Introduction à l'informatique

Cours 7

kn@lri.fr
http://www.lri.fr/~kn

université
PARIS-SACLAY

Plan

Résumé des épisodes précédents ...

En Python, nous savons manipuler :

- Des types scalaires (entiers, flottants, booléens)
- Des chaînes de caractères
- Des tableaux

On va compléter la panoplie en montrant :

- D'autres types de données
- Certaines opérations avancées sur les valeurs

Les tuples

Il est souvent pratique de vouloir regrouper un certain nombre de valeurs « dans le même paquet » :

- un point du plan (x, y)
- une date `jj/mm/aaaa`
- un instant `hh:mm:ss` :
- le résultat d'une division quotient, reste
- une URL : *protocole* : / / *nom de domaine* / chemin
- une couleur : `R G B`
- ...

Quelles solutions ?

- Utiliser des variables séparées : `f(a,m,j,hh,mm,ss)` très verbeux
- Utiliser des tableaux : les tableaux sont modifiables on peut écraser des cases

Les tuples (2)

Python propose le type de donnée de tuple (ou n-uplet). On écrit simplement les expressions en utilisant des parenthèses et des virgules.

```
>>> point = (1.5, -3.19)
>>> point
(1.5, -3.19)
>>> point[0]
1.5
>>> point[1]
-3.19
>>> x, y = point
>>> x + y
 -1.69
>>> point[0] = 2.2
 Traceback (most recent call last):
 File "", line 1, in <module>
 TypeError: 'tuple' object does not support item assignment
```

Les tuples (3)

- La fonction `len(t)` renvoie le nombre de composantes
- `()` est le tuple de taille 0, `(v, )` un tuple de taille 1 contenant `v`

(La virgule seule est obligatoire, sinon `(v)` est compris comme `v` entouré de parenthèses « mathématiques ».

- On peut utiliser le `+` et le `*` comme pour des tableaux

```
>>> t = (1, 2) + (3, 4, 5)
>>> len(t)
5
>>> t
(1, 2, 3, 4, 5)
>>> (42, ) * 5
 (42, 42, 42, 42, 42)
>>> (42) * 5
 210
```

Les tuples (exemple)

```
#On représente des points par un couple (x, y)
from math import sqrt

def add_point(p1, p2):
    return (p1[0] + p2[0], p1[1] + p2[1])

def mult_point(p, k):
   return (p1[0] * k, p1[1] * k)

def norm_point(p):
    x, y = p
    return sqrt(x ** 2 + y ** 2)
…
```

Le type None

En Python, la constante `None` est une valeur spéciale indiquant une « absence de valeur ». Peut être utilisée dans plusieurs cas :

- Une fonction qui ne fait pas de `return` renvoie la valeur `None`
- Dans une fonction, on peut vouloir renvoyer `None` plutôt que lever une erreur.

Exemple :

```
def moyenne(tab):
    if len(tab) == 0:
        return None
    total = 0
    for i in range(len(tab)):
        total += tab[i]
    return total / len(tab)


moyenne([1, 2, 3, 4])  #renvoie 2.5
moyenne([])            #renvoie None
```

Le type None (2)

On peut tester qu'une valeur est `None` avec l'opérateur d'égalité (==). Si une fonction peut renvoyer `None`, alors il faut **toujours** tester son résultat avant de l'utiliser, sinon on risque des erreurs:

```
moy = moyenne(tab)
if moy == None:
    print("Erreur, tableau vide !")
else:
    print("Moyenne au carré:", moy * moy)
```

Sans le test, on aurait calculé `None * None` dans le cas d'un tableau vide, ce qui aurait provoqué une erreur.

Les dictionnaires

On a souvent besoin d'associer des clés à des valeurs. Lorsque les clés sont des entiers consécutifs et commençant par 0, on peut utiliser des tableaux :

```
jmois = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
…
```

Mais comment faire lorsque l'on souhaite utiliser d'autres types de clés ?

Par exemple : `"janvier" → 31`, … `"décembre" → 31` ?

C'est une situation très très courante

Les dictionnaires (2)

Python propose le type de donnée de dictionnaire.
Il est similaire aux tableaux, mais les indices peuvent être (presque) n'importe quel type de données Python.
On définit un dictionnaire vide par des `{ }`
On peut pré-remplir le dictionnaire avec la notation `{ k1:v1, ..., kv:vn }`.

```
>>> jours = { 'lundi':1, 'mardi':2, 'mercredi':3 }
>>> jours['mardi']
2
>>> jours
{'lundi':1, 'mardi':2, 'mercredi':3}
>>> jours['jeudi'] = 4
>>> jours
 {'lundi':1, 'mardi':2, 'mercredi':3, 'jeudi' : 4}
>>> jours['jeudi'] = 42
>>> jours
 {'lundi':1, 'mardi':2, 'mercredi':3, 'jeudi' : 42}
```

Les dictionnaires (3)

Accéder à une clé inexistante est similaire à faire un accès invalide dans un tableau. L'opérateur `in` permet de tester si une clé est dans le dictionnaire :

```
>>> jours['toto']
Traceback (most recent call last):
  File "", line 1, in <module>
KeyError: 'toto'
>>> 'mardi' in jours
True
>>> 'toto' in jours
>>> jours
 False
```

On peut utiliser d'autres types de valeur pour les clés (entiers, booléens). L'utilisation la plus fréquente reste les chaînes de caractères.

Attention, comme les tableaux, les dictionnaires sont mutables!

Plan

Boucles sur les collections

On a vu la boucle `for` sur des entiers, en utilisant la fonction `range`:

```
for i in range(len(tab)):
    v = tab[i]
    …
```

La boucle `for` est plus générique que cela. Elle permet d'itérer sur les éléments d'une collection.

```
for v in [42, 56, -1, 28]:
  print(v)    #affiche 42, puis 56, puis -1, puis 28
```

Cela fonctionne avec tous les types de « collections » :

```
for v in "ABCDEFG":
  print(v)    #affiche A, puis B, puis C, …
```

Boucles sur les dictionnaires

On peut itérer sur les dictionnaires. Plus précisément, on peut itérer sur les clés d'un dictionnaire :

```
mois = { 'janvier' : 31, 'février': 28, …, 'décembre' : 31 }
for m in mois:
  print(m, mois[m])    #affiche janvier 31, puis février 28,  …
```

Dans quel ordre les clés sont-elle considérées ?

- Avant Python 3.7 : dans un ordre arbitraire
- À partir de Python 3.7 : dans l'ordre d'insertion

On ne fera pas d'hypothèse sur l'ordre des clés

Tri d'un tableau

On peut trier un tableau avec la fonction prédéfinie `sorted`

```
>>> tab = [10,-1, 100, 13, -5]
>>> sorted(tab)
[-5, -1, 10, 13, 100]
>>> tab
[10,-1, 100, 13, -5 ]
```

Cette fonction renvoie une **copie triée** du tableau, le tableau original reste inchangé.
Sur un **dictionnaire**, cette fonction renvoie le tableau **trié** des clés :

```
>>> dico = { 'd' : 50, 'a' : 40, 'b' : 100 }
>>> sorted(dico)
['a', 'b', 'd']
```

On peut donc parcourir un dictionnaire dans l'ordre croissant des clés avec :

```
for k in sorted(dico):
    …
```

`break` et `continue`

Comme dans d'autres langages, les instructions `break` et `continue` peuvent être utilisées dans des boucles `for` ou `while` pour :

- sortir immédiatement de la boucle, avec `break`
- recommencer au tour de boucle suivant, avec `continue`

```
for i in range(10):
    if i == 7:
        break
    elif i % 2 == 0:
        continue
    print(i)
```

Le code ci-dessus affiche `1`, `3` et `5`.

Modèle mémoire de Python

On appelle modèle mémoire la façon dont les valeurs d'un langage sont représentés comme une séquence d'octet dans l'ordinateur.

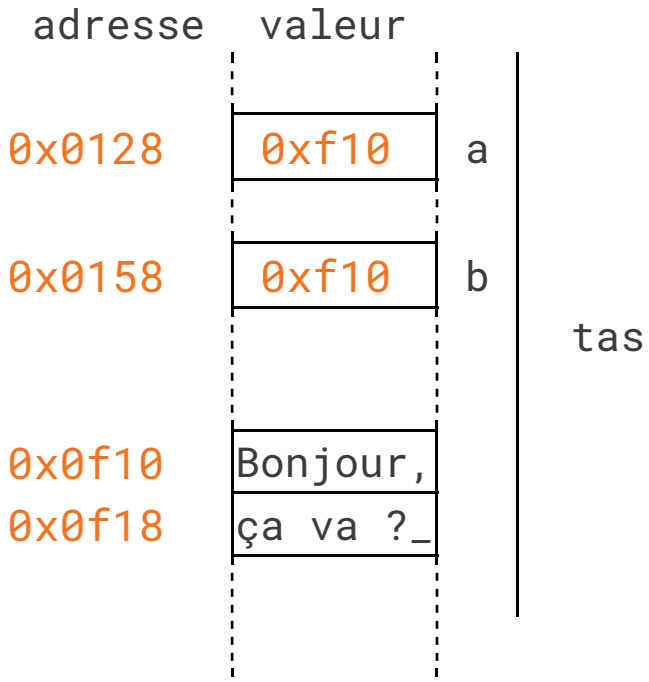
En Python, **toutes** les valeurs sont représentées par l'**adresse mémoire** (i.e. un entier 64 bits ou 8 octets) d'un bloc alloué dans le **tas** (une zone particulière de la mémoire).

```
a = "bonjour, ça va ?"
b = a
```

« En interne », a contient l'adresse d'un bloc mémoire contenant `"bonjour, ça va ?"`. L'affectation `b` place dans la variable `b` une **copie de l'adresse** et non pas une copie du bloc !

Modèle mémoire de Python (2)

En mémoire :



Modèle mémoire de Python (3)

Il y a donc une différence en mémoire entre :

```
a = "bonjour, ça va ?"
b = a
```

et

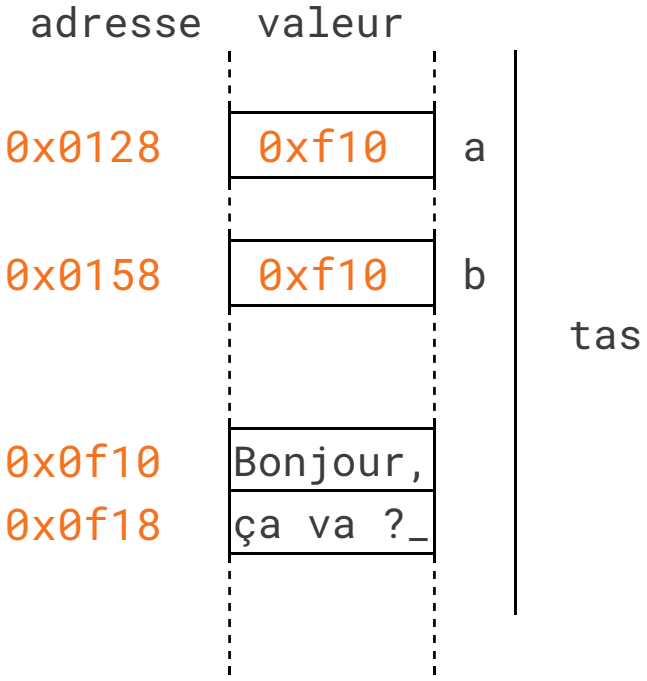
Il y a donc une différence en mémoire entre :

```
a = "bonjour, ça va ?"
b = "bonjour, ça va ?"
```

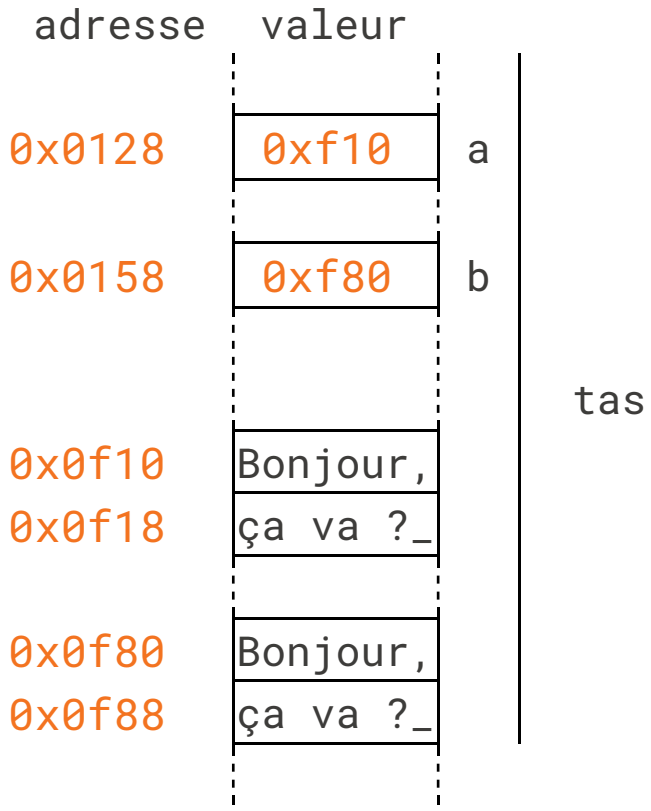
Modèle mémoire de Python (4)

En mémoire :

```
a = "Bonjour, ça va ?"
b = a
```




```
a = "Bonjour, ça va ?"
b = "Bonjour"
b = b + ", ça va ?"
```



Est-ce que tout cela est important ?

Si on utilise un type de donnée immuable (≡ non modifiable) tel que :

- Des entiers
- Des chaînes de caractères
- Des flottants (nombres à virgule)
- Des tuples
- Des booléens
- `None`

Alors ça ne fait pas de différences fondamentale (un peu de consommation mémoire en plus ou en moins).

Est-ce que tout cela est important ? (2)

Si on utilise un type de donnée mutable (≡ modifiable) tel que :

- Des tableaux
- Des dictionnaires

Alors ça ne fait une **énorme** différence. Exemple :

```
a = [1, 2, 3, 4]
b = a
```

et

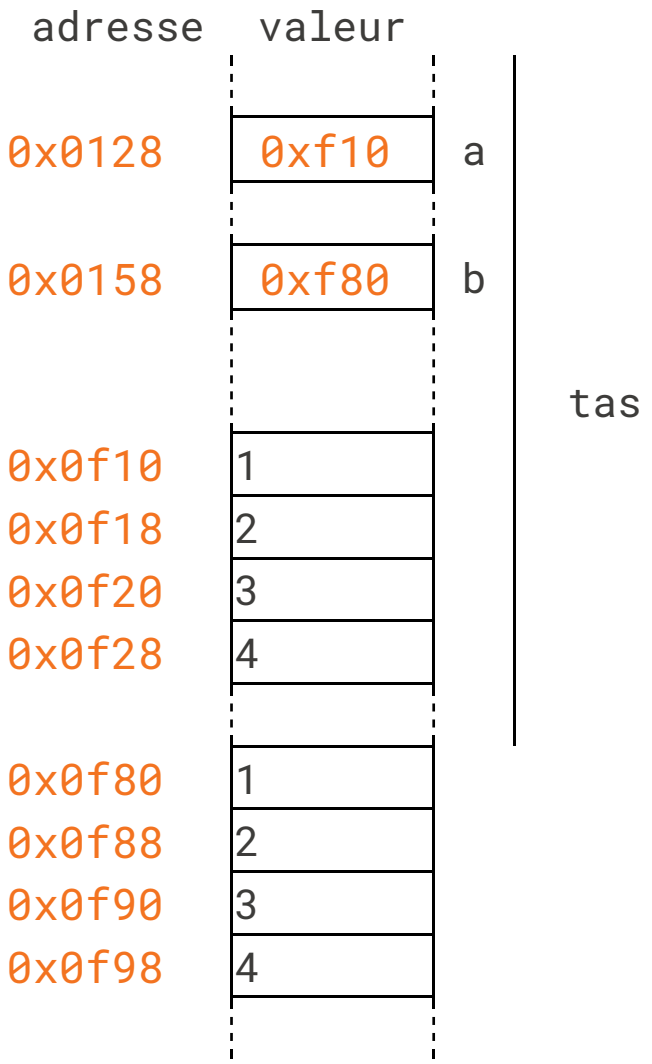
Il y a donc une différence en mémoire entre :

```
a = [1, 2, 3, 4]
b = [1, 2, 3, 4]
```

Modèle mémoire des tableaux Python

```
a = [1, 2, 3, 4]
b = a
```


```
a = [1, 2, 3, 4]
b = [1, 2, 3, 4]
```



Modèle mémoire des tableaux Python (2)

Partage de la même référence :

```
a = [1, 2, 3, 4]
b = a
b[0] = 42
print(a)          #affiche [42, 2, 3, 4]
print(b)          #affiche [42, 2, 3, 4]
```

Copie :

```
a = [1, 2, 3, 4]
b = [1, 2, 3, 4]
b[0] = 42
print(a)          #affiche [1, 2, 3, 4]
print(b)          #affiche [42, 2, 3, 4]
```

Modèle mémoire des tableaux Python (3)

Super dangereux :

Supposons que l'on veuille créer une matrice 3×3, c'est à dire un tableau de tableaux :

```
m = [ [0, 0, 0 ], [0, 0, 0 ], [0, 0, 0 ] ]
m[1][1] = 42
print (m)
# affiche [ [0, 0, 0 ], [0, 42, 0 ], [0, 0, 0 ] ]
# tout va bien


m = [ [ 0, 0, 0 ] ] * 3
m[1][1] = 42
print (m)
# affiche [ [0, 42, 0 ], [0, 42, 0 ], [0, 42, 0 ] ]
# WAT !
```

Modèle mémoire des tableaux Python (4)

Dans le code précédent, l'expression `[ [ 0, 0, 0 ] ] * 3` se comporte comme:

```
tmp = [0, 0, 0 ]

m = [None, None, None]

for i in range(len(m)):
    m[i] = tmp          #la même référence est placée dans les 3 cases!
```

Conclusion sur le modèle mémoire

Il faut être extrèmement **rigoureux** lorsque l'on manipule des tableaux.

Si on écrit des fonctions prenant des tableaux en argument, il faut préciser clairement si elles font des copies ou modifient le tableau.

Il est presque toujours faux d'écrire une chose du style:

```
tab1 = [1, 2, 3, 4, 5]
…
tab2 = tab1                 #c'est un alias, pas une copie.
                            #modifier tab1 modifie aussi tab2
```

Plan

Génie logiciel

Ensemble des méthodes de travail et bonnes pratiques à utiliser dans le cadre du développement d'un logiciel.
C'est une science de génie industriel.

On donne ici quelques petites notions importantes et on montre comment Python nous aide à les respecter.

Le génie logiciel est un aspect important de l'informatique, il est développé tout au cours de la Licence (L1, L2, L3).

Docstrings

Il est très important de documenter ces fonctions :

- Combien d'arguments et avec quels types
- Quelles valeurs peuvent être renvoyées
- Quels sont les cas d'erreur

En Python, on peut mettre une chaîne de caractères comme première « instruction » d'une fonction. Cette chaîne est sauvegardée et peut être affichée au moyen de la commande `help`.

Cette convention est utilisée par toutes les fonction de la bibliothèque standard.

Docstrings (2)

```
>>> def div_mod(a, b):
...     """Prend en argument deux entiers a et b et renvoie
...        le quotient et le reste dans la division de a par b
...        sous la forme d'une paire. Lève une exception si b vaut 0."""
...        return (a // b, a % b)
...
>>> div_mod(10, 3)
(3, 1)
>>> help(div_mod)
Help on function div_mod in module __main__:

div_mod(a, b)
    Prend en argument deux entiers a et b et renvoie
    le quotient et le reste dans la division de a par b
    sous la forme d'une paire. Lève une exception si b vaut 0.

>>> help(len)
Help on built-in function len in module builtins:

len(obj)
    Return the number of items in a container.
```

Assertion

Il est courant de vouloir signaler qu'une valeur n'est pas un argument valide pour une fonction.
On sait déjà utiliser des exceptions pour signaler cela.
L'instruction `assert (e)` évalue l'expression `e`. Si cette dernière est fausse, le programme lève une exception `AssertionError`

```
def div_mod(a, b):
    """Prend en argument deux entiers a et b et renvoie
le quotient et le reste dans la division de a par b
sous la forme d'une paire. Lève une exception si b vaut 0."""
    assert (b != 0)
    return (a // b, a % b)
```

On peut lire `assert` comme « vérifie que ». Ci dessus : « vérifie que `b` est différent de 0.

Modules

En génie logiciel, il est considéré comme une **mauvaise pratique** de mettre dans les mêmes fichier des fonctions qui n'ont rien à voir.
Si on conçoit un jeu :

- Un ensemble de fichier doit contenir uniquement les fonctions liées à l'affichage
- Un ensemble de fichier doit contenir uniquement les fonctions liées aux entrées de l'utilisateur (souris, manette, ...)
- Un ensemble de fichier doit contenir uniquement les fonctions liées à la simulation physique
- ...

Modules en Python

En Python, chaque fichier `.py` définit un **module**, c'est à dire un ensemble de fonction et de variables.

Par défaut, on ne peut référencer que des fonctions et variables du fichier (≡ du module) dans lequel on se trouve.

La directive `import` permet d'importer tout ou partie des fonctions et variables d'un module.

import

La première chose que l'on peut faire est d'importer tout un module.

```
import math

y = math.sin(math.pi / 2)
z = math.sqrt(499)
t = math.log(29)
```

Lorsque l'on écrit `import toto`, l'interprète Python cherche dans le répertoire courant, puis dans les répertoire systèmes (dans cet ordre par défaut) un fichier `toto.py`. S'il le trouve, il l'évalue :

- Toutes les instructions se trouvant directement dans le fichier sont exécutées
- Toutes les fonctions et les variables définies dans ce fichier sont accessibles en les préfixant avec `toto.`

Attention, pour cette raison, il ne faut jamais appeler un de ses fichiers comme un fichier de la bibliothèque standard !

`import` partiel

Parfois, on ne souhaite importer qu'un petit nombre de fonctions.
On peut utiliser la directive `from … import` :

```
from math import sin, sqrt, pi

y = sin(pi / 2)
z = sqrt(499)
```

Dans le code ci-dessus, seul `sin`, `sqrt` et `pi` sont visibles.

La forme : `from foo import *` importe tous les symboles, sans préfixe. Elle est à proscrire dorénavant. Pourquoi ?

Pourquoi utiliser des modules ?

Il y a deux aspects contradictoires :

- Toutes les fonctions doivent avoir un nom distinct
- On doit utiliser des noms les plus courts mais les plus descriptifs possibles.

« Your variable names should be short, and sweet and to the point » (L. Torvalds)
Exemple :

- Le module `math` de Python définit une fonction logarithme. Elle s'appelle `log`
- Le module `logging` de Python définit une fonction permettant d'afficher des messages d'erreurs dans la console et dans des fichiers. Elle s'appelle `log`
(c'est le terme en anglais)

Sans système de module, on aurait du utiliser une convention arbitraire par exemple `math_log` et `console_log`. C'est moche.

Pourquoi import * c'est mal ?


```
from logging import log, WARNING, ERROR
from math import *


…
log (WARNING, "attention !") #Erreur  utilise math.log()
…
log (ERROR, "erreur fatale !")
```

Dans le code ci-dessus, on a masqué involontairement la fonction `log` du module `logging`, par une fonction qui fait complètement autre chose.

Conclusion

Python possède des outils pour aider à écrire du code propre, il faut les utiliser :

- Documenter les fonctions
- Mettre des assertions pour vérifier les arguments passés à une fonction
- Utiliser à bon escient les modules de la bibliothèque standard
- Encapsuler son propre code dans différents modules (pour plus tard)