

Types de base

entier, flottant, booléen, chaîne

```
int 783 0 -192
float 9.23 0.0 -1.7e-6
bool True False
str "Un\nDeux" 'L\''âme'
```

↑ retour à la ligne
↑ multiligne
↑ non modifiable, séquence ordonnée de caractères
↑ échappé
↑ tabulation

Types Conteneurs

- séquences ordonnées, accès index rapide, valeurs répétables

```
list [1,5,9] ["x",11,8.9] ["mot"] []
tuple (1,5,9) 11,"y",7.4 ("mot",) ()
```

↑ non modifiable
↑ expression juste avec des virgules

- sans ordre *a priori*, clé unique, accès par clé rapide ; clés = types de base ou tuples

```
dict {"clé":"valeur"}
dictionnaire {1:"un",3:"trois",2:"deux",3.14:"pi"}
ensemble
set {"clé1","clé2"} {1,9,3,0} set()
```

Identificateurs

pour noms de variables, fonctions, modules, classes...

a..zA..Z suivi de **a..zA..Z_0..9**

- accents possibles mais à éviter
- mots clés du langage interdits
- distinction casse min/MAJ

```
© a toto x7 y_max BigOne
© 8y and
```

Affectation de variables

```
x = 1.2+8+sin(0)
y,z,r = 9.2,-7.6,"bad"
```

↑ valeur ou expression de calcul
↑ nom de variable (identificateur)
↑ noms de variables
↑ conteneur de plusieurs valeurs (ici un tuple)
↑ incrémentation
↑ décrémentation
↑ valeur constante « non défini »

Conversions

type(expression)

```
int("15") on peut spécifier la base du nombre entier en 2nd paramètre
int(15.56) troncature de la partie décimale (round(15.56) pour entier arrondi)
float("-11.24e8")
str(78.3) et pour avoir la représentation littérale → repr("Texte")
bool → utiliser des comparateurs (avec ==, !=, <, >, ...), résultat logique booléen
```

voir au verso le formatage de chaînes, qui permet un contrôle fin

```
list("abc") → ['a','b','c']
dict([(3,"trois"),(1,"un")]) → {1:'un',3:'trois'}
set(["un","deux"]) → {'un','deux'}
```

↑ utilise chaque élément de la séquence en paramètre
↑ utilise chaque élément de la séquence en paramètre

```
":".join(['toto','12','pswd']) → 'toto:12:pswd'
chaîne de jointure séquence de chaînes
"des mots espacés".split() → ['des','mots','espacés']
"1,4,8,2".split(",") → ['1','4','8','2']
chaîne de séparation
```

Indexation des séquences

pour les listes, tuples, chaînes de caractères,...

index négatif	-6	-5	-	-3	-2	-1
index positif	0	1	2	3	4	5

```
lst=[11, 67, "abc", 3.14, 42, 1968]
```

tranche positive 0 1 2 3 4 5 6
tranche négative -6 -5 -4 -3 -2 -1

```
lst[: -1] → [11, 67, "abc", 3.14, 42]
lst[1: -1] → [67, "abc", 3.14, 42]
lst[: :2] → [11, "abc", 42]
lst[: :] → [11, 67, "abc", 3.14, 42, 1968]
```

indication de tranche manquante → à partir du début / jusqu'à la fin

```
len(lst) → 6
```

accès individuel aux éléments par [index]

```
lst[1] → 67
lst[0] → 11 le premier
lst[-2] → 42
lst[-1] → 1968 le dernier
```

accès à des sous-séquences par [tranche début : tranche fin : pas]

```
lst[1:3] → [67, "abc"]
lst[-3: -1] → [3.14, 42]
lst[:3] → [11, 67, "abc"]
lst[4:] → [42, 1968]
```

Logique booléenne

Comparateurs: < > <= >= == !=
≤ ≥ = ≠

```
a and b et logique
a or b les deux en même temps ou logique
not a l'un ou l'autre ou les deux non logique
True valeur constante vrai
False valeur constante faux
```

Blocs d'instructions

```
instruction parente :
├── bloc d'instructions 1...
│   │
│   └── instruction parente :
│       ├── bloc d'instructions 2...
│       │
│       └── instruction suivante après bloc 1
```

Instruction conditionnelle

bloc d'instructions exécuté uniquement si une condition est vraie

```
if expression logique :
    └── bloc d'instructions
```

combinable avec des sinon si, sinon si... et un seul sinon final, exemple :

```
if x==42:
    # bloc si expression logique x==42 vraie
    print("vérité vraie")
elif x>0:
    # bloc sinon si expression logique x>0 vraie
    print("positivons")
elif bTermine:
    # bloc sinon si variable booléenne bTermine vraie
    print("ah, c'est fini")
else:
    # bloc sinon des autres cas restants
    print("ça veut pas")
```

Maths

Opérateurs: + - * / // % **
× ÷ ↑ ↑ a^b
÷ entière reste ÷

```
(1+5.3)*2 → 12.6
abs(-3.2) → 3.2
round(3.57,1) → 3.6
```

angles en radians

```
from math import sin,pi...
sin(pi/4) → 0.707...
cos(2*pi/3) → -0.4999...
acos(0.5) → 1.0471...
sqrt(81) → 9.0
log(e**2) → 2.0 etc. (cf doc)
```

bloc d'instructions exécuté tant que la condition est vraie **Instruction boucle conditionnelle**

while expression logique :
→ bloc d'instructions

```
s = 0
i = 1
```

initialisations avant la boucle

condition avec au moins une valeur variable (ici **i**)

```
while i <= 100:
    # bloc exécuté tant que i ≤ 100
    s = s + i**2
    i = i + 1
```

$s = \sum_{i=1}^{100} i^2$

faire varier la variable de condition !

print("somme:", s) } résultat de calcul après la boucle

attention aux boucles sans fin !

bloc d'instructions exécuté pour chaque élément d'une séquence de valeurs **Instruction boucle itérative**

for variable in séquence :
→ bloc d'instructions

Parcours des valeurs de la séquence

```
s = "Du texte"
cpt = 0
```

initialisations avant la boucle

variable de boucle, valeur gérée par l'instruction **for**

```
for c in s:
    if c == "e":
        cpt = cpt + 1
```

Comptage du nombre de **e** dans la chaîne.

print("trouvé", cpt, "e")

boucle sur dict/set = boucle sur séquence des clés

utilisation des tranches pour parcourir un sous-ensemble de la séquence

Parcours des index de la séquence

- changement de l'élément à la position
- accès aux éléments autour de la position (avant/après)

```
lst = [11, 18, 9, 12, 23, 4, 17]
perdu = []
for idx in range(len(lst)):
    val = lst[idx]
    if val > 15:
        perdu.append(val)
        lst[idx] = 15
```

Bornage des valeurs supérieures à 15, mémorisation des valeurs perdues.

print("modif:", lst, "-modif:", perdu)

Affichage / Saisie

print("v=", 3, "cm :", x, " ", y+4)

éléments à afficher : valeurs littérales, variables, expressions

Options de **print**:

- **sep=" "** (séparateur d'éléments, défaut espace)
- **end="\n"** (fin d'affichage, défaut fin de ligne)
- **file=f** (print vers fichier, défaut sortie standard)

s = input("Directives: ")

input retourne toujours une chaîne, la convertir vers le type désiré (cf encadré Conversions au recto).

len(seq) → nb d'éléments

min(seq) max(seq) sum(seq)

sorted(seq) → copie triée **reversed(seq)** → copie inversée

enumerate(seq) → séquence (index, valeur) pour boucle **for**

spécifique listes **lst.append(item) lst.extend(seq)**

lst.index(val) lst.count(val) lst.pop(idx)

lst.sort() lst.remove(val) lst.insert(idx, val)

Opérations sur séquences

très utilisé pour les boucles itératives **for**

Génération de séquences d'entiers

range([début,] fin [,pas])

par défaut 0 non compris

```
range(5) → 0 1 2 3 4
range(3, 8) → 3 4 5 6 7
range(2, 12, 3) → 2 5 8 11
```

range retourne un « générateur », faire une conversion en liste pour voir les valeurs, par exemple:

```
print(list(range(4)))
```

stockage de données sur disque, et lecture **Fichiers**

f = open("fic.txt", "w", encoding="utf8")

variable nom du fichier mode d'ouverture encodage des

fichier pour sur le disque

les opérations (+chemin...)

cf fonctions des modules **os** et **os.path**

□ 'r' lecture (read)

□ 'w' écriture (write)

□ 'a' ajout (append)...

caractères pour les fichiers textes: utf8 ascii latin1 ...

en écriture **f.write("coucou")**

chaîne vide si fin de fichier

en lecture **s = f.read(4)**

si nb de caractères pas précisé, lit tout le fichier

lecture ligne suivante **s = f.readline()**

f.close() ne pas oublier de refermer le fichier après son utilisation !

très courant : boucle itérative de lecture des lignes d'un fichier texte :

```
for ligne in f:
    # bloc traitement de la ligne
```

nom de la fonction (identificateur) **Définition de fonction**

paramètres nommés

```
def nomfct(p_x, p_y, p_z):
    """documentation"""
    # bloc instructions, calcul de res, etc.
    return res
```

→ valeur résultat de l'appel.

si pas de résultat calculé à retourner : **return None**

les paramètres et toutes les variables de ce bloc n'existent que dans le bloc et pendant l'appel à la fonction (« boîte noire »)

r = nomfct(3, i+2, 2*i) **Appel de fonction**

un argument par paramètre

recupération du résultat retourné (si nécessaire)

```
"{:e}".format(123.728212)
→ '1.237282e+02'
"{:f}".format(123.728212)
→ '123.728212'
"{:g}".format(123.728212)
→ '123.728'
```

directives de formatage

"modele{} {} {}".format(x, y, r) → **str**

"{sélection:formatage!conversion}"

valeurs à formater **Formatage de chaînes**

Paramètre de **sélection** (défaut ordre d'apparition):

2 → argument d'index 2 (le 3^e)

y → argument nommé y

"...".format(x=3, y=2, z=12)

0.nom → attribut nom de l'argument d'index 0

0[nom] → valeur pour la clé nom de l'argument d'index 0

0[2] → valeur pour l'index 2 de l'argument d'index 0

Paramètres de formatage:

- **remplissage**: 1 caractère (suivi par l'alignement!)
 - **alignement**: < gauche, > droite, ^ centré, = sur le signe
 - **signe**: + pour >0 et <0, - seulement pour <0, *espace* pour >0
 - **#**: représentation alternative
 - **largeurmini**: nombre, 0 au début pour remplissage avec des 0
 - **precision**: nombre de décimales pour un flottant, largeur maxi
 - **type**: entiers: **b** binaire, **c** caractère, **d** décimal (défaut), **o** octal, **x** ou **X** hexadécimal... flottant: **e** ou **E** exponentielle, **f** ou **F** point fixe, **g** ou **G** approprié (défaut), % pourcentage
- Paramètre de **conversion**:
- s** → chaîne d'affichage via **str()**
- r** → chaîne de représentation via **repr()**