Entrées / Sorties

- `input` (renvoie une chaine de caractètes `str`) :
  `a = input("Ce que vous voulez : ")`
- `print` (sert à l'affichage)
  `In[] : print('Hello')`
  `Hello`

Types

- `int` (entier) : `783`, `0`, `-192`
- `float` (flottant) : `9.23`, `0.0`, `-1.7e-6`
- `complex` (complexe) : `2.7+3.1j`, `1j`
- `bool` (booléen) : `True`, `False`
- `str` (chaine de caractères) : `"Hello"`
  (séquence ordonnée de caractères, non modifiable)
- `list` (liste) : `[1,5,9]`, `["x",11,8.9]`, `[]`
  (séquence ordonnée, modifiable)
- `tuple` (tuple) : `(1,5,9)`, `"x",11,8.9`, `()`
  (séquence ordonnée, non modifiable)

Types conteneurs

listes, tuples et chaines
Accès index rapide, valeurs répétables.

Aide

`dir(a)` liste d'attributs de `a`.

Affectation de variables

`x=1.2+8`

- A gauche du signe =
  identificateur (nom de variable)
- A droite du signe =
  valeur ou expression de calcul.

Affectation multiple : `x=y=3`.
Affectation parallèle : `x,y,z=1.2,3,"bad"`
Incrémentation / Décrémentation : `x+=1`, `x-=2`.

Conversions de type

- `int("15")`
- `int(15.56)` (troncature de la partie décimale)
  `round(15.56)` pour entier arrondi.
- `float("-11.2e8")`
- `str(78.3)` renvoie `'78.3'`
- `list("abc")` renvoie `['a','b','c']`
- `eval("3+4")` renvoie `7`
  Utile pour convertir la valeur renvoyée par `input`

Maths

Opérateurs : `+`, `-`, `*`, `/`, `**`
`abs` (valeur absolue)
`round` (valeur approchée)

```
import math as m
m.sin(m.pi/4) # 0.707
m.sqrt(4) # 2.0
m.log(m.e**2) # 2.0
```

Complexes

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

Arithmétique

```
17 % 5
17 // 5
```

reste et quotient (division euclidienne de 17 par 5)

Indexation des séquences (listes, tuples et chaines de caractères)

index négatif		-6		-5		-4		-3		-2		-1	
index positif		0		1		2		3		4		5	
`liste=`	[	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(liste)` renvoie 6
  Accès individuel aux éléments par `[index]`
- `liste[1]` renvoie 67, `liste[0]` renvoie 11.
- `liste[-2]` renvoie 42, `liste[-1]` renvoie 1968.
  Accès à des sous-séquences par
  `liste[tranche début:tranche fin:pas]`
- `liste[:-1]` renvoie `[11,67,"abc",3.14,42]`
- `liste[1:-1]` renvoie `[67,"abc",3.14,42]`
- `liste[1:3]` renvoie `[67,"abc"]`
- `liste[-3,-1]` renvoie `[3.14,42]`
- `liste[::2]` renvoie `[11,"abc",42]`
- `liste[:]` renvoie `[11,67,"abc",3.14,42,1968]`
- `liste[:3]` renvoie `[11,67,"abc"]`
- `liste[4:]` renvoie `[42,1968]`

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

Opérations sur conteneurs

- `len(c)` (nombre d'éléments)
- `min(c)`, `max(c)`, `sum(c)`
- `sorted(c)` (copie triée)
- `val in c` (test de présence)
- `reversed(c)` (itérateur inversé)
- `c*5` (duplication), `c1+c2` (concaténation)
- `c.index(val)` (position)
- `c.count(val)` (nombre d'occurences)

Opérations spécifiques aux listes

- `lst.append(item)` (ajout d'un élément à la fin)
- `lst.extend(seq)` (ajout d'une séquence 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 d'un élément à une position et retour de la valeur)
- `lst.sort()` (tri de la liste sur place)
- `lst.reverse()` (inversion de la liste sur place)

Logique booléenne

Comparateurs : `<`, `>`, `<=`, `>=`, `==`, `!=`

- `a and b` (et logique)
- `a or b` (ou logique)
- `not a` (non logique)
- `True` (valeur constante vraie)
- `False` (valeur constante faux)

Instruction conditionnelle

bloc d'instructions exécuté
uniquement si une condition est vraie

```
if condition1:
    # bloc si condition1 vraie
    instruction1
elif condition2:
    # bloc sinon si condition2 vraie
    Instruction2
else:
    # bloc sinon des autres cas restants
    Instruction3
```

Instruction boucle conditionnelle
bloc d'instructions exécuté
tant que la condition est vraie

```
while expression logique:
    bloc d'instructions
```

Exemple :

```
# Initialisation avant la boucle
s=0
i=0
while i <= 100:
    # bloc exécuté tant que i<= 100
    s = s + i**2
    i = i + 1 # faire varier la variable
                de condition de boucle
print("somme: ",s)
```

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 sequence:
    bloc d'instructions
```

- Parcours des **valeurs** de la séquence :
  ```
  for val in liste:
      bloc d'instructions
  ```
- Parcours des **index** de la séquence :
  ```
  for idx in range(len(liste)):
      bloc d'instructions
  ```
- Parcours simultané **index** et **valeur** de la séquence :
  ```
  for idx,val in enumerate(liste):
  ```

Génération de séquences d'entiers

```
range([début,]fin[,pas])
```

```
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.

Définition de fonction

```
def fonction(p1,p2):
    """ documentation """
    # bloc d'instructions
    return res # valeur résultat de l'appel
```

Si pas de résultat calculé à retourner : `return None`

Les paramètres et toutes les variables du bloc suivant `def` n'existent que dans le bloc et pendant l'appel à la fonction.

Appel de fonction

```
r=fonction(2,4)
```

Fonction lambda

`f=lambda x:x**2` (définit la fonction $f : x \mapsto x^2$)

Listes en compréhension

```
liste=[2*i for i in range(10)]
liste=[i for i in range(20) if i%2 == 0]
```

Graphique

`LX` et `LY` sont deux listes (abscisses et ordonnées) de même taille.

```
import matplotlib.pyplot as plt
plt.figure('titre')
plt.plot(LX,LY)
plt.xlim(xmin,xmax)
plt.ylim(ymin,ymax)
plt.axis('equal') # axes orthonormés
plt.show() # affichage de la fenêtre
```

Numpy

```
import numpy as np
```

- `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 de dx.

Numpy : conversion

- `V=np.array([1,2,3])` crée le vecteur V : $(1\,2\,3)$.
- `L=V.tolist()` crée la liste L : $[1, 2, 3]$.
- `M=np.array([[1,2],[3,4]])` crée la matrice M : $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$.
- `L=M.tolist()` crée la liste L : $[[1, 2], [3, 4]]$.
- `M=np.matrix(M)` ou `M=np.mat(M)` transforme le tableau en une matrice : ce changement influe principalement sur le comportement de l'opérateur `*`.

Numpy : extraction d'une partie de matrice

- `M[i]` ou `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.
- `M2=M1.copy()` copie d'un tableau.

Numpy : opération sur les matrices

- Opération terme à terme :
  `M1+M2`, `M1*M2`, `M1**2`
- Multiplication de la matrice M par le scalaire c :
  `c*M`
- Renvoie le produit scalaire de deux vecteurs :
  `V1.dot(V2)` ou `np.dot(V1,V2)`
- Renvoie le produit d'une matrice par un vecteur :
  `M.dot(V)` ou `np.dot(M,V)`
- Renvoie le produit de deux matrices :
  `M1.dot(M2)` ou `np.dot(M1,M2)`
- Renvoie une copie de M transposée :
  `M.transpose()` ou `np.transpose(M)` ou `M.T`
- Calcul du déterminant et de l'inverse :

```
import numpy.linalg as la
la.det(M) # déterminant de M
la.inv(M) # inverse de M
la.solve(A,B) # renvoie X tel que AX=B
```