
'__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__',
'__getattribute__', '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__',
'__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__',
'__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__',
'__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__',
'__truediv__', '__trunc__', '__xor__', 'as_integer_ratio', 'bit_length', 'conjugate',
'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

OOP: Examples - int, float, complex, str

```
>>> print(x.as_integer_ratio())
(2, 1)
```

Yes because 2 is equal to 2 divided by 1

```
>>> y = 3.5  # y is float
>>> print(y.as_integer_ratio())
(7, 2)
```

Yes because 3.5 is equal to 7 divided by 2

OOP: Examples - int, float, complex, str

```
>>> z = 1 + 3j
>>> print(z.real)  # displays z’s real part
1
>>> print(z.imag)  # displays z’s imaginary part
3

>>> s = “I AM ALL CAPITAL LETTERS”
>>> print(s.lower())
“i am all capital letters”
```

OOP: Examples - int, float, complex, str

Conclusion: You see how Python built-in objects like int, float, complex, str contain data (for example the data you assign to it when declaring a variable), and how it contains methods that lets you access certain part of the data or even modify it

OOP: Classes

Question: How do we create our own objects ?

Answer: We first define classes using the class keyword, and we then create objects (instances) of these classes.

For example, I may want to define the notion of a point in a 2-dimensional place, and I want to be able to compute the distance between two points in that space, compute their norm, etc … It does not come natively with the language but OOP allows me to create this abstraction layer.

OOP: Classes

A class generally contains AT LEAST these 2 methods:

- a constructor __init__:
  - your class attributes are introduced inside the constructor

- a destructor __del__:
  - usually implicitly defined by Python, you don't have to worry about it in 99.9% of cases -> just know it's there !

OOP: Classes - Point example

```
import math
class Point:
    def __init__(self, x, y):  # always pass the parameter “self”
        print(“Hey I am the constructor”)
        self.x = x
        self.y = y
    def l2_norm(self):
        return math.sqrt(self.x ** 2 + self.y ** 2)  # access attributes using self
    def l2_dist(self, p2):
        # p2 is a type Point and has the same attributes and method from the class # Point
        return return math.sqrt((self.x - p2.x) ** 2 + (self.y - p2.y) ** 2)
```

OOP: Classes - Point example

Now that we defined a class Point, let’s create two Point **objects** (we say **instances**) and see what we can do with them

```
>>> p1 = Point(1, 2)
Hey I am the constructor
>>> p2 = Point(3, 4)
Hey I am the constructor
>>> print(p1.x)
1
>>> print(p1.y)
2
>>> print(p1.l2_norm())
2.23606797749979  # just sqrt(5)
```

OOP: Classes - Point example

```
>>> print(p2.x)
3
>>> print(p2.y)
4
>>> print(p2.l2_norm())
5
>>> print(p1.l2_dist(p2))
2.8284271247461903  # just sqrt(8)
```

OOP: Classes - Inheritance

Classes can inherit other classes in a parent -> child relationship

For example using an Animal base class we could create two other classes that inherit from Animal, mainly Tiger and Monkey

All tigers are animals but tigers aren't monkeys, and the same holds for monkeys

Both Tiger and Monkey would inherit the Animal attribute but each implement their own specificities

OOP: Classes - Inheritance

```
class Animal:
    def __init__(self):
        self.has_fur = True
```

```
class Tiger(Animal):
    def __init__(self):
        self.is_vegetarian = False
```

```
class Monkey(Animal):
    def __init__(self):
        self.is_vegetarian = True
```

OOP: Classes - Inheritance

```
>>> tiger = Tiger()
>>> monkey = Monkey()
>>> print(tiger.has_fur)
True
>>> print(monkey.has_fur)
True
>>> print(tiger.is_vegetarian)
False
>>> print(monkey.is_vegetarian)
True
```

OOP: Classes - Operators

```
import math
class Point:
        def __init__(self, x, y):  # always pass the parameter “self”
                print(“Hey I am the constructor”)
                self.x = x
                self.y = y
        def l2_norm(self):
                return math.sqrt(self.x ** 2 + self.y ** 2)  # access attributes using self
        def l2_dist(self, p2):
                # p2 is a type Point and has the same attributes and method from the class # Point
                return return math.sqrt((self.x - p2.x) ** 2 + (self.y - p2.y) ** 2)
```

OOP: Classes - **__add__** operator

- The __add__ operator tells Python how to add two objects from the same class together.

- That way you can define addition on any object you'd like

- For math people: This essentially lets you attach a vector space structure on abstract object YOU define ! Pretty cool

Let's see how we do it using our previous Point class example

OOP: Classes - **__add__** operator

```
import math
class Point:
    def __init__(self, x, y):  # always pass the parameter “self”
        print(“Hey I am the constructor”)
        self.x = x
        self.y = y
    def __add__(self, p2):
        return Point(self.x + p2.x, self.y + p2.y)
    def l2_norm(self):
        return math.sqrt(self.x ** 2 + self.y ** 2)  # access attributes using self
    def l2_dist(self, p2):
        # p2 is a type Point and has the same attributes and method from the class # Point
        return return math.sqrt((self.x - p2.x) ** 2 + (self.y - p2.y) ** 2)
```

OOP: Classes - **__add__** operator

```
>>> p1 = Point(1, 2)
Hey I am the constructor
>>>  p2 = Point(3, 4)
Hey I am the constructor
>>> p3 = p1 + p2
<__main__.P object at 0x7f04839550a0>
>>>print(p3.x)
4
>>>print(p3.y)
6
```

p3 is just another Point object defined to be the sum of p1 and p2

OOP: Classes - Operators

There are lots of other operators, a few common ones are:

__add__ addition (+)

__mul__ multiplication (*)

__div__ division (/)

__lt__ less than (<) -> when you write 2 < 5 this is the operator being invoked

__lte__ less than or equal (<=)

__gt__ greater than (>)

__gte__ greater than or equal (>=)

__str__ string operator

etc…

See Python’s official documentation online for an extensive list of these operators

OOP: Classes - **__str__** operator

With implementing the __str__ operator:

```
class A:

        def __str__(self):

                return “This is my class A”

a = A()

>>> print(a)

This is my class A
```

Without implementing the __str__ operator

```
class A:

        pass

a = A()

>>> print(a)

<__main__.A object at 0x7fd82f55a3d0>
```