

Types de Base

int 783 0 -192 0b010 0o642 0xF3
nul binaire octal hexa

float 9.23 0.0 -1.7e-6
×10⁻⁶

bool True False

str "Un\nDeux"
retour à la ligne échappé
 'L\âme'
échappé

bytes b"toto\xfe\775"
hexadécimal octal

Chaîne multiligne :
 ""X\tY\tZ
 1\t2\t3""
tabulation échappée

immuables

Types Conteneurs

▪ **séquences ordonnées**, accès par index rapide, valeurs répétées

list [1, 5, 9] ["x", 11, 8.9] ["mot"]
tuple (1, 5, 9) 11, "y", 7.4 ("mot",)
Valeurs non modifiables (immuables) expression juste avec des virgules → tuple

▪ **conteneurs clés**, sans ordre a priori, accès par clé rapide, chaque clé unique

dictionnaire dict {"clé": "valeur"} dict (a=3, b=4, k="v")
(couples clé/valeur) {1: "un", 3: "trois", 2: "deux", 3.14: "π"}

ensemble set {"clé1", "clé2"} {1, 9, 3, 0} **set** (
clés=valeurs hachables (types base, immuables...) frozenset ensemble immuable vides

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 for**

Conversions

int ("15") → 15 **type** (expression)

int ("3f", 16) → 63 *spécification de la base du nombre entier en 2nd paramètre*

int (15.56) → 15 *troncature de la partie décimale*

float ("-11.24e8") → -112400000.0

round (15.56, 1) → 15.6 *arrondi à 1 décimale (0 décimale → nb entier)*

bool (x) **False** pour x nul, x conteneur vide, x None ou False ; **True** pour autres x

str (x) → "..." chaîne de représentation de x pour l'affichage (cf. *formatage au verso*)

chr (64) → '@' **ord** ('@') → 64 *code ↔ caractère*

repr (x) → "..." chaîne de représentation littérale de x

bytes ([72, 9, 64]) → b'H\t@'

list ("abc") → ['a', 'b', 'c']

dict ([(3, "trois"), (1, "un")]) → {1: 'un', 3: 'trois'}

set ("un", "deux") → {'un', 'deux'}

str de jointure et séquence de **str** → **str** assemblée
 ':'.join(['toto', '12', 'pswd']) → 'toto:12:pswd'

str découpée sur les blancs → **list** de **str**
 "des mots espacés".split() → ['des', 'mots', 'espacés']

str découpée sur **str** séparateur → **list** de **str**
 "1,4,8,2".split(",") → ['1', '4', '8', '2']

séquence d'un type → **list** d'un autre type (par liste en compréhension)
 [int(x) for x in ('1', '29', '-3')] → [1, 29, -3]

Affectation de Variables

1) évaluation de la valeur de l'expression de droite
 2) affectation dans l'ordre avec les noms de gauche
 ⊕ affectation ⇔ **association** d'un nom à une valeur

x=1.2+8+sin(y)

a=b=c=0 affectation à la même valeur

y, z, r=9.2, -7.6, 0 affectations multiples

a, b=b, a échange de valeurs

a, *b=seq } dépaquetage de séquence en
***a, b=seq** } élément et liste

x+=3 *incrémement ⇔ x=x+3*

x-=2 *décrémement ⇔ x=x-2*

x=None valeur constante « non défini »

del x suppression du nom x

Indexation des Conteneurs Séquences

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

index négatif	-5	-4	-3	-2	-1	
index positif		0	1	2	3	4
		10	20	30	40	50
tranche positive	0	1	2	3	4	5
tranche négative	-5	-4	-3	-2	-1	

len (lst) → 5 *Nombre d'éléments*

Accès individuel aux éléments par **lst** [*index*]

lst [0] → 10 ⇒ le premier **lst** [1] → 20

lst [-1] → 50 ⇒ le dernier **lst** [-2] → 40

Sur les séquences modifiables (**list**), suppression avec **del lst** [3] et modification par affectation **lst** [4]=25

Accès à des sous-séquences par **lst** [*tranche début:tranche fin:pas*]

lst [: -1] → [10, 20, 30, 40] **lst** [: : -1] → [50, 40, 30, 20, 10] **lst** [1: 3] → [20, 30] **lst** [: 3] → [10, 20, 30]

lst [1: -1] → [20, 30, 40] **lst** [: : -2] → [50, 30, 10] **lst** [-3: -1] → [30, 40] **lst** [3:] → [40, 50]

lst [: : 2] → [10, 30, 50] **lst** [:] → [10, 20, 30, 40, 50] *copie superficielle de la séquence*

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

Sur les séquences modifiables (**list**), suppression avec **del lst** [3: 5] et modification par affectation **lst** [1: 4]=[15, 25]

Logique Booléenne

Comparateurs: < > <= >= == != (résultats booléens) ≤ ≥ = ≠

a and b et logique *les deux en même temps*

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

⊕ piège : **and** et **or** retournent la valeur de **a** ou de **b** (selon l'évaluation au plus court). ⇒ s'assurer que **a** et **b** sont booléens.

not a non logique

True } constantes Vrai Faux
False }

Blocs d'Instructions

instruction parente:
 bloc d'instructions 1...
 ...
 instruction parente:
 bloc d'instructions 2...
 ...
 instruction suivante après bloc 1

⊕ régler l'éditeur pour insérer 4 espaces à la place d'une tabulation d'indentation.

angles en radians

Imports de Modules/Noms

module **truc** ⇔ fichier **truc.py**

from monmod import nom1, nom2 as fct
 → accès direct aux noms, renommage avec **as**

import monmod → accès via **monmod.nom1** ...

⊕ modules et packages cherchés dans le **python path** (cf **sys.path**)

un bloc d'instructions exécuté, **Instruction Conditionnelle** uniquement si sa condition est vraie

if condition logique :
 → bloc d'instructions

Combinable avec des **sinon si**, **sinon si...** et un seul **sinon** final. Seul le bloc de la première condition trouvée vraie est exécuté.

⊕ avec une variable **x**:

if bool(x) == True : ⇔ **if x :**

if bool(x) == False : ⇔ **if not x :**

Opérateurs

⊕ nombres flottants... valeurs approchées !

Opérateurs: + - * / // % **

Priorités (...)

@ → × matricielle *python3.5+ numpy*

(1+5.3) * 2 → 12.6

abs(-3.2) → 3.2

round(3.57, 1) → 3.6

pow(4, 3) → 64.0

⊕ priorités usuelles

Maths

from math import sin, pi...

sin (pi/4) → 0.707...

cos (2*pi/3) → -0.4999...

sqrt (81) → 9.0 √

log (e**2) → 2.0

ceil (12.5) → 13

floor (12.5) → 12

modules **math, statistics, random, decimal, fractions, numpy, etc.** (cf. doc)

Exceptions sur Erreurs

Signalisation d'une erreur:
raise Exception(...)

Traitement des erreurs:
try :
 → bloc traitement normal
except Exception as e :
 → bloc traitement erreur

⊕ bloc **finally** pour traitements finaux dans tous les cas.

Instruction Boucle Conditionnelle

bloc d'instructions exécuté tant que la condition est vraie

while condition logique:
→ bloc d'instructions

```

s = 0
i = 1
while i <= 100:
    s = s + i**2
    i = i + 1
print("somme:", s)

```

initialisations avant la boucle
condition avec au moins une valeur variable (ici i)
faire varier la variable de condition!

Contrôle de Boucle
break sortie immédiate
continue itération suivante
 bloc **else** en sortie normale de boucle.

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

Instruction Boucle Itérative

bloc d'instructions exécuté pour chaque élément d'un conteneur ou d'un itérateur

for var in séquence:
→ bloc d'instructions

```

s = "Du texte"
cpt = 0
for c in s:
    if c == "e":
        cpt = cpt + 1
print("trouvé", cpt, "e")

```

Parcours des valeurs d'un conteneur
initialisations avant la boucle
variable de boucle, affectation gérée par l'instruction for
Algo: comptage du nombre de e dans la chaîne.

Affichage

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=sys.stdout** print vers fichier, défaut sortie standard

Saisie

s = input ("Directives: ")

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

boucle sur dict/set ⇒ boucle sur séquence des clés
utilisation des tranches pour parcourir un sous-ensemble d'une séquence

Parcours des **index** d'un conteneur 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)

```

Algo: bornage des valeurs supérieures à 15, mémorisation des valeurs perdues.

Parcours simultané **index** et **valeurs** de la séquence:

```

for idx, val in enumerate(lst):

```

Opérations Génériques sur Conteneurs

len(c) → nb d'éléments
min(c) **max(c)** **sum(c)**
sorted(c) → liste copie triée
val in c → booléen, opérateur in de test de présence (**not in** d'absence)
enumerate(c) → itérateur sur (index, valeur)
zip(c1, c2...) → itérateur sur tuples contenant les éléments de même index des c_i
all(c) → True si tout élément de c évalué vrai, sinon False
any(c) → True si au moins un élément de c évalué vrai, sinon False

Spécifique aux **conteneurs de séquences ordonnées** (listes, tuples, chaînes, bytes...)

- reversed(c)** → itérateur inversé
- c*5** → duplication
- c+c2** → concaténation
- c.index(val)** → position
- c.count(val)** → nb d'occurrences

import copy
copy.copy(c) → copie superficielle du conteneur
copy.deepcopy(c) → copie en profondeur du conteneur

Séquences d'Entiers

range([début,] fin [,pas])
 début défaut 0, fin non compris dans la séquence, pas signé et défaut 1

range(5) → 0 1 2 3 4
range(2, 12, 3) → 2 5 8 11
range(3, 8) → 3 4 5 6 7
range(20, 5, -5) → 20 15 10
range(len(séq)) → séquence des index des valeurs dans séq
 range fournit un séquence immuable d'entiers construits au besoin

Opérations sur Listes

modification de la liste originale

- lst.append(val)** ajout d'un élément à la fin
- lst.extend(seq)** ajout d'une séquence d'éléments à la fin
- lst.insert(idx, val)** insertion d'un élément à une position
- lst.remove(val)** suppression du premier élément de valeur val
- lst.pop([idx])** → valeur supp. & retourne l'item d'index idx (défaut le dernier)
- lst.sort()** **lst.reverse()** tri / inversion de la liste sur place

Définition de Fonction

nom de la fonction (identificateur)
paramètres nommés

```

def fct(x, y, z):
    """documentation"""
    # bloc instructions, calcul de res, etc.
    return res

```

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 (penser "boîte noire")

Avancé: **def fct(x, y, z, *args, a=3, b=5, **kwargs):**
 *args nb variables d'arguments positionnels (→ tuple), valeurs par défaut, **kwargs nb variable d'arguments nommés (→ dict)

Appel de fonction

r = fct(3, i+2, 2*i)
 stockage/utilisation une valeur d'argument de la valeur de retour par paramètre

c'est l'utilisation du nom de la fonction avec les parenthèses qui fait l'appel

Avancé:
 *séquence
 **dict

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 itérables sur les clés / valeurs / couples
d.values()
d.items()
d.pop(clé, défaut) → valeur
d.popitem() → (clé, valeur)
d.get(clé, défaut) → valeur
d.setdefault(clé, défaut) → valeur

Opérations sur Ensembles

Opérateurs:
 | → union (caractère barre verticale)
 & → intersection
 - ^ → différence/diff. symétrique
 < <= > >= → relations d'inclusion

Les opérateurs existent aussi sous forme de méthodes.

```

s.update(s2) s.copy()
s.add(clé) s.remove(clé)
s.discard(clé) s.clear()
s.pop()

```

Fichiers

stockage de données sur disque, et relecture

```

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

```

variable nom du fichier mode d'ouverture encodage des caractères pour les fichiers textes:
 fichier pour sur le disque (+chemin...) 'r' lecture (read) utf8 ascii
 les opérations 'w' écriture (write) latin1 ...
 cf modules **os**, **os.path** et **pathlib** 'a' ajout (append) 'x' 'b' 't'

en écriture lit chaîne vide si fin de fichier en lecture

```

f.write("coucou")
f.writelines(list de lignes)
f.read() → caractères suivants si n non spécifié, lit jusqu'à la fin!
f.readlines([n]) → list lignes suivantes
f.readline() → ligne suivante

```

par défaut mode texte t (lit/écrit str), mode binaire b possible (lit/écrit bytes). Convertir de/vers le type désiré!

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

f.flush() écriture du cache **f.truncate([taille])** retaillage

lecture/écriture progressent séquentiellement dans le fichier, modifiable avec:
f.tell() → position **f.seek(position, origine)**

Très courant: ouverture en bloc gardé (fermeture automatique) et boucle de lecture des lignes d'un fichier texte:

```

with open(...) as f:
    for ligne in f:
        # traitement de ligne

```

Opérations sur Chaînes

```

s.startswith(prefix[, début[, fin]])
s.endswith(suffix[, début[, fin]])
s.strip([caractères])
s.count(sub[, début[, fin]])
s.partition(sep) → (avant, sep, après)
s.index(sub[, début[, fin]])
s.find(sub[, début[, fin]])
s.is...() tests sur les catégories de caractères (ex. s.isalpha())
s.upper() s.lower() s.title() s.swapcase()
s.casefold() s.capitalize() s.center([larg, rempl])
s.ljust([larg, rempl]) s.rjust([larg, rempl]) s.zfill([larg])
s.encode(codage) s.split([sep]) s.join(séq)

```

Formatage

directives de formatage valeurs à formater

```

"modele{ } { }".format(x, y, r) → str
"{sélection: formatage! conversion}"

```

□ **Sélection:**

```

2
nom
0.nom
4[clé]
0[2]

```

Exemples:

```

"{: +2.3f}".format(45.72793) → '+45.728'
"{1: >10s}".format(8, "toto") → 'toto'
"{x!r}".format(x="L'ame") → 'L'ame'

```

□ **Formatage:**
 car-rempl. alignement signe larg.mini.precision-larg.max type

```

<> ^ = + - espace 0 au début pour remplissage avec des 0
entiers: b binaire, c caractère, d décimal (défaut), o octal, x ou X hexa...
flottant: e ou E exponentielle, f ou F point fixe, g ou G approprié (défaut),
chaîne: s ... % pourcentage

```

□ **Conversion:** s (texte lisible) ou r (représentation littérale)

bonne habitude : ne pas modifier la variable de boucle