

Chapitre II: Premiers éléments de programmation en Python



BENMANSOUR Asma

Table des matières



I - Objectifs spécifiques du chapitre	4
II - Données et Variables	5
III - Noms de variables et mots réservés	6
IV - Exercice : Nom de variables invalides	8
V - L'affectation	9
VI - Exercice : Affectation	11
VII - Afficher la valeur d'une variable	12
VIII - Les types de variables	13
1. Introduction	13
2. Les types pour les entiers	14
3. Le type réel	14
4. Exercice : Types entier, réel et affichage de variables	16
5. Les chaînes de caractères	16
5.1. Présentation	17
5.2. Opérations de base sur les chaînes	17
5.3. Indexation	18
5.4. Les méthodes sur chaînes	20
6. Exercice : Chaîne de caractères : indexation, affichage et méthodes sur chaînes	22
IX - Opérateurs et expressions	24
1. Les opérateurs de base en Python	24
2. Priorité entre opérateurs	25
X - Exercice : priorité de calcul à l'intérieur d'une expression	27
XI - Les entrées-sorties	28
1. Les entrées	28
2. Les sorties	29

3. Les séquences d'échappement	29
Contenus annexes	31
Solutions des exercices	34
Bibliographie	37

Objectifs spécifiques du chapitre



A l'issu de ce chapitre, l'apprenant sera capable de :

- Mémoriser les règles de construction des identifiants (noms de variables)
- Différencier entre les types de données entier, booléenne, réel et chaîne de caractère
- Manipuler les données grâce à un vocabulaire de mots réservés et grâce à aux différents types de données.
- Citer quelques méthodes s'appliquant sur le type chaîne de caractère.
- Distinguer parmi ces méthodes, celles qui retournent une nouvelle chaîne et celles qui retournent une valeur de type booléenne.
- Mettre en pratique la priorité entre opérateurs arithmétique lors du calcul des expressions mathématiques
- Reconnaître la fonction permettant de saisir des informations, depuis une lecture au clavier.
- Reconnaître et distinguer entre les différentes façon permettant d'afficher des informations sur l'écran.
- Identifier les données à saisir, les données servant au traitement et les données à afficher lors de l'élaboration du programme.

Données et Variables



L'essentiel du travail effectué par un programme d'ordinateur consiste à manipuler des *données*. Ces données peuvent être très diverses, mais dans la *mémoire de l'ordinateur* elles se ramènent toujours en définitive à une *suite finie de nombres binaires*. Pour pouvoir accéder aux données, le programme d'ordinateur quel que soit le langage dans lequel il est écrit fait abondamment usage d'un grand nombre de variables de différents types [1].

Une variable apparaît dans un langage de programmation sous un nom de variable à peu près quelconque (voir ci-après), mais pour l'ordinateur il s'agit d'une référence désignant une adresse mémoire, c'est-à-dire un emplacement précis dans la mémoire vive [1].

À cet emplacement est stockée une valeur bien déterminée. C'est la donnée proprement dite, qui est donc stockée sous la forme d'une suite de nombres binaires, mais qui n'est pas nécessairement un nombre aux yeux du langage de programmation. Cela peut être en fait à peu près n'importe quel "objet" susceptible d'être placé dans la mémoire d'un ordinateur, par exemple : un nombre entier, un nombre réel, un nombre complexe, un vecteur, une chaîne de caractères typographiques, un tableau, une fonction, etc [1].

Noms de variables et mots réservés



Les noms de variables sont des noms que vous choisissez vous-même assez librement. Efforcez-vous cependant de bien les choisir : de préférence assez courts, mais aussi explicites que possible, de manière à exprimer clairement ce que la variable est censée contenir. Par exemple, des noms de variables tels que `altitude`, `altit` ou `alt` conviennent mieux que `x` pour exprimer une `altitude`[1].

Sous Python, les noms de variables doivent en outre obéir à quelques règles simples :

1. Un nom doit débuter par une lettre ou par le caractère de soulignement `_` et constitue une séquence de lettres ($a \rightarrow z$, $A \rightarrow Z$) et/ou de chiffres ($0 \rightarrow 9$) ou de caractères de soulignement [2].
2. Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que `$`, `#`, `@`, etc. sont interdits, à l'exception du caractère `_` (souligne) [1].
3. Il existe 33 mots clés réservés par le langage lui-même, ces derniers ne peuvent pas être utilisés en tant que nom de variable et sont présentés ci-dessous [1] :

<code>and</code>	<code>as</code>	<code>assert</code>	<code>break</code>	<code>class</code>
<code>continue</code>	<code>def</code>	<code>del</code>	<code>elif</code>	<code>else</code>
<code>except</code>	<code>False</code>	<code>finally</code>	<code>for</code>	<code>from</code>
<code>global</code>	<code>if</code>	<code>import</code>	<code>in</code>	<code>is</code>
<code>lambda</code>	<code>None</code>	<code>nonlocal</code>	<code>not</code>	<code>or</code>
<code>pass</code>	<code>raise</code>	<code>return</code>	<code>True</code>	<code>try</code>
<code>while</code>	<code>with</code>	<code>yield</code>		

4. La casse est *significative*: les caractères majuscules et minuscules sont distingués [1].

Attention

Joseph, joseph, JOSEPH sont donc des variables différentes. Soyez attentifs !

 Exemple : ci-dessous quelques identifiants invalides :

1nombre, ma-variable, code%secret

 Conseil

1. Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules (y compris la première lettre). Il s'agit d'une simple convention, mais elle est largement respectée. N'utilisez les majuscules qu'à l'intérieur même du nom, pour en augmenter éventuellement la lisibilité, comme dans *tableDesMatières* [1].
2. Un bon programmeur doit veiller à ce que ses lignes d'instructions soient faciles à lire.

Exercice : Nom de variables invalides



[solution n°1 p.34]

Ci-dessous quelques noms de variables, sélectionnez ceux qui ne sont pas valides c'est à dire qui n'obéissent pas aux règles vu précédemment:

- lambda
- Nom_variable
- 3noms
- nom-variable
- Le_nom_suivi_dun_nombre4

L'affectation




Nous savons désormais comment choisir judicieusement un nom de variable. Voyons à présent comment nous pouvons définir une variable et lui affecter une valeur. Les termes affecter une valeur ou assigner une valeur à une variable sont équivalents. Ils désignent l'opération par laquelle on établit un lien entre le nom de la variable et sa valeur (son contenu) [1].

En Python comme dans de nombreux autres langages, l'opération d'*affectation* est représentée par le signe = , la création d'une variable se fait lors de sa première utilisation qui est toujours une initialisation réalisée grâce à l'opérateur d'affectation [1].


Exemple

```
1 >>> n = 7 # définir n et lui donner la valeur 7
2 >>> msg = "Quoi de neuf ?" # affecter la valeur "Quoi de neuf ?" à msg
3 >>> pi = 3.14159 # assigner sa valeur à la variable pi
```

Les trois instructions d'affectation ci-dessus ont eu pour effet chacune de réaliser plusieurs opérations dans la mémoire de l'ordinateur [1]:

1. créer et mémoriser les noms de variable à savoir n, msg et pi ;
2. leur attribuer un type bien déterminé (ce point sera explicite dans la section suivante).^{p.37} 
3. créer et mémoriser leurs valeurs à savoir le nombre entier 7, la chaîne de caractères *Quoi de neuf ?* et le nombre réel 3,14159 respectivement pour les trois variables ci-dessus.
4. établir un lien (par un système interne de pointeurs) entre le nom de la variable et l'emplacement mémoire de la valeur correspondante.

Fondamental : Noms de variables et leurs valeurs dans la mémoire de l'ordinateur !

Les noms de variables sont des références, mémorisées dans une zone particulière de la mémoire que l'on appelle espace de noms, alors que les valeurs correspondantes sont situées ailleurs, dans des emplacements parfois fort éloignés les uns des autres. Nous aurons l'occasion de préciser ce concept plus loin dans ces pages [1] ^{p.37}  .

Attention

Le signe = utilisé dans les exemples précédents est bien l'opérateur d'affectation. Il s'agit d'une action visant à placer dans le membre de gauche (c'est à dire le stockage mémoire représenté par la variable placée à gauche du signe =) le contenu représenté par l'expression simple ou complexe placée à droite de l'opérateur [1].

Il ne faut surtout pas confondre cet opérateur avec le signe = des mathématiques. Ce dernier sert à affirmer une égalité. L'opérateur d'affectation, lui, n'a pas valeur d'affirmation mathématique mais représente une action [2] p.37 ↗ .

Complément : Affectations multiples

Sous Python, on peut assigner une valeur a plusieurs variables simultanément. Exemple :

```
1 >>> a=b=6
2 >>> a
3 6
4 >>> b
5 6
```

On peut aussi effectuer des affectations parallèles a l'aide d'un seul opérateur :

```
1 >>> z,y=3,7.55
2 >>> z
3 3
4 >>> y
5 7.55
```

Dans cet exemple, les variables z et y prennent simultanément les nouvelles valeurs 3 et 7.55

Exercice : Affectation

VI

[solution n°2 p.34] $a=0$ $b=a+5$ $z=2$ $a=z$

Les instructions ci dessus peuvent être remplacées par une seule instruction, cochez la parmi les instructions ci dessous :

- $a=b=z=2$
- $a=b=z=5$
- $a,b,z=2,5,2$

Afficher la valeur d'une variable

VII

Avant d'aller plus loin dans ce cours, il est nécessaire de vous présenter les différentes façons permettant d'afficher le contenu d'une variable à l'écran mais aussi de vous introduire la fonction `print` (plus de détails seront donnés à cette fonction dans la section : les entrées-sorties). À la suite de l'exemple de la section ci-dessus, nous disposons donc des trois variables `n`, `msg` et `pi`. Pour afficher leur valeur à l'écran, il existe deux possibilités. La première consiste à entrer au clavier le nom de la variable, puis `<Enter>`. Python répond en affichant la valeur correspondante [1] :

```
1 >>> n
2 7
3 >>> msg
4 'Quoi de neuf ?'
5 >>> pi
6 3.14159
7
```

Il s'agit cependant là d'une fonctionnalité secondaire de l'interpréteur, qui est destinée à vous faciliter la vie lorsque vous faites de simples exercices à la ligne de commande. À l'intérieur d'un programme, vous utiliserez toujours la fonction `print()` :

```
1 >>> print(msg)
2 'Quoi de neuf ?'
3 >>> print(n)
4 7
5
```



Remarque

Remarquez la subtile différence dans les affichages obtenus avec chacune des deux méthodes. La fonction `print()` n'affiche strictement que la valeur de la variable, telle qu'elle a été encodée, alors que l'autre méthode (celle qui consiste à entrer seulement le nom de la variable) affiche aussi des apostrophes afin de vous rappeler que la variable traitée est du type « chaîne de caractères » : nous y reviendrons.

Les types de variables

VIII

Introduction	13
Les types pour les entiers	14
Le type réel	14
Exercice : Types entier, réel et affichage de variables	16
Les chaînes de caractères	16
Exercice : Chaîne de caractères : indexation, affichage et méthodes sur chaînes	22

1. Introduction

C'est une notion très importante en programmation. En effet, aux valeurs contenues en mémoire sont associés des *types*. Ces types permettent à la machine de savoir ce qu'elle peut faire avec les valeurs et comment les manipuler [2].

Exemple

En langage C, la déclaration d'une variable de type entier, avec initialisation à une certaine valeur, se fait par la déclaration suivante : `int var = 16` ; cela crée une variable de type entier nommée `var`. En Python, cela donne : `var = 16`

Attention

En Python, contrairement à ce qui se passe pour d'autres langages comme le langage C, il n'y a pas de déclaration de type lors de la création d'une variable ! [2]

Les variables Python sont créées lors de leur première affectation, sans déclaration préalable. Les variables Python ne sont donc pas associées à des types, mais les valeurs qu'elles contiennent, elles, le sont ! C'est au programmeur de savoir le type de la valeur contenue dans une variable à un moment donné pour ne pas faire d'opération incompatible [2].

Exemple

Voici deux instructions successives incompatibles tapées dans l'interpréteur Python :

```
1 >>> var=16
2 >>> var=var+'C'
3 Traceback (most recent call last):
4   File "<pyshell#21>", line 1, in <module>
```

```

5     var=var+'C'
6 TypeError: unsupported operand type(s) for +: 'int' and 'str'

```

Dans la seconde instruction, nous tentons d'ajouter le caractère 'C' au contenu de la variable var qui contient déjà la valeur entière 16. Cela est *incompatible* et l'interpréteur, essayant de réaliser l'addition, nous signale une erreur.

2. Les types pour les entiers

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

1. 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 [3] ^{p.37} .

```

1 >>> 2009 # décimal
2 2009
3 >>> 0b11111011001 # binaire
4 2009
5 >>> 0o3731 # octal
6 2009
7 >>> 0x7d9 # hexadecimal
8 2009

```

2. Le type bool :

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

```

1 >>> 2 > 8 # False
2 False
3 >>> 2 <= 8 < 15 # True
4 True

```

- Les opérations logiques et de comparaisons sont évaluées afin de donner des résultats booléens dans `False`, `True`. Ainsi les opérateurs logiques (concept de *shortcut*) sont: `not`, `or` et `and`. En observant les tables de vérité des opérateurs `and` et `or`, on remarque que [3] :
 1. 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 [3].
 2. De même dès qu'un premier membre a la valeur `True`, l'expression `True or expression2` vaudra `True` [3].

Exemple : Principe du shortcut dans l'évaluation d'une expression

```

1 >>> (3 == 3) or (9 > 24) # True (dès le premier membre)
2 True
3 >>> (9 > 24) and (3 == 3) # False (dès le premier membre)
4 False
5 >>>

```

3. Le type réel

Un réel (`float` en Python) est noté avec un point décimal ou en notation exponentielle (2.718, .02, 3e8, 6.023e23). Ils supportent les mêmes opérations que les entiers et ont une précision finie indiquée dans `sys.float_info.epsilon` [3].

L'import du module `math` autorise toutes les opérations mathématiques usuelles [3]:

```
1 >>> import math
2 >>> print(math.sin(math.pi/4)) # 0.7071067811865475
3 0.7071067811865475
4 >>> print(math.degrees(math.pi)) # 180.0
5 180.0
6 >>> print(math.factorial(9)) # 362880
7 362880
8 >>> print(math.log(1024, 2)) # 10.0
9 10.0
```

4. Exercice : Types entier, réel et affichage de variables

[solution n°3 p.34]

Ordonnez les instructions selon les résultats d'exécutions sur python qui correspond à:

<class 'int'>

3.67

<class 'float'>

<class 'int'>

SyntaxError: invalid syntax

<class 'bool'>

type(b)

type(z)

y=x+1.67

type(x)

z=x

print(y)

type(y)

for=5

b=True

x=2

5. Les chaînes de caractères

Présentation	17
Opérations de base sur les chaînes	17
Indexation	18
Les méthodes sur chaînes	20

5.1. Présentation

Définition

Le type de données non modifiable `str` représente une séquence de caractères *Unicode*. Non modifiable signifie qu'une donnée, une fois créée en mémoire, ne pourra plus être changée [3].

Remarque

Remarquez que l'on peut aussi utiliser le `'` à la place de `"` comme ceci :

```
1 >>> syntaxe1 = "Première forme \n"
2 >>> syntaxe1
3 'Première forme \n'
4 >>> type(syntaxe1)
5 <class 'str'>
6 >>> syntaxe2 = 'deuxième forme\n'
7 >>> syntaxe2
8 'deuxième forme\n'
9 >>> type(syntaxe2)
10 <class 'str'>
```

Attention : Pas de type pour le caractère individuel !

Les chaînes de caractères sont des valeurs composées de plusieurs caractères. Il n'y a pas en Python de type 'caractère individuel' comme en langage C, mais nous pouvons très bien avoir une chaîne ne comportant qu'un seul caractère [2].

5.2. Opérations de base sur les chaînes

- Longueur :

```
1 >>> s="maison"
2 >>> len(s) # la fonction len retourne la longueur de la chaine s
3 6
```

- Concaténation : se fait en utilisant nom de variable de la chaîne 1 + nom de variable de la chaîne 2

```
1 >>> s1="grande"
2 >>> s2="Maison"
3 >>> s3=s1+s2
4 >>> s3
5 'grandeMaison'
```

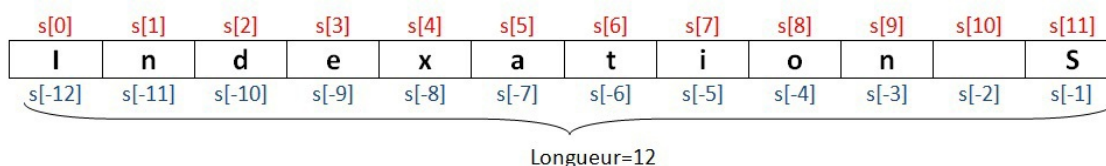
- Répétition : se fait à l'aide de l'opérateur `*` suivi du nombre de répétition souhaité (par exemple 3)

```
1 >>> s4="La!"
2 >>> s5=s4*3#'La!La!La!'
3 >>> print(s5)
4 La!La!La!
```

5.3. Indexation

Pour indexer une chaîne, on utilise l'opérateur [] dans lequel l'index indique la position d'un caractère de la chaîne, dans une chaîne de longueur n, le premier caractère est représenté par l'index 0 et le dernier est représenté par l'index n-1 [3]:

```
1 >>> s="Indexation S"
2 >>> len(s)
3 12
4 >>> s[0]
5 'I'
6 >>> s[1]
7 'n'
8 >>> s[-12]
9 'I'
10 >>> s[-11]
11 'n'
```



L'indexation d'une chaîne



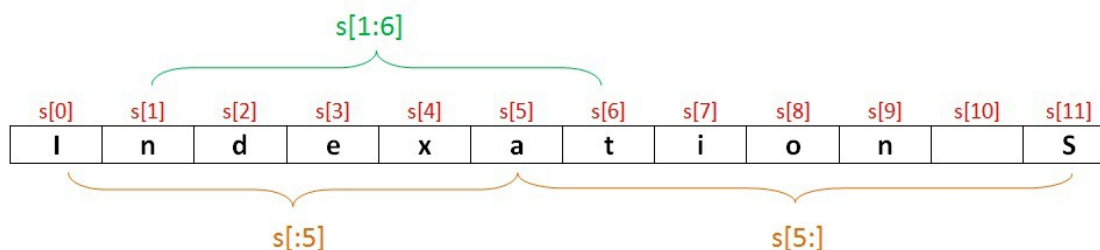
Complément : Extraction de sous-chaînes

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

- s[:ind] permet d'extraire une sous chaîne de s en partant du début à l'indice ind-1.
- s[ind:] permet d'extraire une sous chaîne en partant de l'indice ind jusqu'à la fin de la chaîne.

ind peut aussi représenter un entier positif ou un entier négatif, cela fonctionne de la même manière (voir les exemples ci dessous).

```
1 >>> s="Indexation S"
2 >>> s[1:6] #de l'index 1 à l'index 6 non compris
3 'ndexa'
```



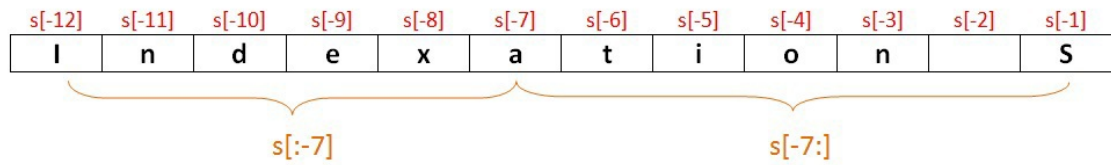
Extraction de sous chaînes (indices positifs)

```
1 >>> s="Indexation S"
```

```

2 >>> s[:5] #du début à l'index 5 non compris
3 'Index'
4 >>> s[5:] #de l'index 5 compris à la fin
5 'ation S'

```



Extraction de sous chaînes (indices négatifs)

```

1 >>> s="Indexation S"
2 >>> s[:5] #du début à l'index -7 non compris
3 'Index'
4 >>> s[5:] #de l'index -7 compris à la fin
5 'ation S'

```

5.4. Les méthodes sur chaînes

Méthodes retournant une valeur de type bool

20

Méthodes retournant une nouvelle chaîne

20

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 [3].

5.4.1. Méthodes retournant une valeur de type bool

Les méthodes suivantes sont à valeur booléenne, c'est-à-dire qu'elles retournent la valeur True ou False [3]:

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

```
1 >>> print("cHAise basSe".isupper()) # False
2 False
```

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

```
1 >>> print("Chaise Basse".istitle()) # True
2 True
```

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

```
1 >>> print("3 chaises basses".isalpha()) # False
2 False
3 >>> print("54762".isdigit()) # True
4 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 :

```
1 >>> print("abracadabra".startswith('ab')) # True
2 True
3 >>> print("abracadabra".endswith('ara')) # False
4 False
```

5.4.2. Méthodes retournant une nouvelle chaîne

Les méthodes ci dessous retournent une nouvelle chaîne, c'est-à-dire qu'elles retournent un type str [3] :

- `lower()`, `upper()`, `capitalize()` et `swapcase()` : retournent respectivement une chaîne en minuscule, en majuscule, en minuscule commençant par une majuscule, ou en casse inversée : # s sera notre chaîne de test pour toutes les méthodes

```

1 >>> s="cHAise basSe"
2 >>> print(s.lower()) # chaise basse
3 chaise basse
4 >>> print(s.upper()) # CHAISE BASSE
5 CHAISE BASSE
6 >>> print(s.capitalize()) # Chaise basse
7 Chaise basse
8 >>> print(s.swapcase()) # ChaISE BASsE
9 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) :

```

1 >>> print(s.center(20, '-')) # ----cHAise basSe----
2 ----cHAise basSe----
3 >>> print(s.rjust(20, '@')) # @@@@@@@@cHAise basSe
4 @@@@@@@@cHAise basSe

```

- `zfill(width)` : complète `ch` à gauche avec des 0 jusqu'à une longueur maximale de `width` :

```

1 >>> print(s.zfill(20)) # 00000000cHAise basSe
2 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 :

```

1 >>> print(s.strip('ce')) # HAise basS
2 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 :

```

1 >>> print(s.find('se b')) # 4
2 4

```

- `replace(old[, new[, count]])` : remplace `count` instances (toutes par défaut) de `old` par `new` :

```

1 >>> print(s.replace('HA', 'ha')) # chaise basSe
2 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 `str.splitlines()` effectue ce travail avec les caractères de fin de ligne :

```

1 >>> print(s.split()) # ['cHAise', 'basSe']
2 ['cHAise', 'basSe']

```

- `join(seq)` : concatène les chaînes du conteneur `seq` en intercalant la chaîne sur laquelle la méthode est appliquée:

```

1 >>> print("".join(['cHAise', 'basSe'])) # cHAise**basSe
2 cHAise**basSe

```

6. Exercice : Chaîne de caractères : indexation, affichage et méthodes sur chaînes

Voici la chaîne de caractère `module="programmation python"`, mettez dans l'ordre les résultats d'exécution des instructions suivantes :

```
>>> len(module)
>>> module[0:14]
>>> module[-20:]
>>> print(module.startswith("pro"))
>>> print(module.endswith('yn'))
>>> print(module.capitalize())
>>> print(module.upper())
>>> print(module)
>>> module=module.upper()
>>> module
>>> print(module.find('ON'))
```

Opérateurs et expressions

IX

Les opérateurs de base en Python

24

Priorité entre opérateurs

25

En programmation, nous manipulons les valeurs et les variables qui les référencent en les combinant avec des opérateurs pour former des expressions [1]. Dans ce qui suit nous allons vous présenter les opérateurs arithmétiques en python ainsi que leurs priorité lors de l'évaluation dans une expression.

1. Les opérateurs de base en Python

Les opérateurs permettent de manipuler les variables en mémoire pour obtenir des résultats [2].

- + addition $a = b+5$ # affecte dans a la valeur représentée par $b+5$
- - soustraction $c = c - f$ # soustrait de c la valeur de f
- * multiplication
- / division réelle $a = 5/2$ # la division ici retourne une valeur réelle 2.5 quelque soit le type des deux opérandes droite et gauche (qu'elles soient réelles, entières ou l'une d'elle de type entier et l'autre réelle)
- // division entière $a = 5 // 2$ # le résultat est 2 car les deux opérandes sont de type entier mais si l'une des deux opérande est réelle $a = 5 .0// 2$ #le résultat sera réelle et exprimé uniquement en fonction du quotient issue de la division (quotient.0) plus précisément 2.0
- % opérateur modulo : reste de la division entière $a = 5 \% 2$ # a est affecté avec le reste de la division de 5 par 2, ce qui donne la valeur 1
- += ajout d'une quantité à une variable $a += 4$ est équivalent à : $a=a+4$
- -= retrait d'une quantité à une variable $a -= 4$ est équivalent à : $a=a-4$

Exemple : Détails du calcul d'une expression

```
1 >>> a, b = 7.3, 12
2 >>> y = 3*a + b/5
3 >>> y
4 24.299999999999997
```


Dans cet exemple, nous commençons par affecter aux variables a et b les valeurs 7,3 et 12. Comme déjà explique précédemment, Python assigne automatiquement le type réel a la variable a, et le type entier à la variable b [1].

La seconde ligne de l'exemple consiste à affecter à une nouvelle variable y le résultat d'une expression qui combine les opérateurs * , + et / avec les opérandes a, b, 3 et 5. Les opérateurs sont les symboles spéciaux utilisés pour représenter des opérations mathématiques simples, telles l'addition ou la multiplication. Les opérandes sont les valeurs combinées à l'aide des opérateurs [1].

Attention : Le type de la valeur retournée par une expression !

Python évalue chaque expression qu'on lui soumet, aussi compliquée soit-elle, et le résultat de cette évaluation est toujours lui-même une valeur. À cette valeur, il attribue automatiquement un type, lequel dépend de ce qu'il y a dans l'expression. Dans l'exemple ci-dessus, y sera du type réel, parce que l'expression évaluée pour déterminer sa valeur contient elle-même au moins un réel [3].

Remarque

Les opérateurs Python ne sont pas seulement les opérateurs mathématiques de base. Nous avons déjà signalé l'existence de l'opérateur de division entière //. Il faut encore ajouter l'opérateur ** pour l'exponentiation, ainsi qu'un certain nombre d'opérateurs logiques, des opérateurs agissant sur les chaînes de caractères, des opérateurs effectuant des tests d'identité ou d'appartenance, etc [1].

2. Priorité entre opérateurs

Lorsqu'il y a plus d'un opérateur dans une expression, l'ordre dans lequel les opérations doivent être effectuées dépend de règles de priorité. Sous Python, les règles de priorité sont les mêmes que celles qui vous ont été enseignées en cours de mathématique. Vous pouvez les mémoriser aisément à l'aide d'une astuce mnémotechnique, l'acronyme PEMMODEDASA [1]:

1. P pour parenthèses. Ce sont elles qui ont la plus haute priorité. Elles vous permettent donc de forcer l'évaluation d'une expression dans l'ordre que vous voulez. Ainsi $2*(3-1) = 4$, et $(1+1)**(5-2) = 8$.
2. E pour exposants. Les exposants sont évalués ensuite, avant les autres opérations. Ainsi $2**1+1 = 3$ (et non 4), et $3*1**10 = 3$ (et non 59049 !).
3. M, MO, DE et D pour multiplication, modulo, division entière et division, qui ont la même priorité. Elles sont évaluées avant l'addition A et la soustraction S.
4. A et S pour addition et soustraction. Ainsi $2*3-1 = 5$ (plutôt que 4), et $2/3-1 = -0.3333...$ (plutôt que 1.0).
5. A pour affectation est effectuée en dernier lieu le résultat du calcul de l'expression sera stockée dans la variable située à gauche du signe =.

Attention

Si deux opérateurs ont la même priorité, l'évaluation est effectuée entre les opérandes deux à deux et

de gauche à droite. Ainsi dans l'expression $59 * 100 // 60$, la multiplication est effectuée en premier entre les opérandes 59 et 100, le résultat étant 5900. Ensuite la machine doit effectuer $5900 // 60$, ce qui donne 98. Si la division avait été effectuée en premier, le résultat aurait été 59 [1].

Exercice : priorité de calcul à l'intérieur d'une expression



[solution n°5 p.36]

Mettez dans l'ordre les calcul effectués lors du calcul de l'expression : $9\%2^{**}2+(5*2+1)-3^{**}2//2$ en tenant compte des priorités entre opérateurs.

8

$3^{**}2$ qui sera remplacé par 9

$12-4$

$1+11$ qui sera remplacé par 12

$9//2$ qui sera remplacé par 4

$2^{**}2$ qui sera remplacé par 4

$(5*2+1)$ qui sera remplacé par 11

$9\%4$ qui sera remplacé par 1

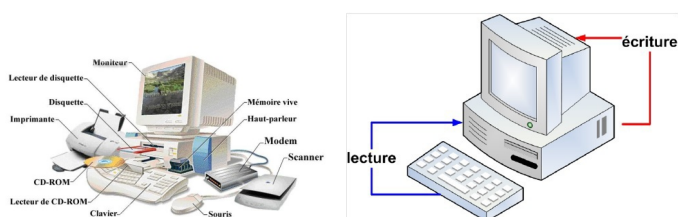
Les entrées-sorties

XI

Les entrées	28
Les sorties	29
Les séquences d'échappement	29

L'utilisateur a besoin d'interagir avec le programme au travers des périphériques d'entrées/sorties. En mode « console », 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 (sur imprimante, sur scanner, sur haut-parleur)[3].

Les entrées-sorties



1. Les entrées

Il s'agit de réaliser une saisie à l'écran : la fonction standard `input()` interrompt le programme, affiche une éventuelle invite et attend que l'utilisateur entre une donnée et la valide par Entrée [3].

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*) [3].

```

1 >>> nb_etudiant = input("Entrez le nombre d'étudiants : ")
2 Entrez le nombre d'étudiants : 23
3 >>> print(type(nb_etudiant)) # <class 'str'> (c'est une chaîne)
4 <class 'str'>
5 >>> f1 = input("\nEntrez un flottant : ")
6 Entrez un flottant : 2.4
7 >>> f1 = float(f1) # transtypage en flottant
8 # ou plus brièvement :
9 >>> f2 = float(input("Entrez un autre flottant : "))
10 Entrez un autre flottant : 4.6
11 >>> print(type(f2)) # <class 'float'>

```

```
12 <class 'float'>
```

2. Les sorties

En mode « calculatrice », Python lit-évalue-affiche, mais la fonction `print()` reste indispensable aux affichages dans les scripts [3]:

```
1 >>> a, b = 2, 5
2 >>> print(a, b) # 2 5
3 2 5
4 >>> print("Somme :", a + b) # Somme : 7
5 Somme : 7
6 >>> print(a - b, "est la différence") # -3 est la différence
7 -3 est la différence
8 >>> print("Le produit de", a, "par", b, "vaut :", a * b)
9 # Le produit de 2 par 5 vaut : 10
10 Le produit de 2 par 5 vaut : 10
11 >>> print() # affiche une nouvelle ligne
12
13 >>> print("On a <", 2**32, "> cas !", sep="###")
14 On a <###4294967296###> cas !
```

Dans `print("On a <", 2**32, "> cas !", sep="###")`, `sep="###"` permet d'afficher à la place d'un espace le séparateur de notre choix, `sep` intervient à l'endroit indiqué par une virgule dans le `print`, la fonction `print` affiche la chaîne de caractère *On a* suivi de `###` suivi du *résultat de 2**32* suivi de `###` suivi de *>cas !*

3. Les séquences d'échappement

À l'intérieur d'une chaîne, le caractère antislash (`\`) permet de donner une signification spéciale à certaines séquences [3]:

Séquence	Signification
<code>\saut_ligne</code>	saut de ligne ignoré
<code>\\</code>	affiche un antislash
<code>\'</code>	apostrophe
<code>\"</code>	guillemet
<code>\a</code>	sonnerie (bip)
<code>\b</code>	retour arrière
<code>\f</code>	saut de page
<code>\n</code>	saut de ligne
<code>\r</code>	retour en début de ligne
<code>\t</code>	tabulation horizontale
<code>\v</code>	tabulation verticale
<code>\N{nom}</code>	caractère sous forme de code Unicode nommé
<code>\uhhhh</code>	caractère sous forme de code Unicode 16 bits
<code>\Uhhhhhhh</code>	caractère sous forme de code Unicode 32 bits

Contenus annexes

> Méthodes retournant une valeur de type bool

Les méthodes suivantes sont à valeur booléenne, c'est-à-dire qu'elles retournent la valeur True ou False [3]:

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

```
1 >>> print("cHAise basSe".isupper()) # False
2 False
```

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

```
1 >>> print("Chaise Basse".istitle()) # True
2 True
```

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

```
1 >>> print("3 chaises basses".isalpha()) # False
2 False
3 >>> print("54762".isdigit()) # True
4 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` :

```
1 >>> print("abracadabra".startswith('ab')) # True
2 True
3 >>> print("abracadabra".endswith('ara')) # False
4 False
```

> Méthodes retournant une nouvelle chaîne

Les méthodes ci dessous retournent une nouvelle chaîne, c'est-à-dire qu'elles retournent un type `str` [3] :

- `lower()`, `upper()`, `capitalize()` et `swapcase()` : retournent respectivement une chaîne en minuscule, en majuscule, en minuscule commençant par une majuscule, ou en casse

inversée :

s sera notre chaîne de test pour toutes les méthodes

```
1 >>> s="cHAise basSe"
2 >>> print(s.lower()) # chaise basse
3 chaise basse
4 >>> print(s.upper()) # CHAISE BASSE
5 CHAISE BASSE
6 >>> print(s.capitalize()) # Chaise basse
7 Chaise basse
8 >>> print(s.swapcase()) # ChaISE BASsE
9 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) :

```
1 >>> print(s.center(20, '-')) # ----cHAise basSe----
2 ----cHAise basSe----
3 >>> print(s.rjust(20, '@')) # @@@@@@@@cHAise basSe
4 @@@@@@@@cHAise basSe
```

- `zfill(width)` : complète `ch` à gauche avec des 0 jusqu'à une longueur maximale de `width` :

```
1 >>> print(s.zfill(20)) # 00000000cHAise basSe
2 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 :

```
1 >>> print(s.strip('ce')) # HAise basS
2 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 :

```
1 >>> print(s.find('se b')) # 4
2 4
```

- `replace(old[, new[, count]])` : remplace `count` instances (toutes par défaut) de `old` par `new` :

```
1 >>> print(s.replace('HA', 'ha')) # chaise basSe
2 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 `splitlines()` effectue ce travail avec les caractères de fin de ligne :

```
1 >>> print(s.split()) # ['cHAise', 'basSe']
```



```
2 ['cHAise', 'basSe']
```

- `join(seq)` : concatène les chaînes du conteneur `seq` en intercalant la chaîne sur laquelle la méthode est appliquée:

```
1 >>> print("".join(['cHAise', 'basSe'])) # cHAise**basSe
2 cHAise**basSe
```

Solutions des exercices



> Solution n° 1

Exercice p. 8

Ci-dessous quelques noms de variables, sélectionnez ceux qui ne sont pas valides c'est à dire qui n'obéissent pas aux règles vu précédemment:

- lambda
- Nom_variable
- 3noms
- nom-variable
- Le_nom_suivi_dun_nombre4

> Solution n° 2

Exercice p. 11

a=0

b=a=5

z=2

a=z

Les instructions ci dessus peuvent être remplacées par une seule instruction, cochez la parmi les instructions ci dessous :

- a=b=z=2
- a=b=z=5
- a,b,z=2,5,2

> **Solution n° 3**

Exercice p. 16

Ordonnez les instructions selon les résultats d'exécutions sur python qui correspond à:

```
<class 'int'>
```

```
3.67
```

```
<class 'float'>
```

```
<class 'int'>
```

```
SyntaxError: invalid syntax
```

```
<class 'bool'>
```

```
x=2
```

```
type(x)
```

```
y=x+1.67
```

```
print(y)
```

```
type(y)
```

```
z=x
```

```
type(z)
```

```
for=5
```

```
b=True
```

```
type(b)
```

1. Il est clair que `type(x)` vient après `x=2`, puisque le type de `x` lui est directement associé au moment une valeur lui est affecté.
2. `type(y)` vient après `y=x+1.67` puisque cette expression calcule la somme d'un réel et d'un entier retournant ainsi un réel stocké dans `y`
3. Par contre l'instruction `type(z)` vient après `z=x` puisque la variable `z` se voit affectée la variable `x` qui est de type entier donc le type de `z` prend instantanément le même type de `x` (typage dynamique fort des variables).
4. `for=5` pose un problème puisqu'on ne peut pas utiliser un mot réservé pour identifier une variable, `for` fait partie des 33 mots clés réservés de python. Cliquer sur *Noms de variables et mots réservés.* - p.

6

> **Solution n° 4**

Exercice p. 22

Voici la chaîne de caractère `module="programmation python"`, mettez dans l'ordre les résultats d'exécution des instructions suivantes :

```
>>> len(module)
```

```
>>> module[0:14]
```

```
>>> module[-20:]
```

```
>>> print(module.startswith("pro"))
```

```
>>> print(module.endswith('yn'))
```

```
>>> print(module.capitalize())
```

```
>>> print(module.upper())
```

```
>>> print(module)
```

```
>>> module=module.upper()
```

```
>>> module
```

```
>>> print(module.find('ON'))
```

20

'programmation '

'programmation python'

True

False

Programmation python

PROGRAMMATION PYTHON

programmation python

'PROGRAMMATION PYTHON'

11

- Remarquer la différence d'affichage de la variable module entre >>> module et >>> print(module.upper()) dans la première nous précise que la variable est une chaîne de caractère grâce aux apostrophes, dans la deuxième la fonction print affiche le contenu de la variable module sans apostrophes.
- print(module.upper()) retourne le résultat d'application de upper() sur la chaîne module donc après l'affichage à l'aide de >>> print(module) nous remarquons que la chaîne initiale n'a pas changé car nous n'avons pas sauvegardé le résultat de retour de la fonction dans la chaîne module, l'instruction suivante permet de le faire :

```
>>> module=module.upper()
```

l'instruction >>> module permet l'affichage de la chaîne module qui est en majuscule !

Pour revoir les méthodes sur chaînes : *cliquez ici - p.31* pour revoir les méthodes qui retournent une valeur booléenne ou *cliquez ici - p.31* pour revoir les méthodes retournant une nouvelle chaîne !

> Solution n°5

Exercice p. 27

Mettez dans l'ordre les calcul effectués lors du calcul de l'expression : $9\%2^{**}2+(5*2+1)-3^{**}2//2$ en tenant compte des priorités entre opérateurs.

(5*2+1) qui sera remplacé par 11

2**2 qui sera remplacé par 4

3**2 qui sera remplacé par 9

9%4 qui sera remplacé par 1

9//2 qui sera remplacé par 4

1+11 qui sera remplacé par 12

12-4

8

- Rappel : les parenthèses ont la plus forte priorité, ensuite la priorité revient à ** (l'exposant) puis la multiplication, la division avec ses variantes (/,,%) et enfin addition et soustraction.
- Si deux opérations ont la même priorité le calcul se fait de gauche à droite par exemple : ici 2**2 et 3**2 ont la même priorité mais python va calculer 2**2 en premier suivi de 3**2.

Bibliographie

XI

[1] Swinnen Gérard, 02/02/2012, "Apprendre à programmer avec python 3", (3e édition), Eyrolles, 435 p, Noire

[2] Jachym, Marc, "Cours de Programmation avec le langage Python: Niveau débutant en programmation", Licence professionnelle – Métrologie dimensionnelle et qualité IUT de St Denis, Université Paris 13.

[3] Cordeau, Bob, Introduction à Python 3, version 2.71828.