Rappels

Cours 0

- v1.2

Lycée polyvalent Franklin Roosevelt, 10 Rue du Président Franklin Roosevelt, 51100 Reims

Nous allons revoir ici les différents éléments primordiaux du Python : les variables et leurs types, les structures de boucles et enfin les fonctions.

1 Les types de variables

Une chose concerne toutes les variables, c'est l'affectation, c'est-à-dire le fait d'associer à une variable, repérée par un nom, une valeur. L'opération universelle d'affectation se fait avec le signe `=`. Le nom de l'objet est toujours à gauche, la valeur qu'il doit prendre à droite.

Attention

- Cela se rapproche du concept d'égalité comme définition en mathématiques. On a donc `a=2` pas du tout équivalent à `2=a` :
```
>>> a = 2
>>> a
2
>>> 2 = a
File "<console>", line 1
SyntaxError: cannot assign to literal
```
La deuxième écriture est une erreur de syntaxe (`to assign` est la traduction anglaise du verbe affecter).
- On peut utiliser n'importe quelle suite de caractères alphanumériques qui ne commence pas par un chiffre comme nom de variable (sauf les mots clés du langage), on a aussi le droit au tiret du bas (underscore, `_`). On évite les accents et alphabets non-latin pour des questions de compatibilités.

Pour connaitre le type d'une variable, on peut utiliser la fonction `type()`.

1.1 Les variables numériques

a) Les opérateurs

Les opérateurs qui concernent les types numériques (entiers, flottants et complexes) sont assez transparents. Il y a l'addition (opérateur `+`), la soustraction (opérateur `-`), la multiplication (opérateur `*`) la division (opérateur `/`) le quotient de la division euclidienne (opérateur `//`), le reste de la division euclidienne (opérateur `%`) et la mise à la puissance (opérateur `**`).

Attention

- La multiplication est toujours explicite en Python : jamais `(p+3)(p+5)`, mais `(p+3)*(p+5)`.
- La division renvoie toujours un flottant, même si le résultat est un entier :
  ```
  >>> 6/3
  2.0
  ```

b) Les types numériques de base : int, float, complex

On distingue les entiers (`int`) par l'absence de point décimal.

Les flottants (`float`) permet de représenter les réels (avec quelques limitations quasi-invisibles pour une utilisation normale, ces dernières étant revues plus tard dans l'année). On les repère grâce au point décimal et à la notation exponentielle qui apparait quand les nombres deviennent très grands ou très petits.

```
>>> 24139842843914108 # int
24139842843914108
>>> 24139842843914108.0 # float "grand"
2.4139842843914108e+16
>>> 0.0000104839284 # float "petit"
1.04839284e-05
```

Python est l'un des rares langages à prévoir un type complexe de base (`complex`). La partie complexe s'écrit grâce à la lettre `j` (en minuscule ou majuscule), comme en Physique ou en Sciences Industrielles. $i^2 = -1$ devient alors :

```
>>> m1=1j**2
>>> print(m1)
(-1+0j)
```

c) Les conversions

Pour convertir d'un type à l'autre, ou pour avoir une conversion à partir d'une chaine de caractères, il existe les fonctions `int()`, `float()` et `complex()`.

1.2 Autres variables : NoneType et bool

`NoneType` est le type de `None`, qui est la valeur renvoyée par une fonction qui a fini sans rencontrer de `return` ou n'a rien après son `return`.

Un booléen (`bool`) ne peut valoir que `True` ou `False`. Les opérations sur les booléens sont aussi relativement transparentes : le NON logique (opérateur `not`), le OU logique (opérateur `or`) et le ET logique (opérateur `and`).

a	b	not(a)	a and b	a or b
False	False	True	False	False
False	True	True	False	True
True	False	False	False	True
True	True	False	True	True

Les booléens sont très souvent le résultat d'opérations de comparaisons, relativement transparentes : l'égalité (opérateur `==`), la différence (opérateur `!=`), et les inégalités strictes (opérateurs `<` et `>`) ou larges (opérateurs `<=` et `>=`).

Attention

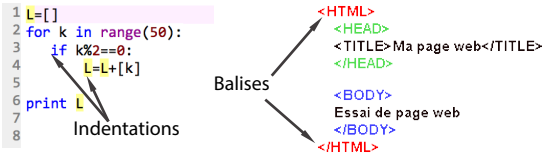
- Ne pas confondre l'affectation `=` avec le test d'égalité `==`. Ces 2 sens de l'égalité n'ont rien à voir entre eux.
- on évite les tests redondants dès qu'on peut les éviter `if test == True` ou `if test == False`. On préférera les plus élégants `if test` ou `if not test` à la place.

2 Les structures

2.1 Indentation structurelle

Habituellement un bon nombre de langages de programmation (HTML, scilab, matlab, LaTeX, etc...) utilisent des balises qui délimitent une fonction. On conseille généralement en plus d'utiliser des indentations (utilisation de la touche tabulation) pour délimiter les fonctions utilisées, car cela facilite la lisibilité.

Le langage Python a pour particularité d'utiliser les indentations a des fins structurelles. Cela permet alors de rendre un programme bien présenté et ainsi facile à relire et/ou à déboguer.



Comparaison entre l'utilisation de l'indentation pour Python et de balises pour le HTML

2.2 Instructions conditionnelles si-alors-sinon (`if`, `elif` et `else`)

Une instruction conditionnelle permet d'exécuter au choix une instruction en fonction de la valeur d'un test, très souvent obtenu par une opération de comparaison.

On utilise pour cela la commande `if`. La sémantique est la suivante :

- `if` est seul : si la condition évaluée qui suit le `if` est vrai alors les instructions indentées juste en dessous sont exécutées.

```
if condition_1:
    instruction_1
```

- `if` est suivi de `else` : si la condition évaluée qui suit le `if` est vrai alors les instructions indentées juste en dessous sont exécutées. Si elle est fausse alors les instructions qui suivent le `else` sont exécutées. Aucune condition ne suit le `else`.

```
if condition_1:
    instruction_1
else:
    instruction_2
```

- `if` suivi d'une ou plusieurs instructions `elif` : le programme sélectionne la condition vraie qui suit le `if` et les `elif` et exécute les instructions associées. Ces conditions sont très souvent *exclusives*. Si toutes les conditions sont fausses alors, le programme exécute l'instruction associée au `else`. Lorsqu'il n'y a pas de `else`, aucune instruction n'est exécutée.

```
if condition_1:
    instruction_1
elif condition_2:
    instruction_2
...
elif condition_n-1:
    instruction_n-1
else:
    instruction_n
```

Attention

On veillera à ne pas oublier les : à la suite de chaque condition et du `else`.

2.3 Boucles tant que (`while`) et boucles pour (`for`)

a) Boucles `while`

```
while condition:
    instructions
```

Une boucle `while` permet de répéter une même instruction *tant qu'une* condition reste vrai. Elle est souvent appropriée pour répéter des instructions sans connaitre au préalable le nombre de fois.

La condition qui figure après le mot clé `while` est évaluée à l'entrée de la boucle :

- si elle est fausse (booléen égal à `False`) alors on sort de la boucle et les instructions ne sont pas exécutées ;
- si elle est vraie (booléen égal à `True`) alors on entre de la boucle et les instructions sont exécutées.

Attention

La terminaison de la boucle `while` n'est pas garantie.

b) Boucles `for`

```
for k in range(deb,fin,pas):
    instructions
```

Une boucle `for` permet de répéter une même instruction un nombre fini de fois. Elle est utilisée pour itérer sur les éléments d'une séquence (par exemple une chaine de caractère `str`, un n-uplet `tuple`, une liste `list` ou un parcours `range`).

Le plus simple est souvent d'utiliser une variable de type entier et la fonction **range** pour créer une séquence ad hoc.

Boucle *pour* classique utilisant un parcours (`range`).

```
for elt in iterable:
    instructions
```

Boucle avec chaque élément de la séquence accessible avec `elt`.

2.4 Les fonctions

Définition *Fonctions*

Une fonction est une suite d'instructions qui dépend de paramètres. Une fonction comporte :

- zéro, une ou plusieurs entrées, appelées arguments,
- une sortie qui peut être sous la forme d'un n-uplets ou d'une liste si besoin.

En Python, une fonction est introduite par le mot-clé **def** suivi du nom de la fonction et des arguments entre parenthèses, séparés par des virgules, puis les 2 points finaux `:`.

Tout ce qui se situe en dessous doit être **indenté** et constitue le **corps de la fonction**. Il représente l'ensemble des instructions qui s'exécutent lorsque l'on fait appelle à la fonction.

Pour renvoyer des variables en sortie de la fonction, on utilise le mot-clé **return**.

Exemple : *Écriture sous la forme d'une fonction de la division euclidienne*

Le programme ci-dessous permet de représenter la fonction de la division euclidienne qui prend en arguments d'entrée a et b et retourne le quotient q et le reste r.

```
1 def div_euclidienne(a,b):
2     q, r = 0, a
3     while r>=b:
4         r, q = r-b, q+1
5     return q,r
```

Attention

Le mot-clé `return` arrête forcément la fonction, même dans une boucle.

2.5 Portées des variables

Définition *Portées des variables et espace des noms*

La notion de *portée d'une variable* est liée à son accessibilité en lecture et en écriture. Python définit des *espaces de noms* locaux, pour chaque fonction. Les variables des niveaux au-dessus sont accessibles en lecture sans restriction, mais pas en écriture.

Pour vous aider à mieux visualiser ces espaces de noms qui s'emboitent les uns dans les autres, on peut penser à un hôtel : si vous êtes dans la rue, tout ce qui se passe dans l'hôtel est inconnu, mais du couloir de la chambre d'hôtel, on peut voir la rue. De même, si vous louez une chambre, vous avez accès au couloir et à la rue, mais vous ne pouvez pas accéder aux autres chambres. Et ce qui est dans la rue ou dans le couloir de l'hôtel, vous ne pouvez pas l'emmener dans votre chambre pour le modifier et le ramener à sa place après.

Exemple :

Exemple avec 4 espaces de noms, dont 1 qui en englobe 2 autres :

- Ne fonctionne pas parce que g n'a pas été modifié dans le bon espace de nom :

```
1  def compareChute(h,L_emplacement):
2
3      g = 0 # pas de pesanteur par défaut
4
5      def metsMoiSur(emplacement):
6          if emplacement == 'la Terre':
7              g = 9.81
8          elif emplacement == 'Mars':
9              g = 3.73
10
11     def dureeChuteLibre():
12         return (2*h/g)**(0.5)
13
14     for empl in L_emplacement:
15         metsMoiSur(empl)
16         t = dureeChuteLibre()
17         res = "Sur " + empl + ", la chute dure " + str(t) + " s"
18         print(res)
19
20 compareChute(10,['la Terre','Mars'])
21 # erreur se finissant par ...
22 # ... line 12, in dureeChuteLibre
23 #     return (2*h/g)**(0.5)
24 # ZeroDivisionError: division by zero
```

- On change notre code pour corriger l'accessibilité en écriture :

```
1  def compareChute(h,L_emplacement):
2
3      g = 0 # pas de pesanteur par défaut
4
5      def metsMoiSur(emplacement):
6          if emplacement == 'la Terre':
7              return 9.81
8          elif emplacement == 'Mars':
9              return 3.73
10
11     def dureeChuteLibre():
12         return (2*h/g)**(0.5)
13
14     for empl in L_emplacement:
15         g = metsMoiSur(empl)
16         t = dureeChuteLibre()
17         res = "Sur " + empl + ", la chute dure " + str(t) + " s"
18         print(res)
19
20 compareChute(10,['la Terre','Mars'])
21 # Sur la Terre, la chute dure 1.4278431229270645 s
22 # Sur Mars, la chute dure 2.315584223237446 s
```