Mémento v2.1.2

Mémento Python 3

Pour Python ≥3.10 *dernière version sur : https://py3memento.lisn.fr/*

Types de Base

entier, flottant, booléen, chaîne, octets

```
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"   Chaîne multiligne :
    retour à la ligne échappé   """X\tY\tZ
    'L\'âme'                    1\t2\t3"""
     ' échappé                  tabulation échappée
bytes b"toto\xfe\775"
      hexadécimal  octal
```

immutables

Types Conteneurs

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

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

Valeurs non modifiables (immutables) — *expression juste avec des virgules →* `tuple`

`str` `bytes` *(séquences ordonnées de caractères / d'octets)* — `""` `b""`

▪ **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, immutables…) — `frozenset` *ensemble immutable* — *vides*

→modules `collections`, `array`, `weakref`…

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`

☹ ~~`3y and for`~~

Variables & Affectation

`=` affectation ⇔ **association** d'un *nom* à une *valeur*

1) évaluation de la valeur de l'expression de droite
2) affectation dans l'ordre avec les noms de gauche

```
x=1.2+8+sin(y)
a=b=c=0          affectation à la même valeur
y,z,r=9,7,0      affectations multiples
a,b=b,a          échange de valeurs
a,*b=seq         dépaquetage de séquence
*a,b=seq         en élément et liste
x+=3   incrémentation ⇔ x=x+3      et *= /= %= …
x-=2   décrémentation ⇔ x=x-2
x=None valeur constante « non défini »
del x  suppression du nom x
```

:= Expression d'affectation, association d'un *nom* à une *valeur* utilisée dans une expression.

```
while (v:=suiv()) is not None:…
```

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") → -1124000000.0
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]
```

Indexation Conteneurs Séquences

listes, tuples, chaînes de caractères, bytes,…

index négatif	-5	-4	-3	-2	-1
index positif	0	1	2	3	4
`lst=`	[10	20	30	40	50]
tranche positive	0	1	2	3	4	5
tranche négative	-5	-4	-3	-2	-1	

Accès à un **éléments** `lst[index]`

`lst[0]`→10 ⇒ *le premier* `lst[1]`→20
`lst[-1]`→50 ⇒ *le dernier* `lst[-2]`→40

Sur les séquences modifiables (`list`) :
suppression élément `del lst[3]`
modification par affectation `lst[4]=25`

Nombre d'éléments `len(lst)`→5 *index à partir de 0*

Sous-séquences `lst[`*tranche début*`:`*tranche fin*`:`*pas*`]`

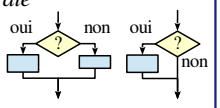
```
lst[:-1]→[10,20,30,40]      lst[::-1]→[50,40,30,20,10]
lst[1:-1]→[20,30,40]        lst[::-2]→[50,30,10]
lst[::2]→[10,30,50]         lst[:]→[10,20,30,40,50] →copie superficielle de la séquence
lst[1:3]→[20,30]
lst[:3]→[10,20,30]
lst[-3:-1]→[30,40]
lst[3:]→[40,50]
```

Indication de tranche manquante → à partir du début / jusqu'à la fin.
Sur les séquences modifiables (`list`) : suppression sous-séquence `del lst[3:5]`
modification par affectation `lst[1:4]=[15,25]`

Instruction Conditionnelle

*un bloc d'instructions exécuté 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.

```
if age<=18:
  etat="Enfant"
elif age>65:
  etat="Retraité"
else:
  etat="Actif"
```

avec une variable `x`:
`if bool(x)==True:` ⇔ `if x:`
`if bool(x)==False:` ⇔ `if not x:`

Imports 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`)

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` `False` constantes Vrai Faux

Blocs d'Instructions

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

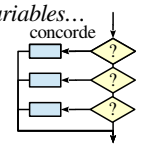
indentation !

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

Instruction Concordance

choix d'un bloc d'instructions à exécuter suivant une concordance avec un motif.
Peut dépaquetter des séquences, positionner des variables…

```
match expression:
    case motif1:
        bloc d'instructions
    case motif2:
```




```
match infos:
  case 'nono':                  → valeur
  case 'bob'|'elsa'|300:        → valeur parmi un choix
  case ['lui','luc']:           → séquence de deux valeurs
  case ['untel',name]:          → 1re valeur, récup 2e dans name
  case ['eux',*names]:          → 1re valeur, récup le reste dans names
  case 'will' if flag:          → valeur avec test supplémentaire
  case str():                   → type ou classe
  case _:                       → tout le reste (dernier cas)
```

Note : on peut utiliser `()` ou `[]` pour les motifs.

Maths

nombres flottants… valeurs approchées ! — *angles en radians*

Opérateurs: `+` `-` `*` `/` `//` `%` `**`
Priorités `(…)` × ÷ ↑ ↑ a^b
÷ entière, reste ÷

`@` → × 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

```
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
```

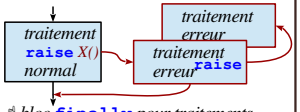
→modules `math`, `statistics`, `random`, `decimal`, `fractions`, `numpy`…

Exceptions sur Erreurs

Signalisation:
`raise` *ExcClass(…)*

Traitement:

```
try:
    bloc traitement normal
except ExcClass 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          # initialisations avant la boucle
i = 1          # condition avec au moins une valeur variable (ici i)
while i <= 100:
    s = s + i**2
    i = i + 1      # faire varier la variable de condition !
print("somme:",s)
```

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

Contrôle de Boucle

```
break       sortie immédiate
continue    itération suivante
```

bloc `else` *en sortie **normale** de boucle.*

attention aux boucles sans fin !

Affichage

```
print(f"{x}cm+{y}m={x/100+y}m")
```

*Exemple avec une chaîne de formatage **f-string**.* `print` *peut afficher plusieurs éléments (valeurs, variables, expressions…) en les séparant par des virgules.*

Paramètres optionnels 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

→module `pprint`…

Saisie

```
s = input("Directives:")
```

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

Opérations Génériques sur Conteneurs

`len(c)` → nb d'éléments
`min(c)` `max(c)` `sum(c)`
`sorted(c)` → `list` *copie* triée
`val in c` → booléen, opérateur `in` de test de présence (`not in` d'absence)
`enumerate(c)` → *itérateur* sur (index/clé, 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`

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

*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'occurences*

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

→modules `collections`, `itertools`, `functools`…

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
```

→modules `heapq`, `bisect`…

Opérations sur Dictionnaires

```
d[clé]=valeur      d.clear()
d[clé] → valeur    del d[clé]
```

Opérateurs: `|`→ fusion `|=` → mise à jour

```
d.keys()
d.values()   →vues itérables sur les clés / valeurs / couples
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 `&` → intersection
`-` `^` → différence/diff. symétrique
`<` `<=` `>` `>=` → relations d'inclusion

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

Certains opérateurs existent aussi sous forme de méthodes.

Fichiers

stockage de données sur disque, et relecture

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

variable fichier pour les opérations

nom du fichier sur le disque (+chemin…)

mode d'ouverture
- `'r'` lecture (read)
- `'w'` écriture (write)
- `'a'` ajout (append)
- …`'+'` `'x'` `'b'` `'t'`

encodage des caractères pour les *fichiers textes*: `utf8` `ascii` `latin1` …

→modules `pathlib`, `os`, `os.path`

écriture

```
f.write("coucou")
f.writelines(list de lignes)
```

lecture

lit chaîne vide si fin de fichier

```
f.read([n])          → 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 **refermer 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 avec un *context manager*) et boucle de lecture des lignes d'un fichier texte :

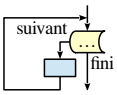
```
with open(…) as f:
    for ligne in f :
        # traitement de ligne
```

Plusieurs fichiers: `with (open() as f1, open() as f2):`

Instruction Boucle Itérative

*bloc d'instructions exécuté **pour chaque** élément d'un conteneur ou d'un itérable*

```
for var in séquence :
    bloc d'instructions
```



Parcours des **valeurs** d'un conteneur

```
s = "Du texte"   # initialisations avant la boucle
cpt = 0
    variable de boucle, affectation gérée par l'instruction for
for c in s:
    if c == "e":
        cpt = cpt + 1
print(f"trouvé {cpt} 'e'")
```

Algo: comptage du nombre de e *dans la chaîne.*

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

Parcours des **index** et **valeur** d'un conteneur séquence, permet :
- 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 i,v in enumerate(lst):
    if v > 15:
        perdu.append(v)
        lst[i] = 15
print(f"modif:{lst}-modif:{perdu}")
```

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

*Note : Parcours des **index** de la séquence avec* `range(len(lst))`

Programmation fonctionnelle, expressions d'itérables

```
map(f,seq)    → (f(x) for x in seq)
filter(f,seq) → (x for x in seq if f(x))
```

bonne habitude : ne pas modifier la variable de boucle

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(3,8)` → 3 4 5 6 7
`range(2,12,3)` → 2 5 8 11
`range(20,5,-5)` → 20 15 10
`range(len(séq))` → *séquence des index des valeurs dans séq*

range fournit un séquence immutable d'entiers construits au besoin

Définition de Fonction

nom de la fonction (identificateur)
paramètres nommés

```
def fct(x,y,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 (penser "boite noire")

***affectation variable globale `xxx` dans** la fonction → déclaration* `global xxx` *dans son bloc*

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`)*
Et : `/` → arguments avant sont positionnels, `*` → arguments après sont nommés

Appel de fonction

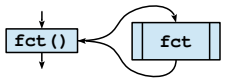
```
r = fct(3,i+2,2*i)
```

stockage/utilisation de la valeur de retour

une valeur d'argument par paramètre

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

*Avancé: *séquence **dict*



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)
s.removeprefix(pref)  s.removesuffix(suf)  s.format(…)
```

Formatage *f-string*

préfixe `f` → chaîne de formatage "f-string"

```
f"{x}+{y}={x+y:.2f}" → str
```

{expression : formatage ! conversion}

- **Expression** : variable, appel de fonction… toute expression Python.
  Valeurs considérées lors de l'évaluation de la *f-string* à l'exécution.

Exemples
```
x,t1,t2=45.72793,"toto","L'ame"
f"{x:+2.3f}" → '+45.728'
f"{t1:>10s}" → '      toto'
f"{t2!r}" → '"L\'ame"'
```

- **Formatage** :

car-rempl. *alignement* *signe* *larg.mini*`.`*précision~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… *flottants*: `e` ou `E` exponentielle, `f` ou `F` point fixe, `g` ou `G` approprié (défaut), *chaînes*: `s` … `%` pourcentage

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