Informatique

Le langage Python

I Quelques généralités

- Introduction d'un commentaire avec #
- On écrit une instruction par ligne et les blocs sont regroupés par le même niveau d'indentation.
- Pour afficher du texte et le contenu d'une variable en console, on utilise la fonction `print` en séparant les expressions passées en argument par des virgules. Exemple :

```
>>> a = 45 * 2
>>> print( "Le coût est " , a, "€")
Le coût est  90 €
```

- Importation de modules avec la syntaxe au choix :
  - `import module` : importation d'un ensemble de fonctions. L'appel à ces fonctions doit obéïr à la syntaxe : `module.fonction(var1, var2,...)`
  - `import module as alias` : permet l'usage d'un nom local et la syntaxe de l'appel est simplifié : `alias.fonction(var1, var2,...)`
  - `from module import f, g` : l'appel aux fonctions `f`, `g` est simplifié car on n'a plus besoin de préciser de quelle librairie elles proviennent : `f(var1, var2,...)`

II Glossaire

- **séquence** : une collection ordonnée de valeurs où chaque valeur est identifiée par un indice entier.
- **élément** : l'une des valeurs dans une séquence.
- **indice** : une valeur entière utilisée pour sélectionner un élément dans une séquence, comme un caractère dans une chaîne. En Python, les indices commencent à 0.
- **tranche** : une partie d'une chaîne spécifiée par une gamme d'indices.
- **chaîne vide** : une chaîne sans aucun caractère et de longueur 0, représentée par deux guillemets simples.
- **immuable** : la propriété d'une séquence dont les éléments ne peuvent être modifiés.
- **parcourir** : itérer à travers les éléments dans une séquence, en exécutant une opération du même type sur chacun.
- **recherche** : un modèle de parcours qui s'arrête quand il trouve ce qu'il recherche.
- **compteur** : une variable utilisée pour compter quelque chose, généralement initialisée à zéro, puis incrémentée.

III Les types numériques

Les principaux types numériques	Exemples
Le type `int` : Python sait calculer sur les entiers relatifs arbitrairement grands (essayer de calculer 200 !), mais si l'on en abuse les opérations algébriques élémentaires ne peuvent plus être considérées comme s'exécutant en temps constant.	`>>> type(5)` `<class 'int'>`

Le type `float` : Python utilise aujourd'hui sur toutes les plates-formes les flottants double précision définis par la norme IEEE 754, stockés sur 64 bits (8 octets). 1 bit pour le signe, 11 bits pour l'exposant de 2 et 52 bits pour la partie "après la virgule" de la mantisse (normalisée en binaire sous la forme 1, . . . ; le 1 avant la virgule est donc un 53e bit, dit implicite).
Comme $\log_{10} 2^{53} \approx 16$, la précision dans le système décimal est d'environ 16 chiffres significatifs.
Attention ! Le fait qu'un ordinateur calcule en base 2 peut occasionner quelques surprises : par exemple 0.1 + 0.2 - 0.3 renvoie $5.551115123125783e-17$. . . ce qui vaut 0 aux erreurs d'arrondis près.
	`>>> type(5.6)`
	`<class 'float'>`
	
	`>>> erreur = 1e-16`
	`>>> a = 0.1 + 0.2 - 0.3`
	`>>> if abs(a) < erreur :`
	`...  a = 0`
	`...`
	`>>> a # à 1e-16 près !`
	`0`

Opérations	Commentaires	Exemples

`+, -, *, /`	Il s'agit des opérateurs algébriques courants.
	L'ordre de priorité usuel est respecté.
	Attention aux limites engendrées par la représentation des nombres en machine....
		`>>> 2*3-1`
		`5`
		`>>> 2*(3-1)`
		`4`
		`>>> 3*1.2`
		`3.5999999999999996`

`**`	Puissance.
	L'exposant peut être un entier relatif ou un nombre à virgule.
		`>>> 3**2 ; 3**(-2)`
		`9`
		`0.1111111111111111`
		`>>> 25**0.5`
		`5.0`

`e` ou `E`	Exponentiation.
	Comme sur une calculatrice ! On peut écrire `e` ou `E`, c'est indifférent.
	Le résultat est de type `float`.
		`>>> 5e3`
		`5000.0`
		`>>> 5e-3`
		`0.005`

`//`	`a//b` renvoie le quotient de la division entière de `a` par `b`.	`>>> 5//2` `2`
Opérateur modulo `%`	Soient a et b deux entiers naturels, b non nul, `a%b` renvoie le reste de la division entière de `a` par `b`.	`>>> 7%2` `1`
`divmod(a,b)` renvoie le doublet `(q,r)` où `q` est le quotient et `r` le reste.		`>>> divmod(7,2)` `(3, 1)`
`abs(n)` renvoie la valeur absolue de `n`		`>>> abs(-34.3)` `34.3`
`round(nombre,k)` arrondi le nombre à `k` chiffres après la virgule		`>>> round(34.493, 1)` `34.5`

`int(n)` convertit si possible `n` en un nombre entier. Il s'agit d'un transtypage.		`>>> int(34.8)`
		`34`
		`>>> int(True)`
		`1`

`float(n)` convertit si possible `n` en un nombre à virgule flottante. C'est du transtypage. Le séparateur décimal est le point « . » et non la virgule.		`>>> float(45)` `45.0`

`var @= e`	`@` désigne un opérateur arithmétique : `+,-,*,/,//,%,**`.
	`var @= e` produit le même résultat que : `var = var @ e` .
		`>>> a=5`
		`>>> a += 1`
		`>>> a`
		`6`

IV Affectation (ou assignation d'une valeur à une variable)

Une variable est une zone de la mémoire de l'ordinateur dans laquelle une valeur est stockée. Aux yeux du programmeur, cette variable est définie par un nom, alors que pour l'ordinateur, il s'agit en fait d'une adresse, c'est-à-dire d'une zone particulière de la mémoire.

Noms de variable (identificateur) : les règles

Sous Python, les noms de variables doivent obéir à quelques règles simples mais strictes :

1. Un nom de variable est une séquence de lettres (a → z, A → Z) et de chiffres (0 → 9), qui doit toujours commencer par une lettre.
2. Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que `$`, `#`, `@`, etc. sont interdits, à l'exception du caractère _ (souligné).
3. La casse est significative (différence entre un caractère minuscule et majuscule).

Les mots clés et les fonctions internes

Les mots réservés (ou mots clés) sont les mots du langage Python , ils ne peuvent pas être utilisés comme identificateur. Il y a en 30 en tout, en voici quelques uns : `and`, `def`, `if`, `else`, `for`, `import`, `in`, `return`, `while`....
À cette liste, il faut ajouter les trois valeurs constantes `True`, `False`, `None` qui sont également considérés comme des mots réservés.
De même, il existe des fonctions intégrées dans le langage Python qui ne doivent pas être redéfinies, exemple : `len`, `print`, `range`, `abs` ...

Affectation ou assignation

Une variable est l'association d'un identificateur et d'une valeur stockée en mémoire sous forme d'une écriture binaire. Cette association est réalisée par l'affectation d'une valeur à la variable. C'est une instruction :

LA SYNTAXE D'UNE AFFECTATION EST : <VARIABLE> = <VALEUR>

En Python, le typage des variables est dynamique : on n'a pas besoin de déclarer au préalable quel est le type de la <variable>, l'interpréteur détermine le type à la volée lors de l'exéction du code.

Quelques opérations d'affection	**Commentaires**	**Exemples**
Affectation simultanée `var1 = var2 = var3 = ... = e`	Toutes les variables var_i prennent la valeur e	`>>> x = y = 7*3`
Affectation multiple `var1, var2 ... = e1, e2, ...`	À chaque variable var_i Python a assigné la valeur e_i	`>>> a , b = 4 , 8.33`

Permutation
`var1 , var2 = var2 , var1`
	En Python, pas besoin d'une variable supplémentaire pour faire une permutation !	`>>> a , b = 4 , 8.33`
		`>>> a , b = b , a`
		`>>> print('a=',a,' et b=',b)`
		`a= 8.33  et b= 4`

Le type d'une variable définit les opérations qu'elle supporte. Exemples de types :		

`>>> type('bonjour')`
`<class 'str'>`
`>>> type(5)`
`<class 'int'>`
	`>>> type(True)`
	`<class 'bool'>`
	`>>> type([1, 2.5, 'Hi!'])`
	`<class 'list'>`
		`>>> type(3.0 * 4)`
		`<class 'float'>`
		`>>> type((1,2))`
		`<class 'tuple'>`

L'objet `None` possède lui aussi un type `NoneType` (dont il est le seul représentant), il ne correspond intuitivement à « rien », et permet de formaliser lorsqu'une fonction ne renvoie rien à son appel.		

V Les booléens, les constantes `True` et `False`

Les constantes `True` et `False` du type `bool`

Le type booléen permet d'effectuer des tests. Il comprend deux constantes : `True` et `False`. Les expressions booléennes sont des expressions logiques.

Table de vérité des opérateurs logiques :

not	
False	True
True	False

and	False	True
False	False	False
True	False	True

or	False	True
False	False	True
True	True	True

Opérateurs de comparaison

`P or Q`	`P` et `Q` sont des expressions booléennes ; si `P` est évaluée à `True`, `Q` n'est pas évaluée et (`P or Q`) est vraie. Si `P` est évaluée à `False`, `Q` est évaluée et (`P or Q`) est vraie ssi `Q` est vraie.
`P and Q`	`P` et `Q` sont des expressions booléennes ; si `P` est évaluée à `True`, `Q` est alors évaluée. Si `Q` est vraie, (`P and Q`) est vraie, fausse sinon.
`not(P)`	`P` est une expression booléenne ; `not(P)` est la négation de `P`
`E1 == E2`	teste l'égalité lorsqu' `E1` et `E2` sont des expressions (numériques, booléennes ou autres). Exemple : est-ce que `n` est divisible par 5 ? s'écrit : `n % 5 == 0` ; Python renvoie `True` ou `False`.
`E1 != E2`	retourne la négation de (`E1 == E2`). La réponse à la question : « Est-ce que `n` est différent de 34 ? » s'écrit `n != 34` ; Python renvoie `True` ou `False`

`E1 < E2`
`E1 <= E2`
`E1 > E2`
`E1 >= E2`
	Opérateurs de comparaisons, exemple : est-ce que n est positif ? s'écrit `n>0` ; Python renvoie `True` ou `False`.
	Pour tester si une variable est encadrée par deux valeurs, on peut écrire : `valeur1 < variable < valeur2` ; expression qui vaut `True` ou `False`

`e in S`	teste l'appartenance de `e` à un conteneur `S` ; retourne une erreur de type si `S` n'est pas un conteneur. Ex : `a in (1,2,3)` remplace efficacement `a==1 or a==2 or a==3`.

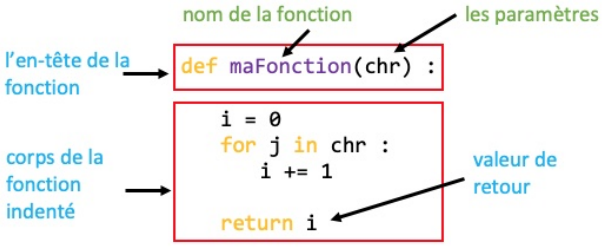
Véracité d'une expression

- Si on convertit un nombre non nul en booléen alors, il prendra la valeur `True`, s'il s'agit du nombre zéro, alors, il prendra la valeur `False`.
- Si on convertit une chaîne de caractères (ou une liste) vide en booléen, elle prendra la valeur `False`, toute autre chaîne (ou liste) prendra la valeur `True` ;
- Si on convertit un booléen en un nombre, `True` prendra la valeur 1, et `False` la valeur 0 ;
- Il existe une constante assez particulière en Python : `None` (l'unique constante du type `NoneType`). Cette constante a pour valeur « rien ». Si on convertit `None` en booléen, on obtient la valeur `False`.

VI Les fonctions

VI.1 La syntaxe et les règles à suivre

Une fonction est un ensemble d'instructions regroupées sous un nom qui renvoie un résultat lorsque son exécution se termine.
En Python, l'exécution d'une fonction `f(x)` évalue d'abord `x` puis exécute `f` avec la valeur calculée.
Le type d'une fonction est `function` et la définition d'une fonction doit respecter les règles suivantes :



les règles

1. La ligne d'en-tête commence par le mot-clé **`def`** suivi du nom de la fonction, de parenthèses entourant les paramètres séparés par des virgules, et du caractère « deux points ».
2. Le bloc d'instructions, indenté par rapport à la ligne de définition forme le corps de la fonction.
3. Le retour à la ligne signale la fin de la définition de la fonction.

VI.2 Ce que retourne une fonction

L'instruction **`return`** suivie d'une expression **provoque l'arrêt** de la fonction et **le retour de l'expression à l'endroit même où la fonction a été appelée**. Une fonction renvoie toujours une valeur et une seule (si cette valeur est composite, elle est renvoyée sous la forme d'un tuple). L'accès aux différents éléments constituant le tuple peut se faire par dépaquetage ou par leur indice dans le tuple.

```
>>> def maFonction(x):
...     return x**2 , 2*x
...
>>> maFonction(5)[0]
25
>>> car,dble = maFonction(5)
>>> print(car, dble)
25 10
```

L'emploi du mot clé **`return`** est facultatif, en cas d'absence, la fonction retourne **`None`**. Analyser l'exemple suivant :

```
>>> def maFonction(x):
...     print (x**2 , 2*x)
...
>>> carre,double = maFonction(5)
25 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot unpack non-iterable NoneType object
```

VI.3 Les expression passées en argument

Lors de la définition d'une fonction, il est possible de définir des valeurs par défaut pour un ou plusieurs paramètres, et ainsi la fonction peut être appelée en spécifiant seulement une partie des arguments, les autres seront pris égaux aux valeurs par défaut.

```
>>> def prod(a, b=2, c=1):
...     return a*b*c
...
>>> print(prod(5), prod(5,b=3), prod(5,b=3, c=2))
10 15 30
```

Si on spécifie la valeur de `b`, Python calcule avec cette valeur (c'est ce qui s'est passé avec `prod(5,b=3)`) et sinon il assigne 2 à `b` (c'est ce qu'il s'est passé avec `prod(5)`). Lorsqu'il y a plusieurs paramètres dont certains avec une valeur par défaut, ces derniers doivent impérativement être placés après les autres. Pour éviter toute ambiguïté, il faut les nommer : `b=3` par exemple.

VI.4 Spécification

On peut décrire une fonction de manière systématique en fournissant sa *spécification*. Pour cela, on doit décrire deux choses :

- **les préconditions** d'une fonction sont toutes les conditions qui doivent être satisfaites avant de pouvoir appeler la fonction, que ce soit sur des variables globales ou sur ses paramètres ;
- **les postconditions** d'une fonction sont toutes les conditions qui seront satisfaites après appel de la fonction, si les préconditions étaient satisfaites, que ce soit sur des variables globales ou sur l'éventuelle valeur renvoyée.

Voyons comment spécifier la fonction `isDivisor` en définissant ses pré- et postconditions :

```
>>> # Test de la divisibilité d'un nombre par un autre
>>> # Pre  : d et n sont deux entiers positifs
>>> #        d != 0
>>> # Post : la valeur renvoyée vaut True si d divise n,
>>> #        et False sinon
>>> def isDivisor(d, n):
...     return n % d == 0
...
```

Pour appeler la fonction, il faut donc lui fournir deux nombres entiers positifs en paramètres, et s'assurer que d soit différent de zéro. Dans ce cas, après avoir appelé la fonction, la valeur qu'elle aura renvoyée contiendra True si d est un diviseur de n et False sinon.

La spécification d'une fonction

La spécification d'une fonction contient donc toute la documentation nécessaire pour l'utiliser correctement. S'il ne faut pas devoir lire le corps de la fonction pour comprendre ce qu'elle fait et comment l'utiliser, c'est que la spécification est bien écrite.

VI.5 Annotations

Les en-têtes des fonctions peuvent être annotés pour préciser les types des paramètres et du résultat. Les annotations sont surtout utiles dans un but de documentation. Ainsi,

```
def uneFonction(n:int, X:[float], c:str, u) -> np.ndarray:
```

signifie que la fonction `uneFonction` prend quatre paramètres n , X , c et u , où n est un entier, X une liste de nombres à virgule flottante et c une chaîne de caractères ; le type de u n'est pas précisé. Cette fonction renvoie un tableau numpy .

Attention ! l'annotation de fonctions est optionnelle et elle n'est là qu'à titre indicatif : même si lors de l'appel de la fonction, le paramètre n'est pas du même type que celui inscrit dans la définition, Python ne générera pas d'erreurs. À ce titre, on notera aussi que donner des types comme annotations n'est qu'une convention. Annoter des paramètres avec des chaînes de caractères ne provoquera pas d'erreur par exemple.

VII Les structures conditionnelles

Syntaxe

Une instruction conditionnelle dans Python est de l'une des formes suivantes où, en allant de gauche à droite, on ajoute des instructions optionnelles :

if condition :	**if** condition :	**if** condition1 :
.... bloc d'instructions	.... bloc d'instructions 1	.... bloc d'instructions 1
#suite du code	**else** :	**elif** condition2:
	.... bloc d'instructions 2	.... bloc d'instructions 2
	#suite du code	**elif** conditionx :
		.... bloc d'instructions x
		else :
		.... bloc d'instructions x+1
		#suite du code

Chaque `condition` présente dans l'en-tête est une **expression booléenne** qui ne peut être évaluée qu'à `True` ou à `False`.

Sémantique

• `if` seul : si la condition est évaluée à True, alors le bloc d'instructions est exécuté, sinon le bloc est ignoré ;

• `if` suivi de `else` : si la condition est évaluée à True, le bloc d'instructions 1 est exécutée et le bloc d'instructions 2 est ignoré. Si la condition est évaluée à False, c'est le bloc d'instructions 2 qui est exécuté et la 1 ignorée. On imagine que la condition est nécessairement évaluée soit à True soit à False. Ainsi, un seul et unique bloc d'instructions sera nécessairement exécuté ;

• `if` suivie d'une ou plusieurs instructions `elif` : le programme sélectionne successivement les conditions suivant les mots `elif` et, dès qu'une évaluation retourne True, le bloc d'instructions correspondant est exécuté et l'instruction conditionnelle termine là ; à défaut si `else` est présente, les instructions correspondantes sont exécutées ; dans le cas contraire aucune instruction n'est exécutée.

UNE DES CONDITIONS AU PLUS EST EVALUEE A TRUE,
UN SEUL BLOC D'INSTRUCTIONS AU PLUS EST EXECUTÉ.

Analysez l'exemple suivant

```
>>> a = 5
>>> if a > 5:
...     a = a + 1
... elif a == 5:
...     a = a + 1000
... else:
...     a = a - 1
...
>>> a
1005
```

VIII Les chaînes de caractères et les tuples

Définitions

Un **tuple** est une suite d'éléments quelconques, séparés par une virgule et éventuellement encadrés par des parenthèses. On ne peut ni ajouter ni retrancher ni modifier les éléments d'un tuple (il est dit immuable).
Le mot clé `return` utilisé dans les fonctions, renvoie un objet de type tuple si l'expression retournée contient au moins deux données.
Une chaîne de caractère est une variable de type `str`, c'est également un objet non modifiable, c'est une séquence ordonnée de caractères.

Construction « à la main » avec une affectation

```
>>> t = 3,4*3 ; t
(3, 12)
>>> type(t)
<class 'tuple'>
>>> mot = 'bonjour'
>>> type(mot)
<class 'str'>
```

La virgule qui sépare deux éléments suffit à construire un tuple. On encadre les éléments d'un tuple par des parenthèses.
Une chaîne de caractère est encadrée par des guillemets simples ou doubles. Une chaîne vide (de valeur booléenne `False`) est simplement constituée de guillemets ouvrant et fermant : `''` est une chaîne vide.

Construction par concaténation + ou duplication *

```
>>> 'bonjour' + ' ' + 'le monde'
'bonjour le monde'
>>> (3,4) * 2
(3, 4, 3, 4)
```

Les opérateurs + et * permettent de concaténer des chaînes (entres elles) ou des tuples (entre eux) pour en générer de nouveaux (ailleurs en mémoire). On ne concatène pas une chaîne avec un tuple.

Accès aux éléments de la liste par indice positif valide

```
>>> mot = 'bonjour'
>>> mot[2]
'n'
```

Les différentes données dans une chaîne ou un tuple sont repérées par leur indice, c'est un entier qui donne leur position dans la séquence. Attention, le premier élément a pour indice 0.

Accès aux éléments de la séquence par tranchage (slicing)

```
>>> chr = 'Jacques Decour'
>>> chr[3 : 12 : 2]
'qe eo'
>>> chr[ : 12 ]
'Jacques Deco'
```

`S[i :j :k]` crée une vue de la séquence `S` (un tuple ou une chaîne de caractères), sélection des éléments de `S` allant de l'indice i (inclus) à l'indice j (exclu) par pas de k.

- Si la valeur de i n'est pas renseignée, alors, i prend la valeur 0
- Si la valeur de j n'est pas renseignée, alors le tranchage inclus le dernier élément de L
- Si la valeur de k n'est pas renseignée, alors le tranchage avance par pas de 1.

La fonction len , str, et opérateurs de comparaison

```
>>> t = 3,3*4; len(t)
2
```

La fonction `len` renvoie le nombre d'éléments de la séquence qui lui est passée en argument.

```
>>> n=10
>>> fich='graph'+str(n)+'.pdf'
```

Opération de transtypage : `str(x)` convertit x en chaine de caractères

```
>>>'alpha' < 'beta'
True
```

Opérateurs de comparaisons : < , > Le tri par ordre alphabétique est possible.

Le caractère d'échappement

```
>>> print("ceci est un \t test")
ceci est un      test
>>> print("Il m\'a dit \"Attention\" ")
Il m'a dit "Attention"
```

Le signe \ permet de transformer le caractère qui suit, par exemple,

- `\n` est un saut de ligne (new) , `\t` une tabulation
- `\"` ou `\'` guillemet qui ne ferme pas la chaine

IX La boucle conditionnelle

Syntaxe

```
while condition C(x1, x2 …xn) :
.... instruction1 # les instructions sont indentées pour former un bloc
.... instruction2
.... .
.... .
.... .
.... dernière instruction du bloc while
.... dans ce bloc, les variables xi sont susceptibles d'être modifiées
# On sort de la boucle while grace à la fin de l'indentation
```

La `condition` est une expression booléenne de la forme `C(x1,x2,...xn)` où les `(xi)` sont des variables présentes dans le corps du programme ; La `condition` est donc susceptible d'être modifiée par le corps de la boucle.

Sémantique

Le mot clé **while** permet de répéter une séquence d'instructions, en boucle, tant qu'une condition est vérifiée. La `condition`, une expression booléenne, qui figure après le mot clé **while** est évaluée :

— Si elle prend la valeur **False**, le programme sort de la boucle et passe à la suite du code (s'il y en a une) ; la suite d'instructions (instruction1, instruction2 etc) est donc ignorée.

— Si elle prend la valeur **True**, la suite d'instructions est exécutée et le programme retourne en début de boucle (où la condition est à nouveau évaluée) ; et ce, tant que la condition prend la valeur **True**.

le mot clé break

Le mot-clé **break** a pour effet de sortir de la boucle immédiatement. Testez le code suivant pour comprendre !

```
print(" *** Avant la boucle *** ")
entree="bidule"
while entree != ' ' :          # tant que la chaine entree n'est pas vide
    print(" *** dans la boucle ***")
    entree=input( 'Taper un mot : ' )    # on demande a l'utilisateur d'entrer une chaine
    if entree =='ouistiti' :               # s'il entre le mot ouistiti
        print('J'ai vu un ouistiti !')
        break                             # on quitte la boucle while et ...
print("*** Apres la boucle ***")           # on reprend la suite du code
```

Exemple et usage

On privilégie l'usage de la boucle **while** lorsqu'on ne connaît pas à l'avance le nombre de tour qu'il faudra exécuter pour terminer la tâche (exemple : la recherche du zéro d'une fonction par la méthode de Newton). Un exemple classique : l'algorithme de la division euclidienne

```
>>> def division_Euclidienne(a,b) :
...     q = 0
...     while a >= b :
...         a -= b        # on retranche b à a q fois tant que ce qu'il reste est >=b
...         q += 1
...         r = a          # ainsi, ce qu'il reste s'identifie à r
...     return q, r      # et q s'identifie au quotient
...
```

X Les objets itérables et la boucle `for`

Syntaxe générale

Les objets **itérables** sont des structures que l'on peut parcourir (listes, chaînes de caractères, etc). Une boucle `for` en Python réalise une itération sur les éléments d'un conteneur (=un objet itérable).

```
>>>   for <var> in <conteneur> :        # ligne d'en-tête qui se termine par :
...           bloc d'instructions       # instructions exécutées à chaque tour
...     sortie de la boucle             # fin de l'indentation
```

Parcourir le `<conteneur>` par ses éléments : `for <element> in <conteneur> :`

Analyser l'exemple suivant :

```
>>> Caps = [ 'Paris', 'Londres', 'Berlin']  J'aime Paris
>>> for ville in Caps :                     J'aime Londres
...     print(" J'aime " + ville)           J'aime Berlin
...
```

Parcourir le `<conteneur>` par les index de son contenu : `for k in range(len(<conteneur>) :`

`for k in range(n) :` se lit « Pour tout k entier allant de 0 à n-1 »

```
>>> Caps = [ 'Paris', 'Londres', 'Berlin'] ...
>>> n = len(Caps)                          0 Paris
>>> for k in range(n) :                    1 Londres
...     print(k,  Caps[k])                 2 Berlin
```

Un itérateur très important : La fonction `range`

La fonction `range` renvoie **l'énumération** d'une séquence d'entiers en prenant très peu de place en mémoire. La fonction `range` peut prendre entre 1 et 3 arguments entiers :

- `range(b)` énumère tous les entiers allant de 0 à b - 1 ;
- `range(a, b)` énumère tous les entiers allant de a à b - 1 ;
- `range(a, b, c)` énumère les entiers tous les entiers allant de a à b - 1 par pas de c ;

Cette commande ne mémorisera que le code permettant d'engendrer les valeurs, si on veut « voir » ces valeurs, il faut les stocker dans une liste grâce à un transtypage :

```
>>> list(range(5))          >>> list(range(5,10))       >>> list(range(0,10,3))
[0, 1, 2, 3, 4]             [5, 6, 7, 8, 9]           [0, 3, 6, 9]
```

Parcourir le `<conteneur>` par ses éléments et les index : la fonction enumerate `enumerate`

```
>>> Caps = [ 'Paris', 'Londres', 'Berlin']   (0, 'Paris')
>>> for e in enumerate(Caps) :               (1, 'Londres')
...     print(e)                             (2, 'Berlin')
...
```

Quelques calculs avec la boucle `for` et itération sur `range(n)`

Une somme : $\sum_{k=0}^{n} a_k$	`s=0`
	`for k in range(0,n+1) :`
	`  s = s + ak`

Un produit : $\Pi_{k=0}^{n} a_k$	`p=1`
	`for k in range(0,n+1) :`
	`  p = p * ak`

Construire une liste : $[a_0, a_1, \ldots, a_n]$	`L=[]`
	`for k in range(0,n+1) :`
	`  L.append(ak)`

Une suite récurrente : $u_{n+1} = f(u_n)$	`u = u0`
	`for k in range(1,n+1) :`
	`  u = f(u)`

récurrence d'ordre deux : $u_{n+2} = f(u_{n+1}, u_n)$	`u, v = u0, u1`
	`for k in range(2,n+1) :`
	`  u, v = v, f(v,u)`

XI Opérations sur les listes

Définition

Sous Python, on peut définir une liste comme une collection ordonnée d'éléments de type pouvant être variés, séparés par une virgule, l'ensemble étant enfermé dans des crochets . Une liste est une séquence modifiable et **itérable** (on peut parcourir une liste par ses éléments ou par leurs indices).

Construction « à la main » avec une affectation

```
>>>L1=[]
>>>L2=[1, [10,20], 'h', True, 1]
>>>type(L2)
<class 'list'>
```

`L1` est une liste vide.
`L2` est une liste composée de 5 éléments de types variés, dont une liste. Les listes peuvent s'imbriquées.

Construction par concaténation + ou duplication *

```
>>>La = [1,2,3] ; Lb=['a', 'b']
>>>Ltot = La + Lb
[1, 2, 3, 'a', 'b']
```

Les opérateurs + et * permettent de concaténer des listes pour en générer une nouvelle (ailleurs en mémoire).

Construction par ajout de donnée en fin de liste : `L.append(<var>)`

```
>>>La = [1,2,3]
>>>La.append('plus') ; La
[1,2,3,'plus']
```

La méthode `append` ajoute l'expression `<var>` à la fin de la liste `L` et ne retourne rien (`None`).

Construction par compréhension : `[ f(x) for x in L if C(x) ]`

On dit que l'on définit un sous-ensemble de la liste `L` en compréhension lorsqu'on le caractérise comme étant l'ensemble des éléments de `L` vérifiant une certaine relation `f(x)` sous une certaine condition `C(x)`. Exemple : Soit la liste des $(k-2)^2$ pour k pair variant dans $[0, N[$ où N est donnée. Cette liste se programme en Python selon : `>>>L = [(k-2)**2 for k in range (N) if k\%2==0]`

Accès aux éléments de la liste par indice positif valide

```
>>>L2=[1, [10,20], 'h', True, 1]
>>>L2[3] ; L2[1][0]
True
10
```

Les différentes données dans une liste sont repérées par leur indice, c'est un entier qui donne leur position dans la liste. Attention, le premier élément a pour indice 0. On accède aux données d'une liste par son nom, suivi de l'indice de la donnée entre crochets.

Accès aux éléments de la liste par tranchage (slicing)

`L[i :j :k]` crée une vue de `L`, sélection des éléments de `L` allant de l'indice `i` (inclus) à l'indice `j` (exclu) par pas de `k`. Si la valeur de `i` n'est pas renseignée, alors, `i` prend la valeur 0. Si la valeur de `j` n'est pas renseignée, alors le tranchage inclus le dernier élément de L. Si la valeur de `k` n'est pas renseignée, alors le tranchage avance par pas de 1.

Modification du contenu de la liste : affectation `L[i] = var` ou `L[i :j] = t`

```
>>>La = [1,2,3]
>>>La[0] = La[1]*4
>>>La
[1,8,3]
>>>Ltot = [1,2,3,'a', 'b']
>>>Ltot[:2] = ['aa', 'bb']
>>>Ltot
['aa','bb',3,'a', 'b']
```

Syntaxe : `L[i] = var` l'élément d'indice i est remplacé par la donnée var qui peut être calculée « à la volée » Syntaxe : `L[i :j] = t`

la tranche des éléments d'indices `i` à `j` est remplacée par le contenu de la séquence `t` (`t` doit être une donnée itérable)

La fonction `len` et la méthode `pop`

```
>>>len(Ltot)
5
```

La fonction `len` renvoie le nombre d'éléments dans la liste. Si elle est vide, `len` retourne 0 (équivalent à `False`)

```
>>>Ltot.pop(); Ltot
b
['aa','bb',3,'a']
```

L'instruction `lst.pop()` retourne le dernier élément de la liste `lst` et le supprime de la liste `lst`.

XII Opérations sur les dictionnaires

Définition : un dictionnaire est une collection non ordonnée de relations entre clés et valeurs

Un dictionnaire contient une collection non ordonnée, modifiable de paires clé-valeur (ou items). On reconnaît un dictionnaire à sa syntaxe : `{clé_1 : valeur_1,... clé_n : valeur_n}`. Les dictionnaires sont de type `dict`. Les clés peuvent être de type `str`, `int`, `float`, tuple de nombre, tuple de tuple de nombre... mais pas de type `list` ou un tuple de liste. Les valeurs peuvent être de n'importe quel type.

Instantiation : voici quelques exemples possibles

```
>>>dico = {}
```
Création d'un dictionnaire vide

```
>>>dico = {'prenom':'Jacques', 'nom':'Decour'}
```
Création « à la main »

```
>>>Mon_dico = dict('un'=1, 'deux'=2)
```
On peut utiliser la fonction de transtypage `dict`.

```
>>>liste = [('un',1),('deux',2),('trois',3)]
>>>Mon_dico = dict(liste)
{'un' :1,'deux':2,'trois':3}
```

```
>>> {a:a**2 for a in range(11) if a%2==0}
{0: 0, 2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
```
Par compréhension de dictionnaire : `{ cle(x) : valeur(x) for x in L if C(x) }`

Les clés : accès, itération et test de présence

```
>>> mon_dico = {'prenom':'Jacques', 'nom':'Decour'}
>>> mon_dico.keys()
dict_keys(['prenom', 'nom'])
```
La méthode `keys` donne accès aux clés du dictionnaire.

```
>>> for objet in mon_dico :
...     print(objet)
...
prenom
nom
```
La variable d'itération d'une boucle for sur un dictionnaire est une clé.

```
>>> 'prenom' in mon_dico
True
```
Le mot clé `in` permet de tester l'appartenance d'une clé mais pas d'une valeur.

Cette opération est à coût constant (= indépendant de la taille du dictionnaire).

Les valeurs : accès et itération

```
>>> mon_dico = {'prenom':'Jacques', 'nom':'Decour'}
>>> mon_dico['prenom']
'Jacques'
```
On peut accéder à une valeur connaissant sa clé.

```
>>> mon_dico.values()
dict_values(['Jacques', 'Decour'])
>>> #for objet in mon_dico.values() :
>>> #    print(objet)
```
La méthode `values` donne accès aux valeurs contenues dans le dictionnaire.

Les items : ajout ou modification, accès et itération

```
>>>dictionnaire[clé]=nouvelle_valeur
```
Si `clé` existe, sa valeur sera modifiée en `nouvelle_valeur`. Sinon, un nouvel item est crée.

```
>>> mon_dico = {'prenom':'Jacques', 'nom':'Decour'}
>>> mon_dico.items()
dict_items([('prenom', 'Jacques'), ('nom', 'Decour')])
```
La méthode `items` donne accès aux couples clé-valeur contenues dans le dictionnaire.

```
>>> #for cle,valeur in mon_dico.items() :
>>> #    print(cle,valeur)
```

La fonction `len` et la méthode `get`

```
>>> mon_dico = {'prenom':'Jacques', 'nom':'Decour'}
>>> len(mon_dico)
2
```
La fonction `len` renvoie la longueur d'un dictionnaire, c'est-à-dire le nombre de paires : clé-valeur (ou de clés).

L'instruction `valeur=dictionnaire.get(clé [, e=None])` où le second argument facultatif vaut `None` par défaut, renvoie la valeur correspondante à la clé si elle existe, sinon elle renvoie la valeur de `e`.

XIII Portée lexicale

On distingue les variables locales et les variables globales. Lorsqu'une expression fait référence à une variable à l'intérieur d'une fonction, Python cherche la valeur définie à l'intérieur de la fonction et à défaut la valeur dans l'espace global du module.

```
>>> def maFonction(x):
...     a = x**2 -2*x + 1
...     return a
...
>>> print(maFonction(5) , 'a = ',a)
16 a =  1005
```

La **variable locale** est créée à l'intérieur de la fonction et elle n'est pas accessible à l'extérieur de la fonction :

Expliquez l'erreur !

```
>>> k=2
>>> def maFonction_k(x):
...     a = x**2 -k*x + 1
...     return a
...
>>> maFonction_k(5)
16
```

La **variable globale** contient les objets prédéfinis en Python ainsi que les variables créées à l'extérieur de la fonction. Elles sont en général accessibles à l'intérieur de la fonction et peuvent encore être utilisées après l'exécution de celle-ci.

```
>>> k=10
>>> def maFonction_k2(x):
...     k=2
...     a = x**2 -k*x + 1
...     return a
...
>>> print(maFonction_k2(5) , 'k = ',k)
16 k =  10
```

Si, dans une fonction, on crée une variable (locale) ayant même nom qu'une variable globale déjà définie, il s'agit d'une nouvelle variable locale qui est créée.

```
>>> k=10
>>> def maFonction_k2(x):
...     k = k + 1
...     a = x**2 -k*x + 1
...     return a
...
>>> maFonction_k2(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in maFonction_k2
UnboundLocalError: local variable 'k' referenced before assignment
```

Une variable globale est accessible et modifiable dans l'environnement du programme où elle est définie. Une variable globale est accessible en lecture par une fonction définie dans le même environnement, mais elle est non modifiable par cette fonction. En effet, étudier l'exemple suivant :

```
>>> k=1
>>> def maFonction_kGlobal(x:float)->float:
...     global k
...     k = k+1
...     a = x**2 -k*x + 1
...     return a
...
>>> print(maFonction_kGlobal(5), 'k = ', k)
16 k =  2
```

Pour créer ou pour modifier une variable globale à l'intérieur d'une fonction, on utilise le mot clé `global` pour déclarer la variable dont le nom suit comme globale. L'instruction `global i,j` permet de désigner deux variables globales i et j dans une fonction.

XIV Assertion

Une assertion est une aide de détection de bugs dans les programmes. Pour étudier la notion d'assertion, nous allons nous appuyer sur l'exemple suivant.

Sans assertion

Soit la fonction `moyenne` qui admet comme argument `L` une liste non vide de nombres et retourne la moyenne de `L`.

```
>>> def moyenne(L) :
...     s=0
...     n=len(L)
...     for i in range(n):
...         s+=L[i]
...     moy = s/n
...     return moy
...
```

Nous allons écrire une fonction `moyenne2` qui a la même spécification que `moyenne`. Nous allons annoter l'en-tête de la fonction pour préciser le type des données attendues en entrée et fournies en retour. Nous allons utiliser des assertions pour vérifier le type de `L`, le nombre d'éléments de `L` ainsi que le type des éléments de `L` .

- La fonction `moyenne2` peut être appelée si le type de `L` est `list`. On ajoute la ligne suivante dans le code :
  `assert type(L) == list`
  Le programme teste si `type(L) == list`.
  - Si la condition est vérifiée, le programme continue à s'exécuter normalement.
  - Si la condition n'est pas vérifiée (on dit qu'on a une *levée de l'assertion*), alors le programme Python s'arrête et affiche le message d'erreur : `assert type(L)==list AssertionError`
- La fonction `moyenne2` peut être appelée si le nombre d'éléments de `L` est strictement positif. On ajoute la ligne :
  `assert len(L)>0`
  Le programme teste si len(L)>0, sinon on aurait une division par 0 dans la fonction.
  - Si la condition est vérifiée, le programme continue à s'exécuter normalement.
  - Si la condition len(L)>0 n'est pas vérifiée (on dit qu'on a une *levée de l'assertion*), alors le programme Python s'arrête et affiche le message d'erreur : `assert len(L)>0 AssertionError`

Avec assertions

La même fonction avec assertions donne :

```
>>> def moyenne2(L :list)->float :
...     assert type(L) == list
...     assert len(L) > 0
...     s=0
...     n=len(L)
...     for i in range(n):
...         assert type(L[i]) == float or type(L[i]) == int
...         s+=L[i]
...     moy = s/n
...     return moy
...
```

On supprime les assertions dans la version finale du programme Python.

XV Opérations sur les fichiers

Ouverture : la commande `open`	
`nom = open("fichier.txt","mode" )`	`open` commande d'ouverture du fichier passé en argument. `nom` est la variable qui identifie le fichier manipulé. `"fichier.txt"` est le nom du fichier manipulé. `"mode"` désigne le mode d'ouverture.
Ouverture en mode écriture seule : usage de la méthode `write`	

`f = open("fichier.txt","w" )`
`f.write("début")`
`f.write("suite de la 1ère ligne\n")`
`f.write("2ème ligne")`
`f.close()`
	pointe en début de fichier
	Si le fichier existe, il est écrasé. Sinon, il est crée.
	`\n` permet de sauter une ligne
	
	indispensable pour enregistrer.

`f = open("fichier.txt","a" )`
`f.write("suite de la 2ème ligne")`
`f.close()`
	Si le fichier n'existe pas il est créé ;
	sinon les données écrites le seront à la fin du fichier.
	indispensable pour enregistrer.

Ouverture en mode lecture seule : usage des méthodes `read`, `readline` et `readlines`	

`f = open("fichier.txt","r" )`
`tout = f.read()`
`f.close()`
	pointe en début de fichier ; read() lit tout le fichier, y compris les caractères de saut de lignes, le convertit en chaînes de caractères et l'affecte à `tout` ; indispensable pour enregistrer le fichier.

`f = open("fichier.txt","r" )`
`n_car = f.read(n)`
`m_car_suiv = f.read(m)`
`f.close()`
	pointe en début de fichier ; read(n) lit, convertit en chaines de caractères les `n` premiers caractères du fichier et l'affecte à `n_car`.
	idem avec les `m` caractères suivants.
	indispensable pour libérer le fichier.

`f = open("fichier.txt","r" )`
`ligne = f.readline()`
`f.close()`
	pointe en début de fichier
	lit le fichier ligne par ligne (en incluant le caractère de fin de ligne)
	indispensable pour enregistrer le fichier.

`f = open("fichier.txt","r" )`
`lignes = f.readlines()`
`f.close()`
	pointe en début de fichier
	fournit la liste des lignes du texte.
	indispensable pour enregistrer le fichier.

Mise en œuvre avec un fichier csv (pour comma-separated value)

Dans un fichier csv, chaque ligne de texte correspond à une ligne d'un tableau de données et un caractère spécial (le plus souvent une virgule) sépare les colonnes. Par exemple,imaginons le fichier planetes.txt qui contient le texte suivant (planète et rayon en km) :

```
Mercure , 2439
Venus , 6052
Terre , 6378
Mars , 3396
```

On commence par découper le texte en lignes :

```
>>> planetes = open('planetes.txt', 'r')
>>> lignes = planetes.readlines()
>>> planetes.close()
```

et chaque ligne doit ensuite être découpée en colonnes. Pour cela, on utilise la méthode `split` qui découpe une chaîne de caractères en une liste de sous-chaînes, le séparateur de ces sous-chaînes étant indiqué en paramètre.

Dans le cas de notre fichier csv cela donne :

```
>>> tab = []
>>> for chn in lignes:
.... tab.append(chn.split(','))
```

À cette étape, `tab` est une liste de listes de la forme : `[ ['Mercure', ' 2439\n '], ..]`

Il reste à convertir le deuxième terme de chacune de ces listes en un entier :

```
>>> for lst in tab:
.... lst[1] = int(lst[1])
```

et la liste `tab` est maintenant prête à être utilisée :

```
[['Mercure', 2439], ['Venus', 6052],['Terre', 6378], ['Mars', 3396]]
```

XVI Mutabilité

Les listes et les dictionnaires (et les tableaux `numpy`) sont des objets mutables contrairement aux chaînes de caractères et tuples. Ainsi, ce qui est étudié ici pour les listes est également valable pour les dictionnaires.

`>>>L1 = [1,2,3]`
`>>>id(L1)`
`4448440520`
`>>>t1 = (1,2,3)`
`>>>id(t1)`
`4448266352`
	L' affectation est une instruction qui réalise les opérations suivantes :
	1) Création d'un objet (appelé `obj`) de type `list` à une adresse mémoire. Cet objet possède un identifiant (adresse mémoire), un type et une valeur.La valeur de `obj` vaut : `[1, 2, 3]`. 2) Création de la variable `L1`. 3) Association de la variable `L1` avec l'objet `obj` contenant la valeur `[1,2,3]`.
	Même principe avec un objet non mutable. L' adresse mémoire s'obtient avec `id`.

`>>>L1[0] = 4`
`>>>id(L1)`
`4448440520`
`>>>t1 = (4,2,3); id(t1)`
`4448266640`
`>>>L2 = L1`
`>>>id(L2) == id(L1)`
`True`
	Une liste est mutable, on peut modifier le contenu de l'emplacement mémoire référencé par le nom L1 mais pas l'adresse elle-même.
	La variable `L1` contient uniquement la référence de l'objet `obj`.
	En revanche, là, on recrée un nouveau référencement à une adresse différente.
	L'instruction `L2=L1` n'affecte pas `[1, 2, 3]` à `L2` mais
	crée un nouveau référencement vers **le même** emplacement en mémoire !
	La copie est très rapide puisqu'on n'occupe pas deux fois plus de place mémoire.

`>>>t2 = t1`
`>>>id(t1) == id(t2)`
`True`
	Idem avec des objets non mutables mais ici, il n'y a pas de risques car il n'est pas possible de modifier `t1` ou `t2`. Il faut recréer un nouvel emplacement en mémoire.

Modifier un objet mutable qui a plusieurs référencements

`>>>L1[0] = 1`
`>>>L2`
`[1,2,3]`
`>>>t1 = (1,2,3)`
`>>>t2`
`(4,2,3)`
	Cette instruction va modifier la variable `L2` puisque
	ces deux noms référencent la même adresse.
	
	Ce n'est pas un problème pour un tuple qui est un objet non mutable.
	Si on veut modifier `t1`, on n'a pas d'autre choix que de le recréer entièrement.

Comment copier un élément mutable ?

Il est nécessaire de recréer une liste qu'on veut copier. Cette copie peut se faire par exemple à l'aide du slicing `[:]`, puisque ceci recrée une nouvelle liste. De manière équivalente à `L3 = L1[:]` on peut écrire `L3 = L1.copy()` dont la syntaxe est peut-être plus explicite.

Le caractère superficiel de la copie peut nécessiter « une copie profonde »

`>>>L1=[[1,2,3], [4,5]]`
`>>>L2 = L1[:]`
`>>> id(L1) == id(L2)`
`False`
`>>> id(L1[0]) == id(L2[0])`
`True`
	Considérons maintenant le cas d'une liste dont les éléments sont eux-mêmes des objets mutables et créons-en une copie.
	L1 et L2 référencent deux emplacements mémoires distincts.
	mais `L1[i]` et `L2[i]`
	référencent toujours le même objet mutable et toute modification
	de l'un entrainera automatiquement une modification identique de l'autre.

`from copy import deepcopy`
`L3 = deepcopy(L1)`
`id(L1[0]) == id(L3[0])`
`False`
	Dans ces situations on utilise un module spécialisé dans la copie :
	le module copy et plus précisément la fonction deepcopy de ce module
	qui réalise une copie profonde d'un objet mutable.

XVII Annexe : le module numpy

Constantes , fonctions réelles usuelles	
Exemples de commande	**Résultat**
`import numpy as np`	On importe le module `numpy` avec l'alias `np`
`np.pi`	valeur approchée de π.
`np.sqrt(x)`	racine carré de `x`
`np.exp(x)` ; `np.log(x)`	exponentielle et logarithme népérien
`np.sin(x)` ; `np.arcsin(x)`	fonctions trigonométriques, angles en radians.
`np.mean(v)` ; `np.std(v)`	moyenne et écart-type des valeurs dans `v`.
Tableaux	
`>>> tab1D = np.array([1,2,7])`	création d'un tableau 1D (ou vecteur)

`>>> tab2D = np.array([[1,2,3],[4,5,6]])`	création d'un tableau 2D (ou matrice)
	`array([[1, 2, 3],`
	`[4, 5, 6]])`

`>>> np.linspace(0,1,6)` `array([0., .2, .4, .6, .8, 1.])`	création d'une subdivision à valeurs décimales régulièrement espacées.
`>>> np.arange(0, 2, 0.5)` `array([0., .5, 1., 1.5 ])`	Même principe qu'avec `range`, ici les valeurs peuvent être à virgule flottante.
`>>> np.zeros(5)` `array([0., 0., 0., 0., 0.])`	création d'un tableau 1D rempli de zéros. Il existe aussi la fonction `ones`

`>>> np.zeros((2,3))`
`array([[ 0., 0., 0.],`
`[ 0., 0., 0.]])`
	création d'un tableau 2D rempli de zéros. Même principe avec la fonction `ones`.

Quelques opérations sur les tableaux	
`>>> np.arange(5) + np.arange(0, 50, 10)` `array([ 0, 11, 22, 33, 44])`	Les opérations algébriques `+`, `-`, `*`, `/` se font terme à terme.
`>>> np.dot(np.arange(5),np.arange(0, 50, 10))` `300`	La fonction `dot(A,B)` réalise le produit matriciel $A \cdot B$ tel qu'il est défini en maths.

`>>> A = np.array([[1, 2, 3], [4, 5, 6]])`
`>>> A.shape`
`(2, 3)`
	L'attribut `shape` donne la taille d'une matrice : nombre de lignes, nombre de colonnes.

À retenir 1 – Propriété importante : les fonctions sont vectorialisées

Dans le module `numpy` les fonctions mathématiques sont **vectorialisées**. Cela signifie qu'une telle fonction $x \to f(x)$, accepte en arguments des tableaux numériques `T` et retourne les tableaux `([ f(T[0]), ... f(T[n-1]) )]`. Analysez l'exemple suivant :

```
>>> a = np.arange(1,5)
>>> np.log(a)
array([ 0.,  0.69314718,  1.09861229,  1.38629
```

Si f est une fonction scalaire, on peut la vectorialiser : `f_vect=np.vectorize(f)`

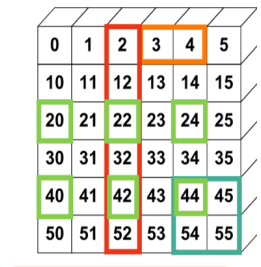
Usage du *slicing* :

```
>>> a[0,3:5]
array([3,4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,22,52])

>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```



Le slicing créé des vues, pas des copies

Affectation et copie :

Si on écrit `tab_bis = tab` où `tab` est un tableau numpy (objet mutable) alors on crée deux références pour le même objet : si on modifie `tab` dans la suite du programme, alors `tab_bis` est également modifié puisque `tab` et `tab_bis` font référence à la même adresse mémoire. Pour dupliquer un tableau et rendre les deux répliques indépendantes, il faut utiliser la fonction `copy` du module qui porte le même nom : `tab_bis = copy.copy(tab)`.

XVIII Annexe : le module matplotlib

XVIII.1 La fonction plot

a) Le principe général

Les outils pour les tracés graphiques sont dans la bibliothèque **matplotlib**. Les interfaces graphiques permettant de gérer les figures sont dans le module **pyplot** (il est aussi possible d'utiliser pylab) de cette bibliothèque. C'est lui que nous chargerons avec la bibliothèque **numpy** pour bénéficier des fonctionalités très pratiques de ce module.

```
>>> import numpy as np
>>> import matplotlib . pyplot as plt
```

À retenir 2

La règle pour tous les tracés 2D avec **matplotlib**, est :

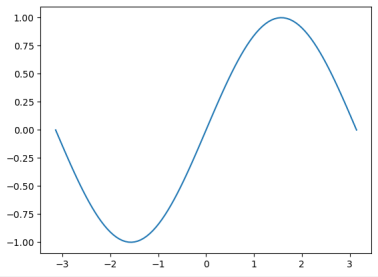
1. on se donne deux tableaux (ou listes) de valeurs numériques : `X=[ x1, x2, ..., xn ]` et `Y=[ y1, y2, ..., yn ]` de même taille et
2. on trace le polygone reliant les points `(x1,y1) , (x2,y2), (x3,y3), ... , (xn,yn)` à l'aide de l'instruction : `plt.plot( X, Y )`

b) Exemple : tracer la fonction $t \to \sin(t)$ sur $[-\pi;\pi]$

1. On commence par créer le tableau des abscisses : `>>> X = np.linspace(-np.pi ,np.pi ,100)`

   La fonction `linspace(start, stop, N)` de la bibliothèque **numpy** génère un tableau commençant par `start`, terminant par `start`, contenant `N` points régulièrement espacés.
2. Ensuite, on crée le tableau des ordonnées en appliquant la fonction `np.sin()` (i.e. la fonction sinus définie dans le module NumPy est vectorielle, et donc applicable à des tableaux) à chaque élément de `X`. Pour cela, on va appliquer directement `np.sin()` à un tableau, ce qui produit le tableau des images : `>>> Y = np.sin(X)`
3. Enfin, on demande le tracé de la courbe joignant tous les points de coordonnées `(X[k]; Y[k])` via la commande : `>>> plt.plot (X,Y)`

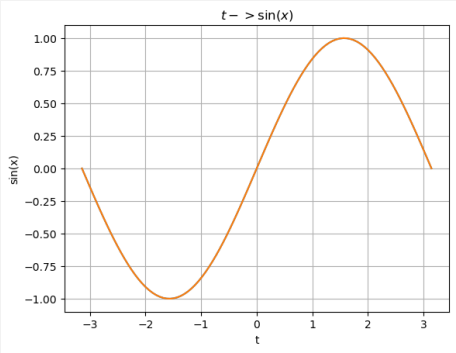
```
import matplotlib.pyplot as plt
import numpy as np
X = np.linspace(-np.pi, np.pi, 100)
Y = np.sin(X)
plt.plot(X, Y)
plt.show()
```



c) Quelques commandes courantes

Tracer une ligne brisée :	`plt.plot([x1 ,... , xn ],[ y1 ... , yn ])`
Mettre des titres :	`plt.title('Titre de la figure ')`
Noms des axes :	`plt.xlabel('Nom de l\'axe des x'); plt.ylabel('axe des y')`
Visualiser le résultat :	`plt.show()`
Sauvegarder la figure dans un fichier :	`plt.savefig('nom_de_fichier.pdf ')`
Mettre une grille :	`plt.grid()`

```python
import matplotlib.pyplot as plt
import numpy as np
X = np.linspace(-np.pi, np.pi, 100)
Y = np.sin(X)
plt.title ('$t->\sin(x)$')
plt.xlabel ('t')
plt.ylabel ('sin(x)')
plt.grid()
plt.plot(X, Y)
plt.savefig('sinus2.png')
plt.show()
```



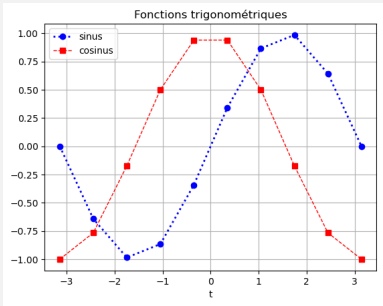
d) Quelques options pour décorer la courbe

Il faut préciser les options de la fonction `plot`. Voici quelques options courantes :

- **Couleur de la courbe** : `color='black'` donne un tracé en noir ; les autres couleurs disponibles sont : `'blue'`, `'green'`, `'red'`, `'cyan'`, `'magenta'`, `'yellow'`, `'white'`
- **Style de trait** : `linestyle='-'` donne un trait continu, ; les autres styles disponibles sont : `'--'`, `' :'`, `'-.'`
- **Épaisseur du tracé** : `linewidth='2'` donne un trait plus épais (la valeur par défaut est 1)
- **Marques pour les points** : `marker='x'` matérialise la position de chaque point par une croix ; les autres styles disponibles sont : `'o'`, `'+'`, `'*'`, `'v'`, `'.'`, `'^'`, `'<'`, etc. . . .
- **Légende** : `label='Courbe théorique'` définit le nom associé à un tracé ; moyennant quoi la commande `plt.legend()` affiche un cartouche avec les différents styles de tracés suivis du nom associé par l'option `label`. On peut choisir la position du cartouche en précisant par exemple `plt.legend(loc=2)` avec 1 pour en haut à droite ou 2 pour en haut à gauche etc. Par défaut, on laisse Python placer le cartouche.

```python
import matplotlib.pyplot as plt
import numpy as np

X = np.linspace(-np.pi, np.pi, 10)
Ys = np.sin(X)
Yc = np.cos(X)
plt.title ('Fonctions trigonométriques')
plt.xlabel ('t')
plt.grid()
plt.plot(X, Ys, color = 'blue', linestyle = ':', linewidth='2', marker = 'o', label='sinus')
plt.plot(X, Yc, color = 'red', linestyle = '--', linewidth='1', marker = 's', label='cosinus')
plt.legend() # à placer après plt.plot()
plt.show()
```

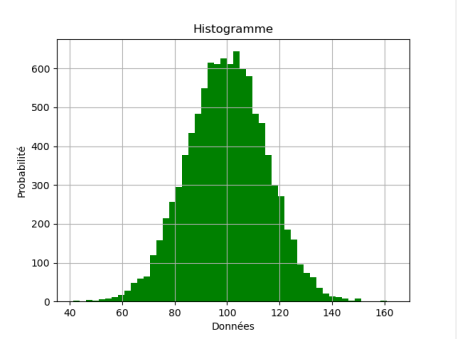


Remarques : On peut aussi concaténer les options concernant le tracé dans une unique chaine de caractères. Analyser l'exemple suivant : `plt.plot(x, y, 'r+:')` : le tracé est rouge (`r`), les points sont marqués par + et reliés par des pointillés ':'

XVIII.2 La fonction `hist`

Soit x un tableau (ou une liste) de valeurs dont on souhaite représenter la distribution.

```
import numpy as np
import matplotlib.pyplot as plt
# simulation d'un tableau de valeur aléatoire x
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)
# histogramme des données
plt.hist(x, bins=50, color='g')
plt.xlabel('Données')
plt.ylabel('Probabilité')
plt.title('Histogramme')
plt.grid(axis='y')
plt.show()
```



On crée l'histogramme à l'aide de la méthode `plt.hist(x, options)`.
Voici quelques paramètres de cette méthode, sachant que seul le tableau de valeurs est obligatoire !

- `x` : tableau contenant les valeurs de l'échantillon,
- `bins=10` : le nombre d'intervalles donc de barres ; optionnel, la valeur par défaut est 10.
- `range` : donne les valeurs limites de l'axe des abscisses `(xmin, xmax)` ;
- `rwidth` : largeur des barres sous la forme d'une fraction de l'intervalle ; optionnel, la valeur par défaut est 1.
- `color`, `edgecolor` : couleur des barres et de leur contour ; optionnel
- `label` : permet de légender l'histogramme ; optionnel. Attention, son utilisation nécessite l'appel de la méthode `plt.legend()` dans la suite du programme

On peut ensuite afficher un titre, légender les axes tout comme on le ferait sur un graphique grâce aux méthodes `plt.title(), plt.xlabel(), plt.ylabel()`....

XVIII.3 Les autres fonctions

On cite les autres types de graphes :

- `plt.scatter` pour représenter un nuage de points,
- `bar` pour tracer des diagrammes à barre,
- `pie` pour tracer des diagrammes camemberts.

Les commandes `plt.semilogx()`, `plt.semilogy()` et `plt.loglog()` mettent respectivement le graphe à l'échelle logarithmique simple en x, logarithmique simple en y et double échelle logarithmique.

```
import numpy as np
import matplotlib.pyplot as plt
x=np.linspace(1,1000,50)
plt.loglog()
plt.plot(x,1./x)
plt.plot(x,1./x**2)
plt.savefig('log.png')
plt.show()
```