Mémento Python 3

Types de base

entier, flottant, complexe, booléen, chaîne

```
  int   783     0     -192
float   9.23    0.0    -1.7e-6
complex 2.7+3.1j  1j
 bool   True    False
  str   "Un\nDeux"   'L\'âme'
```

- `\n` retour à la ligne
- `\'` échappé
- multiligne `"""X\tY\tZ` / `1\t2\t3"""` (`\t` tabulation)
- *non modifiable, séquence ordonnée de caractères*
- 10^{-6}

Types Conteneurs (listes, tuples, chaînes)

- 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

`str` en tant que séquence ordonnée de caractères

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"
x+=3
x-=2
x=None
```

- `x` nom de variable (identificateur)
- `1.2+8+sin(0)` valeur ou expression de calcul
- `y,z,r` noms de variables
- `9.2,-7.6,"bad"` conteneur de plusieurs valeurs (ici un tuple)
- `x+=3` incrémentation, `x-=2` décrémentation
- `x=None` valeur constante « non défini »

Conversions

`type(`*expression*`)`

```
int("15")
int(15.56)
float("-11.24e8")
str(78.3)
```

`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)

`str(78.3)` et pour avoir la représentation littérale → `repr("Texte")`

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

`bool` → utiliser des comparateurs (avec `==`, `!=`, `<`, `>`, …), résultat logique booléen

`list("abc")` → *utilise chaque élément de la séquence en paramètre* → `['a','b','c']`

`":".join(['toto','12','pswd'])` → `'toto:12:pswd'`

- `":"` chaîne de jointure
- `['toto','12','pswd']` séquence de chaînes

`"Un blanc final \n".strip()` → `"Un blanc final"`

`"des mots  espacés".split()` → `['des','mots','espacés']`

`"1,4,8,2".split(",")` → `['1','4','8','2']`

- `","` chaîne de séparation

Indexation des 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]` → `[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]`
- `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

indentation !

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.

```
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")
else:
    # bloc sinon des autres cas restants
    print("ça veut pas")
```

Maths

nombres flottants… valeurs approchées ! *angles en radians*

Opérateurs: `+` `-` `*` `/` `**` (× ÷ a^b)

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

Complexes

```
z=1+2j
z.real
z.imag
z.conjugate()
abs(z)
```

Opérations spécifiques aux entiers

`17 % 5` reste et
`17 // 5` quotient
dans la div. eucl. de 17 par 5

Instruction boucle conditionnelle

bloc d'instructions exécuté tant que la condition est vraie

```
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
```
faire varier la variable de condition !

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

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

attention aux boucles sans fin !

contrôle de boucle :

`break` sortie immédiate `continue` itération suivante

Instruction boucle itérative

bloc d'instructions exécuté pour chaque élément d'un conteneur 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
print("trouvé",cpt,"'e'")
```
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 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.

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

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

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).

Génération de séquences d'entiers

très utilisé pour les boucles itératives for

```
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)))
```

Opérations sur conteneurs (listes, tuples, chaînes)

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

*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'occurences
```

Définition de fonction

nom de la fonction (identificateur) — 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 (« *boite noire* »)

Appel de fonction

```
r = nomfct(3,i+2,2*i)
```
un argument par paramètre

récupération du résultat renvoyé (si nécessaire)

Opérations spécifiques aux listes

modification de la liste originale

```
lst.append(item)     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 d'un élément à partir de sa valeur
lst.pop(idx)         suppression de l'élément à une position et retour de la valeur
lst.sort()  lst.reverse()   tri / inversion de la liste sur place
```

Listes par compréhension

```
lst = [2*i for i in range(10)]
lst = [i for i in range(20) if i%2 == 0]
```
lst = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

Fichiers

stockage de données sur disque, et relecture

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

variable fichier pour les opérations

nom du fichier sur le disque, chemin, relatif ou absolu

mode d'ouverture
- `'r'` lecture (read)
- `'w'` écriture (write)
- `'a'` ajout (append)…

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

en **écriture**

```
f.write("coucou")
```
*fichier texte → lecture / écriture de **chaînes** uniquement, convertir de/vers le type désiré*

en **lecture**

```
s = f.read(4)
```
chaîne vide si fin de fichier — si nb de caractères pas précisé, lit tout le fichier

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

```
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 de traitement de la ligne
```

Ce mémento est fourni à titre indicatif. Il ne faut le considérer :

- *ni comme exhaustif (en cas de problème sur un exercice particulier, si une fonction ou une commande indispensable était absente de la liste, l'interrogateur pourrait aider le candidat),*
- *ni comme exclusif (une fonction ou une commande absente de cette liste n'est pas interdite : si un candidat utilise à très bon escient d'autres fonctions MAIS sait aussi répondre aux questions sur les fonctions de base, il n'y a pas de problème),*
- *ni comme un minimum à connaître absolument (l'examinateur n'attend pas du candidat qu'il connaisse parfaitement toutes ces fonctions et ces commandes).*

Les fonctions et commandes présentées doivent simplement permettre de faire les exercices proposés aux candidats.

L'examinateur n'attend pas du candidat une connaissance encyclopédique du langage Python, mais une utilisation raisonnée des principes algorithmiques et une mise en pratique des connaissances de base.

L'utilisation de l'aide en ligne est encouragée, mais ne doit pas masquer une ignorance sur ces aptitudes.

Aide

```
help(a) → aide sur a      dir(a) → liste d'attributs de a      F1
help("module.obj") → aide sur obj de module, sans avoir besoin d'importer le module
```

Mémento numérique Python 3

```
import matplotlib.pyplot as plt
```
→ charge le module pyplot sous le nom `plt`

`plt.figure('titre')` → crée une fenêtre de tracé vide

`plt.plot(LX,LY,'o-b')` → trace le graphique défini par les listes LX et LY (abscisses et ordonnées)
- couleur : `'b'` (**blue**), `'g'` (**green**), `'r'` (**red**), `'c'` (cyan), `'m'` (**magenta**), `'y'` (yellow), `'k'` (**black**)
- type de ligne : `'-'` (trait plein), `'--'` (pointillé), `'-.'` (alterné)...
- marque : `'o'` (rond), `'h'` (hexagone), `'+'` (plus), `'x'` (croix), `'*'` (étoile)...

`plt.xlim(xmin,xmax)` → fixe les bornes de l'axe x

`plt.ylim(ymin,ymax)` → fixe les bornes de l'axe y

`plt.axis('equal')` → change les limites des axes x et y pour un affichage avec des axes orthonormés (le tracé d'un cercle donne un cercle)

`plt.show()` → affichage de la fenêtre

`plt.savefig(fichier)` → sauve le tracé dans un fichier (le suffixe du nom fichier peut donner le format ; par exemple, `'image.png'`)

```
import numpy as np
```
→ charge le module numpy sous le nom `np`

Construction de tableaux (de type ndarray)

`np.zeros(n)` → crée un vecteur dont les n composantes sont nulles

`np.zeros((n,m))` → crée une matrice $n \times m$, dont les éléments sont nuls

`np.eye(n)` → crée la matrice identité d'ordre n

`np.linspace(a,b,n)` → crée un vecteur de n valeurs régulièrement espacées de a à b

`np.arange(a,b,dx)` → crée un vecteur de valeurs de a incluse à b exclue avec un pas dx

`M.shape` → tuple donnant les dimensions de M

`M.size` → le nombre d'éléments de M

`M.ndim` → le nombre de dimensions de M

`M.sum()` → somme de tous les éléments de M

`M.min()` → plus petit élément de M

`M.max()` → plus grand élément de M

argument **axis** optionnel : 0 → lignes, 1 → colonnes :

`M.sum(0)` → somme des lignes

`M.min(0)` → plus petits éléments, sur chaque colonne

`M.max(1)` → plus grands éléments, sur chaque ligne

```
import numpy.linalg as la
```

`la.det(M)` → déterminant de la matrice carrée M

`la.inv(M)` → inverse de M

`la.eigvals(M)` → valeurs propres de M

`la.matrix_rank(M)` → rang de M

`la.matrix_power(M,n)` → M^n (n entier)

`la.solve(A,B)` → renvoie X tel que $A\,X = B$

```
import scipy.integrate as spi
```

`spi.odeint(F,Y0,LT)`
→ renvoie une solution numérique du problème de Cauchy $\mathbf{Y}'(t) = \mathbf{F}(\mathbf{Y}(t),t)$, où $\mathbf{Y}$ est un vecteur d'ordre n, avec la condition initiale $\mathbf{Y}(t_0) =$ `Y0`, pour les valeurs de t dans la liste `LT` de longueur k commençant par t_0, sous forme d'une matrice $n \times k$

`spi.quad(f,a,b)[0]` → renvoie une évaluation numérique de l'intégrale : $\int_a^b f(t)\,dt$

Conversion ndarray <-> liste

`V = np.array([1,2,3])` → V : vecteur (1 2 3)

`L = V.tolist()` → L : liste [1, 2, 3]

`M = np.array([[1,2],[3,4]])` → M : matrice $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

`L = M.tolist()` → L : liste [[1, 2], [3,4]]

Extraction d'une partie de matrice

`M[i], M[i,:]` → ligne de M d'index i

`M[:,j]` → colonne de M d'index j

`M[i:i+h,j:j+l]` → sous-matrice $h \times l$

Copier un tableau avec la méthode copy :

`M2 = M1.copy()`

`M1+M2, M1*M2, M**2` → opérations « terme-à-terme »

`c*M` → multiplication de la matrice M par le scalaire c

`M+c` → matrice obtenue en ajoutant le scalaire c à chaque terme de M

`V1.dot(V2)` / `np.dot(V1,V2)` → renvoie le produit scalaire de deux vecteurs

`M.dot(V)` / `np.dot(M,V)` → renvoie le produit d'une matrice par un vecteur

`M1.dot(M2)` / `np.dot(M1,M2)` → renvoie le produit de deux matrices

`M.transpose()` / `np.transpose(M)` → renvoie une copie de M transposée (ne modifie pas M)

`M.trace()` / `np.trace(M)` → renvoie la trace de M

Fonctions mathématiques usuelles

`np.exp, np.sin, np.cos, np.sqrt` *etc.*

→ fonctions qui s'appliquent sur des réels ou des complexes, mais aussi sur des vecteurs et des matrices (s'appliquent à chaque terme), qui sont optimisées en durée de calcul.

Rappel : ce mémento est fourni à titre indicatif. Il ne faut le considérer ni comme exhaustif, ni comme exclusif, ni comme un minimum à connaître absolument (l'examinateur n'attend pas du candidat qu'il connaisse parfaitement toutes ces fonctions et ces commandes).

Mémento Python 3 – Compléments

Modules de tirages pseudo-aléatoires : `random` et `numpy.random`

`random.random()` ⟶ Valeur flottante dans l'intervalle [0,1[(loi uniforme)

`random.randint(a,b)` ⟶ Valeur entière entre `a` inclus et `b` inclus (équiprobabilité)

`random.choice(L)` ⟶ Un élément de la liste L (équiprobabilité)

`random.shuffle(L)` ⟶ `None`, mélange la liste L « **en place** »

```
import numpy as np
```

`np.random.rand(`n_0`,…,`n_{d-1}`)` ⟶ Tableau $(n_0 \times \ldots \times n_{d-1})$ de flottants dans [0,1[(loi uniforme)

`np.random.randint(a,b,shp)` ⟶ Tableau de forme `shp` d'entiers entre `a` inclus et `b` **exclu** (équiprobabilité)

`np.random.randint(n,size=d)` ⟶ Vecteur de dimension `d` d'entiers entre `0` et `n-1`.

`np.random.normal(m,s,shp)` ⟶ Tableau de forme `shp` de flottants tirés selon une loi normale de moyenne `m` et d'écart-type `s`

`np.random.uniform(a,b,shp)` ⟶ Tableau de forme `shp` de flottants tirés selon une loi uniforme sur l'intervalle [`a` , `b` [

Rappel : ce mémento est fourni à titre indicatif. Il ne faut le considérer ni comme exhaustif, ni comme exclusif, ni comme un minimum à connaître absolument (l'examinateur n'attend pas du candidat qu'il connaisse parfaitement toutes ces fonctions et ces commandes).

Autres méthodes utiles de la classe `str` (chaîne de caractères)

```
"nomfic.txt".replace(".txt",".png")  ⟶ "nomfic.png"
"nomfic.txt".endswith(".png")        ⟶ False
"data08.txt".startswith("data")      ⟶ True
"{} -> {}".format("A","B")           ⟶ "A -> B"
"{1:} -> {0:}".format("A","B")       ⟶ "B -> A"
"image{:04d}.png".format(12)         ⟶ "image0012.png"
```

Format entier avec 4 chiffres complétés par des zéros

```
"L : {:.3f} m".format(0.01)          ⟶ "L : 0.010 m"
```

Format flottant avec 3 chiffres après la « virgule »

```
"m : {:.2e} kg".format(0.012)        ⟶ "m : 1.20e-02 kg"
```

Format scientifique avec 2 chiffres après la « virgule »

```
r"rep\trajet\nom.py"  ⟶  "rep\\trajet\\nom.py"
```

Le préfixe r signifie *"raw string"* (tous les caractères sont considérés comme de *vrais* caractères)

Attributs et méthodes complémentaires sur les tableaux `numpy.ndarray`

Soit un exemple de tableau : `T = numpy.array([[1,-2,-7],[0,3,-4]])`

`T.dtype` ⟶ `dtype('int32')` Type de données : entier codé sur 4 octets

`T.astype(float)` ⟶ Conversion en tableau de *flottants*

```
array([[ 1., -2., -7.],
       [ 0.,  3., -4.]])
```

`T.astype(complex)` ⟶ Conversion en tableau de *complexes*

```
array([[ 1.+0.j, -2.+0.j, -7.+0.j],
       [ 0.+0.j,  3.+0.j, -4.+0.j]])
```

`T >= 1` ⟶ Comparaison des valeurs à une valeur donnée

```
array([[ True, False, False],
       [False,  True, False]], dtype=bool)
```

`(T > -3)*(T < 3)` ⟶ Plusieurs comparaisons

```
array([[ True,  True, False],
       [ True, False, False]], dtype=bool)
```

`T.flatten()` ⟶ Mise « à plat » du tableau (un seul indice)

```
array([ 1, -2, -7,  0,  3, -4])
```

`T.argmax()` ⟶ `4` `T.argmin()` ⟶ `2` Indices du maximum et du minimum dans le tableau mis à plat

`numpy.unravel_index(T.argmax(),T.shape)` ⟶ `(1,1)` Position du maximum

Statistiques : `T.mean()` ⟶ `-1.5` `T.mean(axis=0)` ⟶ `array([0.5,0.5,-5.5])`

`T.std()` ⟶ `3.304…` (écart-type σ_n) `T.std(ddof=1)` ⟶ `3.619…` (écart-type σ_{n-1})