Table des matières

1 Rappels de base sur Python **3**
- 1.1 Quelques principes généraux - comparaison avec C . . . 3
- 1.2 Les types usuels . . . 3
  - 1.2.1 Les entiers et flottants . . . 3
  - 1.2.2 Les booléens . . . 3
  - 1.2.3 Chaînes de caractères . . . 4
  - 1.2.4 Conversions . . . 4
- 1.3 Structures de contrôle . . . 4
  - 1.3.1 Affectation, affectations multiples . . . 4
  - 1.3.2 Tests . . . 5
  - 1.3.3 Les boucles . . . 5
- 1.4 Fonctions . . . 6
  - 1.4.1 Définitions . . . 6
  - 1.4.2 Autres situations . . . 7
  - 1.4.3 Variables locales et globales . . . 7
  - 1.4.4 Typage des arguments, du retour - Signature d'une fonction . . . 8
- 1.5 Listes . . . 8
  - 1.5.1 Création . . . 8
  - 1.5.2 Accès aux éléments . . . 8
  - 1.5.3 Ajout d'un élément et concaténation . . . 8
  - 1.5.4 Les n-uplets (tuple) . . . 9
  - 1.5.5 Listes et fonctions . . . 9
  - 1.5.6 Quelques méthodes sur les listes en complément . . . 10
- 1.6 Représentation interne des variables en Python . . . 10
- 1.7 Dictionnaires . . . 11
  - 1.7.1 Comparaison listes et dictionnaires . . . 11
- 1.8 Modules . . . 12
  - 1.8.1 Module math . . . 13
  - 1.8.2 Autres modules usuels . . . 13
- 1.9 Fichiers . . . 13
  - 1.9.1 Lecture et écriture . . . 13

2 Divers **15**
- 2.1 Commentaires, variables . . . 15
- 2.2 Assertions . . . 15

Chapitre 1 | Rappels de base sur Python

Ces rappels ne sont pas écrits dans l'ordre - ce n'est pas un cours. Certaines parties utilisent des éléments qui sont rappelés dans les paragraphes suivants.

I | Quelques principes généraux - comparaison avec C

Quelques différences principales entre C et Python (tout n'est pas totalement vrai - ou disons parfois incomplet mais l'essentiel pour vous est là)

- langage interprété et pas compilé, entièrement objet
- typage dynamique en Python, pas besoin de déclaration de type (nulle part, ni pour les variables, ni pour les fonctions : ni pour les arguments, ni pour le retour). Une même variable peut changer de type à tout moment.
- structuration du code : obtenue par l'indentation du bloc de code (pas de délimiteur { et } ou autre - et ce dans toutes les situations que ce soit une boucle/un test/le corps d'une fonction. En général, on place 4 caractères d'espacement (les éditeurs usuels remplace la tabulation par ces 4 espaces).

```
if x==2:
    y = x+1
else:
    y = x+2
```

- pas de pointeurs, pas besoin de gérer la mémoire à la main

II | Les types usuels

II.1 | Les entiers et flottants

OPÉRATIONS SUR LES ENTIERS ET FLOTTANTS

`+,-,*`	opérations classiques	
`//`	division entière	
`/`	division flottante	convertit si besoin
`%`	modulo (reste de la division euclidienne)	
`abs(x)`	valeur absolue	int, float, complex
`pow(x,y)` ou `x**y`	puissance	avec int, float et complex

II.2 | Les booléens

- Deux valeurs définies : *True* et *False* (correspondent aux entiers 1 et 0)
- opérateurs :
  - → `bool1 and bool2` : évalue d'abord `bool1` et si l'expression est vraie, évalue alors `bool2` (appelé évaluation paresseuse - très utilisé lors des tests afin notamment de controler qu'une variable prend une valeur acceptable)
  - → `bool1 or bool2` : même principe sur l'évaluation, s'arrête dès qu'une condition est vraie
  - → `not bool1`
- le test d'égalité se fait avec == (pas d'affectation dans les instructions - `if k=1` : renverra une erreur).

II.3 | Chaînes de caractères

Définition d'une chaîne de caractères

Une chaîne de caractères est définie sous la forme `'chaîne'` ou `"chaîne"` - on peut également la définir avec des triples guillemets ou triples apostrophes : on tape alors tout ce qu'on veut, sur plusieurs lignes si l'on souhaite. Cette façon est notamment utilisée pour documenter convenablement une fonction.

Opérations sur les chaînes de caractères

OPÉRATIONS SUR LES CHAÎNES

`s+t`	concaténation des deux chaînes
`s*n`	création d'une chaîne ou s est copiée n fois
`s[i]`	éléments en position i - le premier est le 0 (uniquement en lecture)
`s[i:j]`	tranche entre l'élément i (inclus) et j (exclu)
`s[i:j:k]`	tranche entre l'élément i (inclus) et j (exclu) avec un pas de k
`len(s)`	longueur de s
`s in t`	test d'appartenance

```
>>> c1 = 'Python'
>>> c2 = 'MPI'
>>> c1+' '+c2
'Python MPI'
>>> c2*3
'MPIMPIMPI'
```

Remarque : les chaînes de caractères sont des objets immuables (non mutable), c'est-à-dire qu'une fois créé, l'objet ne peut plus être modifié (toute modification crée un nouvel objet). Notamment, il n'est pas possible de remplacer un caractère de la chaîne `c` via une affectation `c[position]=...`

II.4 | Conversions

Il suffit en général d'une commande sous la forme `type(donnees)` :

```
>>> chaîne='123'
>>> int(chaîne)+3
126
>>> float(chaîne)+3
126.0
>>> str(23+12)
'35'
```

III | Structures de contrôle

III.1 | Affectation, affectations multiples

On utilise = pour affecter une variable. L'expression de droite est tout d'abord évaluée, puis son résultat est affecté à la variable.

On peut effectuer une affectation à plusieurs variables en même temps

```
>>> a,b = 2,'test'
>>> a
2
>>> b
'test'
```

On peut également utiliser n'importe quel itérable pour affecter plusieurs variables successivement (par dépaquetage), du moment qu'on a le bon nombre de variables :

```
>>> a,b = [2,3]
>>> l1,l2,l3 = range(3)
>>> e,f = (2,5)
```

C'est en fait ce qu'il se passe lors de `a,b = 2,3` : la partie droite est évaluée, cela retourne un tuple `(2,3)`, puis l'affectation multiple se fait.

Cela permet, par exemple, d'échanger le contenu de deux variables :

```
a,b = b,a
# la partie droite est évaluée, mise dans un tuple et le résultat est
   affecté par dépaquetage dans a et b
```

III.2 | Tests

Syntaxe

```
if test1:
    debut_bloc_1
    if sous_test1:
        sous_bloc_interne
    else:
        sous_bloc_sinon
    fin_bloc_1
```

UTILISATION DE L'*évaluation paresseuse*

sur un test `if (cond1) and (cond2):`, on détermine d'abord le résultat du premier test. S'il est faux, le second n'est pas évalué. C'est notamment utile lorsqu'on doit vérifier que l'indice dans un tableau est acceptable. Si L est un tableau de taille n, pour s'assurer qu'on n'aura pas d'erreur `Index out of range`, on peut utiliser : `if (i<n) and (L[i]...)`

III.3 | Les boucles

Boucles inconditionnelles

```
for x in iterateur:
```

où `iterateur` est un objet « itérable », c'est-à-dire qu'il dispose d'une méthode qui permet de parcourir ses éléments les uns après les autres. Pour les objets itérables usuels :

- `range(n)` (entier de 0 à $n-1$), `range(a,b)` (entiers de a à $b-1$), `range(a,b,pas)` (entiers $a+k.pas$ strictement inférieurs à b)

```
for i in range(1,10):
    ....
    # i prend les valeurs de 1 à 9
```

- les listes : x va décrire chacun des éléments de la liste,

```
for x in ['a','b','c']:
    ....
    # x prend successivement les valeurs 'a', 'b' et 'c'
```

- les chaînes : x va décrire chaque caractère de la chaîne

MODIFICATION DU COMPTEUR

on peut modifier la valeur de x à l'intérieur de la boucle, en revanche au passage suivant, x prendra systématiquement la valeur suivante de l'itérateur (si x décrit les entiers de 0 à 9 et x en est à la valeur 4, même si on modifie x, au passage suivant il prendra la valeur suivante à savoir 5)

Remarque : la fonction `range(10)` ne crée pas la liste des entiers de 0 à 9 mais seulement un objet itérable qui va permettre d'obtenir ces entiers les uns après les autres. Si on veut créer la liste de ces entiers, on peut forcer la conversion en liste :

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

Boules conditionnelles (tant que)

```
while test:
    bloc1
```

Tant que la condition `test` est vérifiée, on effectue le contenu de la boucle. Il vaut mieux que les variables qui interviennent dans la condition soient modifiées dans le corps de la boucle si on ne veut pas se lancer dans une boucle infinie...

Sortie d'une boucle

On dispose de `break` et `return` lorsqu'on veut sortir « brutalement » d'une boucle. Le premier sort simplement de la boucle et continue le programme qui suit, le second (uniquement dans une fonction), quitte définitivement la fonction

```
def fonction(n):
    for i in range(n):
        print(i)
        if i>5:
            break
    print("On est sorti")

def fonction2(n):
    for i in range(n):
        print(i)
        if i>5:
            return
    print("On est sorti")
```

```
>>> fonction(10)
0
1
...
5
6
On est sorti
>>> fonction2(10)
0
1
...
5
6
```

IV | Fonctions

IV.1 | Définitions

```
def f(x):
    traitement
    return(...)    # pas obligatoire
```

Il n'y a pas de différence entre fonction et procédure, donc une fonction n'est pas obligée de renvoyer quelque chose (par défaut la valeur `None` est retournée). Elle peut cependant renvoyer n'importe quel type d'objet (et même renvoyer des objets de types différents suivant les valeurs de x, ce qui n'est pas forcément conseillé)

On peut définir des fonctions prenant plusieurs paramètres en entrée :

```
>>> def f(x,y):
        return x*y
>>> f(3,4)
12
```

On peut également mettre des arguments avec des valeurs par défaut à la fin des arguments :

```
def f(x,n=1):
    ....
```

on peut l'appeler soit avec `f(2)` (dans ce cas n aura la valeur 1 dans la fonction), soit avec deux arguments `f(2,3)`

IV.2 | Autres situations

Définition d'une fonction avec l'opérateur lambda

On peut avoir besoin d'une fonction simple (par exemple pour la transmettre en argument d'une autre) sans avoir envie de lui donner un nom (et la définir avec `def`). Pour cela, on dispose de l'opérateur `lambda`

```
>>> f = lambda x:x*x
>>> f(2)
  4
>>> (lambda x,y : x+y)(3,4)
7
```

Fonction dans une fonction

On peut avoir besoin localement d'une fonction dans une autre (avec même éventuellement un argument de la fonction extérieure qui peut être utilisée). Par exemple, on veut manipuler une fonction qui dépend d'un paramètre :

```
def essai(n):
    def f(x):
    # utilise la valeur de n transmise en argument de essai
        return 1/(n*n+x*x)
    ...
```

IV.3 | Variables locales et globales

```
>>> def f():
...     a=3
...     print(a)
>>> a=1
>>> f()
3
>>> a
1
```

La variable `a` affectée dans la fonction `f` est locale à cette fonction. Elle est même totalement oubliée lorsqu'on quitte la fonction :

En résumé (presque exact), on a

- les arguments passés à une fonction sont créés en tant que variable locale,
- toute variable affectée dans une fonction est locale
- une fonction cherche d'abord dans les variables locales à la fonction (ou à celles d'une fonction englobante) puis dans les variables globales,
- on peut préciser qu'on veut utiliser la version globale d'une variable (et ainsi la modifier globalement) en précisant son statut global dans la fonction (avant de l'utiliser) avec le mot clé `global` (sous la forme `global variable`) - c'est évidemment à utiliser plus que le moins possible (on se retrouve avec une variable modifiée sans forcément le savoir... dangereux).

IV.4 | Typage des arguments, du retour - Signature d'une fonction

On peut faire apparaître un « typage » des fonctions. Par exemple

```
def multiplie(chaine: str, nombre: int) -> str:
    return nombre*chaine
```

Dans notre utilisation, ce typage n'est qu'une indication (Python de ne va pas contrôler le type des arguments - ça peut être utilisé par d'autres programmes d'analyse ou de vérification de code).

Les types usuelles : `int`, `float`, `str`, `list` (et `list[int]` pour des listes d'entiers), `tuple`, `dict`, `Callable` (pour les fonctions)

V | Listes

V.1 | Création

Une liste est un ensemble ordonné (chaque élément à un numéro d'ordre) d'éléments de types hétérogènes. Elles sont définies entre crochets et les éléments sont séparés par une virgule. On peut créer une liste

- directement : `L = [1,2,3]`
- par conversion d'un objet itérable : `L = list(range(10))`, `L=list("une chaine de caractères")`
- par concaténation de listes : `L = L1+L2`
- par copie d'une liste `L = L1.copy()`
- par concaténation multiple `L = [0]*5` (donne `[0,0,0,0,0]` : très utile pour initialiser une liste à un certain nombre d'éléments).
- par compréhension : `[ fonction(x) for x in iterable ]`
- par compréhension et filtrage : `[ fonction(x) for x in iterable if conditions]`

Par exemple, on veut construire tous les carrés des éléments plus grand que 2 d'une liste

```
>>> l=[0,4,1,-3,7,8]
>>> [x*x for x in l if x>=2]
[16, 49, 64]
```

V.2 | Accès aux éléments

La numérotation des éléments d'une liste est exactement la même que pour les chaînes, à la différence que cette fois on peut accéder aux éléments à la fois en lecture et en écriture : si `l` est une liste

ACCÈS AUX ÉLÉMENTS

`l[i]`	éléments en position i - le premier est le 0
`l[i:j]`	tranche entre l'élément i (inclus) et j (exclu)
`l[i:j:k]`	tranche entre l'élément i (inclus) et j (exclu) avec un pas de k
`l[i:]`	tranche entre l'élément i (inclus) et la fin
`l[:j]`	tranche entre le début et l'élément j (exclu)
`len(s)`	taille de la liste

V.3 | Ajout d'un élément et concaténation

- Lorsqu'on veut ajouter en fin d'une liste `l`, on peut le faire grâce à l'opérateur + :

```
>>> l=[4,5,6]
>>> l+[8]
[4, 5, 6, 8]
>>> l
[4, 5, 6]
```

l'opération `l+[8]` crée une nouvelle liste et ne modifie pas `l`. Si on n'affecte pas le résultat, on a perdu la modification. On peut le réaliser entre deux listes, et même concaténer plusieurs fois une même liste :

```
>>> l=[4,5,6]
>>> m=[0,1]
>>> l+m
[4,5,6,0,1]
>>> m*3
[0,1,0,1,0,1]
>>> [0]*10
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- Avec la méthode `append` de la classe `list`. Cette fois on modifie effectivement le contenu de la liste

```
>>> l=[4,5,6]
>>> l.append(8)
>>> l
[4, 5, 6, 8]
```

IMPORTANT - CONCATÉNATION ET CRÉATION

- si on connaît la taille à l'avance, autant créer directement la liste de cette taille plutôt que d'ajouter des éléments les uns après les autres (si la liste est de taille N, on la crée avec `L=[0]*N` par exemple)
- pour ajouter un élément à une liste, on utilise la méthode `append` plutôt que la création complète d'une nouvelle liste avec l'opérateur + (sauf raisons valables)

V.4 | Les n-uplets (tuple)

Il existe un type proche de celui des listes : les tuples (ou n-uplets). On remplace les crochets par des parenthèses lors de leur définition. La grosse différence est que les données ne sont pas modifiables. C'est notamment ce que renvoie un `return` si on donne plusieurs valeurs séparées par des virgules. Les opérations sont similaires à celles sur les listes

```
>>> t = (3,6,9)
>>> s = 2,3  # transformé en tuple
>>> s
(2,3)
>>> t[1]
6
>>> s+t
(2,3,3,6,9)
```

L'avantage d'être non mutable est qu'un tel objet peut être utilisé comme clé d'un dictionnaire.

V.5 | Listes et fonctions

Pour les explications, lire le paragraphe suivant. Il faut continuer à dissocier la liste et le contenu de la liste dans la fonction appelée. L'exécution de

```
def f(liste):
    liste[1]="modif"
    # la fonction ne retourne rien

l=[0,1,2]
f(l)
print(l)
```

donne à l'affichage

```
[0, 'modif', 2]
```

- La liste `l` est transmise en argument de la fonction `f` sous un identifiant `liste` local à la fonction est créé et pointe au même endroit que la liste `l`.

- L'affectation `liste[1]="modif"` change le contenu de la liste, mais ne modifie pas son identifiant si bien que `liste` pointe toujours au même endroit que `l`

```
def f(liste):
    print("avant", liste)
    liste=[4,5,6]
    print("après", liste)
```

```
>>> l=[0,1,2]
>>> f(l)
avant [0, 1, 2]
après [4, 5, 6]
>>> l
[0, 1, 2]
```

V.6 | Quelques méthodes sur les listes en complément

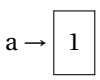
méthode	commentaires
utilisables	
`l.append(a)`	ajoute l'élément `a` à la fin de la liste `l`
`l.pop(indice)`	retourne l'élément d'indice `indice` et le retire de la liste `l`. Par défaut, c'est le dernier élément qui est retiré. Si la liste est vide, la méthode renvoie une erreur
`l.copy()`	copie la liste `l` de façon superficielle (les éléments ne sont pas récursivement copiés)
`l.sort()`	trie la liste `l`
à éviter	
`l.insert(indice,a)`	insert l'élément `a` à l'indice `indice`. Si cet indice dépasse la taille de la liste, l'élément est ajouté à la fin
`l.extend(iter)`	ajoute à la fin de `l` la liste créée à partir de l'objet itérable `iter`
`l.index(a)`	retourne l'indice de l'élément `a` dans la liste `l` s'il est présent dans cette liste et une erreur sinon
`l.count(a)`	compte le nombre doccurrences de l'élément `a` dans la liste `l`

VI | Représentation interne des variables en Python

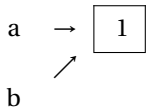
Une partie un peu plus difficile et plus spécifique au langage Python...

```
>>> a=1
>>> b=a
>>> a=2
>>> b
1
```

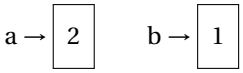
Cela semble tout à fait logique. Que se passe-t-il lorsqu'on valide `a=1` : un emplacement mémoire avec l'entier 1 est créé et *a* est un alias qui va pointer vers cet emplacement. On peut le représenter ainsi :



Après `b=a`, on se trouve dans la situation

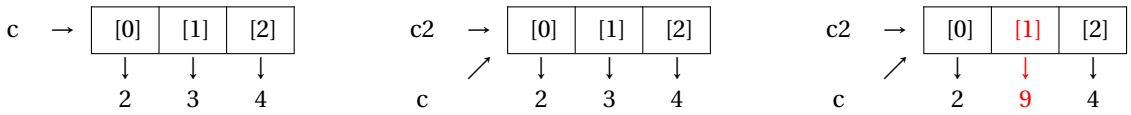


l'affectation finale `a=2` nous amène à la situation




```
>>> c=[2,3,4]
>>> c2=c
>>> c2[1]=9
>>> c
[2, 9, 4]
```

L'évolution des listes `c` et `c2` est représentée ainsi :



Les deux listes `c` et `c2` ont toujours le même identifiant car on n'a pas modifié la liste `c2` mais seulement le lien vers la valeur sur laquelle point le deuxième élément de la liste.

VII | Dictionnaires

Un dictionnaire est un ensemble constitué de couples « clé :valeur ». Il n'est pas ordonné. Les clés sont uniques et à chacune de ces clés est associée une valeur. Les clés doivent être des objets non mutables - en général on utilise des entiers, des chaînes de caractères ou des tuples, les valeurs peuvent être totalement quelconques.

DICTIONNAIRES : CRÉATION ET MANIPULATION

`dict()` ou `{}`	dictionnaire vide
`{c1:v1, c2:v2, ...}`	création d'un dictionnaire avec les clés c_i et les valeurs v_i
`d[cle]`	accès à la valeur associée à la clé `cle` (en lecture et écriture). Lors d'une affectation, la clé est créée si elle n'existe pas
`d.keys()`	ensemble des clés du dictionnaires (objet itérable)
`d.values()`	les valeurs du dictionnaires (objet itérable)
`d.items()`	les couples clé/valeurs du dictionnaires (objet itérable)
`len(d)`	taille du dictionnaire (nombre de clés)
`k in d` (ou `k not in d`)	teste si la clé `k` est présente (absente) dans le dictionnaire
`d.copy()`	réalise une copie superficielle du dictionnaire
`d.pop(cle)`	retire le couple clé/valeur du dictionnaire et renvoie la valeur
`d.popitem()`	retire et renvoie un couple clé/valeur du dictionnaire (le dernier créé)
`del d[cle]`	efface le couple clé/valeur associé à la clé

VII.1 | Comparaison listes et dictionnaires

L'accès aux éléments d'un dictionnaire est très efficace (utilisation d'une fonction de « hachage »). Les points à retenir (en simplifiant, notamment sur la fonction de hachage) et en notant `n` la taille de la liste/dictionnaire :

- une liste est intéressante lorsque l'ordre des éléments est important, ou que l'indexation par des entiers est importante - à l'opposé, une dictionnaire permet d'avoir un ensemble de clés totalement quelconque,
- le test d'appartenance est en $O(1)$ pour un dictionnaire, alors qu'il peut être en $O(n)$ pour une liste
- les temps d'accès à un élément donné est en $O(1)$ pour les deux (un peu plus rapide pour les listes),
- le temps de suppression d'un élément est en $O(1)$ pour un dictionnaire et en $O(n)$ pour une liste (sauf si c'est le dernier élément qu'on retire avec `pop()`)

```
>>> d = { 'a':3, 'b':5, 'c':4 }
>>> d['a']
3
```

```
>>> d['e']
KeyError
>>> 'b' in d
True
>>> d['e'] = 5 ; d['b']=3*d['e']
>>> d
{'a': 3, 'b': 15, 'c': 4, 'e': 5}
```

On peut avoir des clés qui sont des tuples, mais pas des listes (une clé doit être immuable)

```
>>> d = {} ; d[(0,0)]=1 ; d[(1,0)]=2 ; d[(2,3)] = 8 ; d
{(0, 0): 1, (1, 0): 2, (2, 3): 8}
>>> d[[2,1]] = 0
TypeError: unhashable type: 'list'
```

VIII | Modules

On peut importer des objets d'un module (un fichier d'instructions Python) de différentes façons et à différents endroits de la mémoire :

```
>>> import math
```

Le module `math` est chargé en mémoire et tout est placé dans « un espace de noms », c'est-à-dire (en simplifié) une zone de mémoire propre qui ne se mélange pas avec les données et variables déjà affectées. Pour accéder à un objet de cet espace de noms, on utilise son préfixe, ici le nom du module chargé :

```
>>> math.sin(0)
0
```

On peut l'importe sous un autre nom

```
>>> import math as m
>>> m.sin(0)
0
```

On peut importer un sous-module d'un module. On le fait fréquemment lorsqu'on utilise de très grosses bibliothèques comme Numpy/Matplotlib

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

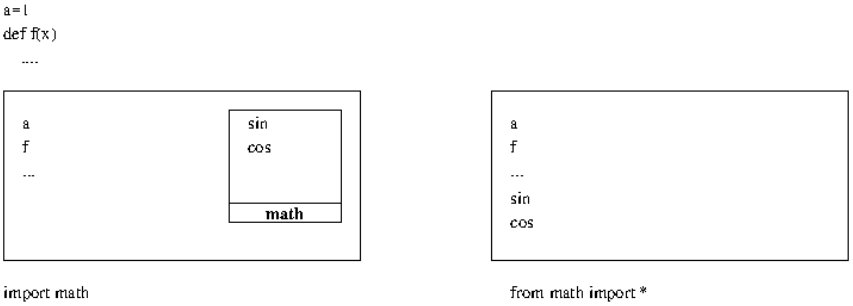
le sous-module `pyplot` de `matplotlib` est importé dans l'espace de noms « plt ».

Enfin, on peut directement importer des fonctions ou des modules complets dans l'espace de noms courant

```
>>> from math import *
# ou
# >>> from math import sin,cos,pi
# pour importer seulement ces fonctions
```

On a alors directement accès aux objets :

```
>>> sin(pi)
1.2246467991473532e-16
```



VIII.1 | Module math

- les constantes : `pi, e`
- quelques fonctions classiques : `sqrt, exp, log, log10`, ainsi que `ceil, floor, fabs, factorial` (pour un entier)
- les fonctions trigonométriques diverses : `sin, cos, t an, asin, acos, atan`
- les fonctions hyperboliques : `sinh, cosh, tanh, asinh, acosh, atanh`
- quelques fonctions spéciales : `gamma, lgamma` ($\ln\circ\Gamma$), `erf` (avec $\text{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$)
- quelques autres plus ou moins utilisées...

Il existe un module assez proche pour traiter le cas des fonctions d'une variable complexe (module `cmath`).

VIII.2 | Autres modules usuels

MODULES

numpy	pour le calcul numérique essentiellement basé sur le type tableau `array`
matplotlib	tout ce qui concerne l'affichage de graphiques avec plusieurs sous-modules (notamment `matplotlib.pyplot`)
deepcopy	permet la copie « profonde » de listes...
os	pour l'interaction avec le système
random	pour l'aléatoire (on utilisera plutôt `numpy.random`)
time	pour mesurer le temps (obtenir l'heure, le temps d'exécution d'une fonction)

IX | Fichiers

On utilise un fichier en l'ouvrant à partir de la fonction `open`. Cela crée alors un objet avec différentes méthodes. Pour avoir l'aide complète : `help(file)`. Il y a plusieurs paramètres possibles. Les plus standards sont les deux premiers : le nom du fichier et le type d'ouverture (en lecture, en écriture, en ajout...). Par défaut le type est « lecture » (`'r'`). Les autres types usuels sont `'w'` (write) pour l'ouverture en écriture (en vidant le fichier) et `'a'` (append) pour l'ajout à la fin du fichier (et création si le fichier n'existe pas).

Dans cette partie, on va uniquement s'intéresser aux fichiers texte.

IX.1 | Lecture et écriture

Une fois un objet fichier créé, on peut lire les données ou en écrire. À la fin on ferme le fichier :

```
>>> fichier=open("test.txt","w")
>>> fichier.write("Une chaîne dans le fichier\nUne deuxième ligne.")
46   # la méthode retourne le nombre de caractères écrits
>>> fichier.close()
```

Lorsqu'on regarde le contenu du fichier `test.txt`, on obtient (le caractère `\n` est un retour à la ligne).

```
Une chaîne dans le fichier
Une deuxième ligne.
```

On peut ensuite relire ce fichier :

```
>>> fichier=open("test.txt")
>>> a=fichier.read() # on lit l'intégralité du fichier
>>> type(a)
<class 'str'>
>>> a
'Une chaîne dans le fichier\nUne deuxième ligne.'
```

```
>>> print(a)
Une chaîne dans le fichier
Une deuxième ligne.
```

méthode	description
`read(nombre)`	lit les caractères du fichier, avec un nombre maximal si l'argument est donné. Chaîne vide lorsqu'il n'y a plus rien. Lit l'intégralité si nombre n'est pas précisé
`readline()`	lit une ligne complète du fichier (retourne une chaîne). Vide si à la fin
`readlines()`	lit les lignes du fichier dans une liste (de chaînes)
`write(chaine)`	écrit la chaîne dans le fichier
`writelines(lignes)`	écrit les lignes de la liste transmise (pas de retour à la ligne ajouté à la fin de chaque ligne)
`close()`	ferme le flux
`tell()`	indique la position courante dans le fichier

Un fichier texte ouvert en lecture possède également les propriétés d'un itérateur : on parcourt alors les lignes du fichier :

```
fichier=open("test.txt")
for ligne in fichier:   # on parcourt les lignes
    traitement...
```

Chapitre 2 | Divers

I | Commentaires, variables

Quelques conseils en vrac - la plupart ne sont pas des obligations, mais des conventions usuelles pour une lecture facilitée pour tous :

- choisir des noms de variables explicites :
  - → on peut évidemment prendre des noms simples pour des variables de compteur sans signification (`i,j,k...`)
  - → pour des variables plus importantes, on les écrira soit en minuscule : `droite`, `tableau` ..., soit sous forme capitalisée si le mot est composé : `TableauEntiers`, `BorneDroite` (ou éventuellement avec des underscores : `tableau_entiers`, `borne_droite`).
  - → les variables écrites en majuscule désigne en général des constantes, souvent définies au début de programme : `NOMBRE = 6` (le but est de fixer la valeur en début et de ne plus y toucher)
- **commenter ses programmes** pour tout ce qui n'est pas immédiat
- espacement après les symboles :
  - → = et == : des espaces autour

```
if x == 3:
    a = 2
else:
    a = 1
```

  - → deux-points, point-virgule : collés à gauche, un espace à droite

```
def test():
    if x>0: return(1)
    a = 1 ; c = 3
```

  - → parenthèses, crochets : pas d'espace avant ni à l'intérieur

```
>>> f(2)  # oui - et pas f (2) ni f( 2 )
>>> T[i]  # oui - et pas T [i] ni T[ i ]
```

  - → virgule : un espace après

```
>>> f(2, 3)
>>> a = (2, 3, 4)
>>> x, y = 2, 3
```

    sauf sans le cas particulier d'un tuple à un seul élément `a=(2,)`
  - → autres opérateurs `+,-,*,/...` : 0 ou 1 espace... pour que ça reste facile à lire (mais si on met un espace d'un coté, on en met un de l'autre) - ne pas en mettre trop non plus

```
a = 2*x+1
a = 2*x + 1
a = 2 * x + 1 # trop d'espaces
i = i + 1
i = i+1
i += 1 # mais pas i +=1 ou i+= 1
```

II | Assertions

L'un des moyens simples pour pallier ce défaut de contrôle des arguments est d'utiliser la fonction `assert`. Un exemple (plus ou moins pertinent sur les choix des assertions) pour illustrer :

```
def renverse(t,k):
    """..."""
    assert(type(t)==list), 'le premier argument doit être une liste'
    assert(len(t)>0), 'la liste doit être non vide'
    assert(k>=2), 'on doit inverser au moins 2 éléments'

    t2 = []                  # nouveau tableau
    n = min(k,len(t))        # nombre d'éléments à vraiment inverser (t trop petit)
    for i in range(n):
        t2.append(t[n-1-i])
    for i in range(n,len(t)):
        t2.append(t[i])
    return t2
```

On teste :

```
>>> renverse([],3)
...
AssertionError: le tableau doit être non vide
>>> renverse("abcdef",4)
...
AssertionError: le premier argument doit être une liste
>>> renverse([3,8,2,6,4,1,9,10],4)
[6, 2, 8, 3, 4, 1, 9, 10]
```

Remarque : Une autre utilisation des assertions est de pouvoir écrire un jeu de tests pour une fonction/un programme