INFORMATIQUE TRONC COMMUN

MPSI-PCSI-PTSI
MP-PC-PSI-PT

Nicolas Audfray
Jean-Loup Carré
Stéphane Legros
Vojislav Petrov
Julien Reichert
Marc Rezzouk

PARCOURS PRÉPAS

INFORMATIQUE TRONC COMMUN

MPSI-PCSI-PTSI
MP-PC-PSI-PT

EDISCIENCE

Création de couverture : Studio Dunod

11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-084100-4

Avant-propos

Cet ouvrage s'adresse aux élèves des classes préparatoires scientifiques aux grandes écoles (filières MPSI, PCSI, PTSI, MP, PC, PSI et PT). Il pourra également intéresser les étudiants d'autres filières (TSI, TPC) ainsi que les étudiants préparant le CAPES d'informatique.

Le livre est divisé en neuf chapitres, couvrant l'intégralité des programmes d'informatique des deux années de classes préparatoires, les six premiers chapitres contenant le programme de première année et les trois derniers celui de seconde année. Cet ouvrage est rédigé en utilisant le langage Python. Les chapitres et exercices sont numérotés à partir de zéro pour respecter les conventions de Python.

Chaque chapitre commence par une partie nommée *L'essentiel du cours*. On y présente les points les plus importants du cours à la manière de fiches. La première chose à faire est de connaître cette partie.

On trouve ensuite une partie nommée *Les méthodes à maîtriser*. Elle présente les méthodes en rapport avec le cours, illustrées d'un ou de plusieurs exemples et à savoir mettre en pratique.

La plupart des chapitres comprennent un *questionnaire à choix multiples* ou un *questionnaire vrai/faux*. Ceux-ci permettront d'identifier rapidement d'éventuelles lacunes.

Un large choix d'*exercices*, de niveaux variés, est ensuite présenté. Une correction est proposée pour chacun d'entre eux.

Les chapitres proposent ensuite des *Travaux pratiques*, dont la difficulté est progressive. Ils portent sur des thèmes très variés, englobant des problèmes issus de l'informatique, mais également des mathématiques, de la physique, de la chimie, des sciences de l'ingénieur et de la biologie. Progresser en informatique demande de la pratique, et le lecteur est invité à réaliser ces travaux sur machine, essayer des codes, déchiffrer les messages d'erreurs que pourrait lui envoyer la machine pour progresser dans sa maîtrise syntaxique du langage, effectuer des tests en cherchant davantage les limites de ses programmes qu'une validation superficielle avant de consulter le corrigé. Certaines corrections de ces Travaux pratiques sont incluses dans le présent ouvrage, et les autres se trouvent sur la page dédiée à celui-ci du site de Dunod (`https://www.dunod.com/ean/9782100841004`).

Nous avons utilisé certains pictogrammes tout au long de cet ouvrage :

 pour attirer l'attention du lecteur sur une remarque spécifique,

 pour attirer l'attention du lecteur sur des pièges potentiels,

 pour apporter au lecteur des précisions sur des points de syntaxe utiles dans le cadre de l'exemple traité mais non essentiels à retenir,

 pour indiquer une question ou un exercice assez difficile.

Des vidéos pour vous aider à réussir en prépa

Pour réussir vos concours, vous devrez mettre en œuvre des compétences disciplinaires ***(hard skills)***, mais aussi des ***soft skills***, ces compétences transversales qui vous permettront de tenir le bon rythme. La collection ***Parcours Prépas*** vous offre six vidéos pour vous préparer à réussir dès la première année et faire la différence le jour J par la maîtrise de votre énergie (physique, émotionnelle, mentale), par l'entretien de votre motivation et par vos méthodes de travail.

*Tout d'abord deux vidéos méthodologiques d'**Alexis Brès**. Professeur agrégé de physique-chimie en MP2I (lycée Hoche, Versailles), il est aussi correcteur et concepteur de sujets pour la banque du concours e3a-Polytech ; ancien correcteur du concours d'entrée aux ENS. Auteur de L'Oral de physique aux concours des ENS et de Polytechnique (Dunod).*

http://dunod.link/jvy7mqd

Vidéo 1 : Apprendre à apprendre Comment mobiliser efficacement son cours ?

Comment apprendre un cours ? Comment savoir si on l'a vraiment compris ? Comment le mobiliser dans les TD et dans les épreuves ? Comment créer du lien entre les connaissances pour se forger une intuition de la solution et gagner un temps précieux ? Autant de questions-réponses abordées dans cette vidéo. Une méthodologie particulièrement adaptée à l'apprentissage des cours de physique, de mathématiques ou de sciences industrielles.

http://dunod.link/z0psk69

Vidéo 2 : Écrit, oral : aborder sereinement la résolution d'un problème

Si les exigences d'un sujet d'écrit et d'un oral peuvent sembler assez différentes, il existe des techniques communes pour aborder ces épreuves sans stress. Cette vidéo fournit :

- des techniques pour apprivoiser la résolution d'un problème de physique : modalités de décryptage du sujet et de mobilisation du cours ;
- des recommandations sur le fond et la forme pour gagner la confiance des correcteurs ;
- des tactiques cohérentes pour gagner des points ;
- des points de vigilance concernant la préparation des khôlles et des oraux.

*Ensuite quatre vidéos « soft skills » pour aborder la prépa comme le ferait un sportif de haut niveau. Ces vidéos ont été conçues par **Stéphane Fassetta**, fondateur de Syprium, coach professionnel, préparateur mental de sportifs de haut niveau, professeur d'aïkido. Auteur de Nos 8 profils énergétiques (InterÉditions).*

http://dunod.link/80x2gwu

Vidéo 3 : Les cinq piliers de l'énergie, ou comment réussir le marathon de la prépa ?

La prépa, c'est un peu comme le sport de haut niveau : plus le temps passe, plus le niveau ou les contraintes augmentent. Maîtriser son énergie, c'est donc faire un usage optimum

de ses ressources pour tenir le rythme des deux années, s'adapter à la diversité des situations et réussir ses épreuves. Cette vidéo présente les dimensions de notre énergie et les cinq piliers pour l'entretenir. La capacité à se ressourcer sur ces cinq piliers est une compétence à développer dès votre arrivée en prépa.

http://dunod.link/sicy8u3

Vidéo 4 : Gérer efficacement son temps en prépa

En prépa, on manque toujours de temps. L'enjeu est donc de gérer efficacement cette ressource pour atteindre les objectifs de vos différentes échéances.

Cette vidéo fournit des repères pour :

- trouver sa propre organisation personnelle : techniques de planification, objectifs SMART... ;
- développer sa capacité d'attention, essentielle à la compréhension, à la mémorisation, à la gestion de la charge mentale et à votre avancement ;
- connaître ses propres biorythmes pour un apprentissage efficient, en capitalisant sur les acquis de la chronobiologie.

http://dunod.link/p5maym6

Vidéo 5 : Gérer son stress et développer la confiance en soi pour les concours

Comme dans le sport de haut niveau, la préparation d'un concours soumet votre énergie à rude épreuve. Si une certaine pression est stimulante pour doper ses performances, l'installation dans un stress chronique compromet à la fois votre santé et vos chances de réussite.

Cette vidéo permet :

- d'identifier les sources externes et internes de son propre stress ;
- de comprendre le rôle du stress comme mécanisme naturel d'adaptation de l'organisme face à une situation déstabilisante et/ou à fort enjeu ;
- d'apprendre à reconnaître certains symptômes physiques, émotionnels ou cognitifs du stress pour prévenir l'épuisement ;
- de connaître les possibilités de régulation physique et mentale du stress ;
- d'entretenir passionnément sa motivation pour préserver durablement la confiance en soi, quelles que soient vos contre-performances.

http://dunod.link/vncd3c5

Vidéo 6 : Techniques respiratoires et de préparation mentale pour préparer les concours

La capacité à se relaxer ou à récupérer quand il le faut est essentielle pour tenir le rythme de préparation d'un concours.

Grâce à cette vidéo :

- vous saurez mettre en œuvre différentes techniques respiratoires adaptées à la récupération et à la dynamisation ;
- vous disposerez de deux techniques de préparation mentale pour conserver un état d'esprit positif, limiter votre niveau de stress et améliorer vos capacités d'attention.

Table des matières

CHAPITRE 0

Bases de programmation Python

Dans ce chapitre nous donnons les bases de programmation en langage Python, censées être un pré-requis pour le programme d'informatique de tronc commun de CPGE.

L'essentiel du cours

■ 0 Introduction

Python est un langage de programmation interprété, en opposition aux langage compilés comme le langage C par exemple. C'est-à-dire qu'il est exécuté ligne par ligne. Les éventuelles erreurs de programmation ne sont alors détectées qu'à l'exécution de la ligne de code problématique.
Pour qu'un code Python soit valide (pour la machine) et lisible (par nous), il y a un certain nombre de règles de présentation à suivre :

(0) Indentation (1 tabulation = 4 espaces) : pour indiquer les blocs du code (utiliser `\` pour un retour à la ligne sans changer de bloc). L'indentation est capitale en Python, il n'y a pas de mot-clé ou de parenthèses pour structurer les blocs (`if…endif` en scilab par exemple).

(1) Espaces : il est conseillé de mettre un espace autour de `=, !=, ==, <, >,` *etc.* et éventuellement autour des opérateurs arithmétiques. On place également un espace après une virgule (et pas avant) dans un tuple ou une liste : `(1, 2, 3)` et `[a, b, c]`. Il n'y a pas d'espace entre une fonction et la parenthèse : `f(x)` et pas `f (x)`. Cela permet une lecture plus aisée mais ne remet pas en cause l'exécution du code.

(2) Docstring (chaîne de documentation) : elle permet de préciser ce que fait la fonction, on entoure le texte sur plusieurs ligne par `"""` et `"""`. On ne peut que recommander de documenter correctement les fonctions (voir chapitre 2), et plus généralement le code écrit, pour une compréhension plus rapide par une autre personne ou tout simplement pour comprendre son propre code quelques temps plus tard.

(3) Commentaires : placés judicieusement, ils permettent de comprendre le code.

(4) Noms : on choisira des noms de variables ou de fonctions suffisamment clairs pour qu'à la première lecture du code, on comprenne instantanément leur rôle. Ceci permettra par exemple de ne pas surcharger en commentaires.

■ 1 Expressions

Définition

Une **expression** est une formule qui renvoie un résultat.

Les opérations usuelles (plus, moins, fois, divisé, puissance, notées `+`, `-`, `*`, `/` et `**`) ainsi que des fonctions (par exemple `abs`) peuvent être utilisées dans les expressions. Exemples :

```
>>> (5 * 9) / 5 + 3**2
18.0

>>> abs(abs(-10)**3 + (-3)**7)
1187
```

Division euclidienne généralisée

Étant donné un réel $b > 0$, pour tout réel a, il existe deux nombres $q \in \mathbb{Z}$ et $r \in [0, b[$ tels que $a = bq + r$. En Python, le quotient q est calculé avec l'opération `//` et le reste r avec `%`.

■ 2 Types de base

Les types de base manipulés en Python sont :

- `int` : les entiers relatifs ;
- `float` : les nombres à virgule aussi appelés nombres flottants ;
- `complex` : les complexes, représentés par une paire de `float` ;
- `bool` : les booléens `True` et `False` ;
- `NoneType` : le type de `None`.

Le calcul est exact sauf pour les types `float` et `complex`. Pour ces deux types, les calculs peuvent mener à des erreurs d'arrondis.

Les opérations de comparaison `==`, `!=`, `<`, `<=`, `>`, `>=` renvoient un booléen. Les opérations usuelles sur les booléens (`and`, `or` et `not`) sont disponibles en Python.

Astuce : `2 < 7 < 5` est une abréviation de `2 < 7 and 7 < 5`.

La valeur `None` est renvoyée par les fonctions n'ayant aucun résultat. Par défaut, il n'est pas affiché dans la console. Par exemple, l'expression `print(2)` affiche `2` puis renvoie `None`.

■ 3 Instructions

Définition

Une instruction est un ordre donné à l'ordinateur, une instruction ne donne pas de valeur.

L'affectation ou assignation est une instruction qui associe une valeur à une variable. Les variables affectées peuvent ensuite être utilisées dans des expressions. Exemple :

```
>>> x = 3

>>> (x + 1) / 2
2.0
```

Ce qui est à gauche et ce qui est à droite du `=` n'ont pas le même rôle.
De plus, `=` et `==` ne doivent pas être confondus.

Le branchement conditionnel. Exemple :

```
if x > 0:
    print(1)
```

Ce code affiche `1` si `x` est strictement positif, et ne fait rien sinon.
Le corps du `if` (ici `print(1)`) est indenté, il n'est exécuté que si la condition du `if` (ici `x>0`) est évaluée à `True`. Il est aussi possible d'ajouter un `else`. Le corps du `else` ne sera exécuté que si la condition du `if` est évaluée à `False`. Le corps du `if` ou du `else` peut contenir plusieurs lignes. Exemple :

```
if x > 0:
    print(1)
else:
    x = 0
    print(2)
```

Définition

Les **boucles** permettent de répéter une suite d'instructions. Il en existe deux types : les boucles `while` et les boucles `for`.

La boucle `while` répète une suite d'instructions tant qu'une condition est vraie.

```
y = 123
while y != 0:
    print(y)
    y = y // 10
```

Dans ce code, on affiche `y` puis on le quotiente par `10` tant qu'on n'a pas obtenu zéro. Le code va afficher `123` puis `12` puis `1`.
La boucle `for` permet de répéter une suite d'instructions un nombre prédéterminé de fois.

```
for k in range(5):
    print(k)
```

Le code précédent affiche tous les entiers de 0 inclus à 5 exclu.

■ 4 Séquence

Une séquence est une suite finie de valeurs. Il existe plusieurs types de séquences en Python, dont voici quelques exemples :

- les chaînes de caractères : `"azerty"` ou `'azerty'` ou `'''azerty'''` ou `"""azerty"""` ;
- les tuples : `(1, 7, 58)` ;
- les listes : `[1, 7, 58]`.

Une chaîne de caractères ne contient que des caractères. Un tuple ou une liste peut contenir des données de n'importe quel type (voire des données de types différents). Les opérations suivantes sont communes à toutes les séquences `s` et t :

- lecture de l'élément numéro `k`, `s[k]` (le premier élément porte le numéro 0) ;

- extraction de la sous-séquence de l'élément numéro `i` inclus à l'élément numéro `j` exclu, `s[i:j]` ;
- concaténation de deux séquences, `s + t` ;
- concaténation de `n` copies d'une même séquence, `s * n` ou `n * s`.

Lors de la lecture d'un élément, il est possible d'utiliser des numéros négatifs, `-1` pour le dernier élément, `-2` pour l'avant-dernier…
Lors de l'extraction d'une sous-séquence, il est possible d'ajouter un pas `k`, `s[i:j:k]`. Omettre `i` signifie commencer la sous-séquence au début, omettre `j` signifie finir la sous-séquence à la fin.

Au contraire des chaînes de caractères et des tuples, les listes sont modifiables, et disposent d'opérations supplémentaires :

- modification de l'élément numéro `k` de la liste : `s[k] = x` ;
- ajout d'un élément `x` en fin de liste : `s.append(x)` ;
- suppression d'un élément ou de toute une sous-séquence : `del s[k]` ou `del s[i:j:k]` ;
- suppression du dernier élément (ou du numéro `k`) en renvoyant sa valeur : `s.pop()` ou `s.pop(k)`.

Les opérations `s[i:j:k] = t` et `s.extend(t)` peuvent être utiles à connaître.

L'expression `s + t` ne modifie pas `s` mais crée une nouvelle liste. L'instruction `s += t` modifie `s` et lui ajoute `t` à la fin.

L'affectation `M = L` ne copie pas la séquence `M` ! Elle crée une nouvelle variable, une sorte d'alias qui pointe vers la même adresse : à bien retenir pour les listes (et les objets mutables en général). C'est l'adresse de `L` qui est utilisée et pas sa valeur. Ceci peut être gênant puisque toute modification d'une liste est répercutée sur l'autre comme sur l'exemple qui suit.

```
>>> L = [5, 2, 3] # On crée une liste L
>>> M = L # M est un alias de la liste L
>>> id(L), id(M) # On regarde l'adresse d'enregistrement de L et M
(53566400, 53566400) # Les 2 listes ont la même adresse mémoire
>>> L[0] = 8 # On modifie un élément de L
>>> L
[8, 2, 3]
>>> M
[8, 2, 3] # La modification est répercutée sur M
>>> M[0] = 9 # On modifie un élément de M
>>> L
[9, 2, 3] #  La modification est répercutée sur L
>>> M
[9, 2, 3]
```

La bibliothèque `copy` permet de copier des séquences, en profondeur si nécessaire, afin de pallier les problèmes montrés ci-dessus.

Pour un exemple d'utilisation de cette notion, voir le TP **1.0** p. 74.

■ 5 Bibliothèques (modules) Python

♦ 5.0 Chargement des bibliothèques

Définition

Les **modules** (ou **bibliothèques** ou encore **librairies**) regroupent des fonctions et constantes utilisables dès lors que le module a été importé dans le code à exécuter, ainsi que des types de données particuliers.

L'importation se fait avec l'instruction `import` qui peut être utilisée de différentes façons :

```
import math # bibliothèque des fonctions et variables mathématiques
import numpy as np # Bibliothèque de calcul numérique
from matplotlib.pyplot import * # Bibliothèque de tracé graphique
from random import random, randint # Bibliothèque de fonctions pseudo aléatoirs

# On pourra ainsi à partir de ces différents chargements de bibliothèques
# utiliser les fonctions et variables suivantes :

math.tan(math.pi / 4)
np.log(math.e**3) # logarithme népérien et exponentielle
plot(X, Y) # tracé de Y en ordonnée en fonction de X en abscisse
randint(1, 10) # tire aléatoirement un entier dans l'intervalle [1, 10]
```

Certaines fonctions sont définies dans plusieurs modules. Pour maîtriser l'origine de l'utilisation d'une fonction, il est préférable de l'appeler en spécifiant sa bibliothèque. Par exemple la fonction sinus est définie dans les modules `numpy` et `math`. On l'appellera donc de la manière suivante :

```
import math
import numpy as np  # alias très classiquement utilisé

math.sin(x)  # utilisation de la fonction sinus du module math
np.sin(x)  # utilisation de la fonction sinus du module numpy plus complet : sin(np.array([...])) possible
```

Si on veut éviter ce genre de problèmes on peut importer le module `numpy` en entier par exemple (`from numpy import *`) et uniquement les fonctions utiles de l'autre bibliothèque.

♦ 5.1 Module de tracé graphique `matplotlib.pyplot`

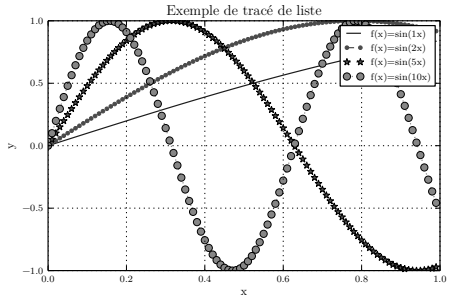
Le langage Python permet de tracer des courbes via le module `matplotlib` et plus particulièrement de son sous-module `pyplot` que l'on peut importer à l'aide de la commande

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

Ce sous-module et en particulier sa fonction principale `plot` utilisent une syntaxe proche de celle utilisée par les logiciels de calcul numérique courant MATLAB® et Scilab® utilisés en SII.

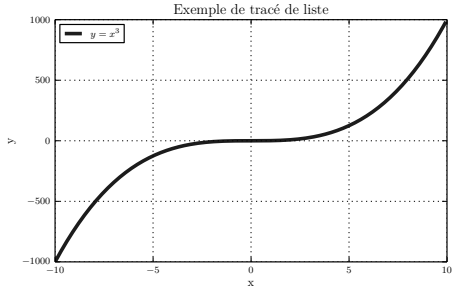
Un petit exemple en utilisant des listes pour décrire les styles de lignes et de marqueurs ainsi que la légende. On peut également faire la même chose pour les couleurs par exemple.

```
import matplotlib.pyplot as plt
import numpy as np
style_ligne = ['solid', 'dashed', 'dashdot', 'dotted']
style_marker = [' ', '.', '*', 'o']
k = [1, 2, 5, 10]
for i in range(len(k)):
    x = []
    y = []
    for j in range(100):
        x.append(j / 100)
        y.append(np.sin(k[i] * x[j]))
    plt.plot(x, y, linestyle=style_ligne[i],
             marker=style_marker[i], label='f(x)=sin('
             + str(k[i]) + 'x)')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Exemple de tracé de liste')
plt.legend()
plt.grid()
plt.show()
```



Il est possible de tracer des figures à partir de listes de valeurs numériques de type `float` :

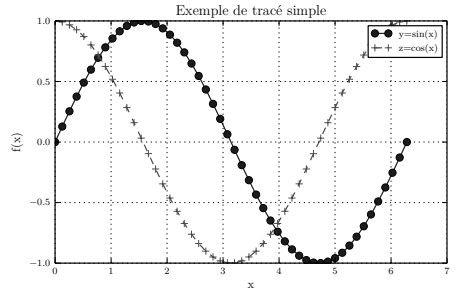
```
import matplotlib.pyplot as plt
x = []
y = []
for i in range(-100, 101):
    x.append(i / 10)
    y.append((i / 10)**3)
plt.plot(x, y, label='$y=x^3$', linewidth=3)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Exemple de tracé de liste')
plt.legend(loc=2)
plt.grid()
plt.show()
```



En réalité, la fonction `plt.plot` convertit le cas échéant ses deux premières entrées en tableau `array`.
On utilise très souvent la fonction `np.linspace` pour générer des subdivisions régulières.

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

x = np.linspace(0, 2 * np.pi, 51)
y = np.sin(x)
z = np.cos(x)
plt.plot(x, y, '-o', label='y=sin(x)')
plt.plot(x, z, '--+', label='z=cos(x)')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Exemple de tracé simple')
plt.legend()
plt.grid()
plt.savefig('trace_simple.png')
```



Nous n'avons ici que présenté des tracés de courbes élémentaires, mais `matplotlib` est très riche en possibilités graphiques (tracé d'histogrammes, courbes polaires, diagrammes log-log, cartographies, etc.). Il est impossible de traiter tous les cas ici et nous vous encourageons à consulter la documentation de `matplotlib` en fonction de vos besoins.

■ 6 Fonctions et variables

♦ 6.0 Définir sa propre fonction

L'instruction `def` permet de définir une fonction. Exemple :

```
def ma_fonction(arg1, arg2):
    s = arg1 + arg2
    p = arg1 * arg2
    return p % s
```

Une fonction doit avoir :

- zéro, un ou plusieurs arguments (ici `arg1` et `arg2`),
- des instructions à exécuter,
- une valeur de retour : la valeur de l'expression après le `return`.

L'instruction `return` arrête l'exécution de la fonction, même si elle est exécutée au milieu d'une boucle.
L'absence de `return` sous-entend un `return None` à la fin de la fonction.
L'instruction `return None` peut être abrégée en `return`.

L'expression `lambda x : x+2` renvoie la fonction qui à `x` associe `x+2`.

♦ 6.1 Variables locales, variables globales

Chaque instruction de la forme `nom = ...` ou `for nom in ...`, présente dans le code d'une fonction, crée, lors de l'appel de la fonction, une variable locale nommée `nom`. Une telle variable n'est définie que pendant l'exécution de la fonction et peut tout à fait posséder le même nom qu'une variable qui existait avant l'appel, sans qu'il y ait de confusion entre ces deux variables.

On observe également ce comportement si `nom` est le nom d'un des arguments de la fonction ; ainsi, Python accepte le code ci dessous, qui calcule le n-ième nombre harmonique $H_n = \sum_{k=1}^{n} \frac{1}{k}$:

```
def H(n):
    s = 0
    while n != 0:
        s += 1 / n
        n = n - 1

    return(s)
```

```
>>> n = 100000

>>> H(n)
12.090146129863408

>>> n
100000
```

Si un nom de variable apparaît ailleurs dans le code de la fonction, le compilateur reconnait une variable globale, qui est donc une variable utilisée dans le corps d'une fonction mais non définie dans ce code. Ainsi, les différentes constantes utilisées dans la modélisation d'un problème sont en général des variables initialisées en début de programme, puis utilisées comme variables globales des différentes fonctions. Cependant, on peut parfois souhaiter qu'une fonction modifie une variable globale `nom`. Comme une instruction `nom = ...` crée automatiquement une variable locale `nom`, il faut préciser dans la définition de la fonction que la variable `nom` est globale, ce qui se fait en ajoutant la ligne `global nom` au début du code de la fonction. On trouvera des exemples d'utilisations de variables globales dans le TP 0.0.

La distinction entre variables locales et globales s'applique également aux variables d'une sous-fonction : une variable locale d'une fonction peut être déclarée comme variable « globale » d'une de ses sous-fonctions. On utilisera alors le mot-clé `nonlocal` au lieu de `global`.

■ 7 Fichiers

Le chemin d'un fichier peut être défini à partir du répertoire courant (chemin relatif), ou à partir de la racine de l'ordinateur (chemin absolu). La fonction `getcwd` du module `os` renvoie le chemin absolu du répertoire courant. Il est également possible de modifier le répertoire courant, en utilisant la fonction `chdir` du module `os` (les exemples qui suivent correspondent à deux systèmes d'exploitation différents) :

```
>>> import os

>>> os.getcwd()
'/Users/OrdiFixe'

>>> os.chdir("Doc/Python")

>>> os.getcwd()
'/Users/OrdiFixe/Doc/Python'
```

```
>>> import os

>>> os.getcwd()
'C:/Users/OrdiFixe'

>>> os.chdir("Doc/Python")

>>> os.getcwd()
'C:\\/OrdiFixe/Doc/Python'
```

Pour ouvrir un fichier, on utilise la fonction `open` en lui indiquant le chemin (relatif ou absolu) du fichier. Nous utiliserons ici les trois options `"w"`, `"r"` et `"a"`, selon que l'on veut écrire dans le fichier, lire le fichier ou ajouter du texte au fichier. L'instruction

```
monfichier = open("exemple.txt", option)
```

crée un flux nommé `monfichier` qui va permettre, selon l'option choisie, d'écrire ou de lire dans le fichier `exemple.txt` du répertoire courant.
L'écriture et la lecture se font ensuite en utilisant les instructions :

```
monfichier.write("chaine de caractere")  # pour écrire à la suite du fichier

ligne = monfichier.readline()  # pour lire la ligne suivante du fichier

texte = monfichier.read()  # pour lire la totalité (restante) du fichier

lignes = monfichier.readlines() # pour obtenir la liste des lignes restantes dans le fichier

morceau = monfichier.read(n) # pour lire les n caractères qui suivent dans le fichier
```

Une fois le travail à effectuer terminé, il ne faut pas oublier de fermer le fichier, à l'aide de la méthode `close` :

```
monfichier.close()
```

C'est à ce moment qu'un fichier ouvert en écriture est physiquement créé. Si un fichier portant le même nom est déjà présent à l'endroit choisi, ce dernier sera écrasé sans avertissement.

L'encodage du fichier (ASCII, UTF8 ou encodage plus exotique) peut être précisé en argument optionnel de la fonction `open` pour que les caractères accentués (ou cyrilliques ou arabes) soient correctement traités.

Le fonctionnement de ces fonctions est explicité dans l'exemple suivant : on crée un fichier contenant quelques lignes de texte (tout d'abord en mode `"w"` puis en mode `"a"`, puis on ouvre ce fichier pour vérifier son contenu. On rappelle que le caractère « passage à la ligne » est `\n`.

```
>>> monfichier = open("Haïku.txt", "w",encoding='utf-8')
>>> monfichier.write(Un vieil étang et\nUne grenouille qui plonge,\n")
45 # nombre de caractères écrits
>>> monfichier.close()
>>> monfichier = open("Haïku.txt", "a")
>>> monfichier.write("Le bruit de l'eau.\n")
19 # nombre de caractères écrits
>>> monfichier.close()
>>> monfichier = open("Haïku.txt", "r",encoding='utf-8')
>>> monfichier.read(5)
'Un vi'
>>> monfichier.readline()
'eil étang et\n'
>>> monfichier.readlines()
['Une grenouille qui plonge,\n', 'Le bruit de l'eau.\n']
>>> monfichier.close()
```

Il est également possible d'utiliser un fichier ouvert en lecture comme un itérateur, l'itération se faisant alors sur les lignes du fichier :

```
>>> for L in open("Haïku.txt", "r",encoding='utf-8'): print(L)
Un vieil étang et
Une grenouille qui plonge,
Le bruit de l'eau.
```

■ 8 Hasard

Les bibliothèques `random` et `numpy.random` permettent de générer des nombres aléatoires selon diverses lois (uniforme, binomiale, etc.). Elles utilisent toutes les deux Mersenne Twister (un PRNG). Ces bibliothèques sont conçues pour le calcul et pas pour le chiffrement.
Certaines fonctions sont communes à ces bibliothèques, comme `random()` (qui tire un flottant au hasard dans $[0, 1[$) ou `choice(L)` (qui tire au hasard un élément de `L`) ou `shuffle(L)` (qui mélange [1] `L`).

La fonction `randint(a, b)` tire un entier aléatoire (selon une loi uniforme) compris entre `a` inclus et `b` inclus pour la bibliothèque `random`. Mais elle tire un entier aléatoire compris entre `a` inclus et `b` **exclu** pour la bibliothèque `numpy.random`.

1. Le mélange « épuise » rapidement les générateurs pseudo-aléatoires (mais pas les TRNG). Pour une discussion sur ce sujet, voir la section 3.4.2 (Random Sampling and Shuffling) de [**Knu13**].

La bibliothèque `numpy.random` permet de simuler toutes les lois usuelles (voir la documentation de `numpy`).

```
import numpy.random as rd
# Simule une loi binomiale, somme de n Bernoulli indépendantes de paramètres $p=0.2$.
B = rd.binomial(10, 0.2)
# Simule une loi normale d'espérance (moyenne) 5 et d'écart type 1.
N = rd.normal(5, 1)
P = rd.poisson(7)  # Simule une loi de Poisson de paramètre 7.
```

Les bibliothèques `secret` et `os` (fonctions `getrandom` et `urandom`) génèrent du hasard de meilleure qualité utilisable, par exemple, pour de la cryptographie.

Les méthodes à maîtriser

Méthode 0.0 : Parcourir une liste

Lorsqu'on veut parcourir une liste (ou une chaîne de caractères, ou un tuple) on se demande : « *Est-il utile d'accéder aux indices de cette liste ?* »

- Si oui, on choisit d'itérer la liste par indices : `for i in range(len(L)):`
- Si non, on choisit d'itérer la liste par éléments : `for x in L:`

Lorsqu'on souhaite utiliser simultanément deux éléments successifs d'une liste, le parcours par indices est préférable. On doit faire attention à ne pas accéder à des éléments de la liste qui n'existeraient pas, en s'arrêtant avant le dernier indice le cas échéant.

Exemples d'application

- Calculer la somme des termes d'une liste `L` : l'accès aux indices est inutile, on choisit le parcours `for x in L:`
- Calculer l'espérance [1] d'une variable aléatoire X à valeurs dans $[\![0, n-1]\!]$ si la liste `L` contient les probabilités des évènements $X = i$ (`L[i]` contient $\mathbb{P}(X = i)$) : l'accès aux indices est indispensable, on choisit le parcours `for i in range(len(L)):`
- Tester si une liste L est triée dans l'ordre croissant : on a besoin d'accéder simultanément à un élément et à son successeur lors de notre parcours. On choisit un parcours par indices et on s'arrête à l'avant-dernier élément : `for i in range(len(L)-1):`

Méthode 0.1 : Créer une liste dont on connaît une expression du terme général

Une première solution est d'utiliser un simple `for`.

```
L = []
for k in range(n):
    L.append(f(k))
```

Une solution alternative consiste à utiliser une **liste par compréhension**.

```
L = [f(k) for k in range(n)]
```

Les deux codes font la même chose.

Exemple d'application

`[k**2 for k in range(5)]` renvoie `[0, 1, 4, 9, 16]`.

1. L'espérance $\mathbb{E}(X)$ d'une variable aléatoire X à valeurs dans $[\![0, n-1]\!]$ est définie par $\mathbb{E}(X) = \sum_{i=0}^{i=n-1} i\,\mathbb{P}(X = x_i)$ où $\mathbb{P}(X = i)$ désigne la probabilité que X vaille i.

Méthode 0.2 : Choisir entre une boucle while et une boucle for

Lorsque le nombre d'itérations est connu à l'avance ou lorqu'on connait la suite des valeurs à parcourir, on préfère une boucle `for`.
Lorsque le nombre d'itérations est inconnu, on choisit (par dépit !) une boucle while.

Avec `return`, il est possible d'interrompre une fonction au milieu d'un `for`. Le `for` est donc aussi adapté au cas où on connait un majorant du nombre d'itérations.

Exemples d'application

- Pour trouver l'ensemble des diviseurs d'un entier, on utilisera une boucle `for` : la borne supérieure à tester est connue (n, ou, en rusant un peu, $\sqrt{n}$).
- Pour connaître le plus petit entier n tel que la suite récurrente $u_{n+1} = u_n^2 + 1$ dépasse 10^7 pour un $u_0 > 0$ donné on utilisera une boucle `while`.
- Pour tester si un élément est présent dans une liste, une boucle `while` devrait a priori être envisagée, étant donné qu'on interrompt le parcours de cette liste dès lors qu'on trouve l'élément ou lorsqu'on a épuisé tous les éléments. Néanmoins, une boucle `for` peut aussi être utilisée dans le cadre d'une fonction, le `return` ayant comme effet bénéfique d'interrompre le parcours de la liste en cas de succès.

Méthode 0.3 : Écrire une boucle while

Pour écrire une boucle `while`, on prend gare à :

- bien définir la condition de poursuite de la boucle `while`. Il arrive que la condition d'arrêt soit plus simple à formuler, il est alors aisé de formuler la condition de poursuite de la boucle en écrivant `not(condition arrêt)` ;
- faire en sorte que la boucle finisse, en n'oubliant pas de modifier la variable sur laquelle porte la condition d'arrêt ;
- réfléchir si les variables en sortie de la boucle sont bien celles qu'on souhaitait avoir, et éventuellement les modifier, ou changer le moment où on modifie la variable sur laquelle porte la condition d'arrêt.

Exemple d'application

Étant donnée une liste $L = [p_0, \ldots, p_{n-1}]$ d'entiers naturels non nuls et un entier naturel N, calculer le plus petit indice k (s'il existe) tel que $N \leqslant \sum_{i=0}^{k} p_i$.

Par convention, k sera égal à -1 si $N > \sum_{i=0}^{n-1} p_i$.

```
def calcul_indice(L, N):
    n = len(L)
    S, k = L[0], 0 # S = L[0] et k=0
    while( S < N and k + 1 < n): # la seconde condition assure l'existence de L[k] à la ligne 7
        k += 1
        S += L[k] # on veut que S = L[0]+...+L[k]
    if S < N: # on est arrivé au bout de la liste sans dépasser N
        return -1
    else: return k # La somme a dépassé N à l'instant k
```

Méthode 0.4 : Calculer une somme

Pour calculer une somme, on :
- définit une variable `s` que l'on initialise à zéro ; cette variable jouera le rôle d'accumulateur ;
- choisit un parcours par élément ou par indice ;
- écrit le corps de boucle, l'instruction `s = s + ...` étant répétée ;
- effectue un éventuel traitement de s en sortie de boucle.

Exemple d'application

Calculer $\frac{1}{n}\sum_{i=1}^{n}\frac{1}{i^2}$.

```
def ma_somme(n):
    s = 0
    for i in range(1, n + 1):
        s = s + 1 / (i**2)
    return s / n
```

Méthode 0.5 : Tester ses fonctions ; débugger

Lorsqu'on écrit un programme (ou qu'on passe un oral de concours !) il est essentiel de tester – dans la mesure du possible – ses fonctions au fur et à mesure.
Pour ce faire, on appelle depuis la console les fonctions à tester avec des arguments simples et des retours que l'on peut facilement prévoir à la main.
Il ne faut pas hésiter à lancer plusieurs tests, en n'ommettant pas de tester des cas limites (comme une liste vide passée en argument).
Dans le cas où la fonction n'a pas le comportement attendu, si le premier temps – indispensable – de réflexion ne permet pas de trouver l'erreur, on peut soit faire appel aux fonctions de débuggage de l'éditeur employé, soit débugger la fonction à la main en traçant les valeurs des variables critiques, en ajoutant des affichages à l'écran à l'aide de la fonction `print`.

Exemple d'application

Tester si une chaîne de caractères contient la lettre 'e' ou la lettre 'E'.

Un élève produit le code suivant :

```
def test(s):
    for i in range(len(s)):
        if i == 'e' or i == 'E':
            return True
        else:
            return False
```

Une bonne première batterie de tests consiste à vérifier cette fonction avec quelques chaînes simples, e.g., `'info'`, `'ETE'`, `'the'`. La fonction renvoie `False` tout le temps. En plaçant un `print(i)` juste avant le test, on se rend compte qu'on itère sur des entiers et non des caractères. Ceci permet de corriger le premier problème.

La même batterie de tests, sur la fonction corrigée, révèle que certaines chaînes ne sont pas correctement traitées. On peut alors conclure quant à la seconde erreur commise ici (le `else`) et rectifier son programme :

```
def test(s):
    for i in range(len(s)):
        if s[i] == 'e' or s[i] == 'E':
            return True
    return False
```

Méthode 0.6 : Tracer la courbe représentative d'une fonction f

Pour tracer la courbe représentative d'une fonction f, on :

(0) importe les modules nécessaires à ce travail :
`import numpy as np`
`import matplotlib.pyplot as plt`

(1) détermine l'intervalle $I = [a, b]$ sur lequel il est pertinent de tracer la courbe représentative de la fonction f ;

(2) discrétise l'intervalle I en le découpant en sous-intervalles de même longueur h.
Cela peut par exemple se faire à l'aide de la fonction `arange` du module `numpy` :
`T = np.arange(a, b, h)`

(3) calcule la liste des images des points de `T` par la fonction f. Cela peut se faire à l'aide d'une construction de liste par compréhension :
`Y = [f(t) for t in T]`

(4) on représente finalement la courbe :
`plt.plot(T, Y)`
`plt.show()`

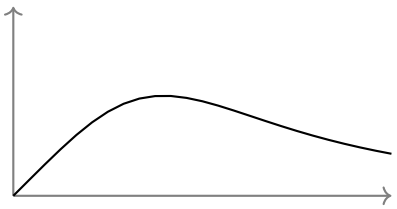
Exemple d'application

Tracer sur $[0, 2]$ la courbe représentative de $f(x) = \dfrac{x}{1 + x^3}$.

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

def f(x):
    return x / (1 + x**3)

T = np.linspace(0, 2, 201)
Y = [f(t) for t in T]
plt.plot(T, Y)
plt.show()
```



Méthode 0.7 : Tracer une courbe paramétrée

Pour tracer une courbe paramétrée $(x(t), y(t))$, on :

(0) importe les modules nécessaires à ce travail :
```
import numpy as np
import matplotlib.pyplot as plt
```
(1) détermine l'intervalle $I = [a, b]$ du paramètre t sur lequel il est pertinent de tracer la courbe paramétrée. Lorsque les fonctions x et y sont toutes deux 2π-périodiques, l'intervalle $[0, 2\pi]$ convient notamment.

(2) discrétise l'intervalle I en le découpant en sous-intervalles de même longueur h.
Cela peut par exemple se faire à l'aide de la fonction `arange` du module `numpy` :
```
T = np.arange(a, b, h)
```
(3) calcule la liste des images des points de `T` par les fonctions x et y. Cela peut se faire à l'aide d'une construction de liste par compréhension :
```
X = [x(t) for t in T]
Y = [y(t) for t in T]
```
(4) on représente finalement la courbe :
```
plt.plot(X, Y)
plt.show()
```

Exemple d'application

Tracer la courbe paramétrique définie par $\left\{ \begin{array}{rcl} x & = & \sin^3\theta \\ y & = & \cos\theta - \cos^4\theta \end{array} \right.$ **sur l'intervalle** $\theta \in [-\pi, \pi]$.

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

T = np.linspace(-np.pi, np.pi, 501)

X = np.sin(T)**3
Y = np.cos(T) - np.cos(T)**4

plt.title('jolie courbe, non ?')
plt.plot(X, Y, linewidth=3)
plt.show()
```