


40h

Introduction - Prise en main

 Les bases du langage Python

Programmation Orientée Objet

20h



Mini-projet

Mes sources :

Formations Python - Jean-Luc CHARLES et Eric DUCASSE



Cours PO S6 - 2022- Gilles GUETTE

Docs Python  [www](#) 

Cours Programmation Python – Laurent POINTAL



Diverses sources sur le web...

Cours Programmation Python – Éric BERTHOMIER



Le langage Python

- ▶ Le langage Python est constitué :
 - de **mots clefs**, qui correspondent à des instructions élémentaires (**for**, **if**...);
 - de **littéraux** : valeurs constantes de types variés (25, 1.e4, 'abc'...);
 - de **types intrinsèques** (**int**, **float**, **list**, **str**...);
 - d'**opérateurs** (=, +, -, *, /, %...);
 - de **fonctions intrinsèques** (*Built-in Functions*) qui complètent le langage.
- ▶ L'utilisateur peut créer :
 - des **classes** : nouveaux types qui s'ajoutent aux types intrinsèques;
 - des **objets** : entiers, flottants, chaînes, fonctions, programmes, modules... instances de classes définies par l'utilisateur ;
 - des **expressions** combinant identificateurs, opérateurs, fonctions...

3

Mots clefs du langage

Doc Python > Language Reference > Identifiers and keywords

docs.python.org/reference/lexical_analysis.html#keywords

Il n'y a que 35 mots clefs (*key words*) dans le langage Python 3.11

2.3.1. Keywords

The following identifiers are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

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

4

Mots clefs du langage

Boucles for boucle while boucle continue tour suivant break sortie de boucle	Définitions d'objets def définition d'une fonction return fin fonction, renvoyer valeur class définition d'une classe nonlocal change la portée d'une variable global définit une variable globale
Tests if test else alternative elif test combiné	Opérateurs logiques and ET logique or OU logique not négation

5

Les opérateurs

Principaux opérateurs

Opérateurs arithmétiques		
+		addition
-		soustraction
*		mutiplication
/		division (Python3 : toujours division flottante)
//		quotient de la division entière
**		puissance
%		modulo
<< n, >> n		décalage à gauche, à droite de n bits

6

Principaux opérateurs

Opérateurs de comparaison	
<, <=	inférieur strict à, inférieur ou égal à
>, >=	supérieur strict à, supérieur ou égal à
==	égal à
!=	différent de
Opérateurs logiques	
and	ET
not	négation
or	OU
&	ET bit à bit
^	XOR bit à bit
	OR bit à bit
~	complément à 2

7

Principaux opérateurs

Autres Opérateurs	
is	même identité?
is not	identité différente?
in	appartient à?
not in	n'appartient pas à?
Opérateurs avec affectation	équivalence
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y
x //= y	x = x // y
x **= y	x = x ** y
x %= y	x = x % y

8

Les Fonctions Intrinsèques

Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

docs.python.org/library/functions.html

Built-in Functions			
A	E	L	R
abs()	enumerate()	len()	range()
aiter()	eval()	list()	repr()
all()	exec()	locals()	reversed()
any()			round()
anext()	F	M	S
ascii()	filter()	map()	set()
	float()	max()	setattr()
B	format()	memoryview()	slice()
bin()	frozenset()	min()	sorted()
bool()			staticmethod()
breakpoint()	G	N	str()
bytearray()	getattr()	next()	sum()
bytes()	globals()		super()
		O	T
C	H	object()	tuple()
callable()	hasattr()	oct()	type()
chr()	hash()	open()	
classmethod()	help()	ord()	
compile()	hex()	P	V
complex()		pow()	vars()
	I	print()	
D	id()	property()	Z
delattr()	input()		zip()
dict()	int()		
dir()	isinstance()		
divmod()	issubclass()		
	iter()		
			__import__()

9

Les Entrées/Sorties clavier, écran (Input/Output)

- ▶ Les Entrée/Sortie clavier/écran permettent le dialogue programme/utilisateur.
- ▶ La fonction `input(' message')` est utilisée :
 - pour afficher un message à l'écran;
 - pour **capturer** la **saisie clavier** et la renvoyer comme un `str`.

```
>>> rep = input("Entrer un nombre: ")
Entrer un nombre : 47
>>> rep
'47'
>>> rep == 47
False
>>> type(rep)
<class 'str'>
>>> x = float(rep); x
47.0
>>> type(x); x == 47
<class 'float'>
True
```

▶ `input` renvoie un `str`

▶ `float` permet de convertir un `str` en nombre flottant.

10

Les Entrées/Sorties clavier, écran (*Input/Output*)

- Formatage des chaînes de caractères avec la méthode `str.format`

```
"chaîne de formatage contenant des {...}".format(objet1, objet2, ...)
```

```
>>> print("nom: {:s}, age: {:4d}, taille: {:5.2f} m".format("Smith", 42, 1.73))
nom: Smith, age:  42, taille: 1.73 m
>>> print("nom: {1:s}, age: {2:4d}, taille: {0:5.2f} m".format(1.73, "Smith", 42))
nom: Smith, age:  42, taille: 1.73 m
```

- Exemples de spécifications de formatage :

.nf	n décimales, format flottant
.ne	n décimales, format scientifique
n.de	n caractères avec d décimales, format scientifique
s	chaîne de caractères
d	entier en base 10
g	choisit le format le plus approprié

- Le formatage "à la printf du C" peut aussi se faire avec l'opérateur %

```
>>> x=1.2e-3 ; print("la variable %s a pour valeur: %8.3E" % ("x", x))
la variable x a pour valeur: 1.200E-03
```

Exercices des chapitres 1 et 2 (3 en option)

11

Types intrinsèques conteneurs

- Collections ordonnées : les séquences (*sequences*)

les listes `list`

les tuples `tuple`

les chaînes de caractères `str`

- Collections sans ordre

les dictionnaires `dict`

les ensembles `set`

Tous les conteneurs Python sont des objets **itérables** : on peut les parcourir avec une boucle `for`.

12

- Collection **ordonnée** d'**objets quelconques** (conteneur hétérogène).
- Une liste n'est pas un vecteur (\sim classe `ndarray` du module `numpy`).
- `len(•)` renvoie le nombre d'éléments de la liste `•`.
- **Indexation** : `•[i]` renvoie l'élément de rang `i` de la liste `•`
 - le premier élément de la liste `•` a le rang 0, le dernier le rang `len(•)-1`.

```
>>> L1 = [10, 11, 12, 13]
>>> L1.append(14)
>>> L1
[10, 11, 12, 13, 14]
>>> L1[0]
10
>>> L1[-1]
14
>>> len(L1)
5
```

```
>>> L1[4]
14
>>> L1[5]
...IndexError: list index out of range
>>> L1[-4]
11
>>> L1[-5]
10
>>> L1[-6]
...IndexError: list index out of range
```

13

- **Sous-liste (Slicing)** :

<code>•[i:j]</code>	\sim sous-liste de <code>i</code> inclus à <code>j</code> exclu
<code>•[i:j:k]</code>	\sim sous-liste de <code>i</code> inclus à <code>j</code> exclu , par pas de <code>k</code>

- Indexation positive, **négative** :

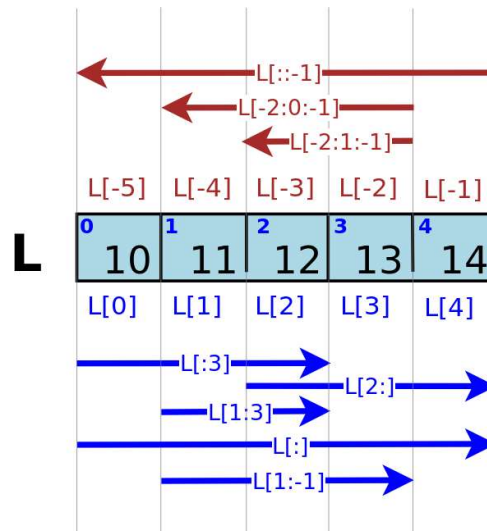
`0 ≤ i ≤ len(•)-1` \sim rang dans la liste

`-len(•) ≤ i ≤ -1` \sim -1 : le dernier, -2 : l'avant dernier...

```
>>> L1 = [10, 11, 12, 13, 14]
>>> L1[1:3]
[11, 12]
>>> L1[:3]
[10, 11, 12]
>>> L1[1:]
[11, 12, 13, 14]
>>> L1[:] # toutes les valeurs de L1
[10, 11, 12, 13, 14]
```

```
>>> L1 = [10, 11, 12, 13, 14]
>>> L1[::-2]
[10, 12, 14]
>>> L1[::-1]
[14, 13, 12, 11, 10]
>>> L1[-2:1]
[]
>>> L1[-2:1:-1]
[13, 12]
```

14



15

Copie : **Référence**, **copie superficielle** ou **copie profonde**

`L2 = L1` nom supplémentaire L2 pour l'objet nommé L1

```
>>> L1 = [10, 11, 12] ; L2 = L1      # L1 et L2 : 2 noms d' un même objet
>>> id(L1), id(L2), L2 is L1
(50660560, 50660560, True)
>>> L2[0] = 0
>>> L1, L2
([0, 11, 12], [0, 11, 12])
```

Copie : **Référence**, **copie superficielle** ou **copie profonde**

`L3 = L1[:]` nouvel objet **list** nommé L3, initialisé par L1 (*shallow copy*)

```
>>> L1 = [10, [11, 12]] ; L3 = L1[:]  # Shallow copy !
>>> id(L1), id(L3), L3 is L1
(50660560, 51420768, False)
>>> L3[0] = 0; L3[1][0] = -1
>>> L1, L3
([10, [-1, 13]], [0, [-1, 13]])
```

16

Copie : Référence, copie superficielle ou **copie profonde**

```
from copy import deepcopy
L4 = deepcopy(L1)
```

nouvel objet **list** nommé L4, copie complète de l'objet L1 (*deep copy*)

```
>>> L1 = [10, [11, 12]]
>>> from copy import deepcopy
>>> L4 = deepcopy(L1)           # Deep copy !
>>> id(L1), id(L4), L4 is L1
(53980336, 53982200, False)
>>> L4[0] = 0
>>> L4[1][0] = -1
>>> L1, L4
([10, [12, 13]], [0, [-1, 13]])
```

17

- Une liste est **itérable**, on peut la parcourir avec une boucle **for** :

```
>>> L1 = [1, 2, 3]
>>> for a in L1:
>>>     print(a+2)
```

```
3
4
5
```

```
>>> col = ["red", "green", "blue"]
>>> for i, c in enumerate(col):
>>>     print(i, "Color: " + c)
```

```
0 Color: red
1 Color: green
2 Color: blue
```

- L'opérateur **+** concatène les listes :

```
>>> L1 = [1, 2, 3]
>>> L2 = [4, "a", 5]
>>> L1 + L2
[1, 2, 3, 4, 'a', 5]
```

- L'opération **list*n** ou **n*list** concatène **n** copies de la liste :

```
>>> [0]*10
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

18

- La classe `list` possède des méthodes utiles
(cf `dir(list)` et `help(list.)`)

```
>>> help(list.remove)
Help on method_descriptor:
remove(...)
    L.remove(value) -> None -- remove first occurrence of value.
    Raises ValueError if the value is not present.
```

```
>>> L1 = [10, 11, 12, 13, 14]
>>> del L1[2]
>>> L1
[10, 11, 13, 14]
>>> L1.remove(11)
>>> L1
[10, 13, 14]
>>> L2 = [10, 11, 10, 13, 10]
>>> L2.count(10)
3
```

```
>>> L2.remove(10)
>>> L2
[11, 10, 13, 10]
>>> L2.remove(10)
>>> L2
[11, 13, 10]
>>> L2.remove(10)
>>> L2
[11, 13]
>>> L2.remove(10)
... list.remove(x): x not in list
```

19

- `dir(*)` permet d'afficher "ce qu'il y a dans" * (classe, objet, module...).
- On peut utiliser toutes les méthodes (publiques) d'une classe sur un objet instance de la classe : `objet.methode(...)`

```
>>> L1 = [4, 2, 1, 3] # L1 est un objet liste (instance de la classe liste)
>>> type(L1)
<class 'list'>
>>> dir(L1)
['_add_', ..., 'append', 'clear', 'copy', 'count', 'extend', 'index',
 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> dir(list)
['_add_', ..., 'append', 'clear', 'copy', 'count', 'extend', 'index',
 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> L1.sort() ; L1 # objet.méthode_de_la_classe
[1, 2, 3, 4]
>>> L1.append(5) ; L1 # objet.méthode_de_la_classe
[1, 2, 3, 4, 5]
>>> L1.reverse() ; L1 # objet.méthode_de_la_classe
[5, 4, 3, 2, 1]
```

20

Séquences : la Classe **tuple**

- Un tuple est une liste **non mutable** :
 - ses **éléments** ne supportent pas la **ré-affectation** ;
 - mais ses **éléments mutables** peuvent être **modifiés**.

```
>>> a = ()          # tuple vide
>>> b = (2,)        # 1-tuple
>>> c = 2,          # 1-tuple
>>> d = 2, 3, 4     # tuple implicite
>>> d
(2, 3, 4)
>>> t=(1, "non", [1,2,3])
>>> t
(1, 'nm', [1, 2, 3])
>>> t[0]=2 # ré-affectation élément
...TypeError: 'tuple' object does
not support item assignment

>>> t[1]="oui" # ré-affectation élément!
...TypeError: 'tuple' object does not
support item assignment
>>> t[1][0]="N" # élément str non mutable
...TypeError: 'str' object does not
support item assignment
>>> t[2]=[3,4,5] # ré-affectation élément
...TypeError: 'tuple' object does not
support item assignment
>>> t[2][0]=-3 # élément list mutable
>>> t
(1, 'nm', [-3, 2, 3])
```

21

Séquences : la Classe **str**

- Chaîne de caractères = **liste** de caractères...
- Même principe d'indexation que les objets **list** :

```
>>> s = "abcdef"
>>> s[0], s[-1]
('a', 'f')
>>> s[:]
'abcdef'
>>> s[1:-1], s[1:-1:2]
('bcde', 'bd')
```

- De nombreuses méthodes utiles sont proposées par la classe **str** :

```
>>> dir(str)
[... 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

22

Collection non-ordonnée : la Classe **dict**

- Collection sans ordre de paires clef, objet : sert à retrouver un objet par sa clef

```
>>> cyll = {"L": 1.2, "D": 0.5, "unit": "m"} ; cyll
{'L': 1.2, 'unit': 'm', 'D': 0.5}
>>> cyll["L"]
1.2
>>> cyll["D"] = 0.6 ; cyll
{'D': 0.6, 'L': 1.2, 'unit': 'm'}
```

- La classe **dict** possède plusieurs méthodes utiles (**dir(dict)** et **help(dict.xxx)...**)

```
>>> cyll["d"] = 0.1 ; cyll      # création de la clef !
{'d': 0.1, 'L': 1.2, 'unit': 'm', 'D': 0.6}
>>> "mass" in cyll, "D" in cyll
(False, True)
>>> del cyll["d"]
>>> list(cyll.items())
[('L', 1.2), ('unit', 'm'), ('D', 0.6)]
>>> list(cyll.keys())
['L', 'unit', 'D']
>>> list(cyll.values())
[1.2, 'm', 0.6]
```

23

Collection non-ordonnée : la Classe **dict**

- Un dictionnaire est **itérable** : on peut le parcourir avec une boucle **for**.

```
>>> for k in cyll:
    print(k)
L
unit
D
>>> for k, v in cyll.items():
    print(k, v)
L 1.2
unit m
D 0.6
```

- méthode **get** pour spécifier la valeur à renvoyer si la clef est absente :

```
>>> x = cyll['age']
Traceback (most recent call last):
...
KeyError: 'age'
>>> x = cyll.get('age', 0)
>>> x
0
```

24

Collection non-ordonnée : la Classe **set**

- Collection non-ordonnée d'**items uniques**

```
>>> s = set([1, 5, 12, 6])
>>> t = set(' Lettres ')
>>> t
{' ', 'e', 'L', 's', 'r', 't'}
>>> u = {"a", "b", "c"} ; u
{'b', 'a', 'c'}
```

- Les objets **set** sont créés avec le constructeur de la classe **set** et un argument de type **list**
- On peut aussi énumérer les éléments entre accolades.

- La classe **set** propose des opérations ensemblistes

```
>>> t = set([5, 6, 7])
>>> s | t      # Union
{1, 5, 6, 7, 12}
>>> s & t      # Intersection
{5, 6}
>>> s - t      # dans s, mais pas dans t
{1, 12}
>>> s ^ t      # dans s ou (exclusif) t
{1, 12, 7}
```

Exercices du chapitre 4

25

Mots clefs du langage

Boucles <div> <div>for</div> <div>boucle</div> </div> <div> <div>while</div> <div>boucle</div> </div> <div> <div>continue</div> <div>tour suivant</div> </div> <div> <div>break</div> <div>sortie de boucle</div> </div>	Définitions d'objets <div> <div>def</div> <div>définition d'une fonction</div> </div> <div> <div>return</div> <div>fin fonction, renvoyer valeur</div> </div> <div> <div>class</div> <div>définition d'une classe</div> </div> <div> <div>nonlocal</div> <div>change la portée d'une variable</div> </div> <div> <div>global</div> <div>définit une variable globale</div> </div>
Tests <div> <div>if</div> <div>test</div> </div> <div> <div>else</div> <div>alternative</div> </div> <div> <div>elif</div> <div>test combiné</div> </div>	Opérateurs logiques <div> <div>and</div> <div>ET logique</div> </div> <div> <div>or</div> <div>OU logique</div> </div> <div> <div>not</div> <div>négation</div> </div>

26

Définition et appel des fonctions

- ▶ **Définition** : mot clef `def` ... terminé par le caractère `:`
 - ▶ le corps de la fonction est **indenté** d'un niveau (bloc indenté);
 - ▶ les objets définis dans le corps de la fonction sont locaux à la fonction;
 - ▶ une fonction peut renvoyer des objets avec le mot clef `return`.
- ▶ **Appel** : nom de la fonction suivi des parenthèses (. .)

```
>>> q = 0
>>> def divide(a, b):
    q = a // b
    r = a - q*b
    return q, r

>>> x, y = divide(5, 2)
>>> q, x, y
(0, 2, 1)
```

- ▶ La variable `q` est définie en dehors de la fonction (valeur = 0)
- ▶ L'opérateur `//` effectue une division entière
- ▶ La variable `q` définie dans la fonction est locale !

27

Définition et appel des fonctions

- ▶ **Portée d'une variable** : mots clefs `global`, `local`, `nonlocal`
 - ▶ `local`, spécifique à une fonction (et n'en sort pas);
 - ▶ `nonlocal` : peut sortir mais n'est pas globale (donc reste interne à la fonction mère);
 - ▶ `global` : là c'est disponible pour tout votre script ou code.

```
x=0 #défini en globale par défaut
def externe():
    x = 1 #défini en locale par défaut
    def interne():
        nonlocal x #nonlocal, se réfère à la variable de la fonction mère
        x=2
        print("interne:",x)

    interne()
    print("externe:",x)
externe()
print("globale:",x)
```

✓ 0.0s Python

```
interne: 2
externe: 2
globale: 0
```

28

Définition et appel des fonctions

► Passage des arguments par référence

- à l'appel de la fonction, les arguments (objets) sont transmis par référence;
- un objet mutable transmis en argument peut être modifié par la fonction;
- un objet non mutable transmis en argument ne peut pas être modifié par la fonction.

<pre>>>> def f0(a, b): a = 2 b = 3 print("f0-> a: {}, b: {}".format(a,b)) return >>> a=0; b=1 >>> a, b (0, 1) >>> f0(a, b) f0-> a: 2, b: 3 >>> a, b (0, 1)</pre>	<pre>>>> def f1(a): for i,e in enumerate(a): a[i] = 2.*e return >>> a=[1, 2, 3] >>> id(a) 140693918338312 >>> f1(a) >>> id(a) 140693918338312 >>> print(a) [2.0, 4.0, 6.0]</pre>
---	--

29

Définition et appel des fonctions

► Arguments positionnels

- à l'appel de la fonction, les arguments passés sont des objets
- chaque objet correspond au paramètre de même **position** (même **rang**).

```
>>> def f2(a, b, c):
    print("a: {}, b: {}, c: {}".format(a,b,c))
    return
>>> f2(0) ...TypeError: f2() missing 2 required positional
           arguments: 'b' and 'c'
>>> f2(0,1) ...TypeError: f2() missing 1 required positional
           argument: 'c'
>>> f2(0,1,2)
a: 0, b: 1, c: 2
```

30

Définition et appel des fonctions

► Arguments nommés

- à l'appel, les arguments passés sont des affectations des noms des arguments;
- l'ordre de passage des arguments nommés est indifférent;
- on peut mixer argument(s) positionnel(s) et argument(s) nommé(s);
- souvent utilisé avec des paramètres ayant des valeurs par défaut;
- après un argument ayant une valeur par défaut, tous ceux qui suivent doivent aussi avoir une valeur par défaut.

<pre>>>> f2(c=3, a=1, b=2) a: 1, b: 2, c: 3 >>> def f3(a, b=2, c=2): print("a: {}, b: {}, c: {}".format(a, b, c)) return >>> f3(0) a: 0, b: 2, c: 2</pre>	<pre>>>> f3(c=1, a=2, b=3) a: 2, b: 3, c: 1 >>> f3(0, c=5, b=4) a: 0, b: 4, c: 5 >>> f3(0, c=5) a: 0, b: 2, c: 5</pre>
--	---

31

Définition et appel des fonctions

► Arguments multiples avec le caractère *

Utilisé quand on ne connaît pas à l'avance le nombre d'arguments qui seront passés à une fonction.

```
def f4(*x):
    print("argument reçu :", x)
    return
```

```
>>> f4(1, 2, "a")
argument reçu: (1, 2, 'a')      # 3 arguments reçus
>>> L=[5, "b", 6]
>>> f4(L)
argument reçu: ([5, 'b', 6],)   # UN seul argument : la liste L
>>> f4(*L)
argument reçu: (5, 'b', 6)      # *L permet de 'défaire' (unpacking) L
                                # 3 arguments reçus
```

32

Définition et appel des fonctions

► Arguments multiples : caractères **

- utilisé pour transmettre un nombre quelconque d'arguments nommés;
- la fonction reçoit un objet de type `dict`;
- utilise le caractère spécial `**`.

```
def f5(**x):
    print("argument(s) reçu(s):", x)
    return
```

```
>>> f5(a=2, b=1, z=2)
argument(s) reçu(s): {'a': 2, 'b': 1, 'z': 2}
>>> d={"age":22, "poids":54, "yeux":"bleu"}
>>> f5(**d)
argument(s) reçu(s): {'yeux': 'bleu', 'age': 22, 'poids': 54}
```

Exercices du chapitre 5

33

Documentation d'une fonction

Pour écrire la documentation d'une fonction, il suffit d'ajouter une chaîne de caractères directement à la suite de la déclaration de la signature de la fonction. On appelle ce texte la **docstring**. C'est ce texte qui est affiché lorsqu'on passe le nom de la fonction à la fonction `help()`.

```
>>> def dire_bonjour_a(nom):
|     """Affiche un message qui dit bonjour au nom qui est passé en paramètre"""
|     print ("Bonjour", nom)
|
|     help(dire_bonjour_a)
|
| ✓ 0.0s Python
```

Help on function dire_bonjour_a in module __main__:

```
dire_bonjour_a(nom)
    Affiche un message qui dit bonjour au nom qui est passé en paramètre
```

Des extension existent pour faciliter la rédaction de ces docstring



```
1 def division(a,b):
2     """summary_
3
4     Args:
5         a (_type_): _description_
6         b (_type_): _description_
7
8     Returns:
9         _type_: _description_
10
11     return a/b
```

34

Mots clefs du langage - suite -

importation d'un Module <code>import</code> module entier <code>from</code> objets d'un module	Objets <code>del</code> détruire une référence sur un objet <code>in</code> parcourir <code>is</code> comparer
gestion des Exceptions <code>assert</code> définir une assertion <code>raise</code> créer une exception <code>try</code> traitement exception <code>except</code> traitement exception <code>finally</code> traitement exception	Autre <code>pass</code> ne rien faire... <code>with</code> objet dans un contexte <code>as</code> utilisé avec <code>with</code> , <code>import</code> ... <code>yield</code> fonction générateur <code>lambda</code> fonction <i>inline</i>

35

Les Modules

- Un **module** est un **fichier Python**, contenant constantes, fonctions, classes...
- Un module est importé avec `import nomModule`
 - création d'un **espace de nom** (*namespace*) qui contiendra (préfixe) tous les objets contenus dans le module
 - **toutes les définitions contenues dans le module sont exécutées**
 - un identifiant (`nomModule`) est associé à l'espace de nom du module, dans le programme appelant.

Fichier test.py :

```
a = 12
def bonjour():
    print("Hello World!")
    return
```

Import du module test :

```
>>> import test
>>> x = test.a; print(x)
12
>>> test.bonjour()
Hello World!
```

36

Les Modules

- Variations possibles pour importer tout ou partie d'un module :

- Module **entier** importé avec son espace de nom (*alias* possible) :

```
import module [as alias]
```

- Module **entier** importé, dans l'espace de nom courant :

```
from module import *
```

- **Sélection d'items** du module, importés dans l'espace de nom courant :

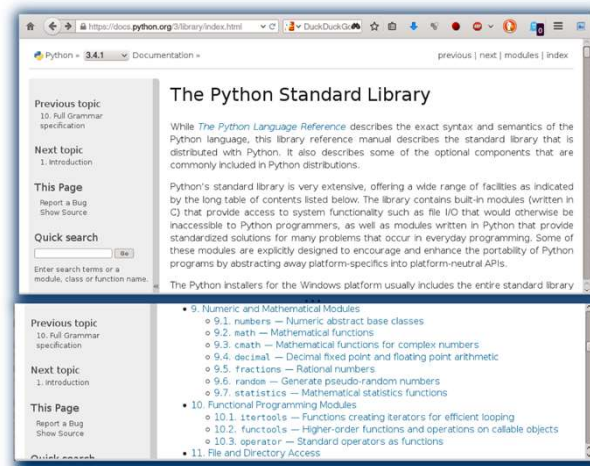
```
from module import item1 [as alias], item2 [as alias]...
```

<pre>>>> import math >>> math.sin(math.pi/4) 0.7071067811865475 >>> import math as m >>> m.sin(m.pi/4) 0.7071067811865475</pre>	<pre>>>> from math import * >>> pi 3.141592653589793 >>> sin(pi/4) 0.7071067811865475 >>> log(1) 0.0</pre>	<pre>>>> from math import pi, sin, log >>> pi 3.141592653589793 >>> sin(pi/4) 0.7071067811865475 >>> log(1) 0.0</pre>
---	--	---

37

Les Modules de la bibliothèque standard (plus de 200... [_](#))

- Doc en ligne Python *Library Reference* docs.python.org/library/index.html



38

Module standard **math**

- Donne accès aux constantes et fonctions mathématiques usuelles (trigonométriques, hyperboliques, log, exp...)

```
>>> import math
>>> dir(math)
['_doc_', '_name_', '_package_', 'acos', 'acosh', 'asin', 'asinh',
'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees',
'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod',
'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma',
'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh', 'trunc']
```

39

Module standard **os**

- Propose des fonctions pour interagir avec le système d'exploitation.
Exemple : manipulation des répertoires et des fichiers

```
>>> import os
>>> dir(os)
['... ' chdir', ..., ' getcwd', ..., ' listdir', ..., ' mkdir', ..., ' rmdir' ]
```

- **chdir** (*change directory*) : change le répertoire de travail D
- **getcwd** (*get current directory*) : renvoie le répertoire de travail D
- **listdir** (*list directory*) : liste le contenu d'un répertoire ('.' : le répertoire

```
>>> os.getcwd()
'/home/jlc'
>>> os.chdir(' /tmp/data' )
>>> rep = os.getcwd(); rep
'/tmp/data'
>>> os.listdir(rep)
[' f1.txt', ' f2.dat', ' f1.dat' ]
>>> ' f3.dat' in os.listdir(rep)
False
>>> ' f1.dat' in os.listdir(rep)
True
```

```
>>> os.getcwd()
'C:\\Users\\jlc'
>>> os.chdir(' C:/tmp/data' )
>>> rep = os.getcwd(); rep
'C:\\temp\\data'
>>> os.listdir(rep)
[' f1.dat', ' f2.dat', ' f2.txt' ]
>>> ' f3.dat' in os.listdir(rep)
False
>>> ' f1.dat' in os.listdir(rep)
True
```

40

Les exceptions

Lorsqu'un programme génère une erreur, le mécanisme des **exceptions** peut permettre au programme de « rattraper » les erreurs, de détecter qu'une erreur s'est produite et d'agir en conséquence afin que le programme ne s'arrête pas

<pre>a=2 b=a+c print(b)</pre> <p>0.0s</p> <p>NameError Cell In[16], line 2 1 a=2 ----> 2 b=a+c 3 print(b)</p> <p>NameError: name 'c' is not defined</p>	<pre>a=2 b=a+'e' print(b)</pre> <p>0.0s</p> <p>TypeError Traceback (most recent call last): Cell In[17], line 2 1 a=2 ----> 2 b=a+'e' 3 print(b)</p> <p>TypeError: unsupported operand type(s) for +: 'int' and 'str'</p>	<pre>a=0 b=1/a print(b)</pre> <p>0.0s</p> <p>ZeroDivisionError Cell In[18], line 2 1 a=0 ----> 2 b=1/a 3 print(b)</p> <p>ZeroDivisionError: division by zero</p>
---	---	--

Les exceptions les plus fréquentes :

IndentationError	SyntaxError	AttributeError	ImportError
NameError	IndexError	KeyError	TypeError
ValueError	OSError	...	

41

try...except...

Les clauses **try** et **except** fonctionnent ensemble. Elles permettent de tester (try) un code qui peut potentiellement poser problème et de définir les actions à prendre si une exception est effectivement rencontrée (except).

```
try:
    x = int(input("Entrez le numérateur : "))
    y = int(input("Entrez le dénominateur : "))
    print("Résultat de la division : ",x/y)
except ValueError :
    print("la valeur entrée doit être un entier")
except ZeroDivisionError :
    print("division par zéro impossible")
```

Python

```
Entrez le numérateur : 2
Entrez le dénominateur : 3
Résultat de la division : 0.6666666666666666
```

```
Entrez le numérateur : 3
Entrez le dénominateur : z
la valeur entrée doit être un entier
```

```
Entrez le numérateur : 3
Entrez le dénominateur : 0
division par zéro impossible
```

42

try...except...else...finally

```
try:
    # ... instructions à protéger
except type_exception_1:
    # ... que faire en cas d'erreur de type type_exception_1
except (type_exception_i, type_exception_j):
    # ... que faire en cas d'erreur de type type_exception_i ou type_exception_j
except type_exception_n:
    # ... que faire en cas d'erreur de type type_exception_n
except:
    # ... que faire en cas d'erreur d'un type différent de tous
    #   Les précédents types
else:
    # ... que faire lorsqu'aucune erreur n'est apparue
finally:
    #... sera toujours exécuté, erreur ou pas
```