

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\l'âme'
```

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

↑ échappé
↑ tabulation

Types Conteneurs

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

```
list [1,5,9] ["x",11,8.9] ["mot"] []
tuple (1,5,9) 11,"y",7.4 ("mot",) ()
dict {"clé":"valeur"} {}
set {"clé1","clé2"} {1,9,3,0} set()
```

↑ non modifiable
↑ expression juste avec des virgules
↑ en tant que séquence ordonnée de caractères
↑ dictionnaire couples clé/valeur
↑ ensemble

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)

on peut spécifier la base du nombre entier en 2nd paramètre

troncature de la partie décimale (round(15.56) pour entier arrondi)

et pour avoir la représentation littérale → repr("Texte")
voir au verso le formatage de chaînes, qui permet un contrôle fin

→ utiliser des comparateurs (avec ==, !=, <, >, ...), résultat logique booléen

utilise chaque élément de la séquence en paramètre → ['a', 'b', 'c']

utilise chaque élément de la séquence en paramètre → {1:'un', 3:'trois'}

→ 'toto:12:pswd'

chaîne de jointure séquence de chaînes

→ ['des', 'mots', 'espacés']

→ ['1', '4', '8', '2']
chaîne de séparation

Indexation des séquences

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

index négatif	-6	-5	-4	-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	

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]

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

Sur les séquences modifiables, utilisable pour suppression del lst[3:5] et modification par affectation lst[1:4]=['hop', 9]

Logique booléenne

Comparateurs: < > <= >= == !=

≤ ≥ = ≠

a and b et logique
les deux en même temps

a or b ou logique
l'un ou l'autre ou les deux

not a 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

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)
```

Opérateurs: + - * / // % **

× ÷ ↑ ↑ a^b

÷ entière reste ÷

(1+5.3)*2 → 12.6

abs(-3.2) → 3.2

round(3.57, 1) → 3.6

bloc d'instructions exécuté tant que la condition est vraie

while expression logique :

s = 0
i = 1 } initialisations avant la boucle

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

while i <= 100:

bloc exécuté tant que $i \leq 100$

s = s + i2**
i = i + 1 } faire varier la variable de condition !

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

attention aux boucles sans fin !

Contrôle de boucle

break sortie immédiate

continue itération suivante

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

bloc d'instructions exécuté pour chaque élément d'une séquence de valeurs ou d'un itérateur

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

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

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

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(c) → nb d'éléments

min(c) **max(c)** **sum(c)**

sorted(c) → copie triée

for idx, val in enumerate(c):

→ bloc d'instructions

val in c → booléen, opérateur **in** de test de présence (**not in** d'absence)

Spécifique aux **conteneurs de séquences** (listes, tuples, chaînes) :

reversed(c) → itérateur inversé **c*5** → duplication **c+c2** → concaténation

c.index(val) → position

c.count(val) → nb d'occurrences

Opérations sur conteneurs

Note: Pour dictionnaires et ensembles, ces opérations travaillent sur les clés.

Boucle directe sur index et valeur en même temps

modification de la liste originale

lst.append(item)

lst.extend(seq)

lst.insert(idx, val)

lst.remove(val)

lst.pop(idx)

lst.sort() **lst.reverse()**

ajout d'un élément à la fin

ajout d'une séquence d'éléments à la fin

insertion d'un élément à une position

suppression d'un élément à partir de sa valeur

suppression de l'élément à une position et retour de la valeur

tri / inversion de la liste sur place

Opérations sur listes

Opérations sur dictionnaires

d[clé]=valeur **d.clear()**

d[clé]→valeur **del d[clé]**

d.update(d2) } mise à jour/ajout des couples

d.keys() } vues sur les clés,

d.values() } valeurs, couples

d.items() } valeurs, couples

d.pop(clé)

Opérations sur ensembles

Opérateurs:

| → union (caractère barre verticale)

& → intersection

- ^ → différence/diff symétrique

< <= > >= → relations d'inclusion

s.update(s2)

s.add(clé) **s.remove(clé)**

s.discard(clé)

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

□ 'r' lecture (read)

caractères pour les

les opérations

(+chemin...)

□ 'w' écriture (write)

fichiers textes:

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

□ 'a' ajout (append)...

utf8 ascii latin1 ...

en écriture

f.write("coucou")

chaîne vide si fin de fichier

en lecture

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

lecture ligne suivante

si nb de caractères pas précisé, lit tout le fichier

f.read(4)

s = f.readline()

s = f.readline()

f.close() } Fermeture automatique Pythonesque : **with f as open(...)** :

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

for ligne in f:

→ bloc de traitement de la ligne