

Module EA4 – Éléments d'Algorithmique II

Micro-cours de PYTHON

Dominique Poulalhon

`dominique.poulalhon@irif.fr`

Université Paris Cité

L2 Informatique & DL Info-Bio, Info-Jap, Math-Info

Année universitaire 2023-2024

PRÉSENTATION RAPIDE

un langage interprété

- lancement de l'interpréteur interactif, pratique pour faire de petits tests rapides : exécuter `python3`¹(ou `ipython3`, ou un IDE comme `pyzo`, ou un notebook en ligne (`Jupyter`, `Google Colab`)...)

```
>>> a = 2
>>> a *= 2
>>> a
4
>>> a *= 2
>>> a
8
```

- scripts dans des fichiers avec comme première ligne la référence de l'interpréteur, par exemple : `#!/usr/bin/python3`
ou (plus portable) : `#!/usr/bin/env python3`

1. `python` correspond à la version 2 de `PYTHON`, qui n'est pas compatible avec la version courante (version 3), donc à éviter

PRÉSENTATION RAPIDE

langage conçu pour être *lisible* et *épuré* :

- peu de constructions syntaxiques
- blocs marqués uniquement par l'indentation (qui est donc cruciale)

langage de haut-niveau permettant l'écriture rapide de prototypes :

- pas de déclaration de variables
- typage dynamique
- types de base très souples et puissants
- nombreux modules spécialisés

langage multiparadigme : à la fois impératif, objet et fonctionnel

LE TYPAGE

typage dynamique : déterminé à l'exécution et évolutif

```
>>> a = 8
>>> type(a)
<class 'int'>
>>> a += 3.5
>>> type(a)
<class 'float'>
```

LE TYPAGE

typage dynamique : déterminé à l'exécution et évolutif

```
>>> a = 8
>>> type(a)
<class 'int'>
>>> a += 3.5
>>> type(a)
<class 'float'>
```

typage fort : on ne peut pas mélanger les types n'importe comment

```
>>> a = 8
>>> "a_vaut_" + a ## provoque une erreur de type
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>> "a_vaut_" + str(a)
a vaut 8
```

LES TYPES PRINCIPAUX

- `int` : entiers *de longueur non bornée*
- `float` : flottants sur 64 bits
- `bool` : booléens `True` et `False`
- `str` : chaînes de caractères
- `list` : listes, non nécessairement homogènes
- `tuple` : k-uplets
- `set` : ensembles
- `dict` : dictionnaires (ou tables d'association)

LES ENTIERS

```
>>> p = 2
>>> p **= p ; p # mettre p à la puissance p
4
>>> p **= p ; p
256
>>> p **= p ; p
32317006071311007300714876688669951960444102669715484032130345
42752465513886789089319720141152291346368871796092189801949411
95591504909210950881523864482831206308773673009960917501977503
89652106796057638384067568276792218642619756161838094338476170
47058164585203630504288757589154106580860755239912393038552191
43333896683424206849747865645694948561760353263220580778056593
31026192708460314150258592864177116725943603718461857357598351
15230164590440369761323328723122712568471082020972515710172693
13234696785425806566979350459972683529986382155251663894373355
43602135433229604645318478604952148193555853611059596230656
```

LES CHAÎNES DE CARACTÈRES

```
>>> chaine = 'cou'+ "cou" # concaténation (noter les 2 syntaxes)
>>> chaine[3]              # accès comme dans un tableau
'c'
>>> chaine[3] = 'd'        # mais non mutable (ie modifiable)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> chaine.capitalize()    # objets, avec des méthodes
'Coucou'
>>> chaine = """ coucou
... """                    # définir des chaînes sur plusieurs lignes
>>> len(chaine)
12
>>> chaine[11]
'\n'
>>> chaine.strip()         # enlever les blancs de part et d'autre
'coucou'
```

Remarque : pas de type « caractère »

LES LISTES

manipulables comme des tableaux *et* comme des listes chaînées

```
>>> l = [ 1, 3, 7, 'coucou', 4]
>>> l[2]
7
>>> l[1:4]
[3, 7, 'coucou']
>>> l.insert(2, "salut")
>>> l
[1, 3, 'salut', 7, 'coucou', 4]
>>> l.append(3)
>>> l
[1, 3, 'salut', 7, 'coucou', 4, 3]
>>> l.count(3)
2
```

LES k-UPLETS (*tuples*)

similaires aux listes *mais* non mutables

```
>>> t = (1, 3, 7, 'coucou', 4)
>>> t[2]
7
>>> t[1:4]
(3, 7, 'coucou')
>>> t[2] = 5                                # objet non modifiable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t += (1,2,3)                             # création d'un nouvel objet
>>> t
(1, 3, 7, 'coucou', 4, 1, 2, 3)
>>> t.count(3)
2
```

LES CONDITIONNELLES

en **PYTHON**, les blocs sont définis uniquement par l'**indentation**

```
if test1 :  
    # bloc d'instructions  
elif test2 :  
    # bloc d'instructions  
elif test3 :  
    # bloc d'instructions  
...  
else :  
    # bloc d'instructions
```

(seul le premier bloc est obligatoire)

LES BOUCLES

en **PYTHON**, les blocs sont définis uniquement par l'**indentation**

```
while test :  
    # bloc d'instructions
```

```
for elt in sequence_iterable : # liste, tuple, chaîne...  
    # bloc d'instructions
```

LES BOUCLES

```
while test :  
    # bloc d'instructions
```

```
for elt in sequence_iterable : # liste, tuple, chaîne...  
    # bloc d'instructions
```

En particulier, pour itérer sur une plage d'entiers :

```
>>> for i in range(1, 4) :  
...     print('cou' * i)  
...  
cou  
coucou  
coucoucou
```

LES BOUCLES

```
while test :  
    # bloc d'instructions
```

```
for elt in sequence_iterable : # liste, tuple, chaîne...  
    # bloc d'instructions
```

En particulier, pour itérer sur une plage d'entiers :

```
>>> for i in range(1, 4) :  
...     print('cou' * i)  
...  
cou  
coucou  
coucoucou
```

il est possible de définir un bloc d'instructions à exécuter après la boucle seulement si elle termine sans appel à `break`, introduit par `else`

LES FONCTIONS

en **PYTHON**, les blocs sont définis uniquement par l'**indentation**

```
def ma_fonction (liste_des_parametres) :  
    # bloc d'instructions
```

```
>>> def truc(x) : # pas de déclaration de type  
...     return x + x # ou 2 * x, ou x * 2  
...  
>>> truc(3)  
6  
>>> truc('cou') # magique!  
'coucou'
```

la fonction `dir()`

- sans argument, liste les noms des objets/fonctions définis
- avec un argument, liste ses attributs

la fonction `dir()`

- sans argument, liste les noms des objets/fonctions définis
- avec un argument, liste ses attributs

la fonction `help()`

- sans argument, lance l'aide interactive
- avec un argument, affiche l'aide spécifique de cet argument

la fonction `dir()`

- sans argument, liste les noms des objets/fonctions définis
- avec un argument, liste ses attributs

la fonction `help()`

- sans argument, lance l'aide interactive
- avec un argument, affiche l'aide spécifique de cet argument

aide des fonctions : définie par une chaîne de caractères suivant immédiatement l'entête

```
>>> def truc(x) :  
...     """ fonction de test """  
...     return 2 * x
```

En résumé...

- `dir()` pour savoir ce qui est disponible

En résumé...

- `dir()` pour savoir ce qui est disponible
- `help()` pour savoir comment ça marche

En résumé...

- `dir()` pour savoir ce qui est disponible
- `help()` pour savoir comment ça marche
- et pour en apprendre (beaucoup) plus sur le langage :
<https://docs.python.org/fr/3/tutorial/>