## Les mots réservés

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| and | del | from | none | true | as | elif | global |
| nonlocal | try | assert | else | if | not | while | break |
| except | import | or | with | class | false | in | pass |
| yield | continue | finally | is | raise | def | for | |

lambda return

## Notion d'expression

Définition

Une expression est une portion de code que l'interpréteur Python peut évaluer pour obtenir une valeur.

Les expressions peuvent être simples ou complexes. Elles sont formées d'une combinaison de littéraux, d'identifiants et d'opérateurs.

Exemples de deux expressions simples et d'une expression complexe

```
id1 = 15.3
id2 = maFonction(id1)
if id2 > 0:
        id3 = math.sqrt(id2)
```

## Les types de données entiers

Python 3 offre deux types entiers standard : int et bool.

Le type int

Le type int n'est limité en taille que par la mémoire de la machine.
Les entiers littéraux sont décimaux par défaut, mais on peut aussi utiliser les bases suivantes :

```
>>> 2009 # décimal
2009
>>> 0b11111011001 # binaire
2009
>>> 0o3731 # octal
```

Un entier écrit en base 10 (par exemple 179) peut se représenter en binaire, octal et hexadécimal en utilisant les syntaxes suivantes :

```
>>> 0b10110011 # binaire
179
>>> 0o263 # octal
179
```

Le type bool

- Deux valeurs possibles : False, True.
- Opérateurs de comparaison : ==, !=, >, >=, < et <= :

## Les opérations arithmétiques

Les principales opérations :

```
20 + 3 # 23
20 - 3 # 17
20 * 3 # 60
20 ** 3 # 8000
20 / 3 # 6.666666666666667
20 // 3 # 6 (division entière)
20 % 3 # 2 (modulo)
abs(3 - 20) # valeur absolue
```

Bien remarquer le rôle des deux opérateurs de division :

/ : produit une division flottante ;
// : produit une division entière.

## Les opérateurs logiques

Opérateurs de comparaison : ==, !=, >, >=, < et <= :

Exemple :

```
2 > 8 # False
```

Opérateurs logiques (concept de *shortcut*) : not, or et and.

En observant les tables de vérité des opérateurs and et or on remarque que :

- Dès qu'un premier membre a la valeur False, l'expression False and expression2 vaudra False. On n'a donc pas besoin de l'évaluer ;
- De même dès qu'un premier membre a la valeur True, l'expression True or expression2 vaudra True.

Cette optimisation est appelée « principe du *shortcut* » :

```
(3 == 3) or (9 > 24) # True (dès le premier membre)
(9 > 24) and (3 == 3) # False (dès le premier membre)
```

Les opérations logiques et de comparaisons sont évaluées afin de donner des résultats booléens dans False, True.

## Les expressions booléennes

Une expression booléenne a deux valeurs possibles : False ou True.

Python attribue à une expression booléenne la valeur False si c'est :

- la constante False ;
- la constante None ;
- une séquence ou une collection vide ;
- une donnée numérique de valeur 0.

Tout le reste vaut True.

## **Les types de données flottants**

### Le type float

- Un float est noté avec un point décimal ou en notation exponentielle :
  - 2.718
  - .02
  - 3e8
  - 6.023e23

- Les flottants supportent les mêmes opérations que les entiers.
- Les float ont une précision finie indiquée dans sys.float_info.epsilon.

- L'import du module math autorise toutes les opérations mathématiques usuelles :

```
import math
print(math.sin(math.pi/4)) # 0.7071067811865475
print(math.degrees(math.pi)) # 180.0
```

### Le type complex

- Les complexes sont écrits en notation cartésienne formée de deux flottants.
- La partie imaginaire est suffixée par j :

```
print(1j) # 1j
print((2+3j) + (4-7j)) # (6-4j)
print((9+5j).real) # 9.0
```

- Un module mathématique spécifique (cmath) leur est réservé :

```
import cmath
print(cmath.phase(-1 + 0j)) # 3.14159265359
print(cmath.polar(3 + 4j)) # (5.0, 0.9272952180016122)
```

## **Les variables**

Dès que l'on possède des *types de données*, on a besoin des *variables* pour stocker les données.
En réalité, Python n'offre pas la notion de variable, mais plutôt celle de *référence d'objet*. Tant que l'objet n'est pas modifiable (comme les entiers, les flottants, etc.), il n'y a pas de différence notable. On verra que la situation change dans le cas des objets modifiables...

### Définition

Une variable est un identifiant associé à une valeur. Informatiquement, c'est une référence d'objet située à une adresse mémoire.

**L'affectation**

Définition

On affecte une variable par une valeur en utilisant le signe = (qui *n'a rien à voir* avec l'égalité en math !). Dans une affectation, le membre de gauche reçoit le membre de droite ce qui nécessite d'évaluer la valeur correspondant au membre de droite avant de l'affecter au membre de gauche.

```
a = 2 # prononcez : a ”reçoit” 2

b = 7.2 * math.log(math.e / 45.12) - 2*math.pi
```

La valeur d'une variable, comme son nom l'indique, peut évoluer au cours du temps (la valeur antérieure est perdue) :

```
a = a + 1 # 3 (incrémentation)
```

L'affectation a un effet (elle modifie l'état interne du programme en cours d'exécution) mais n'a pas de valeur (on ne peut pas l'utiliser dans une expression) :

```
>>> a = 2

>>> x = (a = 3) + 2
```

La comparaison a une valeur utilisable dans une expression mais n'a pas d'effet (l'automate interne représentant l'évolution du programme n'est pas modifié) :

```
>>> x = (a == 3) + 2

>>> x
```

**Les variantes de l'affectation**

Outre l'affectation simple, on peut aussi utiliser les formes suivantes :

```
# affectation simple

v = 4

# affectation augmentée

v += 2 # idem à : v = v + 2 si v est déjà référencé

# affectation de droite à gauche

c = d = 8 # cibles multiples
```

**Les chaînes de caractères**

Définition

Le type de données non modifiable str représente une séquence de caractères Uni14 code.
*Non modifiable* signifie qu'une donnée, une fois créée en mémoire, ne pourra plus être changée.
Trois syntaxes de chaînes sont disponibles.
Remarquez que l'on peut aussi utiliser le ' à la place de ”, ce qui permet d'inclure une notation dans l'autre :

```
    syntaxe1 = ”Première forme avec un retour à la ligne \n”
    syntaxe2 = r”Deuxième forme sans retour à la ligne \n”
    syntaxe3 = ”””
    Troisième forme multi-ligne
    ”””
guillemets = ”L'eau vive”
apostrophes = 'Forme ”avec des apostrophes”'
```

## Les opérations

- Longueur :

```
s = ”abcde”
len(s) # 5
```

- Concaténation :

```
s1 = ”abc”
s2 = ”defg”
s3 = s1 + s2 # ’abcdefg’
```

- Répétition :

```
s4 = ”Fi! ”
s5 = s4 * 3 # ’Fi! Fi! Fi! ’
print(s5)
```

## Fonctions *vs* méthodes

On peut agir sur une chaîne (et plus généralement sur une séquence) en utilisant des fonctions (notion procédurale) ou des méthodes (notion objet).

- Pour appliquer une fonction, on utilise l’opérateur () appliqué à la fonction :

```
ch1 = ”abc”
long = len(ch1) # 3
```

- On applique une méthode à un objet en utilisant la notation pointée entre la donnée/- variable à laquelle on applique la méthode, et le nom de la méthode suivi de l’opérateur () appliqué à la méthode :

```
ch2 = ”abracadabra”
ch3 = ch2.upper() # ”ABRACADABRA”
```

## Méthodes de test de l’état d’une chaîne ch

Les méthodes suivantes sont à valeur booléenne, c’est-à-dire qu’elles retournent la valeur True ou False.
La notation *[xxx]* indique un élément optionnel que l’on peut donc omettre lors de l’utilisation de la méthode.

- isupper() et islower() : retournent True si ch ne contient respectivement que des majuscules/minuscules :

```
print(”cHAise basSe”.isupper()) # False
```

- istitle() : retourne True si seule la première lettre de chaque mot de ch est en majuscule :

```
print(”Chaise Basse”.istitle()) # True
```

- isalnum(), isalpha(), isdigit() et isspace() : retournent True si ch ne contient respectivement que des caractères alphanumériques, alphabétiques, numériques ou des espaces :

```
print(”3 chaises basses”.isalpha()) # False
print(”54762”.isdigit()) # True
```

- startswith(prefix[, start[, stop]]) et endswith(suffix[,start[, stop]]) : testent si la sous-chaîne définie par start et stop commence respectivement par prefix ou finit par suffix :

```
print(”abracadabra”.startswith(’ab’)) # True
print(”abracadabra”.endswith(’ara’)) # False
```

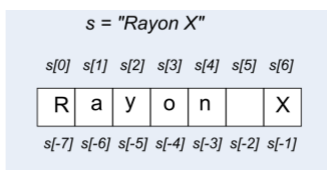
Méthodes retournant une nouvelle chaîne

- lower(), upper(), capitalize() et swapcase() : retournent respectivement une chaîne en minuscule, en majuscule, en minuscule commençant par une majuscule, ou en casse inversée :
  *s = "cHAise basSe" # s sera notre chaîne de test pour toutes les méthodes*
  - *print(s.lower()) # chaise basse*
  - *print(s.upper()) # CHAISE BASSE*
  - *print(s.capitalize()) # Chaise basse*
  - *print(s.swapcase()) # ChaISE BASsE*
- expandtabs([tabsize]) : remplace les tabulations par tabsize espaces (8 par défaut).
- center(width[, fillchar]), ljust(width[, fillchar]) et rjust(width[, fillchar]) : retournent respectivement une chaîne centrée, justifiée à gauche ou à droite, complétée par le caractère fillchar (ou par l'espace par défaut) :
  - *print(s.center(20, '-')) # ----cHAise basSe----*
    *print(s.rjust(20, '@')) # @@@@@@@@cHAise basSe*
- zfill(width) : complète ch à gauche avec des 0 jusqu'à une longueur maximale de width :
  - *print(s.zfill(20)) # 00000000cHAise basSe*
- strip([chars]), lstrip([chars]) et rstrip([chars]) : suppriment toutes les combinaisons de chars (ou l'espace par défaut) respectivement au début et en fin, au début, ou en fin d'une chaîne :
  - *print(s.strip('ce')) # HAise basS*
- find(sub[, start[, stop]]) : renvoie l'index de la chaîne sub dans la sous-chaîne start à stop, sinon renvoie -1. rfind() effectue le même travail en commençant par la fin. index() et rindex() font de même mais produisent une erreur (*exception*) si la chaîne n'est pas trouvée :
  - *print(s.find('se b')) # 4*
- replace(old[, new[, count]]) : remplace count instances (toutes par défaut) de old par new :
  - *print(s.replace('HA', 'ha')) # chaise basSe*
- split(seps[, maxsplit]) : découpe la chaîne en maxsplit morceaux (tous par défaut). rsplit() effectue la même chose en commençant par la fin et striplines() effectue ce travail avec les caractères de fin de ligne :
  - *print(s.split()) # ['cHAise', 'basSe']*
- join(seq) : concatène les chaînes du conteneur seq en intercalant la chaîne sur laquelle la méthode est appliquée:
  - *print("**".join(['cHAise', 'basSe'])) # cHAise**basSe*

Les chaînes de caractères : indexation simple

Pour indexer une chaîne, on utilise l'opérateur [ ] dans lequel l'**index**, un entier signé qui commence à 0 indique la position d'un caractère :

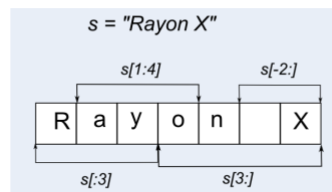
```
s = "Rayon X" # len(s) ==> 7
print(s[0]) # R
print(s[2]) # y
print(s[-1]) # X
print(s[-3]) # n
```




Extraction de sous-chaînes

L'opérateur [ ] avec 2 ou 3 index séparés par le caractère : permet d'extraire des sous-chaînes (ou tranches) d'une chaîne :

```
s = "Rayon X" # len(s) ==> 7
s[1:4] # 'ayo' (de l'index 1 compris à 4 non compris)
s[-2:] # ' X' (de l'index -2 compris à la fin)
s[:3] # 'Ray' (du début à l'index 3 non compris)
s[3:] # 'on X' (de l'index 3 compris à la fin)
s[::2] # 'RynX' (du début à la fin, de 2 en 2)
```



## Les données binaires

### Les types binaires

Python 3 propose deux types de données binaires : byte (non modifiable) et bytearray (modifiable).
Une donnée binaire contient une suite de zéro ou plusieurs octets, c'est-à-dire d'entiers non signés sur 8 bits (compris dans l'intervalle [0...255]). Ces types sont bien adaptés pour stocker de grandes quantités de données. De plus Python fournit des moyens de manipulation efficaces de ces types.
Les deux types sont assez semblables au type str et possèdent la plupart de ses méthodes.
Le type modifiable bytearray possède des méthodes communes au type list.

Exemples de données binaires et de méthodes :

```
# données binaires
b_mot = b”Animal” # chaîne préfixée par b : type byte
print(b_mot) # b’Animal’
for b in b_mot:
print(b, end=” ”) # 65 110 105 109 97 108 (cf. table ASCII)
print()
bMot = bytearray(b_mot) # retourne un nouveau tableau de bytes...
```

## Les entrées-sorties

L'utilisateur a besoin d'interagir avec le programme. En mode « console » (on verra les interfaces graphiques ultérieurement), on doit pouvoir *saisir* ou *entrer* des informations, ce qui est généralement fait depuis une lecture au clavier. Inversement, on doit pouvoir *afficher* ou *sortir* des informations, ce qui correspond généralement à une écriture sur l'écran.

### Les entrées

Il s'agit de réaliser une *saisie* à l'écran : la fonction standard input() interrompt le programme, afficher une éventuelle invite et attend que l'utilisateur entre une donnée et la valide par *Entrée* .

La fonction standard input() effectue toujours une saisie en *mode texte* (la saisie est une chaîne) dont on peut ensuite changer le type (on dit aussi *transtyper*) :

```
nb_etudiant = input(”Entrez le nombre d’étudiants : ”)
print(type(nb_etudiant)) # <class ’str’> (c’est une chaîne)
f1 = input(”\nEntrez un flottant : ”)
f1 = float(f1) # transtypage en flottant
```

Les sorties

Python *lit-évalue-affiche*, mais la fonction print() reste indispensable aux affichages dans les scripts :

```
import sys
a, b = 2, 5
print(a, b)                                  # 2 5
print(”Somme :”, a + b)                      # Somme : 7
print(a - b, ”est la différence”)           # -3 est la différence
print(”Le produit de”, a, ”par”, b, ”vaut :”, a * b) # Le produit de 2 par 5 vaut : 10
```

**Les séquences d’échappement**

À l’intérieur d’une chaîne, le caractère antislash (****) permet de donner une signification spéciale à certaines séquences :

| Séquence | Signification |
|---|---|
| \saut_ligne | saut_ligne saut de ligne ignoré |
| \\ | affiche un antislash |
| \’ | apostrophe |
| \ ” | guillemet |
| \a | sonnerie (*bip*) |
| \b | retour arrière |
| \f | saut de page |
| \n | saut de ligne |
| \r | retour en début de ligne |
| \t | tabulation horizontale |
| \v | tabulation verticale |
| \N{nom} | caractère sous forme de code Unicode nommé |
| \uhhhh | caractère sous forme de code Unicode 16 bits |
| \Uhhhhhhhh | caractère sous forme de code Unicode 32 bits |
| \ooo | caractère sous forme de code octal |
| \xhh | caractère sous forme de code hexadécimal |