transparents_calculatricePython

September 18, 2020

1 La calculatrice Python

Comme tout langage de programmation, Python permet de manipuler des données grâce à un vocabulaire de mots réservés et grâce à des types de données.

Ce chapitre présente les règles de construction des identificateurs, les types de données simples (les conteneurs seront examinés plus tard) ainsi que les types chaînes de caractères.

Pour finir, ce chapitre s'intéresse aussi aux notions non triviales de variable, de référence d'objet et d'affectation.

1.1 Les deux modes d'exécution d'un code Python

Le mode le plus direct et le plus intuitif d'exécution de Python est l'utilisation interactive de l'interpréteur. C'est le *shell Python* (ou "console Python"), aussi appelé *mode interprété*.
Lorsque l'on tape la commande `python` dans une console, une invite apparaît. L'interpréteur attend vos instructions, les exécute quand vous avez tapé sur la touche entrée. C'est ce que l'on appelle la *boucle d'évaluation* (en anglais REPL pour Read-Eval-Print Loop).

```
>>> 5 + 3   # Python affiche l'invite. L'utilisateur tape une expression
8           # Python évalue et affiche le résultat ...
>>>         # ... puis réaffiche l'invite.
```

Mais dès que l'on travaille avec plus de quelques lignes de code, le mode interprété devient malcommode. On passe en *mode script* : on enregistre un ensemble d'instructions Python dans un fichier grâce à un éditeur. (On parle alors de scipt Python.)
Ce script est exécuté ultérieurement (et autant de fois que l'on veut) par une commande ou par une touche du menu de l'éditeur, il peut être corrigé puis réexecuté dans son ensemble.

1.2 Les commentaires

Un programme *source* est destiné à l'être humain. Pour en faciliter la lecture, il doit être judicieusement présenté et commenté de façon pertinente.
La signification de parties non triviales doit être donnée par un *commentaire*. En Python, un commentaire commence par le caractère # et s'étend jusqu'à la fin de la ligne :

[1]:
```
# ---------------------
# Voici un commentaire
#  ---------------------
```

```
9 + 2 # en voici un autre
```

[1]: 11

1.3 Identificateurs et mots clés

1.3.1 Identificateurs

Définition:
Un **identificateur** Python est une suite non vide de caratères, de longueur quelconque, formée d'un caractère de début (n'importe quelle lettre Unicode ou le caractère souligné) et de zéro au sens de "aucun") ou plusieurs caractères de continuation (lettre Unicode, caractère souligné ou chiffre).

Attention :
Les identificateurs sont sensibles à la casse et ne doivent pas faire partie des mots réservés de Python.

Le choix d'un bon identificateur est important car il doit permettre, lors de la rédaction et de la lecture du code, de comprendre ce qu'il représente ; c'est un point qu'il ne faut pas négliger.
On peut avoir des identificateurs très courts comme `x`, `y`, `z`, `a`, `b`, `c`, ... pour autant que leur sens dans le contexte soit pertinent (`x` pour une valeur de calcul ; `a`, `b`, `c` pour des coefficients d'une équation ; `f` pour une fonction quelconque ...) - on évitera de les utiliser par facillité si cela n'a pas de sens, tout comme des `var1`, `var2`, `var3` ... pour lesquels il devient difficile de mémoriser l'utilisation au bout de quelques lignes de programme, et qui sont de ce fait souvent causes d'erreurs.

En général quelques caractères permettent déjà de donner du sens comme `som` (somme), `fct` (fonction), `maxi`, `mini`, `stop`, `start`. Mais il ne faut pas hésiter à les allonger si nécessaire : `maxi_x`, `augmentation_son` ...

1.3.2 Style de nommage

Il est important d'utiliser une politique cohérente de nommage des identificateurs. Voici le style utilisé dans le cadre de ce cours (conforme a la PEP8) :

- `NOM_DE_MA_CONSTANTE` pour les constantes ;
- `maFonction`, `maMethode` pour les fonctions et les méthodes ;
- `MaClasse` pour les classes;
- `UneExceptionError` pour les exceptions ;
- `nom_de_ma_variable` pour les variables et pour les autres identificateurs

Exemples :

[2]:
```
NB_TEMS = 12            # appelé "UPPER_CASE_WITH_UNDERSCORES"
class MaClasse: pass    # appelé "CamelCase" ou "CapitalizedWords"
def maFonction(): pass  # appelé "mixedCase"
mon_id = 5              # appelé "lower_case_with_underscores"
```

Pour ne pas prêter à confusion, éviter d'utiliser les caractères `l` (minuscule), `O` et `I` (majuscules) seuls. Enfin, on évitera d'utiliser les notations suivantes :

```
_xxx     # usage interne
__xxx    # attribut de classe
__xxx__  # nom spécial réservé
```

1.3.3 Les mots réservés de Python3

La version 3.6 de Python compte 33 mots clés :

```
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
```

1.4 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 (représentant directement des valeurs), d'identificateurs et d'opérateurs.

Par exemple :

```
>>> id1 = 15.3
>>> id2 = 4 + 3 sin(pi)
>>> id3 = "-"*10 + "Titre" + " "*10
```

1.5 Les types de données entiers

Définition :
Le **type** d'un objet Python détermine de quelle sorte d'objet il s'agit.
La fonction `type` fournit le type d'une valeur.

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

1.5.1 Le type `int`

Le type `int` n'est limité en taille que par la mémoire de la machine.
(Dans la plupart des autres langages, les entiers sont codés sur un nombre fixe de bits et ont un domaine de définition limité auquel il convient de faire attention. Par exemple, un entier signé sur 16 bits représente un nombre entre -32 768 et +32 767.)

Les entiers littéraux sont représentés en décimal par défaut, mais on peut aussi utiliser les bases suivantes :

[3]:
```
# décimal     (base 10) par défaut :
print('symboles     (decimal) : 2013           -> entier :',2013)
```

```
# binaire     (base  2) avec le préfixe 0b :
print('symboles     (binaire) : 0b11111011101  -> entier :',0b11111011101)
# octal       (base  8) avec le préfixe 0o :
print('symboles       (octal) : 0o3735         -> entier :',0o3735)
# hexadécimal (base 16) avec le préfixe 0x :
print('symboles (hexadecimal) : 0x7dd          -> entier :',0x7dd)
# représentations binaire, octale et hexadécimale de l'entier 179 :
bin(179), oct(179), hex(179)
```

```
symboles     (decimal) : 2013           -> entier : 2013
symboles     (binaire) : 0b11111011101  -> entier : 2013
symboles       (octal) : 0o3735         -> entier : 2013
symboles (hexadecimal) : 0x7dd          -> entier : 2013
```

[3]:
```
('0b10110011', '0o263', '0xb3')
```

Ces dernières opérations correpondent à des changements de bases de représentation d'un entier.

Opérations arithmétiques Les principales opérations :

[4]:
```
print('addition         : 20 + 3  =' , 20 + 3 ,
      '\t soustraction    : 20 - 3  =', 20 - 3)
print('multiplication   : 20 * 3  =' , 20 * 3 ,
      '\t puissance       : 20 ** 3 =', 20 ** 3)
print('valeur absolue   : abs(3 - 20) =',abs(3 - 20),
      '\t modulo          : 20 % 3  =', 20 % 3,end="\n\n")
print('division entiere : 20 // 3 = ', 20 // 3,
    '\t division          : 20 / 3  =', 20 / 3,end="\n\n")
print('division entiere et modulo : divmod(20,3) =', divmod(20,3))
5+2
```

```
addition         : 20 + 3  = 23          soustraction    : 20 - 3  = 17
multiplication   : 20 * 3  = 60          puissance       : 20 ** 3 = 8000
valeur absolue   : abs(3 - 20) = 17      modulo          : 20 % 3  = 2

division entiere : 20 // 3 =  6          division          : 20 / 3  =
6.666666666666667

division entiere et modulo : divmod(20,3) = (6, 2)
```

[4]:
```
7
```

[5]:
```
5+3
```

[5]:
```
8
```

Bien remarquer le rôle des deux opérateurs de division :

- `/` : produit une division flottante, même entre deux entiers. (Ceci est une différence majeure avec de nombreux autres langages où une division entre deux entiers est une division obligatoirement entière.)
- `//` : produit une division entière.

Remarquons que la fonction prédéfinie `divmod` qui prend deux entiers et renvoie la paire `q`, `r` où `q` est le quotient et `r` le reste de leur division, évite d'utiliser `//` pour obtenir `q` et `%` pour `r`

Les bases arithmétiques Certaines bases sont couramment employées : la base 2 (système binaire) en électronique numérique, les bases 8 et 16 (système octal et hexadécimal) en informatique, la base 60 (système sexagésimal) dans la mesure du temps et des angles.

> **Définition** :
> En arithmétique, une **base n** désigne la valeur dont les puissances successives interviennent dans l'écriture des nombres, ces puissances définissant l'ordre de grandeur de chacune des positions occupées par les chiffres composants tout le nombre.
> Par exemple, en base n : $(57)_n = (5 \times n^1) + (7 \times n^0)$

Puisqu'un nombre dans une base n donnée s'écrit sous la forme d'addition des puissances successives de cette base, on peut facilement effectuer une changement de base :

$$(57)_{16} = (5 \times 16^1) + (7 \times 16^0) = (87)_{10}$$

$$(28)_{10} = (3 \times 8^1) + (4 \times 8^0) = (34)_8$$

Inversement, dans le deuxième exemple, pour passer de la base 10 à la base n, il faut connaître les puissances successives de n.

1.5.2 Le type `bool`

Les opérateurs booléens de base En Python les représentations littérales des valeurs booléennes sont notées `False` et `True`. Les opérateurs de base sont notés respectivement `not` (non), `and` (et) et `or` (ou).

opérateur unaire `not`

a	not(a)
False	True
True	False

Opérateur binaires `or` **et** `and`

a	b	a or b	a and b
False	False	False	False
False	True	True	False
True	False	True	False
True	True	True	True

Voici les principales caractéristiques du type `bool` :

- deux valeurs possibles : `False` (faux), `True` (vrai) ;
- conversion automatique (ou "transtypage", en anglais `cast`) des valeurs des autres types vers le type booléen : zéro (quel que soit le type numérique), les chaînes et conteneurs vides, la constante `None`, les objets dont une méthode spéciale `__bool__` ou `__len__` retourne `0` ou faux sont convertis en booléen `False` ; toutes les autres valeurs sont converties en booléen `True` ;
- opérateurs de comparaison entre deux valeurs comparables, produisant un résultat de type `bool` : == (égalité), `!=` (différence), >, >=, < et <=

[6]:
```
print('2 > 8       :', 2 > 8)
print('2 <= 8 < 15 :', 2 <= 8 < 15)
```

```
2 > 8       : False
2 <= 8 < 15 : True
```

- opérateurs logiques : `not`, `or` et `and`. En observant les tables de vérité des opérateurs `and` et `or`, on remarque que :
  - dès qu'un premier membre à la valeur `False`, l'expression `False and expression2` vaudra `False`. On n'a donc pas besoin d'évaluer `expression2`,
  - de même dès qu'un premier membre à la valeur `True`, l'expression `True or expression2` vaudra `True`, quelque soit la valeur de `expression2` ;

Cette optimisation est appelée "principe de shortcut" ou évaluation "au plus court", elle est automatiquement mise en oeuvre par Python lors de l'évaluation des expressions booléennes :

[7]:
```
print('(3 == 3) or  (9 > 24) :', (3 == 3) or  (9 > 24))
print('(9 > 24) and (3 == 3) :', (9 > 24) and (3 == 3))
```

```
(3 == 3) or  (9 > 24) : True
(9 > 24) and (3 == 3) : False
```

> **Attention** :
> Pour être sûr d'avoir un résultat booléen avec une expression reposant sur des valeurs transtypées, appliquer la fonction `bool` sur l'expression. En effet, lorsqu'il rencontre des valeurs non booléennes dans une expression logique. Python effectue des conversions de type automatique sur les données mais le résultat de l'évaluation utilise les valeurs d'origine.

Par exemple :

[8]:
```
# 'a' est automatiquement transtypé en booléen :
print("'a' or  False :", 'a' or False)
# 0 et 56 sont automatiquement transtypés en booléen :
print(" 0  or  56    :", 0 or 56)
b = 0
# b est automatiquement transtypé en booléen :
print(" b  and 3>2   :", b and 3>2)
```

```
'a' or  False : a
 0  or  56    : 56
 b  and 3>2   : 0
```

1.6 Les types de données flottants

Remarque :
La notion mathématique de réel est une notion idéale. Ce graal est impossible à atteindre en informatique. On utilise une représentation interne (normalisée) permettant de déplacer la virgule grâce à une valeur d'exposant variable. On nommera ces nombres des nombres à virgule flottante, ou, pour faire plus court, des flottants.

1.6.1 Le type `float`

- Un `float` est noté avec un point décimal (jamais avec une virgule) ou, en notation exponentielle, avec un `e` ou un `E` symbolisant le "10 puissance" suivi des chiffres de l'exposant. Par exemple : `2.718`, `.02`, `3E10` `-1.6e-19`, `6.023E23`.
- Les flottants supportent les mêmes opérations que les entiers
- Ils ont une précision limitée.
- L'import du module standard `math` autorise toute les opérations mathématiques usuelles. Par exemple :

[9]:
```
import math
print("math.sin(math.pi/4)   :", math.sin(math.pi/4))
print("math.degrees(math.pi) :", math.degrees(math.pi))
print("math.factorial(9)     :", math.factorial(9))
print("math.log(1024,2)      :", math.log(1024,2))
type(math.factorial(9))
```

```
math.sin(math.pi/4)   : 0.7071067811865475
math.degrees(math.pi) : 180.0
math.factorial(9)     : 362880
math.log(1024,2)      : 10.0
```

[9]:
```
int
```

Note : Nous apprendrons ultérieurement comment ne pas avoir à spécifier le préfixe `math.` à chaque fois.

1.6.2 Le type `complex`

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

[10]:
```
print("1j :", 1j)
print("(2+3j) + (4-7j)", (2+3j) + (4-7j))
print("(9+5j).real :", (9+5j).real)
print("(9+5j).imag :", (9+5j).imag)
```

```
print("abs(3+4j) :", abs(3+4j))
```

```
1j : 1j
(2+3j) + (4-7j) (6-4j)
(9+5j).real : 9.0
(9+5j).imag : 5.0
abs(3+4j) : 5.0
```

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

[11]:
```
import cmath
print("cmath.phase(-1+0j) :", cmath.phase(-1+0j))
print("cmath.polar(3+4j) :", cmath.polar(3+4j))
print("cmath.rect(1.,cmath.pi/4) :", cmath.rect(1.,cmath.pi/4))
print("cmath.sqrt(3+4j) :", cmath.sqrt(3+4j))
```

```
cmath.phase(-1+0j) : 3.141592653589793
cmath.polar(3+4j) : (5.0, 0.9272952180016122)
cmath.rect(1.,cmath.pi/4) : (0.7071067811865476+0.7071067811865475j)
cmath.sqrt(3+4j) : (2+1j)
```

1.7 Variables et affectation

1.7.1 Les variables

Pour stocker des données, on utilise des *variables*. Plus précisément, Python n'offre pas la notion de variable comme adresse mémoire identifiée (comme avec le langage C) mais plutôt celle de références d'objets par des noms.

> **Définition** :
> Une **variable** est un identificateur associé à une valeur. En Python, c'est une **référence d'objet**.

Tant que l'objet n'est pas modifiable (entier, flottant, chaîne de caractère, etc.), il n'y a pas de différence notable entre variable et référence. On verra que la situation change dans le cas des objets modifiables...
Une variable spéciale : `_`, utilisable uniquement en mode interactif, contient le résultat de la dernière opération :

```
>>> 2 * 7 - 5
9
>>> _ + 3   # equivalent a 9 + 3
12
```

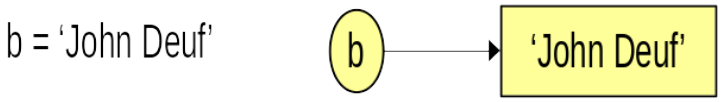
1.7.2 L'affectation (ou assignation)

> **Syntaxe** :
> On affecte une valeur à une variable en utilisant le signe d'égalité (=).
> Attention : l'affectation n'a rien à voir avec l'égalité en math!

[12]:
```
a = 2
e = a == 2       # True
print('e :',e)
b = 'John Deuf'
print("b :",b)
```

```
e : True
b : John Deuf
```

Remarque :
Puisqu'une variable est une référence, pour ne pas confondre affectation et égalité mathématique, `b = 'John Deuf'` pourra se dire "b pointe sur 'John Deuf' ".
On illustre ce mécanisme par un cercle qui contient un nom de référence et un rectangle une valeur.



La valeur d'une variable, comme son nom l'indique, peut évoluer au cours du temps par des opérations de réaffectation. Dans de telles opérations, la valeur antérieure est perdue :

[13]:
```
a = 3 * 7
print("a :",a)
b = 2 * 2
print("b :",b)
a = b + 5
print("a :",a)
```

```
a : 21
b : 4
a : 9
```

Le membre de droite d'une affectation étant évalué avant de réaliser l'affectation elle-même, la variable affectée peut se trouver en partie droite et c'est sa valeur avant l'affectation qui est utilisée dans le calcul :

[14]:
```
a = 2
a = a + 1        # incrémentation
print("a :",a)
a = a -1         # décrémentation
print("a :",a)
```

```
a : 3
a : 2
```

Un bon exemple d'utilisation d'un même nom de variable en partie gauche et en partie droite est le calcul du terme suivant d'une suite numérique :

[15]:
```
u_n = 4
u_n = 3 * u_n + 1
print("u_n :",u_n)
```

```
u_n : 13
```

1.7.3 Attention : affecter n'est pas comparer !

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) :

[16]:
```
c = True                # c a été créé et référence la valeur True
s = (c=True) and True   # On ne peut pas affecter c dans une expression
```

```
  File "<ipython-input-16-3df6f9d93ccb>", line 2
    s = (c=True) and True  # On ne peut pas affecter c dans une expression
         ^
SyntaxError: invalid syntax
```

La comparaison à une valeur (de type `bool`) est utilisable dans une expression mais n'a pas d'effet :

[ ]:
```
c = 'a'
s = (c == 'a') and True
print("s :", s)
print("c :", c)
```

1.7.4 Les variantes de l'affectation

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

[17]:
```
# affectation simple :
v =  4
# affectation augmentée. Idem à v = v + 2 si v est déjà référencé :
v += 2
print("v :", v)
# d reçoit 8, puis c reçoit d. Ils référencent la même donnée (alias) :
c = d = 8
print("c, d :", (c,d))
# affectation parallèle par décapsulation d'un n-uplet (tuple) :
e, f = 2.7, 5.1
print("e :",e,"\t f :",f)
a = 3
```

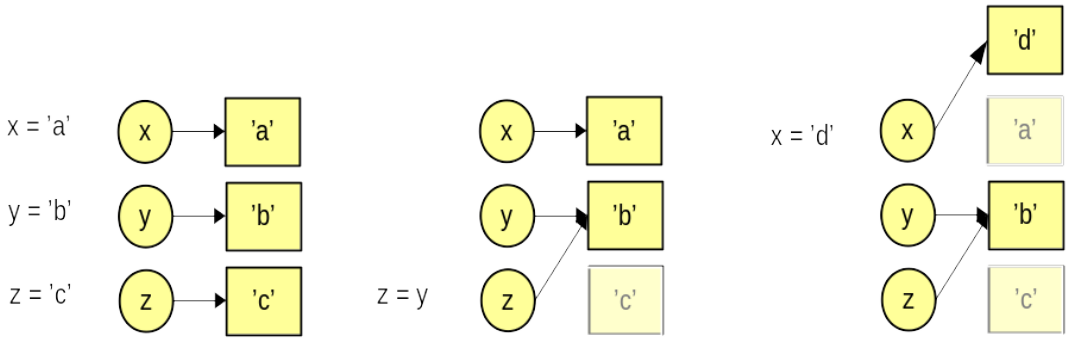
```
# toutes les expressions sont évaluées avant la première affectation :
a, b = a + 2, a * 2
print("a :",a,"  \t b :",b)
```

```
v : 6
c, d : (8, 8)
e : 2.7          f : 5.1
a : 5            b : 6
```

Remarque :
Dans une affectation, le membre de gauche pointe sur le membre de droite, ce qui nécessite d'évaluer la valeur du membre de droite **avant** de l'affecter au membre de gauche. On voit dans l'affectation parallèle l'importance de cette séquence temporelle.

1.7.5 Représentation graphiques des affectations

Les affectations relient les identificateurs aux données : si une donnée en mémoire n'est plus reliée, le ramasse-miettes (garbage collector en anglais) de Python la supprime automatiquement (car son nombre de références tombe à zéro :



(a) Trois affectations

(b) La donnée 'c' est supprimée lorsque z référence y.

(c) La donnée 'a' est supprimée lorsque x référence 'd'.

1.7.6 Suppression d'une variable

Puisque tout est dynamique en Python, il est possible au cours de l'exécution de supprimer une variable, donc de supprimer un nom qui référence une données. L'instruction qui permet cela est `del`.

[18]:
```
a = 3
del a
a
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)
```

```
<ipython-input-18-4c6f95158088> in <module>
  1 a = 3
  2 del a
----> 3 a


NameError: name 'a' is not defined
```

1.8 Les chaînes de caractères

1.8.1 Présentation

Définition :
Le type de données **chaînes de caractères**, `str`, est *non modifiable*. Il permet de représenter une séquence de caractères Unicode. *Non modifable* signifie qu'une donnée, une fois créée en mémoire, ne pourra plus être changée ; toute transformation aboutira à la création d'une *nouvelle* valeur distincte. On utilise souvent le terme "immutable".

Les chaînes de caractères sont des valeurs textuelles (espaces, symboles alphanumériques ...) entourées par des guillemets simples ou doubles, ou par une série de trois guillemets simples ou doubles.

L'utilisation de l'apostrophe (') à la place des guillemet (") autorise l'inclusion d'une notation dans l'autre. La notation entre trois guillemets permet de composer des chaînes sur plusieurs lignes contenant elles-mêmes des guillemets simples ou doubles. On verra ultérieurement que cette utilisation est très utile pour documenter des parties de programme.

Exemple :

[20]:
```
guillemets = "L'eau vive"
apostrophes = 'Il a écrit : "Ni le mort ni le vif, mais les merveilleux !"'
doc = """    forme multiligne très utile pour la documentation d'un script
   ou pour inclure un fragment de programme dans une chaîne de caractères :

for i in range(2, 2*n+1):
    if i%2 == 0:   # indice pair
        monge.insert(0, i)
    else:
        monge.append(i)"""
```

1.8.2 Les séquences d'échapement

A l'intérieur d'une chaîne, le caractère antislash (\) permet de donner une signification spéciale à certaines séquences de caractères.

Séquence	Signification
\	saut de ligne ignoré (placé en fin de ligne)
\\	antislash

Séquence	Signification
`\'`	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

Séquence	Signification
`\uhhhh`	caractère sous forme de code Unicode 16 bits sur 4 chiffres hexadécimaux
`\uhhhhhhhh`	caractère sous forme de code Unicode 32 bits sur 8 chiffres hexadécimaux
`\ooo`	caractère sous forme de code octal sur 3 chiffres octaux
`\xhh`	caractère sous forme de code hexadécimal sur 2 chiffres

[21]:
```
print("Affichage 1 :","\N{pound sign} \u00A3 \U000000A3")
print("Affichage 2 :","d \144 \x64")
print("Affichage 3 :",r"d \144 \x64")
```

```
Affichage 1 : £ £ £
Affichage 2 : d d d
Affichage 3 : d \144 \x64
```

Remarque : La notation `r"..."` désactive la signification spéciale du caractère.

1.8.3 Opérations

Outre les fonctions et méthodes que nous allons voir, les quatre opérations suivantes : longueur, concaténation (i.e. mise bout à bout de deux chaînes), répétition et test d'apartenance s'appliquent au type chaînes :

[22]:
```
print('len("abcde") :',len("abcde"))
print('"abc" + "defg" ->',"abc" + "defg")
print('"Fi! " * 3 ->',"Fi! " * 3)
print("'y' in 'Python' ->",'y' in 'Python')
```

```
len("abcde") : 5
"abc" + "defg" -> abcdefg
"Fi! " * 3 -> Fi! Fi! Fi!
'y' in 'Python' -> True
```

1.8.4 Fonctions vs méthodes

On peut agir sur une chaîne en utilisant des fonctions (notion procédurale) communes à tous les types séquences ou conteneurs, ou bien des méthodes (notion objet) spécifiques aux chaînes :

[23]:
```
# len('Les auteurs') : syntaxe d'une fonction :
print("len('Les auteurs') :",len('Les auteurs'))
# "abracadabra".upper() : syntaxe d'une méthode :
"abracadabra".upper()
```

```
len('Les auteurs') : 11
```

[23]:
```
'ABRACADABRA'
```

1.8.5 Méthodes de test de l'état d'une chaîne

Il s'agit de méthodes à valeur booléenne, c'est-à-dire qu'elles retournent la valeur `True` ou `False`.

> **Syntaxe** :
> La notation entre crochets `[xxx]` indique un élément optionnel, que l'on peut donc omettre lors de l'utilisation de la méthode.

Voici quelques exemples de ces méthodes de test.

[24]:
```
# tout en majuscule :
print("'Le petit PRINCE'.isupper() :",'Le petit PRINCE'.isupper())
# chaque mot commence par une majuscule :
print("'Le petit Prince'.istitle() :",'Le petit Prince'.istitle())
# ne contient que des caracteres alphabetique :
print("'Prince'.isalpha()          :",'Prince'.isalpha())
# ne contient que des caracteres numeriques :
print("'Un'.isdigit()              :",'Un'.isdigit())
# commence par ... :
print("'Le Petit Prince'.startswith('Le ')  :",'Le Petit Prince'.startswith('Le␣
 ↪'))
# finit par ... :
print("'Le Petit Prince'.endswith('prince') :",'Le Petit Prince'.
 ↪endswith('prince'))
```

```
'Le petit PRINCE'.isupper() : False
'Le petit Prince'.istitle() : False
'Prince'.isalpha()          : True
'Un'.isdigit()              : False
'Le Petit Prince'.startswith('Le ')  : True
'Le Petit Prince'.endswith('prince') : False
```

1.8.6 Méthode retournant une nouvelle chaîne

Comme les chaînes sont non modifiables ou immutables (c'est-à-dire que leur contenu ne peut pas changer), les méthodes qui les modifient retournent de nouvelles chaînes.

En voici quelques exemples :

[25]:
```
s = 'Le petit PRINCE'; print("original          :",s)
print("tout en minscule  :",s.lower())
print("tout en majuscule :",s.upper())
print("inverser la casse :",s.swapcase())
print("chaîne centrée    :",s.center(31,'~'))
print("chaîne justifiée à droite :",'Le petit PRINCE'.rjust(31,'^'))
print("suppression de caractères en début de chaîne :",'Le petit PRINCE'.
 ↪lstrip('e L'))
# découpe la chaîne suivant le séparateur :
'Le petit PRINCE'.split()  # (séquence d'espace par défaut)
```

```
original          : Le petit PRINCE
tout en minscule  : le petit prince
tout en majuscule : LE PETIT PRINCE
inverser la casse : lE PETIT prince
chaîne centrée    : ~~~~~~~~Le petit PRINCE~~~~~~~~
chaîne justifiée à droite : ^^^^^^^^^^^^^^^^Le petit PRINCE
suppression de caractères en début de chaîne : petit PRINCE
```

[25]:
```
['Le', 'petit', 'PRINCE']
```

1.8.7 Méthode retournant un indice

`find(sub[, start[, stop]])` renvoie l'indice de la chaîne `sub` dans la sous-chaîne allant de `start` à `stop` (`stop` non inclus), sinon renvoie `-1`. La méthode `rfind` effectue le même travail en commençant par la fin.
Les méthodes `index` et `rindex` font de même mais produisent une erreur (*exception*) si la chaîne `sub` n'est pas trouvée :

[26]:
```
print('indice trouve :','Le petit Prince'.find('Pr'))
'Le petit Prince'.index('bad')                    # génération d'une exception
```

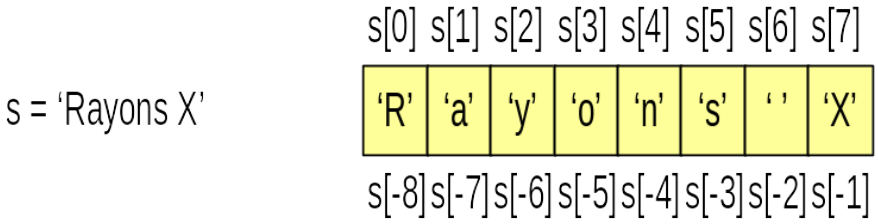
```
indice trouve : 9
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-26-f6af1a8603f5> in <module>
  1 print('indice trouve :','Le petit Prince'.find('Pr'))
----> 2 'Le petit Prince'.index('bad')                    # génération
↪d'une exception


ValueError: substring not found
```

1.8.8 Indexation simple

Syntaxe :
L'opérateur d'indexation utilise la notation `[index]` dans lequel `index` est un entier signé qui commence à `0` et indique la position d'un caractère.

L'utilisation de valeurs d'indice négatives permet d'accéder aux caractères par la fin de la chaîne :

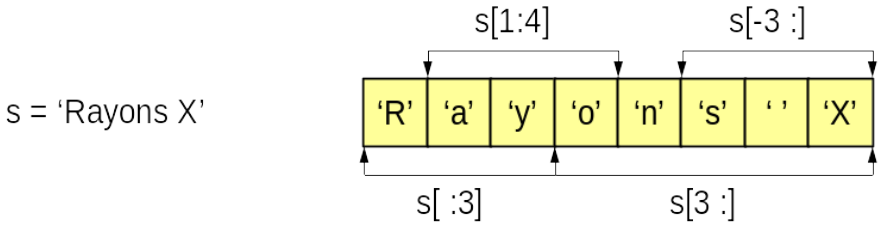



```
[ ]: s = "Rayons X" ;print("s       :", s)
     print("s[0]  :", s[0])    # premier caractère
     print("s[2]  :", s[2])    # troisième caractère
     print("s[-1]   :", s[-1]) # dernier caractère
     print("s[-3]   :", s[-3]) # antepénultième caractère
```

1.8.9 Extraction de tranches

Syntaxe :
L'opérateur d'extraction de tranche `[debut:fin]` ou `[debut:fin:pas]`, dans lequel `debut` et `fin` sont des indices de caractères et `pas` est un entier signé, permet d'extraire des tranches.

L'omission de `debut` ou de `fin` permet de spécifier du début (ou respectivement de la fin) de la chaîne.
Par exemple :




```
[ ]: s = "Rayons X" ;  print("s       :", s)
     print("s[1:4]  :", s[1:4]) # de 1 a 4 non compris
     print("s[-3:]  :", s[-3:]) # de -3 compris à la fin (i.e. les trois derniers)
     print("s[:3]   :", s[:3])  # du début à l'indice 3 non compris  (i.e. les trois␣
      ↪premiers)
     print("s[3:]   :", s[3:])  # de l'indice 3 compris a la fin
```

```
print("s[::2]  :", s[::2]) # du debut à la fin par pas de 2
print("s[::-1] :", s[::-1])# de la fin au debut en pas inverse
```

Avez-vous compris ? (Effacer l'affichage des sorties et faire ce qui suit.)

[ ]:
```
print("s[::]     :", s[::])     # ???
print("s[::-1]   :", s[::-1])   # ???
print("s[1:4:-1] :", s[1:4:-1]) # ???
print("s[:1:-1]  :", s[:1:-1])  # ???
print("s[4::-1]  :", s[4::-1])  # ???
print("s[4:1:-1] :", s[4:1:-1]) # ???
print("s[1:4:1]  :", s[1:4:1])  # ???
```

1.8.10 Opérateur de formatage des chaînes

Pour mettre en forme une chaîne de caractères (par exemple en vue de l'afficher), plusieurs solutions existent. Voici la syntaxe la plus simple (héritée du langage C) qui utilise l'opérateur %.

> **Syntaxe** : `format % valeurs` Où `format` est une chaîne contenant des spécificateurs de format et `valeurs` est un n-uplet (tuple) ou un dictionnaire contenant une ou plusieurs valeurs. Chaque spécificateur est formé du caractère % suivi d'un caractère dont la signification est donnée dans le taleau ci-après. A l'affichage, cette spécification est remplace, dans l'ordre, par son correspondant dans `valeurs`.

Par exemple :

[ ]:
```
ch = 'trois'
"%s %d %s" % ('One', 2, ch+' Go!')
```

Les spécificateurs de format % sont donnés dans le tableau suivant :

Caractères	Format de sortie
`d`, `i`	entier décimal signé
`u`	entier décimal non signé
`o`	entier octal non signé (sans le préfixe 0o)
`x`, `X`	entier hexadécimal non signé écrit en minuscule (`x`) ou en majuscule (`X`)
`e`, `E`	flottant forme exponentielle
`f`, `F`	flottant forme décimale
`g`, `G`	comme `e` si l'exposant est supérieur à 4, comme `f` sinon
`c`	caractère simple
`s`	chaîne interprétée par l'application de la fonction `str`
`r`	chaîne interprétée par l'application de la fonction `repr`
`%`	le caractère littéral %

Par exemple :

[ ]:
```
print("'%s' % (1 / 3.0)           :", '%s' % (1 / 3.0))
print('"%.2f" % (1 / 3.0)         :', "%.2f" % (1 / 3.0))
# formatage à l'aide d'un dictionnaire :
print('"%(toto)s" % dict(toto=12) :', "%(toto)s" % dict(toto=12))
```

1.8.11 Affichage formaté

La méthode `format` permet de contrôler finement la création de chaînes formatées. On l'utilisera pour un affichage via `print`, pour un enregistrement dans un fichier via `write`, ou dans d'autres cas de flux de sortie. Chaque paire d'accolades désigne un champ qui est rempli avec la valeur d'un des paramètres de la méthode `format`, suivant les directives données entre les accolades.

Remplacements simples :

[ ]:
```
print("{} {} {}".format("zero","un","deux")) # affiche : zero un deux

#formatage d'une chaîne pourusage ultérieurs
chain = "{2} {0} {1}".format("zero","un","deux")
print(chain) # affiche : deux zero un

print("Je m'appelle {}.".format("Bob"))      # affiche : Je m'appelle Bob.
print("Je m'appelle {{{}}}.".format("Bob")) # affiche : Je m'appelle {Bob}.
print("{}".format("-"*10))                   # affiche : ----------

# adaptation du message de saisie :
msg = "Saisir la {}e valeur : "
lst = []  # creation d'une liste vide
for i in range(1, 3):
    lst.append(input(msg.format(i)))
```

Remplacement avec champs nommés :

[27]:
```
a, b = 5, 3
print("The story of {c} and {d}".format(c=a+b,d=a-b)) # The story of 8 and 2
```

```
The story of 8 and 2
```

Formatages à l'aide d'une liste (notion introduite dans la suite). Le premier nombre indice l'argument de 'format' à utiliser, le nombre entre crochets spécifie l'indice dans cet argument :

[28]:
```
stock = ['papier', 'enveloppe', 'chemise', 'encre', 'buvard']
print("Nous avons l'{0[3]} et du {0[0]} en stock.\n".format(stock))
```

```
Nous avons l'encre et du papier en stock.
```

Formatages à l'aide d'un dictionnaire (notion introduite dans la suite). Le premier nombre indice l'argument de `format` à utiliser, le texte entre crochets spécifie la clé dans cet argument :

[29]:
```
print("My name is {0[name]}.".format(dict(name='Fred')))

d = dict(poids=12000, animal='éléphant')
print("L'{0[animal]} pèse {0[poids]} kg.".format(d))
```

```
My name is Fred.
L'éléphant pèse 12000 kg.
```

Remplacement avec attributs nommés (notion introduite dans la suite). On utilise alors la notation pointée (i.e. notation avec des points) des espaces de noms, appliquée aux arguments de `format` :

[30]:
```
import math
import sys

print("math.pi = {.pi}, epsilon = {.float_info.epsilon}".format(math, sys)) #␣
 ↪affiche math.pi = 3.141592653589793, epsilon = 2.220446049250313e-16
```

```
math.pi = 3.141592653589793, epsilon = 2.220446049250313e-16
```

Conversions textuelles, `str` et `repr`. La fonction `str` produit une repésentation orientée utilisateur alors que `repr` produit une représentation littérale :

[31]:
```
print("{0!s}    {0!r}".format("texte\n"))
```

```
texte
    'texte\n'
```

Formatages numériques, nombres entiers dans différentes bases, nombres flottants sous différentes notations et différentes précisions :

[32]:
```
print("int :{0:d}; hex :{0:x}; oct :{0:o}; bin :{0:b}".format(42))
```

```
int :42; hex :2a; oct :52; bin :101010
```

[33]:
```
print("int :{0:d}; hex :{0:#x}; oct :{0:#o}; bin :{0:#b}".format(42))
```

```
int :42; hex :0x2a; oct :0o52; bin :0b101010
```

[34]:
```
n = 100
pi = 3.141592653589793
k = -54
print("{:.4e}".format(pi))         # 3.1416e+00
print("{:g}".format(pi))           # 3.14159
print("{:.2%}".format(n/(47*pi))) # 67.73% - Ce format évite de mutliplier par␣
 ↪100.
```

```
3.1416e+00
3.14159
67.73%
```

[35]:
```
msg = "Résultat sur {:d} échantillons : {:.2f}".format(n, pi)
print(msg) # Résultat sur 100 échantillons : 3.14
```

```
Résultat sur 100 échantillons : 3.14
```

[36]:
```
msg = "{0.real} et {0.imag} sont les composantes du complexe {0}.".format(3-5j)
print(msg) # 3.0 et -5.0 sont les composantes du complexe (3-5j).
```

```
3.0 et -5.0 sont les composantes du complexe (3-5j).
```

[37]:
```
print("{:+d} {:+d}".format(n, k)) # +100 -54
```

```
+100 -54
```

[38]:
```
print("{:,}".format(1234567890.123)) # 1,234,567,890.123
```

```
1,234,567,890.123
```

Formatages divers :

[39]:
```
s = "The sword of truth"
print("[{}]".format(s))      # [The sword of truth]
print("[{:25}]".format(s))   # [The sword of truth       ]
print("[{:>25}]".format(s))  # [       The sword of truth]
print("[{:^25}]".format(s))  # [   The sword of truth    ]
print("[{:-^25}]".format(s)) # [---The sword of truth----]
print("[{:.<25}]".format(s)) # [The sword of truth.......]
```

```
[The sword of truth]
[The sword of truth       ]
[       The sword of truth]
[   The sword of truth    ]
[---The sword of truth----]
[The sword of truth...]
```

[40]:
```
lng = 12
print("[{}]".format(s[:lng])) # [The sword of]
m=123456789
print("{:0=12}".format(m))    # 000123456789
```

```
[The sword of]
000123456789
```

1.9 Les types binaires

Python propose deux types de données binaires : `bytes` (non modifiable) et `bytearray` (modifiable).

[41]:
```
print("type('Animal') :", type('Animal'))
print("type(b'Animal') :", type(b'Animal'))
print("type(bytearray(b'Animal')) :", type(bytearray(b'Animal')))
```

```
type('Animal') : <class 'str'>
type(b'Animal') : <class 'bytes'>
type(bytearray(b'Animal')) : <class 'bytearray'>
```

Une donnée binaire contient une suite, éventuellement vide, d'octets, c'est-à-dire une suite d'entiers non signés sur 8 bits (compris chacun dans l'intervalle [0 ... 255]). Ces types "à la C" sont bien adaptés pour stocker de grandes quantités de données ou encore des données ayant une structure définie précisément au niveau des octets ou des bits. 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 aussi des méthodes communes au type `list` que nous verrons par la suite.

1.10 Les entrées-sorties

L'utilisateur a besoin d'interagir avec le programme. En mode "console", on doit 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.

1.10.1 Les entrées

Il s'agit de réaliser une saisie au clavier : la fonction standard `input` interrompt le pogramme, affiche une éventuelle invite à l'écran et attend que l'utilisateur entre une donnée au clavier (affichée à l'écran) et la valide par "entrée".
La fonction `input` effectue toujours une saisie en mode texte (la valeur retournée est une chaîne dont on peut ensuite changer le type (on dit aussi "transtyper") :

[42]:
```
f = input("Entrer un flottant : ")
print('type(f) :',type(f))
g = float(f)  # transtypage
print('type(g) :',type(g))
g
```

```
Entrer un flottant : 3.2
type(f) : <class 'str'>
type(g) : <class 'float'>
```

[42]:
```
3.2
```

Une fois la donnée numérique convertie dans son type "naturel" (`float`), on peut l'utiliser pour faire des calculs.
On rappelle que Python est un langage dynamique (les variables peuvent changer de type au gré des affectations) mais néanmoins *fortement typé* (contrôle de la cohérence des types) :

[43]:
```
i = input("Entrer un entier : ")
print("(i, type(i)) :", (i, type(i)))
iplus = int(input('Entrer un entier : ')) + 1
print("(iplus, type(iplus)) :", (iplus, type(iplus)))
ibug  = input('Entrer un entier : ') + 1
```

```
Entrer un entier : 2
(i, type(i)) : ('2', <class 'str'>)
Entrer un entier : 2
(iplus, type(iplus)) : (3, <class 'int'>)
Entrer un entier : 3
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-43-9260e3512dfc> in <module>
  3 iplus = int(input('Entrer un entier : ')) + 1
  4 print("(iplus, type(iplus)) :", (iplus, type(iplus)))
----> 5 ibug  = input('Entrer un entier : ') + 1


TypeError: can only concatenate str (not "int") to str
```

On voit sur l'exemple précédent que Python n'autorise pas d'additioner une variable de type entier avec une variable de type chaîne.

1.10.2 Les sorties

En mode "calculatrice", Python lit-évalue-affiche, mais la fonction `print` reste indispensable aux affichages dans les scripts. Elle se charge d'afficher la représentation textuelle des informations qui lui sont données en paramètre, en plaçant un blanc séparateur entre deux informations, et en faisant un retour à la ligne à la fin de l'affichage (le séparateur et la fin de ligne peuvent être modifiés) :

[44]:
```
print("Hello world!")
print() # affiche une ligne blanche
a, b = 2, 5
print("Somme :", a + b, ";", a - b, "est la différence et", a * b, "le produit.")
print(a,end="@") # remplace le retour chariot et affiche un autre caractère en␣
 ↪fin de ligne
```

```
Hello world!

Somme : 7 ; -3 est la différence et 10 le produit.
2@
```

La fonction `print` produit des affichages de chaînes et de variables, tant sur les consoles de sortie que dans des fichiers. Très fréquemment, nous aurons besoin d'affichages formatés.