Python rapide

Pierre Karpman

Lycée Champollion MP2I

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Python en ITC

- Approche élémentaire du langage ; peu de constructions à connaître
- Objectif principal : implémenter des algorithmes (relativement simples), fidèlement et proprement
  - La « virtuosité » dans l'écriture de programme n'est pas un objectif

Éléments au programme

- Types : `int`, `float`, `bool`, `str`, `tuple`, `list`, `dict`
- Expressions sur ces types, *via* un certain nombre d'opérateurs
  - `+`, `-`, `*`, `/`, `//`, `%`, `**`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `not`, `or`, `and`
  - Attention à la *surcharge* : `+` pour des `int` ou des `list` n'ont rien à voir entre eux !
- Structures de contrôle : `if`, `while`, `for`
- Définition de fonction : `def`

Aide-mémoire

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/amp.pdf

Focus sur les expressions entières

Soit `ae1`, `ae2` deux expressions python de type `int`, on peut notamment construire les expressions entières suivantes, *via* les opérateurs correspondant :

- Addition : `ae1 + ae2`
- Soustraction : `ae1 - ae2`
- Produit : `ae1 * ae2`
- Quotient de la division entière : `ae1 // ae2` (seul le cas d'opérandes positives est au programme ; dans ce cas, le quotient est toujours associé à un reste positif, cf. ci-dessous). Peut être la cause d'une `ZeroDivisionError`.
- Reste de la division entière : `ae1 % ae2` (seul le cas d'opérandes positives est au programme ; dans ce cas, le reste est toujours positif). Peut être la cause d'une `ZeroDivisionError`.
- Exponentiation : `ae1 ** ae2`

Focus sur les expressions flottantes

Les flottants Python sont *principalement* des flottants IEEE754

Soit `fe1`, `fe2` deux expressions python de type `float`, on peut notamment construire les expressions flottantes suivantes :

- Addition : `fe1 + fe2`
- Soustraction : `fe1 - fe2`
- Produit : `fe1 * fe2`
- Division « réelle » : `fe1 / fe2`
- Exponentiation : `fe1 ** fe2`

Les expressions mixtes sont aussi possibles

- Les flottants ont la précédence (comme en C)

Pythonneries sur les flottants

```
In [1]: (2.**1000)**2
----------------------
OverflowError
In [2]: (2.**1000) * (2.**1000)
Out[2]: inf
In [3]: 1./0.
-----------------
ZeroDivisionError
```

Focus sur les expressions booléennes

Soit `be1`, `be2` deux expressions python de type `bool`, on peut notamment construire les expressions booléennes suivantes :

- Et logique : `be1 and be2`
- Ou logique : `be1 or be2`
- Négation : `not be1`

Soit `fabe1` et `fabe2` deux expressions python de type `int`, `float` ou `bool` (pas nécessairement identiques), on peut notamment construire les expressions booléennes suivantes :

- Test d'égalité `fabe1 == fabe2`
- Test de différence `fabe1 != fabe2`
- Test d'inégalité `fabe1 <= fabe2` ; `fabe1 >= fabe2`
- Test d'inégalité stricte `fabe1 < fabe2` ; `fabe1 > fabe2`

Opérateurs booléens : évaluation paresseuse

Les opérateurs `and` et `or` sont *paresseux* : ils n'évaluent pas la seconde opérande (l'opérande à droite) si le résultat de l'expression peut déjà être déterminé par la seule évaluation de la première (l'opérande à gauche).

Quelques conséquences / utilisations :

- L'expression `(d != 0) and (a % d == 0)` n'entraîne jamais de `ZeroDivisionError`
- L'expression `(x < 7) or (L.pop() == 2)` peut entraîner ou non une modification de la liste `L` en fonction de la valeur de `x`
  - Attention aux *effets de bord*

Focus déclaration / affectation de variable

- `a = 3` : déclaration de la variable `a` et affectation de la valeur `3`, ou modification « absolue » d'une variable `a` préexistante (pas de différence faite en Python…)
  - Si `a` existe déjà, peu importe son type :

    ```
    a = True
    a = 3
    ```

    est parfaitement licite
- `a = a + 1` : modification « relative » de la variable `a` : après exécution (avec succès) de l'instruction, la valeur de `a` sera la valeur précédant l'exécution, augmentée de `1`
  - Si `a` existe déjà, l'opération doit être compatible avec les types :

    ```
    a = (1,2)
    a = a + 1
    TypeError: can only concatenate tuple (not "int") to tuple
    ```

Focus portée des identifiants

En Python, la portée des identifiants est dynamique, et non pas syntaxique

- Une référence à un identifiant est valide si celui-ci a été déclaré *lors de l'exécution du code précédent la référence*
- Conséquence : parfois difficile/impossible d'anticiper si une référence sera valide à la simple lecture du code
- Conséquence : attention aux fautes de frappe !

Illustration

```
In [0]: def suma(b):
   ...:     return a + b
In [1]: suma(3)
---------
NameError
Cell In[1], line 1
----> 1 suma(3)
Cell In[0], line 2, in suma(b)
      1 def suma(b):
----> 2     return a + b

NameError: name 'a' is not defined

In [2]: a = 1
In [3]: suma(3)
Out[3]: 4
```

Focus instructions conditionnelles

Les *blocs* des instructions exécutées dans les différents cas sont délimités *via* l'indentation

```
if be1: # be1 : expression booléenne
    # instructions si be1 vaut True
elif be2: # optionnel
    # instructions si be1 vaut False
    # et be2 vaut True
elif be3: # optionnel
    # instructions si be1, be2 valent False
    # et be2 vaut True
else: # optionnel
    # instructions si be1, be2, be3 valent False
# instructions exécutées en sortie de conditions
```

Focus boucle non bornée

```
while be:
    # instructions du corps de boucle exécutées
    # tant que l'expression booléenne be
    # vaut True
# instructions exécutées en sortie de boucle
```

- On peut *imbriquer* des boucles :

```
while be1: # boucle extérieure/externe
    # ...
    while be2: # boucle intérieure/interne
        # ...
    # ...
# ...
```

- On peut interrompre prématurément *un niveau de boucle* avec l'instruction `break`

Focus boucle bornée

```
for x in I:
    # corps de boucle permettant
    # de traiter itérativement (et un à un)
    # tout élément x de l'objet itérable I
```

- Les instructions du corps de boucle ont accès aux éléments de l'objet `I` et varient (a priori) à chaque itération.
- Les « objets itérables » sont très nombreux en Python ; ceux au programme sont :
  - `range(a)` : les entiers de 0 à $a-1$ (inclus) : la variable `x` prendra successivement les valeurs 0, 1, etc., `range(a, b)` : les entiers de a à $b-1$ (inclus)
  - `range(a, b, c)` : les entiers de a à $b-1$ (pas nécessairement atteint) par incrément de c, possiblement négatif (dans ce cas, la plus petite valeur pouvant être atteinte devient $b+1$)
  - `L` : une liste, `T` : un tuple
  - `D` : les clefs d'un dictionnaire, `D.keys()` : pareil, explicitement, `D.items()` : les couples *(clef, valeur)* d'un dictionnaire
  - `s` : les caractères d'une chaîne de caractères

Focus fonction

Définition

```
def fun(arg1, arg2):
    # corps de la fonction dans lequel
    # arg1 et arg2 sont accessibles comme variables
    return res # renvoie un résultat
```

Re : Portée des variables

- Variables accessibles : arguments de la fonction ; variables *locales* définies dans la fonction ; tout ce qui a été défini globalement par les *exécutions* passées
- Si même nom pour une variable locale et variable globale au contexte appelant, la première *occulte* la seconde

Politique d'appel

- Appel par *valeur* les arguments sont évalués et la fonction appelée sur le résultat

Focus `tuples`

Un *tuple* est une suite finie d'éléments. En Python, la structure `tuple` correspondant à cette notion est *immuable* : une fois un tuple construit, il n'est pas possible de modifier ses éléments. C'est une différence importante avec le type `list`.

```
t = () # tuple vide
t = 1, True # construction d'un tuple
t = (1, True) # variante
t = (1, (True, False)) # les éléments peuvent eux-mêmes être structurés
t = tup1 + tup2 # concaténation de tuples
t = tup1 * i # concaténation répétée (i fois) d'un tuple
t = tup1[a:b] # tuple correspondant aux éléments d'indice a (inclus)
              # à b (exclus) du tuple tup1. Les indices commencent à 0
l = len(t) # nombre d'éléments dans un tuple
x = t[i] # i : indice entre 0 et len(t) - 1. Possible erreur IndexError
a, b = t # équivalent à a = t[0] ; b = t[1], si t a deux (ici) éléments
```

Focus `strings`

Chaînes de caractère (ou *string*) Python $\approx$ tuples d'éléments de même type (lui-même « chaîne de caractère ») $\rightarrow$ immuables, mêmes opérations de concaténation etc. que n'importe quel tuple

```
s = '' # chaîne vide
s = 'toto' # variante : s = "toto"
s = 'toto\n' # \n : caractère spécial « retour à la ligne »
```

Focus `strings` *bis*

- Les chaînes de caractères peuvent être lues ou écrites depuis ou vers un fichier ou une sortie, notamment via la fonction `print`
- La bibliothèque standard du langage Python définit de très nombreuses fonctions pour manipuler les chaînes de caractère. La seule d'entre-elles au programme est la fonction (techniquement, méthode) `split` qui étant donnée une chaîne de caractère renvoie (par défaut) la *liste* des chaînes séparées par des espaces la composant :

```
In :  s = 'il fait beau et chaud'
In :  s.split()
Out:  ['il', 'fait', 'beau', 'et', 'chaud']
In :  W = 'pif paf pouf'.split() # split directement sur un littéral
In :  for w in W:
          print(w)
pif
paf
pouf
```

Focus `lists`

Suites finies *mutables* d'éléments de types divers : en plus des opérations sur les tuples, il est à la fois possible de modifier la valeur (et même le type !) d'un élément d'une liste existante, et d'ajouter ou supprimer des éléments.

```
L = [] # liste vide
L = [1, 2, True] # construction directe
L[0] = 2 # modification d'un élément
L.append(False) # ajout d'un élément en dernière position
x = L.pop() # renvoie et supprime le dernier élément
L = [x for x in I] # construction par compréhension
L = [2*x for x in range(10)] # les entiers pairs de 0 à 18
```

Focus `lists` *bis*

Attention !

Les affectations de liste ne font que créer des *alias* qui *référencent* la *même* liste.

```
In :  L = [1, 2]
In :  G = L # G n'est qu'un « alias » pour L
In :  L[0] = 0
In :  G
Out:  [0, 2] # G « a été modifiée comme L »
In :  mypop(L) # fonction d'unique instruction L.pop()
In :  L
Out:  [0] # L a été modifiée par la fonction
In :  L = [[1]*2]*2
In :  L
Out:  [[1, 1], [1, 1]]
In :  L[0][0] = 0
In :  L
Out:  [[0, 1], [0, 1]]
```

Focus `lists` *ter*

Copie de liste *via* copy

```
In :  L = [1, 2]
In :  G = L.copy()
In :  L[0] = 0
In :  L
Out:  [0, 2]
In :  G
Out:  [1, 2] # G est réellement une autre liste
```

Focus `lists` *quater*

Attention

La copie reste *superficielle*

```
In :  L = [[1, 2], [3, 4]]
In :  G = L.copy()
In :  L[0][0] = 0
In :  L
Out:  [[0, 2], [3, 4]]
In :  G
Out:  [[0, 2], [3, 4]] # semble n'avoir rien changé
In :  L[0] = [1, 2]
In :  L
Out:  [[1, 2], [3, 4]]
In :  G
Out:  [[0, 2], [3, 4]] # G[0] n'a pas été changé
```

`lists` : remarques

- Le type Python `list` est très différent du type OCaml `'a list` : il ne faut pas les confondre
- Les `list`s Python fournissent une implémentation concrète efficace des structures de données abstraites
  - de pile mutable (avec append et pop)
  - de tableau (grâce aux accès aléatoires efficaces)
- Dans un énoncé, on pourrait parfaitement parler de « tableau » ou de « pile » (notions abstraites) pour ensuite les implémenter avec des `list`

Dictionnaires

Ensembles mutables de *valeurs* de types quelconques indexées par des *clefs* de type « hashable » (parmi les types au programme, cela concerne tous les types sauf les listes et les dictionnaires)

```
D = {} # dictionnaire vide
D = {3:4, "a":5}
D[3] # renvoie la valeur associée à la clef 3, si elle existe
     # possible erreur KeyError
D[3] = 5 # ajout ou modification de la valeur associée à la clef 3
len(D) # nombre d'éléments dans le dictionnaire
3 in D # True si la *clef* 3 est présente dans D
```