Mémento Bases Python 3 (v 3.0)

V2.0.4 sur :
https://perso.limsi.fr/pointal/python:memento

Types de Base

entier, flottant, booléen, chaîne

```
int 783  0  -192
```
nul
```
float 9.23 0.0  -1.7e-6
bool True  False
str "Un\nDeux"
```
retour à la ligne échappé
```
'L\'âme'
```
' échappé

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

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

(séquences ordonnées de caractères)

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

`type(`*expression*`)`

```
int("15") → 15
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
```

= 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
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
```

Opérations Génériques sur listes et tuples

```
len(c)→ nb d'éléments
min(c)  max(c)  sum(c)
val in c → booléen, opérateur in de test de présence (not in d'absence)
reversed(c)→ itérateur inversé   c*5→ duplication   c+c2→ concaténation
c.index(val)→ position           c.count(val)→ nb d'occurences
```

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.sort()  lst.reverse()   tri / inversion de la liste sur place
```

Indexation

pour les listes, tuples, chaînes de caractères,…

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	

Nombre d'éléments `len(lst)→5`

index à partir de 0 (de 0 à 4 ici)

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` `False` constantes Vrai Faux

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
```

Saisie

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

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

Maths

nombres flottants, valeurs approchées !

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

Priorités `(…)`

```
(1+5.3)*2→12.6
abs(-3.2)→3.2
round(3.57,1)→3.6
4**3→64.0
```
priorités usuelles

Autres opérations

```
from math import sqrt
sqrt(81)→9.0      √
```

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`)

Blocs d'Instructions

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

indentation ! : → bloc d'instructions

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

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.

avec une variable **`x`**:

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

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

Instruction Boucle Conditionnelle

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

`while` *condition logique* :
→ *bloc d'instructions*

attention aux boucles sans fin !

```
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 !

Algo:

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

Instruction Boucle Itérative

*bloc d'instructions exécuté **pour chaque** élément*

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

Parcours des **valeurs**

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

*initialisations **avant** la boucle*
*variable de boucle, **affectation gérée par l'instruction `for`***

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

Parcours des **index**
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.

bonne habitude : ne pas modifier la variable de boucle

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

```
for idx,val in enumerate(lst):
```

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 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")

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*