LYCÉE LECONTE DE LISLE

Programmation en Python

Vincent Picard

1
Les bases

Python comme calculatrice

- L'**interprète** Python peut être utilisé comme une calculatrice qui **évalue** des **expressions**.

```
>>> (4 + 5) * 4
36
>>> 2 ** 10
1024
>>> 3.14 * 2
6.28
>>> True or False
True
```

- Les **types de base** sur lesquels Python sait travailler sont les **entiers** (int), les **flottants** (float) et les **booléens** (bool).

Les entiers

Le type **int** permet de manipuler les entiers relatifs.

- Addition, multiplication

```
>>> (7 + 2) * (-3)
-27
```

- Puissance

```
>>> 3 ** 4
81
```

- Division euclidenne de a par b : `a // b` donne le **quotient** et `a % b` donne le **reste**

```
>>> 27 // 10
2
>>> 27 % 10
7
>>> -27 // 10
-3
>>> -27 % 10
3
```

Les flottants

Le type **float** permet de manipuler les nombres à virgule.

- Addition, multiplication, division

```
>>> 1.7 + 2.1
3.8
>>> 7.5 * 2
15.0
>>> 25 / 3
8.333333333333334
```

- Puissance

```
>>> 3.0 ** 4
81.0
>>> 17.2 ** 0.5
4.147288270665544
```

Python **convertit implicitement** les entiers en flottants quand c'est nécessaire.

Le calcul flottant n'est pas exact

Les flottants sont codés sur 64 bits (norme IEEE 754), il n'y a *que* 2^{64} valeurs flottantes possibles et il est impossible de représenter $\mathbb{R}$.

Le calcul flottant implique donc des erreurs d'**approximation** qui peuvent poser problème.

- Erreurs d'arrondis

```
>>> 0.1 + 0.2
0.30000000000000004
```

- Dépassement de capacité

```
>>> 1e+309
inf
>>> 1e-324
0.0
```

- Absorption

```
>>> (1.0 + 2.0 ** 53) - 2.0 ** 53
0.0
>>> 1.0 + (2.0 ** 35 - 2.0 ** 35)
1.0 # L'addition n'est pas associative
```

Les booléens

Le type **bool** permet de manipuler les valeurs de vérité `True` et `False`.

- Négation, conjonction (et), disjonction (ou)

```
>>> True or False
True
>>> True and False
False
>>> not(False)
True
```

- Comparaisons `==, !=, <, <=, >, >=`

```
>>> 7 < 4
False
>>> 3 ** 2 + 4 ** 2 == 5 ** 2
True
>>> 0.03 ** 2 + 0.04 ** 2 != 0.05 ** 2
True # ??? Problème...
```

Il ne faut pas effectuer de tests d'égalité sur les flottants

Variables et affectations

- Il est possible de mémoriser des valeurs dans des **variables**. L'opération qui consiste à associer une valeur à une variable est l'**affectation**

```
>>> x = 3
>>> x ** 4
81
```

- Il est possible d'augmenter ou diminuer la valeur d'une variable numérique en une seule instruction.

```
>>> a = 5
>>> a += 3 # équivaut à a = a + 3
>>> a
8
>>> b = 2
>>> b -= 5 # équivaut à b = b - 5
>>> b ** 2
9
```

2
Les structures de contrôle

Les structures de contrôle

- Un programme est constitué d'une **suite d'instructions** qui sont exécutées séquentiellement par l'interprète dans l'ordre.
- Il est possible de maîtriser ce flot d'exécution à l'aide de structures de contrôle :

  a. les instructions conditionnelles (`if`)

  b. les boucles (`for, while`)

  c. les fonctions

Les instructions conditionnelles

- Les **instructions conditionnelles** permettent d'exécuter des portions de code en fonction du résultat de l'évaluation d'un **test**

```
if note >= 10:
    resultat = "accepté"
else:
    resultat = "recalé"
print(resultat)
```

- L'**indentation** en Python fait partie de la **syntaxe** du langage et permet de repérer les blocs d'instructions conditionnelles.
- On peut écrire une instruction conditionnelle sans `else`.

Instructions conditionnelles avec `elif`

- Lorsque le test de la première condition vaut `False` il est possible d'effectuer d'autres tests à l'aide du mot-clé `elif` (sinon si) :

```
if note >= 16:
    mention = "très bien"
elif note >= 14:
    mention = "bien"
elif note >= 12:
    mention = "assez bien"
elif note >= 10:
    mention = "passable"
else:
    mention = "recalé"
print(mention)
```

La boucle `for`

- La boucle `for` permet de **répéter** l'exécution d'un **corps de boucle** un nombre **prédéterminé** de fois :

```
for i in range(a, b):
    #corps de boucle
```

- La variable `i` s'appelle l'**indice de boucle** et prendra successivement chacune des valeurs entre a **inclus** et b **exclu**. Le corps de boucle est donc exécuté $b - a$ fois.
- Calcul de la somme des entiers de 1 à n :

```
n = 10
somme = 0
for k in range(1, n+1):
    somme += k
print(somme)
```

- Il est possible de ne pas donner de valeur pour a dans le `range`, dans ce cas cela signifie qu'on part de $a = 0$.

```
x = 5
for u in range(7):
    x = x + u
    x = 2 * x
print(x)
```

- Cela est particulièrement utile lorsqu'on veut répéter un bloc d'instructions n fois :

```
for _ in range(n):
    print("Bonjour !")
```

La boucle `while`

- La boucle `while` permet de **répéter** l'exécution d'un **corps de boucle** un nombre **indéterminé** de fois :

```
while test:
    #corps de boucle
```

- Le `test` s'appelle **condition de boucle**. Le `test` est d'abord évalué et si sa valeur vaut `True` alors le corps de boucle est exécuté et on recommence. Si le test vaut `False` le programme passe à la suite. En général, on ne sait pas combien de fois le corps de boucle va être exécuté : sinon c'est qu'il faudrait plutôt utiliser un `for`.

- Recherche du premier diviseur > 1 d'un entier n:

```
n = 35
k = 2
while (n % k != 0):
    k += 1
print("Le premier diviseur de", n, "est", k)
```

Les fonctions

- Une fonction est un *sous-programme* qui pourra être exécuté à chaque fois qu'on le souhaite. Une fonction doit d'abord être définie :

```
def afficher_ligne():
    print("*****")
```

- La partie de code qui demande l'exécution du *sous-programme* est appelé **appel de la fonction**.

```
for k in range(3):
    afficher_ligne() # <- appel de la fonction
```

Les fonctions : paramètres

- Le corps d'une fonction peut dépendre de certaines valeurs appelées **paramètres** : ils sont listés entre les parenthèses après le nom de fonction.

```
def afficher_ligne(symbole, n):
    ligne = ""
    for k in range(n):
        ligne = ligne + symbole
    print(ligne)
```

- Lors de l'**appel de fonction** il faut alors préciser la valeur de chacun des paramètres (ces valeurs sont appelés **arguments**)

```
>>> afficher_ligne('#', 7)
#######
>>> afficher_ligne('*', 4)
****
```

Les fonctions : valeur de retour

- Une fonction peut avoir un **résultat** qui est retourné à l'appelant à l'aide de l'instruction `return`.
- L'instruction `return valeur` a deux effets :

  1. Elle termine l'exécution de la fonction en cours
  2. Elle précise que la valeur de retour est `valeur`

- Définition de la fonction *valeur absolue* :

```
def valeur_absolue(x):
    if x < 0:
        return -x
    return x
```

- Appel de fonction avec valeur de retour :

```
>>> valeur_absolue(-7)
7
>>> valeur_absolue(-3) + valeur_absolue(7)
10
```

- Attention à ne pas confondre :
  - `print` qui provoque un affichage à l'écran sans arrêter la fonction
  - `return` qui précise la valeur de retour en arrêtant la fonction
- Erreur classique :

```
def valeur_absolue(x):
    if x < 0:
        print(-x)
    print(x)

>>> valeur_absolue(-7) + 2
7
-7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

- Êtes-vous capable de bien comprendre la raison derrière chaque ligne de ce résulat erroné ?

Variables globales, variables locales

- Les variables définies en dehors des fonctions sont des **variables globales**.
  - Elles sont visibles et utilisables dans toute la suite du programme.
- Les variables définies à l'intérieur des fonctions sont des **variables locales**.
  - Elles ne sont visibles et utilisables qu'à l'intérieur du corps de la fonction dans laquelle elles ont été définies.
  - S'il existe une variable globale de même nom, elle sera **masquée** par la variable locale au sein de la fonction.

```
a = 3 # global
b = 2 # global
def f():
    c = 5 # local
    a = 4 # local qui masque le a global
    print(a + b + c)
f() # -> 11
print(a + b) # -> 5
print(a + b + c) # -> erreur
```

Fonctions prédéfinies

- Pour nous faciliter la vie, certaines fonctions sont prédéfinies en Python, c'est le cas de la valeur absolue :

```
>>> abs(-7) + 3
10
```

- Certaines fonctions prédéfinies sont regroupées dans des **modules** qu'il est nécessaire d'**importer** avant de pouvoir utiliser la fonction. Un module utile est le module `math` :

```
>>> import math
>>> math.sqrt(9)
3.0
>>> math.log(10) # fonction ln
2.302585092994046
```

- Il est possible de donner un *alias* à un module importé pour facilier l'écriture des appels de fonctions :

```
>>> import math as m
>>> m.sqrt(16)
4.0
```

- Il est aussi possible de n'importer que certaines fonctions d'un module avec le mot-clé `from` :

```
>>> from math import log,exp
>>> log(2) + exp(3)
20.77868410374761
```

Dans ce cas, on remarque qu'on ne précise plus le nom du module dans l'appel de fonction.

3
Les listes

Les listes

- Lorsqu'on veut représenter des données complexes, les types de base ne suffisent plus.
- Une solution est de composer plusieurs valeurs (de même type ou non) en une seule à l'aide d'une **structure de donnée linéaire**.
- En Python, la structure de donnée la plus fréquente est la **liste**

```
>>> l = [45, 3, 7] # une liste de trois entiers
```

- Dans une liste ℓ chaque élément est repéré par sa position dans la structure appelée **indice**. En Python, le premier élément a pour indice 0 et le dernier $|\ell| - 1$. La valeur de l'élément d'indice i est obtenu avec la syntaxe $\ell[i]$.

```
>>> l = [45, 3, 7]
>>> l[0] + l[2]
52
```

- La **longueur** d'une liste ℓ est son nombre d'éléments, il est possible d'obtenir cette valeur avec la fonction `len` :

```
l = [5, 4, 3, 2, 1, 0]
for i in range(0, len(l)):
    print(l[i])
```

- Il est possible de construire une nouvelle liste en **concaténant** plusieurs listes :

```
>>> u = [1, 2, 3]
>>> v = [4, 5, 6, 7]
>>> w = u + v + [8, 9, 10]
>>> print(w)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Il est possible de construire une liste en **répétant** une même valeur :

```
>>> [42] * 3
[42, 42, 42]
```

- Une liste peut contenir des valeurs de types différents :

```
l = ["Pentagone", 5, False]
```

Construction de liste par compréhension

- Python permet de créer des listes par compréhension comme ceci :

```
impairs = [ 2*k + 1 for k in range(0, 12)]
```

- Pour toutes les valeurs k entières entre 0 (inclus) et 12 (exclu), la valeur $2k + 1$ sera placé dans la liste (dans cet ordre).

- Exemple

```
>>> carres = [ a ** 2 for a in range(1, 7)]
>>> carres
[1, 4, 9, 16, 25, 36]
```

Extraction de tranche

- Python permet aussi de créer une liste en prenant les valeurs d'une partie, appelée **tranche**, des valeurs d'une liste déjà existante.
- Si ℓ est une liste alors la syntaxe $\ell[a : b]$ construit une nouvelle liste dont les valeurs sont celles de ℓ entre les indices a inclus et b exclu.
- Exemple

```
>>> cube = [a ** 3 for a in range(1, 8)]
>>> cube
[1, 8, 27, 64, 125, 216, 343]
>>> tranche = cube[3:6]
>>> tranche
[64, 125, 216]
```

Les listes sont modifiables

- En Python, les listes sont **modifiables** : il est possible de changer la valeur d'un élément à l'aide l'opérateur d'affectation = :

```
>>> u = [1, 1, 2, 3, 5, 8, 13]
>>> u[4] = 42
>>> u
[1, 1, 2, 3, 42, 8, 13]
```

- On peut ajouter de nouvelles valeurs en fin de listes à l'aide la méthode **append** :

```
>>> u.append(21)
>>> u
[1, 1, 2, 3, 42, 8, 13, 21]
```

- On peut également supprimer une valeur repérée par son indice à l'aide de **del**

```
>>> del u[4]
>>> u
[1, 1, 2, 3, 8, 13, 21]
```

- Remarque : si on modifie une liste ℓ alors les listes obtenues précédemment par extraction de tranches sur ℓ ne sont pas modifiées.

Parcours de liste

- Parcourir une liste consiste à itérer une ou des actions sur chacune des valeurs de la liste. Cela peut être réalisé à l'aide d'une boucle `for`

```
fibo = [1, 1, 2, 3, 5, 8, 13, 21]
somme = 0
for i in range(0, len(fibo)):
    somme += fibo[i]
print(somme)
```

- Les parcours de listes étant fréquents, Python propose une syntaxe simplifiée à cet effet :

```
fibo = [1, 1, 2, 3, 5, 8, 13, 21]
somme = 0
for x in fibo:
    somme += x
print(somme)
```

Les listes pour les piles

- Les listes Python sont un moyen naturel d'implémenter la **structure de données** de **pile**.
- **Création d'une pile vide**

```
>>> pile = []
```

- **Insertion d'éléments dans la pile**

```
>>> pile.append(3)
>>> pile.append(4)
>>> pile.append(5)
```

- **Extraction d'éléments dans la pile**

```
>>> pile.pop()
5
>>> pile
[3, 4]
>>> pile.pop() + 10
14
```

Les listes pour les matrices

- On peut utiliser des listes de listes pour représenter des matrices. Par exemple pour la matrice :

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

  On définir la liste de listes Python :

```
A = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]
```

- Ainsi chaque élément de cette liste correspond à une **ligne** de la matrice. L'expression `A[i]` est une liste représentant la ligne i.

- Pour accéder à la case $A_{i,j}$ il suffit d'utiliser un double indicage :

```
>>> A[1][2]
6
>>> trace = A[0][0] + A[1][1] + A[2][2]
>>> trace
15
```

- On peut déterminer le format d'une matrice avec une bonne utilisation de `len` :

```
>>> B = [[1, 2, 3], [4, 5, 6]]
>>> n = len(B)
>>> m = len(B[0])
>>> print(n, m)
2 3
```

- Attention, il n'est pas possible d'additionner ou de multiplier des matrices sous cette forme (cela reste des listes...). Pour faire de l'algèbre linéaire il vaut mieux utiliser les modules `numpy` et `scipy`.

Matrices par compréhension

- Il est possible d'utiliser la syntaxe de construction de liste par compréhension pour construire des matrices.

- Par exemple, la matrice nulle de format 3×5 peut être obtenue ainsi

```
>>> [[0 for j in range(5)] for i in range(3)]
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

- La matrice $A = (a_{i,j})$ de format 4×3 définie par $a_{i,j} = ij$ peut être définie ainsi :

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \\ 4 & 8 & 12 \end{pmatrix}$$

```
>>> [[i * j for j in range(1, 4)] for i in range(1, 5)]
[[1, 2, 3], [2, 4, 6], [3, 6, 9], [4, 8, 12]]
```

Les structures modifiables sont piégeuses

- Le fait de pouvoir modifier les listes est pratique mais implique certaines difficultés.

- Un **erreur fréquente** est la création d'une matrice par répétition :

```
>>> A = [[0, 0, 0]] * 3
>>> A
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

- Tout semble s'être bien passé, cependant si on modifie une case de la matrice

```
>>> A[0][1] = 5
>>> A
[[0, 5, 0], [0, 5, 0], [0, 5, 0]]
```

- Ainsi on note le **caractère superficiel** de la création par répétition : la même et unique liste a été utilisée 3 fois, modifier l'une des lignes c'est les modifier toutes...

- Il faut utiliser la double compréhension dans ce cas...

4
Autres structures linéaires

Les tuples

- Les **tuples** servent à représenter des n-uplets de valeurs. Il se comportent globalement comme les listes à quelques exceptions près :
  - **Ils ne sont pas modifiables**
  - **On ne peut pas changer leur longueur**
  - On ne peut pas les construire par compréhension
- Exemples

```
>>> triplet = (3, 4, 5)
>>> triplet
(3, 4, 5)
>>> len(triplet)
3
>>> triplet[0] ** 2 + triplet[1] ** 2
25
>>> triplet + (6, 7, 8)
(3, 4, 5, 6, 7, 8)
>>> date = (12, "septembre", 2024)
```

Fonctions avec plusieurs résultats

- Les tuples sont utiles lorsqu'on souhaite écrire une fonction retournant 2 (ou plus) résultats : il suffit de retourner un couple.

```
def division_euclidienne(a, b):
    quotient = a // b
    reste = a % b
    return (quotient, reste)
```

- **Dépaquetage** : lorsqu'une valeur est un couple il est possible d'utiliser la syntaxe suivante pour récupérer facilement chaque membre du couple :

```
>>> (q, r) = division_euclidienne(17, 5)
>>> q
3
>>> r
2
```

Chaînes de caractères

- Les **chaînes de caractères** (`str`) servent à représenter des données textuelles. Elles se comportent globalement comme les listes de caractères. L'élément d'indice i est le i-ème caractère du texte.
- Elles se distinguent des listes par :
  - **Elles ne sont pas modifiables**
  - **On ne peut pas changer leur longueur**
  - On ne peut pas les construire par compréhension
- Une chaîne de caractères se définit en écrivant le texte entre apostrophes `'` ou entre guillemets `"`
- Exemples

```
>>> invite = 'Bonjour '
>>> prenom = "Antoine"
>>> msg = invite + prenom
>>> print(msg)
Bonjour Antoine
```

5
Les dictionnaires

Qu'est-ce qu'un dictionnaire ?

- Un **dictionnaire** est une structure de données qui permet de représenter un ensemble de couples **(clé, valeur)**.
- Les **clés** sont uniques : elles ne peuvent qu'apparaître qu'une seule fois dans la structure.
- À chaque clé est donc associée une unique valeur.
- En Python, un dictionnaire est créé à l'aide d'accolades. Voici un exemple de dictionnaire qui associe à des articles (clés) leur prix de vente (valeur) :

```
>>> prix = {'macatia' : 1.10, 'croissant' : 1.30, 'baguette' : 1.05}
```

- Dans cet exemple, les clés sont des chaînes de caractères et les valeurs des flottants.

Utilisation d'un dictionnaire

- **Lecture de valeurs**

```
>>> 2 * prix['macatia'] + prix['baguette']
3.25
```

- Si la clé n'existe pas dans le dictionnaire cela provoque une erreur :

```
>>> prix['croissant'] + 3 * prix['chocolatine']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'chocolatine'
```

- **Modification d'une valeur**

```
>>> prix['macatia'] = 0.90
>>> prix['baguette'] *= 1.20
>>> prix
{'macatia': 0.9, 'croissant': 1.3, 'baguette': 1.26}
```

- **Ajout d'une nouvelle asssociation clef-valeur**

```
>>> prix['eclair'] = 2.5
>>> prix
{'macatia': 0.9, 'croissant': 1.3, 'baguette': 1.26, 'eclair': 2.5}
```

- **Suppression d'une association**

```
>>> del prix['croissant']
>>> prix
{'macatia': 0.9, 'baguette': 1.26, 'eclair': 2.5}
```

Parcours d'un dictionnaire

- Les clés d'un dictionnaire peuvent être parcourues à l'aide de la méthode **keys** :

```
for article in prix.keys():
    print("Nous vendons :", article)
```

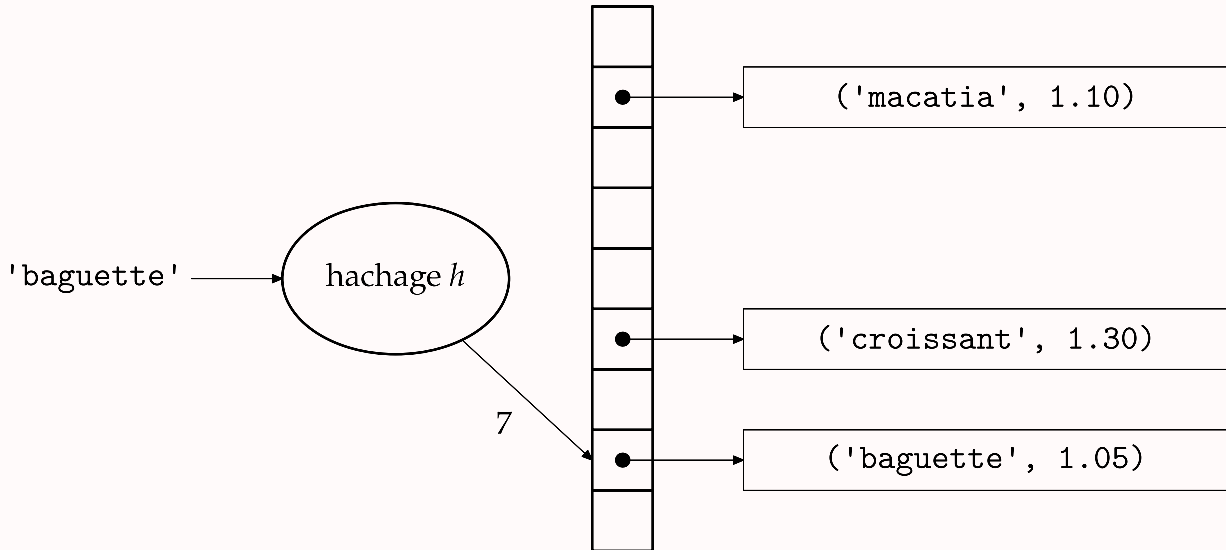
- On peut aussi parcourir toutes les associations (clef, valeur) à l'aide de la méthode **items** :

```
for (article, cout) in prix.items():
    print("Nous vendons : ", article, "au prix de", cout)
```

- Exemple : somme des prix de tous les articles

```
s = 0
for article in prix.keys():
    s += prix[article]
print(s)
```

Comment ça fonctionne ? Les tables de hachage !



- Une **table de hachage** est un **tableau** de cases appelées **alvéoles**.
- Une alvéole sert à mémoriser un (ou plusieurs) couples (clef, valeur)
- Une **fonction de hachage** h : {clefs} → int sert à déterminer le numéro de l'alvéole $h(c)$ dans laquelle sera mémorisée l'association (c, v).

Les tables de hachage sont efficaces

- La fonction de hachage est choisie de manière
  - à être rapide : on considère que le coût du calcul de $h(c)$ est constant
  - à bien répartir les clés sur l'ensemble des alvéoles disponibles : pour éviter les **collisions**
- Différentes stratégies efficaces peuvent être mises en œuvre pour gérer les éventuelles collisions.
- En conclusion, on considère que toutes les opérations sur les dictionnaires (lecture, modification, ajout, suppression) sont de complexité $O(1)$ en pratique.

Les clés ne doivent pas être modifiables

- Pour que la table de hachage fonctionne, la fonction de hachage doit toujours retourner la même valeur pour une clé donnée.
- Il en resulte que les clés doivent être **non modifiables**.
- Ainsi on peut utiliser comme clé :
  - Un entier, un flottant, un booléen
  - Une chaîne de caractères
  - Un tuple (de valeurs non modifiables)
- On ne peut pas utiliser comme clé :
  - Une liste, une tableau, une matrice, ...

```
>>> prix[ [1, 2, 3] ] = 5.6
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

6
Lecture et écriture dans un fichier