PRÉPAS SCIENTIFIQUES

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

CONFORME AU NOUVEAU PROGRAMME

Alain Caignot • Alain Camanes • Lionel Moisan • David Violeau

Informatique

TRONC COMMUN

TOUT-EN-UN

COURS • MÉTHODES • ENTRAÎNEMENTS ET SUJETS • CORRIGÉS

Tout le cours avec :
- Les objectifs-clés du programme
- Les notions à maîtriser
- Les commandes Python et SQL incontournables

Entraînement complet avec plus de 200 exercices :
vrai/faux, exercices d'application et d'approfondissement inédits, adaptés ou issus de sujets de concours

Des outils pour réviser :
- Les synthèses des notions
- Les méthodes et techniques

Tous les corrigés détaillés

OFFERT EN LIGNE

- + de 100 **flashcards** interactives
- Les **synthèses** à télécharger
- Les **scripts Python** interactifs

Vuibert

PRÉPAS SCIENTIFIQUES

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

CONFORME AU NOUVEAU PROGRAMME

Informatique

TRONC COMMUN

TOUT-EN-UN

COURS • MÉTHODES • ENTRAÎNEMENTS ET SUJETS • CORRIGÉS

Alain Caignot est professeur en classes préparatoires scientifiques au collège Stanislas à Paris.

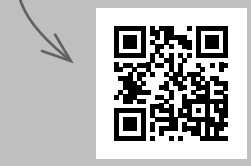
Alain Camanes est professeur en classes préparatoires scientifiques au collège Stanislas à Paris.

Lionel Moisan est professeur à l'université Paris Cité.

David Violeau est professeur en classes préparatoires scientifiques au lycée Janson-de-Sailly à Paris.

Vuibert

Retrouvez notre collection complète ici :



Retrouvez les scripts Python interactifs à télécharger et les ressources numériques utiles pour les exercices au début de chaque chapitre.

ISBN : 978-2-311-21299-0

La loi du 11 mars 1957 n'autorisant aux termes des alinéas 2 et 3 de l'article 41, d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale, ou partielle, faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause, est illicite » (alinéa 1er de l'article 40). Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles 425 et suivants du Code pénal. Le « photocopillage », c'est l'usage abusif et collectif de la photocopie sans autorisation des auteurs et des éditeurs. Largement répandu dans les établissements d'enseignement, le « photocopillage » menace l'avenir du livre, car il met en danger son équilibre économique. Il prive les auteurs d'une juste rémunération. En dehors de l'usage privé du copiste, toute reproduction totale ou partielle de cet ouvrage est interdite. Des photocopies payantes peuvent être réalisées avec l'accord de l'éditeur. S'adresser au Centre français d'exploitation du droit de copie : 20, rue des Grands-Augustins, F-75006 Paris. Tél. : 01 44 07 47 70.



© Vuibert – juillet 2022 – 5, allée de la 2e D.B., 75015 Paris – Site Internet : http://www.vuibert.fr

SOMMAIRE

Ressources numériques

Pour accéder aux ressources numériques en ligne, retrouvez nos codes à flasher tout au long du livre :

- **Flashcards d'auto-évaluation interactives**
- **Scripts Python**
- **Synthèses à télécharger**

Chapitre 1. L'installation et l'utilisation de Python **5**

Cours, p. 6

1 ► Présentation de Python, p. 6 — **2** ► Installation de Python, p. 6 — **3** ► Utilisation de Python, p. 7

Chapitre 2. Bases d'algorithmie **11**

Cours, p. 12

1 ► Notion d'algorithme, p. 12 — **2** ► Les structures de données, p. 12 — **3** ► Les instructions de contrôle, p. 23 — **4** ► Les fonctions, p. 28

Fiche de synthèse, p. 31

Exercices, p. 33

Corrigés, p. 48

Chapitre 3. Manipulation de données **69**

Cours, p. 70

1 ► Import de modules, p. 70 — **2** ► Fichiers, p. 71 — **3** ► Module scientifique, p. 72 — **4** ► Traitement d'images, p. 77

Fiche de synthèse, p. 80

Exercices, p. 82

Corrigés, p. 96

Chapitre 4. Méthodes de programmation **109**

Cours, p. 110

1 ► Algorithmes gloutons, p. 110 — **2** ► Les fonctions récursives, p. 111 — **3** ► Les algorithmes de tri, p. 117

Fiche de synthèse, p. 127

Exercices, p. 128

Corrigés, p. 141

Chapitre 5. Règles de bonne programmation et preuves des algorithmes 157
Cours, p. 158
1 ► Règles de bonne programmation, p. 158 — **2** ► Complexité, p. 160 — **3** ► Preuve de bon fonctionnement, p. 166 — **4** ► Jeux de tests, p. 171
Fiche de synthèse, p. 172
Exercices, p. 174
Corrigés, p. 188

Chapitre 6. Représentation des nombres 207
Cours, p. 208
1 ► Introduction, p. 208 — **2** ► Numération, p. 209 — **3** ► Représentation des nombres entiers en machine, p. 213 — **4** ► Représentation des nombres réels en machine, p. 216 — **5** ► Représentation d'autres informations, p. 222
Fiche de synthèse, p. 223
Exercices, p. 225
Corrigés, p. 234

Chapitre 7. Théorie des graphes 245
Cours, p. 246
1 ► Introduction, p. 246 — **2** ► La notion de graphe, p. 247 — **3** ► Représentations, p. 252 — **4** ► Intermède : piles & files, p. 255 — **5** ► Parcours de graphes, p. 258 — **6** ► Plus courts chemins, p. 265
Fiche de synthèse, p. 273
Exercices, p. 275
Corrigés, p. 294

Chapitre 8. Bases de données 317
Cours, p. 318
1 ► Les motivations, p. 318 — **2** ► L'utilisation d'un logiciel pour manipuler des bases de données, p. 319 — **3** ► La modélisation d'une base de données, p. 320 — **4** ► Les requêtes, p. 326
Fiche de synthèse, p. 338
Exercices, p. 340
Corrigés, p. 348

Chapitre 9. Dictionnaires et programmation dynamique 357
Cours, p. 358
1 ► Le type dictionnaire, p. 358 — **2** ► Manipulation des dictionnaires, p. 359 — **3** ► Les clés d'un dictionnaire doivent être hachables, p. 360 — **4** ► Mémoïsation, p. 361 — **5** ► La programmation dynamique, p. 363 — **6** ► Exemples de problèmes résolus par programmation dynamique, p. 369 — **7** ► Calcul récursif et programmation dynamique, p. 377
Fiche de synthèse, p. 378
Exercices, p. 379
Corrigés, p. 389

Chapitre 10. Intelligence artificielle et jeux à deux joueurs 411
Cours, p. 412
1 ► L'intelligence artificielle, p. 412 — **2** ► Deux algorithmes d'apprentissage statistique, p. 412 — **3** ► Jeux d'accessibilité à deux joueurs sur un graphe, p. 419
Fiche de synthèse, p. 427
Exercices, p. 429
Corrigés, p. 446

CHAPITRE 1

L'installation et l'utilisation de Python

Plan du chapitre

Cours, p. 6 ☐
1 ► Présentation de Python ☐
2 ► Installation de Python ☐
3 ► Utilisation de Python ☐

COURS

1 Présentation de Python

Python est le langage de programmation qui a été choisi par l'Éducation nationale pour introduire une partie du programme d'informatique en CPGE. C'est un langage qui a été créé à la fin de l'année 1989 par Guido Van Rossum. Il a été conçu dans le but d'être lisible et visuellement épuré. Les blocs de programmation sont identifiés par l'indentation, ce qui permet de structurer les algorithmes : toutes les lignes d'un même bloc doivent commencer à la même colonne de texte (en général, on choisit une indentation de quatre espaces). Par exemple :

```
liste = ["simple","puissant","lisible"]                # Python est simple
for adjectif in liste:                                 # Python est puissant
    print("Python est {}".format(adjectif))            # Python est lisible
print("C'est fini")                                    # C'est fini
```

Le but du programme de CPGE n'est pas de faire de vous des spécialistes du langage Python, mais de vous apprendre des méthodes de programmation que vous pourrez réutiliser par la suite dans votre métier, quel que soit le langage que vous utiliserez. Pour cela, Python est tout à fait indiqué, car son avantage principal réside dans le fait qu'il permet une initiation aisée aux concepts de base de la programmation, en minimisant les règles de syntaxe.

Parmi ses autres avantages :

- sa portabilité (il est disponible sous la plupart des systèmes d'exploitation Windows, Mac OS, Linux) ;
- sa gratuité ;
- sa liberté (le fait qu'il soit placé sous une licence libre) ;
- son fonctionnement en console, permettant de tester un programme ligne à ligne.

Actuellement, il existe deux versions stables de Python :

- la version 2.7 est la dernière version majeure dans la série des 2.x et n'est plus maintenue ;
- les versions 3.x (actuellement 3.9, la 3.10 étant en développement). Dans cet ouvrage, nous utiliserons la version 3.8 qui est disponible dans les distributions standards. La plupart des exemples doivent cependant fonctionner avec toutes les versions 3.x.

Python a une autre particularité, il s'agit d'un langage interprété. Chaque ligne de programme que vous écrivez est traitée *à la volée* et *traduite* en langage compréhensible par la machine. C'est le rôle d'un programme auxiliaire : l'interpréteur. L'utilisation de vos programmes nécessite donc d'avoir au préalable installé un interpréteur Python sur votre machine. *A contrario*, il existe des langages compilés. Dans ce cas, l'intégralité des programmes est traduite une fois pour toutes par le compilateur qui va générer un nouveau fichier appelé exécutable. Ce fichier est alors utilisable directement par la machine.

L'avantage des langages interprétés est que l'on peut modifier son programme et voir très rapidement le résultat obtenu *via* l'interpréteur. Un langage compilé nécessite, après chaque modification, de recompiler le programme et de lancer l'exécutable. En revanche, les langages compilés sont généralement plus rapides à l'exécution, car le compilateur optimise l'exécution lors de la traduction en langage machine. Malgré son caractère interprété, Python conserve des performances tout à fait honorables.

2 Installation de Python

L'installation de Python et de tous les modules que nous allons utiliser dépend du système d'exploitation que vous utilisez.

2.1. Sous Windows

Sous Windows, le plus simple est d'utiliser une distribution incluant la plupart des modules. Vous pouvez installer :

- la suite Anaconda, gérée par une entreprise privée, à télécharger et installer depuis le site `www.anaconda.com`. Elle est volumineuse car elle propose de nombreuses librairies et applications annexes ; l'installation peut être un peu longue mais ne pose pas de problème particulier ;
- WinPython, disponible sur `http://winpython.github.io/`, qui utilise Spyder comme éditeur. Il faut choisir une version qui correspond à votre ordinateur (souvent 64 bits).

Les deux distributions sont équivalentes et le choix sera affaire de goût.

Lorsqu'une suite est installée, le cœur de Python est installé sur l'ordinateur (identique à l'installation proposée par `python.org`,) mais un ou plusieurs éditeurs sont disponibles. Dans la suite Anaconda, c'est l'éditeur Spyder qui est proposé par défaut, mais on peut facilement changer d'éditeur si celui-ci ne convient pas (il existe Pyzo, SublimeText, PyScripteur, PyCharm, WingIde, Eric6…). Les installations de ces éditeurs sont décrites sur les sites correspondants.

2.2. Sous Mac OS

Pour installer Pyzo sous Mac OS, il suffit d'aller sur le site dédié `http://www.pyzo.org/` et, dans la section « Downloads », de choisir la version Mac OS (fichier avec l'extension .dmg). Il suffit ensuite de décompresser l'archive en double-cliquant dessus, puis de déplacer le dossier (un simple glisser-déposer) dans le répertoire « Applications ».

2.3. Sous GNU/Linux

La distribution Pyzo `http://www.pyzo.org/` est également disponible sous GNU/Linux mais vous pouvez également choisir d'installer, par l'intermédiaire de votre système de paquets, ceux nécessaires à l'utilisation de Python. Il faut pour cela installer le paquet Python3 ainsi que les paquets relatifs à NumPy, SciPy et Matplotlib afin de pouvoir faire tous les exemples de cet ouvrage.

Si vous n'êtes pas administrateur de votre machine, le plus simple est de télécharger l'archive de Pyzo pour GNU/Linux, de la décompresser dans votre répertoire personnel et ensuite d'exécuter le binaire intitulé Pyzo et présent dans le répertoire où vous avez décompressé l'archive.

2.4. Python sans installation

Si vous souhaitez ne pas installer Python sur votre ordinateur, il est possible de travailler directement en ligne sans installer de logiciel. Plusieurs sites proposent Python en ligne et pour chacun il est préférable de créer un compte pour sauvegarder ses projets.

Le site `https://trinket.io/python3` permet de réaliser des programmes sans s'identifier et l'éditeur Python peut être intégré dans un site web personnel. Il offre la plupart des librairies utilisées dans ce livre.

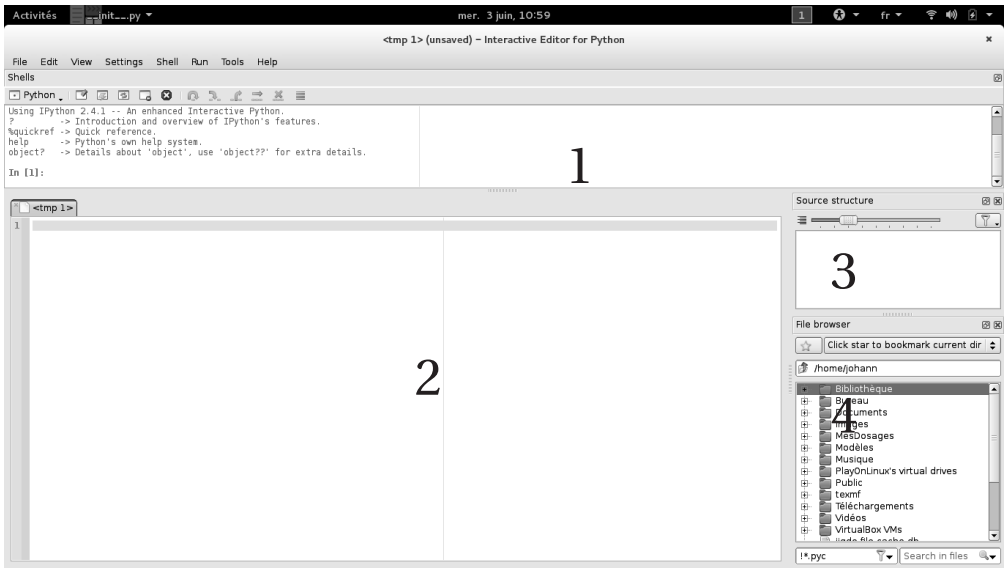
3 Utilisation de Python

3.1. L'éditeur Pyzo

L'utilisation d'un éditeur sur votre machine ou d'un site distant pour programmer en Python requièrent l'utilisation de plusieurs fenêtres de travail :

- une ou plusieurs fenêtres pour renseigner ses scripts Python. Il s'agit d'un éditeur de texte qui propose une coloration syntaxique adéquate et des complétions de code (c'est en général sur ces points que différent les éditeurs) ;
- une fenêtre contenant l'interpréteur Python qui permet soit de tester des commandes, soit d'exécuter des scripts préalablement saisis dans un éditeur. L'interpréteur est facilement repérable car ses lignes commencent toujours par >>>.

La Figure 1.1 présente l'éditeur Pyzo. Sur cette figure, les intitulés des menus sont en anglais. Pour passer en français, il suffit d'aller dans le menu « Settings » puis dans le sous-menu « Select language » et de choisir « French ». Il est nécessaire de redémarrer l'éditeur pour que les changements prennent effet.



► **Figure 1.1.** L'éditeur de Pyzo.

On retrouve ici les différentes fenêtres :

1 : L'interpréteur de commande (IPython sur la figure).

2 : La fenêtre de l'éditeur.

3 : Le squelette du programme écrit dans l'éditeur, qui permet de naviguer facilement dans votre fichier lorsqu'il contient beaucoup de lignes.

4 : Un navigateur de fichiers, qui vous permet d'explorer l'arborescence de votre système de fichiers.

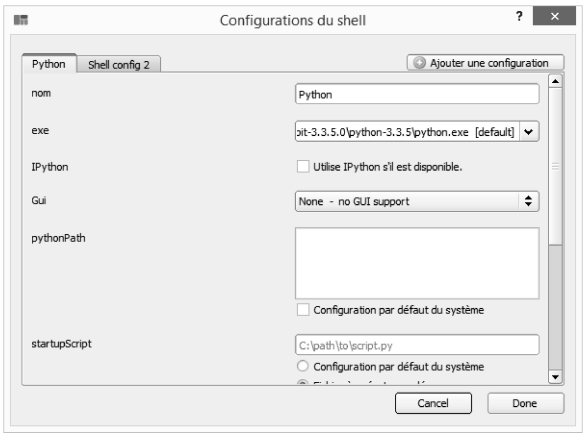
L'éditeur aide à la mise en forme des programmes en Python :

- il propose une autocomplétion des mots que vous êtes en train d'écrire par l'intermédiaire d'un menu déroulant ;
- il ajoute automatiquement une indentation après une instruction le nécessitant ;
- il propose aussi une coloration syntaxique mettant en valeur les mots-clés de Python.

Remarque. Choix de l'interpréteur

Python dispose de plusieurs interpréteurs. Avec Pyzo, il est possible de choisir l'interpréteur de commandes. Par défaut, il s'agit d'IPython, qui permet une utilisation interactive de Python offrant plus de

fonctionnalités. Mais, si vous voulez utiliser l'interpréteur Python de base, il suffit de choisir « Configuration des shells » dans le menu « Shell » (voir Figure 1.2).



► **Figure 1.2.** Configuration des shells sous Pyzo.

Il faut alors décocher la case « Utilise IPython s'il est disponible ». C'est le choix qui a été fait pour la suite de cet ouvrage.

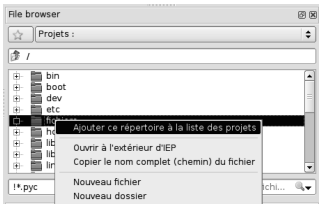
3.2. Écriture et évaluation des programmes

Pour des tests courts, une ou deux lignes maximum, vous pouvez directement utiliser l'interpréteur de commandes, à la manière dont vous utilisez une calculatrice. C'est ce que nous ferons au début de la présentation du langage Python. En revanche, pour des programmes plus longs, il est nécessaire d'utiliser l'éditeur.

Il est important de sauvegarder les programmes écrits dans l'éditeur par l'intermédiaire du menu « Fichiers », puis « Enregistrer » ou « Enregistrer sous ». L'extension utilisée pour les programmes Python est `.py`. Lors de la sauvegarde, cette extension est automatiquement ajoutée au nom de fichier que vous avez choisi.

Pour exécuter votre programme dans l'interpréteur de commandes, il suffit d'aller dans le menu « Exécuter » et de choisir la commande « Exécuter le fichier ». De manière plus rapide, il existe le raccourci *CTRL + E* pour réaliser la même chose ou *CTRL + SHIFT + E* pour exécuter le fichier en se plaçant dans le dossier du fichier automatiquement. Vous pouvez ensuite voir le résultat dans l'interpréteur de commandes situé au-dessus de l'éditeur. De même, le raccourci *ALT + Entrée* permet d'exécuter la sélection.

Il est parfois nécessaire de modifier le répertoire de travail de Pyzo, en particulier lorsqu'il faut lire ou écrire sur les fichiers d'un répertoire donné. Dans le navigateur de fichiers, un clic-droit permet d'ajouter un répertoire à la liste des projets. Il suffit ensuite de sélectionner ce répertoire de projet dans le menu déroulant et enfin, par l'icône étoile, d'aller dans ce dossier. Pour vérifier le changement de dossier, il faut taper `ls` dans l'interpréteur de commandes : la liste des fichiers du répertoire sélectionné doit s'afficher.



► **Figure 1.3.** Changement du répertoire de travail d'IEP.

On peut aussi définir le dossier de travail directement dans un script Python (fichier .py) en tapant

```
import os
os.chdir("Nom_du_chemin_complet")
```

Nous avons présenté les bases de l'utilisation de l'éditeur, nous laissons le lecteur curieux explorer les différents menus de Pyzo afin de découvrir d'autres fonctionnalités non indispensables dans cet ouvrage et que nous ne présenterons donc pas ici.

CHAPITRE 2
Bases d'algorithmie

Plan du chapitre

Cours, p. 12 ☐
- **1** ► Notion d'algorithme ☐
- **2** ► Les structures de données ☐
- **3** ► Les instructions de contrôle ☐
- **4** ► Les fonctions ☐

Fiche synthèse, p. 31 ☐
Exercices, p. 33 ☐
Corrigés, p. 48 ☐

Compétences	Exercices correspondants								
	A	B	C	D	E	F	G	H	I
Analyser et modéliser un problème ou une situation	✓	✓	✓	✓		✓		✓	✓
Imaginer et concevoir une solution	✓	✓	✓	✓	✓	✓	✓	✓	✓
Décrire et spécifier un processus, les données d'un problème									
Mettre en œuvre une solution	✓	✓	✓	✓	✓	✓	✓	✓	✓
Justifier et critiquer une solution				✓					

RESSOURCES NUMÉRIQUES ET SCRIPTS PYTHON DU CHAPITRE

www.lienmini.fr/212990-PYTH-2

FLASHCARDS INTERACTIVES POUR SE TESTER

www.lienmini.fr/212990-FLASH-2

COURS

1 Notion d'algorithme

Un algorithme est la description non ambiguë, en un nombre fini d'étapes, d'un calcul ou de la résolution d'un problème. Il n'est pas nécessaire d'avoir un ordinateur pour suivre les instructions d'un algorithme (l'algorithme d'Euclide que vous avez peut-être appliqué en cours de mathématiques en est un exemple). Cependant, un ordinateur permet de réaliser, en un temps raisonnable, des algorithmes nécessitant un grand nombre d'étapes ou des calculs complexes.

2 Les structures de données

2.1. Types simples

Cette section présente quelques types standards du langage Python.

2.1.1. Les expressions



Définition 2.1. Expression

Une expression est composée de valeurs, de variables, d'opérateurs, de fonctions, et est évaluée en suivant les règles de calcul du langage de programmation pour retourner une *valeur*.

Par exemple, `2 + 3` et `True or False` sont des expressions dont les valeurs sont respectivement `5` et `True`.
Le type du résultat d'une expression est affiché à l'aide de la fonction `type` :

```
>>> type(2 + 3)
<class 'int'>
>>> type(True or False)
<class 'bool'>
>>> type(3.5)
<class 'float'>
```

En Python, le type de la valeur est situé après le mot `class`. Ici, `int`, `bool` et `float` indiquent respectivement que le résultat de l'expression est un entier, un booléen et un flottant (représentation en machine d'un nombre réel).

2.1.2. Les types numériques

Il existe trois types numériques distincts : les entiers, les flottants et les nombres complexes. Nous n'étudierons pas dans cet ouvrage les nombres complexes. Les entiers sont représentés en Python avec une précision infinie et les flottants sont habituellement implémentés en utilisant le type `double` du langage C. Nous présenterons succinctement comment sont représentés les nombres en machine dans le Chapitre 6.

```
>>> 2 + 3
5
>>> 5.1 * 4.2
21.419999999999998
```

On remarque que les calculs sont exacts avec le type entier mais qu'ils sont approchés avec les types flottant et complexe.



Attention ! Séparateur décimal

Le séparateur décimal des flottants, dans la plupart des langages de programmation, est le point et pas la virgule.

Le langage Python supporte une arithmétique mixte : quand un opérateur binaire possède deux opérandes de types différents, l'opérande avec le type le plus « simple » est converti en celui de l'autre opérande. Ces conversions sont effectuées de manière *implicite* dans le langage. Le type entier est plus simple que le type flottant, lui-même plus simple que le type complexe. L'exemple ci-dessous illustre la conversion de l'entier 2 en flottant pour permettre l'addition avec le flottant `6.1` :

```
>>> 2 + 6.1
8.1
```

Les constructeurs `int()` et `float()` peuvent être utilisés pour convertir *explicitement* des nombres, respectivement en entiers et en flottants.

Les opérations du Tableau 2.1 sont définies pour tous les types numériques (sauf les complexes) et sont classées par ordre de priorité croissante.

Python	Résultat
`x + y`	Somme de `x` et `y`
`x - y`	Différence de `x` et `y`
`x * y`	Produit de `x` et `y`
`x / y`	Division de `x` par `y`
`x // y`	Quotient de la division entière de `x` par `y`
`x % y`	Reste de la division entière de `x` par `y`
`-x`	Opposé de `x`
`abs(x)`	Valeur absolue de `x`
`int(x)`	Conversion de `x` en l'entier le plus proche de 0
`float(x)`	Conversion de `x` en flottant
`x**y`	`x` puissance `y`

Tableau 2.1. Les opérations numériques.

N'hésitez pas à tester ces opérations dans un interpréteur Python.

Attention ! Opérateur de multiplication

En Python, la multiplication doit être explicitement indiquée, contrairement aux expressions mathématiques usuelles. Ainsi, si vous tapez l'expression `2(3+4)`, Python renvoie une erreur : l'expression correcte est `2 * (3 + 4)`.

Les expressions suivantes illustrent les opérations usuelles : la multiplication, la division, la division entière et le reste. Les parenthèses ne sont pas toujours obligatoires, car Python applique une priorité sur les opérateurs :

```
>>> 2 * 6.1 + 3
15.2
>>> 10 / 3
3.3333333333333335
```

```
>>> 10 // 3
3
>>> 10 % 3
1
```

Les expressions suivantes illustrent la conversion de flottants en entiers puis d'un entier en flottant.

```
>>> int(5.9)
5
>>> int(-5.9)
-5
>>> float(4)
4.0
```

Lors d'un calcul d'exponentiation (puissance d'entiers), on peut remarquer que le résultat contient un nombre aussi grand que nécessaire de chiffres. Les entiers sont représentés en précision infinie sous Python.

```
>>> 2**60
1152921504606846976
>>> 2**152
5708990770823839524233143877797980545530986496
```



Attention ! Opérateur d'exponentiation

En Python, l'opérateur d'exponentiation est `**` et non `^`, qui est utilisé dans d'autres langages. Attention, ce dernier a une signification en Python et `2^60` renvoie `62`.

Remarque. Choix d'un type pour représenter un nombre

Suivant le problème posé, le choix du type pour représenter un nombre peut être différent :

- si la précision du calcul est importante, alors il vaut mieux utiliser le type entier, car, avec les flottants, la précision est d'environ 15 chiffres significatifs (en base décimale) ;
- pour des grandeurs physiques, le choix des flottants est en général suffisant.

Pour réaliser des opérations mathématiques plus compliquées, on peut utiliser un module particulier, le module `math`.

```
from math import *
sqrt(2) # fonction racine carrée
exp(1) # fonction exponentielle
cos(pi/6)
pow(2,3) # fonction puissance avec conversion implicite en flottants
```

L'utilisation des modules et l'explication de la première ligne seront détaillées dans le Chapitre 3.

Remarque

Pour écrire un commentaire sur une ligne en Python, il suffit de taper le caractère # suivi du commentaire. Cette partie de l'instruction n'est alors pas traitée.

Le module `random` permet de simuler le hasard.

```
from random import *
randint(1,6) # permet de choisir un nombre entier aléatoirement entre 1 et 6 inclus
random() # permet de tirer au hasard un nombre flottant entre 0 et 1
uniform(-2.5,6.8) # même chose entre -2.5 et 6.8
```

2.1.3. Les booléens

En Python, le type `bool` est un sous-type des entiers. Toutes les opérations ou fonctions qui ont un résultat booléen retournent 0 ou `False` pour faux, et 1 ou `True` pour vrai. Le Tableau 2.2 présente les opérateurs booléens par ordre de priorité ascendante.

Python	Résultat
`x or y`	Vrai si `x` ou `y` sont vrais (ou les deux)
`x and y`	Vrai si `x` et `y` sont vrais tous les deux
`not x`	Vrai si `x` est faux et inversement

Tableau 2.2. Les opérateurs booléens.

Remarque. Opérateurs paresseux

Les opérateurs `or` et `and` en Python sont dits paresseux :

- si le premier opérande de `x or y` est `True`, le second opérande n'est pas évalué ;
- si le premier opérande de `x and y` est `False`, le second opérande n'est pas évalué.

```
>>> True or False
True
>>> True or quelconque
True
>>> quelconque or True
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'quelconque' is not defined
>>> False and inconnue
False
>>> not True
False
```

Certains opérateurs sur les types numériques renvoient un résultat booléen. Ce sont les opérateurs de comparaison qui sont listés dans le Tableau 2.3. Ils ont tous la même priorité, mais elle est plus élevée que celle des opérateurs booléens et moins élevée que celle des opérateurs numériques.

Python	Sens
`<`	Strictement inférieur à
`<=`	Inférieur ou égal à
`>`	Strictement supérieur à
`>=`	Supérieur ou égal à
`==`	Égal à
`!=`	Différent de

Tableau 2.3. Les opérateurs de comparaison.

Les expressions ci-dessous illustrent quelques opérateurs de comparaison ainsi que la priorité de ces opérateurs par rapport aux opérateurs booléens et numériques :

```
>>> 2 >= 3
False
>>> 2 >= 3/2
True
>>> not (2 == 3)
True
```

Remarque

Pour faciliter la reprise de vos programmes ou la lecture par un tiers, il convient de parenthéser les expressions dès qu'une ambiguïté est possible.

2.2. Variables

2.2.1. La déclaration et l'initialisation

Les variables ne sont rien d'autre qu'une réservation d'espace mémoire pour stocker des valeurs. Contrairement à d'autres langages de programmation (comme le C ou JAVA), le type des variables Python n'a pas besoin d'être déclaré explicitement pour réserver de l'espace mémoire. Cette opération de réservation d'espace mémoire est appelée *allocation*. L'allocation et la *déclaration* interviennent automatiquement quand une valeur est *assignée* à une variable.
L'opérateur d'affectation est le signe égal : `=`. L'opérande à la gauche de l'opérateur `=` est le *nom* de la variable et l'opérande à la droite est la *valeur* à stocker dans la variable. Le nom de la variable est une suite de caractères alphanumériques ou d'*underscores* (c'est-à-dire les lettres a-z, A-Z, les chiffres 0-9 ou le caractère _, mais ne pas mettre d'accents, ni d'espaces, ni de caractères spéciaux) qui *ne peut pas* commencer par un chiffre. Par exemple, on peut définir une variable nommée `x` ou une variable nommée `x_1`. Python est sensible à la casse, c'est-à-dire qu'il fait la différence entre les majuscules et minuscules. Ainsi, les variables `x` et `X` sont deux variables différentes.

Quelques exemples :

```
>>> compteur = 100   # assignation d'un entier
>>> distance = 25.76 # assignation d'un flottant
>>> verite = True    # assignation d'un booléen
>>> print(compteur, distance, verite) # pour afficher les valeurs de variables
100 25.76 True
```

Ce sont ici nos premières *instructions* qui, contrairement aux expressions, permettent de modifier l'espace mémoire de l'ordinateur. L'ensemble des variables ainsi que les valeurs stockées dans ces variables à un instant t définissent ce que l'on appelle le *contexte*.



Attention ! Mots réservés

Certains mots sont réservés par le langage Python, c'est-à-dire qu'ils ne peuvent pas être utilisés comme nom de variable. En voici la liste :

```
False     None     True     and     as     assert     break     class     continue
def     del     elif     else     except     finally     for     from     global
if     import     in     is     lambda     nonlocal     not     or     pass     raise
return     try     while     with     yield
```

Remarque. Expression et instruction

Le résultat d'une expression étant une valeur, il est tout à fait possible de l'affecter à une variable :

```
>>> a = 3.5 * 4
>>> a
14.0
```

Python permet l'affectation d'une même valeur à plusieurs variables simultanément, mais c'est une particularité du langage Python :

```
>>> a = b = c = 4
>>> print(a, b, c)
4 4 4
```

Une autre particularité syntaxique du langage Python est l'affectation simultanée de plusieurs variables :

```
>>> pas, distance, gagnant = 100, 34.5, False
>>> pas
100
>>> gagnant
False
```

Lorsqu'une variable a été définie, il est ensuite possible de l'utiliser dans une expression :

```
>>> pas = 100
>>> pas + 10
110
```

Lorsque cette expression est évaluée, la variable `pas` est remplacée par la valeur stockée dans l'espace mémoire réservé à cette variable.

Remarque. Choix d'un nom de variable

Pour faciliter la lecture d'un programme, il est conseillé de donner des noms explicites à vos variables. Par exemple, pour un calcul d'aire d'un rectangle, au lieu d'écrire

```
>>> x = 30
>>> y = 10.2
>>> z = x * y
>>> z
306.0
```

préférez

```
>>> hauteur = 30
>>> largeur = 10.2
>>> aire = hauteur * largeur
>>> aire
306.0
```

Lorsque les noms de variables sont longs, utilisez des majuscules pour faciliter la lecture : `fileAttentePourColis` est plus lisible que `fileattentepourcolis`.

2.2.2. L'affectation

L'affectation est le fait de donner une valeur à une variable précédemment déclarée, elle est réalisée à l'aide du même opérateur que l'initialisation : le signe `=`.

```
>>> pas = True # déclaration et initialisation de la variable pas
>>> pas = 50 # affectation d'une valeur à la variable pas
```

En Python, une affectation peut changer le type de la valeur stockée dans la variable. Il est possible d'affecter à une variable le résultat d'une expression dans laquelle cette variable intervient :

```
>>> pas = pas + 10
>>> pas
60
```

Attention, le sens du signe `=` est différent de celui que vous utilisez en mathématiques. L'opérande à gauche du signe `=` est le nom de la variable dans lequel on va stocker le résultat de l'expression de l'opérande de droite : `pas + 10`. Lorsque cette expression est évaluée, la variable `pas` est remplacée par la valeur 50.

Remarque. Raccourcis Python

Au lieu d'écrire `pas = pas + 10`, la syntaxe Python introduit des opérateurs comme `+=`, qui additionne à la variable dans l'opérande de gauche la valeur donnée dans l'opérande de droite. Ainsi, `pas += 10` augmente la variable `pas` de 10. Sur le même principe, pour la plupart des opérateurs classiques, Python définit : `-=`, `*=`, `/=`, etc.

Remarque. Typage statique ou dynamique

Le langage Python permet de modifier le type d'une variable en fonction de la valeur à stocker, en cours d'exécution du programme : le typage est dit *dynamique*.

Beaucoup d'autres langages (C, JAVA) présentent un typage *statique* : le type de la variable doit être indiqué lors de sa déclaration et ne peut plus être modifié, ce qui est assez logique, vu que l'allocation de mémoire dépend du type.
Si le typage dynamique est pratique, il conduit néanmoins à des pertes de performances lors de l'exécution du programme.

2.3. Types séquences

2.3.1. Présentation

Les séquences sont des types informatiques qui permettent de stocker plusieurs valeurs d'une manière efficace et organisée. Nous allons voir trois types séquences de Python : les `list`, les `tuples` ou n-uplets, et les `str` ou chaînes de caractères. Dans les séquences, les données constituent un ensemble fini et ordonné dont les éléments sont indicés par des entiers naturels, en commençant par l'indice 0.
Certaines séquences Python comme les `tuples` et les `str` sont *immuables*, c'est-à-dire que les valeurs stockées dans la séquence ne peuvent pas être modifiées après création. Ainsi, après avoir déclaré et initialisé une variable qui stocke un `tuple`, celle-ci ne peut plus recevoir d'affectation.
Les autres séquences, comme les `list`, sont dites *mutables*. Il est possible de modifier les valeurs stockées dans la séquence et de réaliser des affectations après la déclaration et l'initialisation d'une variable.

Remarque. Valeur
Les sections précédentes ont abordé la notion de *valeur*, qui résulte du calcul d'une expression et qui peut être affectée à une variable. De manière générale, en informatique, le terme de valeur est utilisé pour tous les types rencontrés. Ainsi, une séquence est une valeur de la même manière qu'un entier ou un flottant.

2.3.2. Les listes sous Python

Les listes sont des séquences mutables : les valeurs stockées dans la liste peuvent être modifiées. Une liste se définit entre crochets `[]`. Une liste est soit la liste vide (notée `[]`), soit une succession d'éléments séparés par des virgules :

```
maliste = [2, True, 3, 8.4]
```

Les éléments de la liste sont d'un type quelconque et deux éléments d'une même liste peuvent être de types différents.
On accède à la longueur de la liste (c'est-à-dire à son nombre d'éléments), en temps constant, grâce à la fonction `len` :

```
>>> len(maliste)
4
```

On peut construire une liste à l'aide de plusieurs méthodes :

- en listant ses éléments, comme dans l'exemple précédent ;
- en répétant une liste :

  ```
  >>> maliste1 = 3 * [5, 1]
  [5, 1, 5, 1, 5, 1]
  ```

- en concaténant des listes existantes :

  ```
  >>> maliste2 = maliste + maliste1
  >>> maliste2
  [2, True, 3, 8.4, 5, 1, 5, 1, 5, 1]
  ```

- en partant de la liste vide et en ajoutant, successivement, des éléments à la liste, à l'aide de la méthode `append` :

```
>>> maliste3 = []
>>> maliste3.append(2)
>>> maliste3.append(True)
>>> maliste3.append(3)
>>> maliste3.append(8.4)
>>> maliste3
[2, True, 3, 8.4]
```

- en utilisant une liste par compréhension :

```
>>> [x**2 for x in range(4)]
[0, 1, 4, 9]
```

  Nous reviendrons sur cette technique ultérieurement, après avoir défini les boucles et l'itérateur `range`.

Une fois la liste définie, on peut accéder à chacun de ses éléments, toujours en temps constant, à l'aide de la syntaxe `maliste[i]` :

```
>>> maliste[2]
3
```



Attention !

Le premier élément de la liste est indicé par 0. Le dernier élément de la liste est donc indicé par `len(maliste)-1`.

Remarque. Les indices négatifs

Il est possible d'accéder aux éléments des listes en utilisant des indices négatifs. Par exemple, `maliste[-1]` permet d'accéder au dernier élément de la liste. Cependant, le programme d'informatique se limite à l'utilisation d'indices positifs.

Les opérations courantes sur les listes sont décrites dans le Tableau 2.4.

Opération	Résultat
`x in s`	`True` si un élément de `s` est égal à `x`, `False` sinon
`x not in s`	`False` si un élément de `s` est égal à `x`, `True` sinon
`s + t`	Concaténation de `s` et `t`
`s * n` ou `n * s`	`n` copies de `s` concaténées
`len(s)`	Nombre d'éléments contenus dans `s`
`s[i]`	i-ème élément de `s` (attention, les indices commencent à 0)
`s[i:j]`	Tranche de `s`, séquence composée des éléments `s[k]` tels que $i \leqslant k < j$
`s[i:j:k]`	Tranche de `s`, séquence composée des éléments `s[i + n*k]` tels que $0 \leqslant n < (j-i)/k$
`s[i] = x`	La valeur d'index `i` de la liste `s` est remplacée pas `x`.
`s.append(x)`	Ajoute `x` à la fin de la liste `s`.
`s.pop()`	Renvoie le dernier élément de `s` et le supprime de `s`.

Tableau 2.4. Opérations de modification des listes.

Les opérations sont données dans leur ordre de priorité ascendante, `s` et `t` sont des séquences de même type, `n`, `i`, `j` et `k` sont des entiers et `x` est un objet dont le type répond aux différentes contraintes imposées par `s`.

Les opérations `in` et `not in` ont la même priorité que les opérateurs de comparaisons. La concaténation `+` et la répétition `*` ont la même priorité que les opérateurs arithmétiques correspondants.

Remarque. Les tranches sous Python

Pour les tranches de séquences de la forme `s[i:j]` ou `s[i:j:k]`, si `i` ou `j` est plus grand que `len(s)`, alors on prend `len(s)`. Si `i` ou `j` sont omis, alors on prend les valeurs extrêmes. Par exemple, `s[:10]` est équivalent à `s[0:10]`. Les valeurs de `i` et `j` peuvent être négatives, ces valeurs sont comptées relativement à la fin de la liste. Dans ce cas, on les remplace par `len(s) + i` et `len(s) + j`. Si `k` est négatif, alors la séquence est parcourue en sens inverse.

Attention, les listes sont des objets mutables. Les opérations proposées ci-dessus permettent donc de modifier la liste sans en créer une nouvelle. Ce comportement peut se constater en utilisant la fonction `id` qui renvoie un entier qui identifie de manière unique l'objet durant sa durée de vie :

```
>>> s = [1,2,3,4]
>>> id(s)
139792263928448
>>> s.append(5)
>>> s
[1, 2, 3, 4, 5]
>>> id(s)
139792263928448
>>> s[0] = 10
>>> s
[10, 2, 3, 4, 5]
>>> id(s)
139792263928448
```

Le terme employé pour désigner ces modifications de l'espace mémoire est celui d'*effet de bord* (qui est une mauvaise traduction du terme anglais *side effects*).

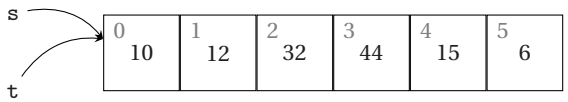


Attention !

En Python, l'affectation de la liste `s` dans la variable `t` *via* l'instruction `t = s` se fait par passage de l'identifiant de `s` dans la variable `t`. La liste `t` fait alors référence à la liste `s` et la modification de l'une entraînera la modification de l'autre.

Des effets inattendus peuvent se produire, comme cela est illustré ci-dessous :

```
>>> s = [10, 12, 32, 44, 15, 6]
>>> t = s
>>> s[0] = 42
>>> s
[42, 12, 32, 44, 15, 6]
>>> t
[42, 12, 32, 44, 15, 6]
```



L'avantage de cette copie est qu'elle est très rapide. Pour éviter ces *effets de bord*, on peut utiliser une *copie par valeur* qui, si elle évite ces problèmes, prend plus de temps et plus d'espace mémoire. La méthode pour effectuer une telle copie est `copy` :

```
>>> s = [1, 2, 3, 4, 5, 6]
>>> t = s.copy()
>>> s[0] = 10
>>> s
[10, 2, 3, 4, 5, 6]
>>> t
[1, 2, 3, 4, 5, 6]
```

Remarque

Les instructions `t = s.copy()` et `t = s[:]` sont identiques.

Attention, cependant, la méthode `copy` ne fonctionne pas récursivement :

```
>>> s = [[0, 1], [2, 3]]
>>> t = s.copy()
>>> s[0][0] = 10
>>> s
[[10, 1], [2, 3]]
>>> t
[[10, 1], [2, 3]]
```

Le module `copy` propose la méthode `deepcopy` qui effectue une copie récursivement, autrement appelée copie *profonde* :

```
>>> import copy
>>> s = [[0, 1], [2, 3]]
>>> t = copy.deepcopy(s)
>>> s[0][0] = 10
>>> s
[[10, 1], [2, 3]]
>>> t
[[0, 1], [2, 3]]
```

Remarque. Modules

Le mot-clé `import` apparaît dans ce programme. Nous verrons plus loin son fonctionnement. Pour l'instant, il suffit juste de savoir qu'il permet d'accéder à de nouvelles fonctions qui ne sont pas déclarées au démarrage de l'interpréteur.

2.3.3. Les n-uplets en Python

Les n-uplets, ou `tuples` en anglais, sont des séquences immuables. Les n-uplets sont délimités par des parenthèses et leurs éléments sont séparés par des virgules. Ils peuvent être construits de différentes manières :

- en utilisant une paire de parenthèses pour désigner le n-uplet vide `()` ;
- en séparant les éléments par une virgule : `a, b, c` ou `(a, b, c)` ou encore `(a, b, c, )`.

Les parenthèses sont optionnelles, à part dans le cas du n-uplet vide ou lorsqu'elles évitent des ambiguïtés syntaxiques.

On peut déterminer la longueur d'un n-uplet ainsi qu'accéder à ses éléments en temps constants. Il est également possible de concaténer des n-uplets. La syntaxe est identique à celle utilisée pour les listes :

```
>>> t = (1, 2, 3, 4)
>>> s = 5, 6, 7, 8
>>> len(t)
4
>>> 2 in t
True
>>> t + s
(1, 2, 3, 4, 5, 6, 7, 8)
>>> t * 3
(1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4)
>>> t[2]
3
>>> t[1:3]
(2, 3)
>>> (t * 3)[::4]
(1, 1, 1)
```



Attention !

Les n-uplets sont des structures de données immuables. Il est donc impossible de modifier l'élément d'un n-uplet :

```
>>> t = (1, 2, 3, 4)
>>> t[0] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Remarque. Dépaquetage des éléments d'un *n*-uplet

En Python, comme dans d'autres langages de programmation, il est possible d'affecter une variable à chaque valeur d'un n-uplet :

```
>>> t = (1,2,3)
>>> a, b, c = t
>>> b
2
```

Cette possibilité est utile avec les fonctions (voir Section 4) qui ont un n-uplet comme valeur de retour.

2.3.4. Les chaînes de caractères en Python

Les données textuelles en Python sont gérées par le type `str` : les chaînes de caractères. Ce sont des séquences *immuables* de caractères. Elles peuvent être définies de plusieurs manières :

- guillemets simples `'On peut y mettre des "guillemets doubles"'`;
- guillemets doubles `"On peut y mettre des 'guillemets simples'"`;
- guillemets triples `"""On peut y mettre des guillemets 'simples' ou "doubles" """`

Les chaînes de caractères avec guillemets triples peuvent contenir des retours à la ligne (notés `"\n"`) ou des tabulations (`"\t"`).

Les fonctions sont identiques à celles utilisées pour les autres séquences et sont illustrées ci-dessous :

```
>>> a = "Ceci est une chaîne de caractères"
>>> a[5]
'e'
>>> a[2:5]
'ci '
>>> len(a)
33
```

Comme les tuples, les chaînes de caractères sont immuables.

2.3.5. Conversions entre types

La fonction `str()` permet de convertir un objet en chaîne de caractères :

```
>>> a = 1.4
>>> b = str(a)
>>> b
'1.4'
```

Les fonctions `list()` et `tuple()` permettent de passer d'un type séquence à un autre :

```
>>> n = 123
>>> ch = str(n)
>>> ch
```

```
'123'
>>> list(ch)
['1', '2', '3']
>>> tuple(ch)
('1', '2', '3')
```

2.3.6. Des méthodes

Nous avons utilisé le terme *méthode* pour les fonctions `append` ou `copy`. Une méthode `methode` s'utilise sur l'objet `objet` de la manière suivante :

```
objet.methode(arguments)
```

Python possède de nombreuses méthodes pour chacun des objets que nous venons de détailler. Ces méthodes sont soit des primitives du langage (comme `append`), disponibles dès l'ouverture de Python, soit présentes dans des modules (comme `deepcopy` du module `copy`).
La documentation en ligne propose un descriptif des méthodes. Citons par exemple :

- `maliste.sort()` qui permet de trier la liste `maliste` sans modifier son adresse ;
- `maliste.reverse()` qui permet de retourner la liste `maliste` sans modifier son adresse ;
- `s.join(maliste)` qui, étant donné une liste `maliste` de chaînes de caractères, les concatène en ajoutant la chaîne `s` entre chacun des éléments :

  ```
  >>> maliste = ["he", "o"]
  >>> "ll".join(maliste)
  'hello'
  ```

- `machaine.lowercase()` qui renvoie la chaîne de caractères `machaine` en minuscules ;
- `machaine.uppercase()` qui renvoie la chaîne de caractères `machaine` en majuscules ;
- `machaine.split(c)` qui renvoie la liste contenant chacune des parties de `machaine` qui sont séparées par la chaîne `c`. Par exemple,

  ```
  >>> s = "hello"
  >>> s.split("ll")
  ['he', 'o']
  ```

Remarque. Chercher de l'aide
Il est impossible de détailler dans cet ouvrage toutes les méthodes existantes. Le lecteur doit être capable de rechercher seul de l'aide sur un sujet donné. Pour cela, la fonction `help()` de Python est très utile. Si l'argument est une chaîne de caractères, alors la chaîne de caractères est recherchée et une page d'aide est affichée. Si l'argument est n'importe quel autre type d'objet, une page d'aide sur l'objet est générée :

```
>>> help("str")
Help on class str in module builtins:
...
>>> help("copy")
Help on module copy:
...
>>> help(1)
Help on int object:
...
```

3 Les instructions de contrôle

Une instruction est une étape élémentaire de programmation qui produit un effet, c'est-à-dire qui indique à l'ordinateur l'action à effectuer. Nous avons déjà vu deux instructions : l'initialisation et l'affectation d'une variable. Les instructions qui suivent (la séquence, le test et la boucle) forment avec les deux premières le noyau impératif du langage Python. La définition de fonctions est aussi une instruction du langage Python, mais nous la détaillerons à part.

3.1. Séquence

La séquence est tout simplement l'instruction qui permet de former une instruction à partir de deux instructions. Une séquence s'écrit sur une seule ligne de l'interpréteur en séparant les deux instructions par un point-virgule ;. Par exemple, la séquence `x = 2; y = 3` permet de réaliser une instruction constituée de deux initialisations de variables. Dans l'éditeur, il suffit d'écrire une instruction par ligne pour réaliser une séquence.

3.2. Instruction conditionnelle

C'est peut-être l'instruction la plus connue en informatique. Son squelette est le suivant :

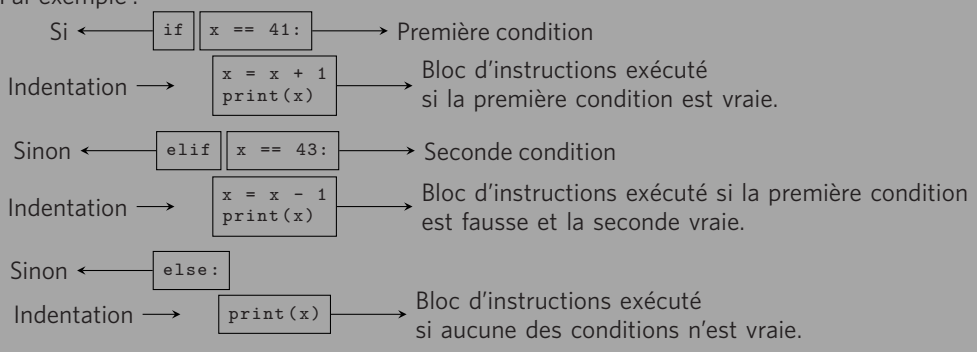
```
if expression1:
    séquence d'instructions si expression1 vraie
elif expression2:
    séquence d'instructions si expression1 fausse et expression2 vraie
else:
    séquence d'instructions si expression1 et expression2 fausses
```

Le `elif` et le `else` sont optionnels. Plusieurs `elif` peuvent être insérés successivement, constituant ainsi un substitut pour l'instruction `switch` ou `case` de certains langages. En général, les expressions utilisées ont une valeur booléenne (`true` ou `false`).



Attention ! Indentation en Python

L'indentation après les deux points est obligatoire en Python et permet de définir la séquence d'instructions qui sera effectuée selon la valeur des expressions. Toutes les instructions qui sont au même niveau d'indentation font partie de la même séquence d'instructions.
Par exemple :



L'oubli d'indentation ou une mauvaise indentation amène à une erreur. L'exécution de

```
if x == 43:
x = x + 1
```

retourne

```
File "<stdin>", line 1, in <module>
    x = x + 1
    ^
IndentationError: expected an indented block
```

Seul le langage Python utilise l'indentation pour isoler les blocs d'instructions. Dans les autres langages, généralement, les blocs sont isolés entre des accolades ou des mots-clés.

3.3. Instructions itératives

3.3.1. Les boucles inconditionnelles

Les boucles inconditionnelles permettent d'itérer une séquence d'instructions un nombre fini prédéterminé de fois. En Python, la fonction `range` permet de générer des suites sur lesquelles itérer :

```
for nb in range(2, 11, 3):
    print(nb)

2
5
8
```

Comme nous pouvons le constater, la fonction `range` en Python prend trois arguments : le nombre de départ, le nombre final (exclu) et le pas (qui peut être négatif). Le premier et le troisième argument sont facultatifs et, s'ils ne sont pas présents, leur valeur par défaut est respectivement 0 et 1. Il est possible de parcourir une liste à l'aide de ses indices en combinant `range` et `len` :

```
>>> a = ["simple", "puissant", "lisible"]
>>> for i in range(len(a)):
...     print(i, "Python est " + a[i])
...
0 Python est simple
1 Python est puissant
2 Python est lisible
```

En Python, l'instruction `for` a un comportement légèrement différent de celui qu'elle peut avoir en Pascal ou en C. Plutôt que d'itérer une suite de nombres ou de donner à l'utilisateur la possibilité de définir à la fois le pas d'itération et la condition d'arrêt, la boucle `for` de Python permet d'itérer sur tous les objets de type séquence dans l'ordre dans lequel ils apparaissent dans la séquence. Par exemple,

```
>>> mots = ["CPGE", "Concours", "Informatique"]
>>> for m in mots:
...     print(m, len(m))
...
CPGE 4
Concours 8
Informatique 12
```

On peut également parcourir les chaînes de caractères, qui sont, comme nous l'avons vu, un type séquentiel.

```
>>> for lettre in "python":
...     print(lettre)
...
p
y
t
h
o
n
```

Remarque

On peut utiliser la fonction `enumerate` pour simplifier le parcours d'une liste si l'indice et la valeur de chaque élément sont recherchés :

```
>>> a = ["simple", "puissant", "lisible"]
>>> for i, val in enumerate(a):
...     print(i, "Python est " + val)
...
0 Python est simple
1 Python est puissant
2 Python est lisible
```

Comme pour l'instruction `if`, la séquence d'instructions effectuée dans la boucle `for` doit être indentée après les deux points et toutes les instructions doivent être au même niveau d'indentation.



Attention ! Modification de la séquence itérée

Lorsque vous itérez sur une liste, il ne faut pas oublier que les listes sont des objets mutables. Il est donc possible de les modifier dans le bloc d'instruction qui suit. Cependant, cela va modifier le parcourt de la boucle. Si vous devez modifier la séquence que vous êtes en train de parcourir dans la boucle, il est conseillé de faire une copie en premier face au risque de bogues difficiles à appréhender.
On pourra comparer le comportement des deux programmes suivants.

```
>>> mots = ['CPGE', 'Concours', 'Informatique']
>>> for m in mots[:]:
...     mots.append(m)
...
>>> mots
['CPGE', 'Concours', 'Informatique', 'CPGE', 'Concours', 'Informatique']
```

et

```
>>> mots = ['CPGE', 'Concours', 'Informatique']
>>> for m in mots:
...     mots.append(m)
...
```

qui ne termine pas, car la liste `mots` s'allonge au fur et à mesure qu'on la parcourt.

Listes en compréhension Attention, les paragraphes suivants présentent une particularité de Python. Cette facilité n'existe pas dans tous les langages de programmation.

La syntaxe des listes en compréhension fournit un moyen rapide de créer des listes. Le contenu de la liste est défini par filtrage du contenu d'une autre liste. Cette syntaxe est très lisible et se rapproche de la notation utilisée en mathématiques :

$$\{i \mid i \in \mathbb{N}, i^2 <= 25\}.$$

Cela se traduit en Python de la manière suivante :

```
>>> [x for x in range(25) if x**2 <= 25]
[0, 1, 2, 3, 4, 5]
```

La création de cette liste peut se faire avec une boucle :

```
res = []
for x in range(n):
    if x**2 <= 25:
        res.append(x)
```

Cependant, les listes par compréhension ont été optimisées en Python et la première syntaxe renvoie le résultat plus rapidement. La définition d'une liste en compréhension permet l'utilisation de plusieurs boucles :

```
>>> [(x, y) for x in range(4) for y in range(4) if x <= y]
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```

Les listes en compréhension sont particulières à Python. Il faut donc être capable de créer une liste sans utiliser ce mécanisme.

3.3.2. Les boucles conditionnelles.

L'instruction `while` est utilisée pour répéter une séquence d'instructions tant qu'une expression est vraie :

```
i = 1
while i < 4:
    i = i + 1

print(i)
4
```

En langage Python, de la même manière qu'avec les autres boucles, toutes les instructions de la séquence doivent être indentées au même niveau après les deux points.



Attention ! Boucle infinie

Il faut être très prudent avec ce genre de boucles : si l'expression testée n'est jamais fausse, la boucle ne se termine jamais. C'est le cas notamment de la boucle suivante :

```
i = 1
while i < 4:
    i = i - 1
```

On veillera donc à ce que la condition proposée après le mot-clé `while` soit modifiée lors du passage dans la boucle.

3.3.3. L'instruction `break`

La syntaxe générale d'une boucle `for` et d'une boucle `while` est :

```
for element in sequence:
    sequence d'instructions
```

```
while expression:
    sequence d'instructions
```

Une instruction `break` dans la séquence d'instructions termine la boucle prématurément. L'instruction `break`, quoique pratique dans certains cas, ne facilite pas la relecture et la compréhension du programme. Il faut donc l'éviter autant que possible.

Par exemple, le programme

```
>>> for lettre in "hello":
...     if lettre == "l":
...         break
...     print(lettre)
...
h
e
```

peut également être écrit

```
>>> mot = "hello"
>>> i = 0
>>> while mot[i] != "l":
...     print(mot[i])
...     i += 1
...
h
e
```

4 Les fonctions

Dans les algorithmes, certaines tâches doivent être effectuée un grand nombre de fois. La plupart du temps, on crée des fonctions qui permettent de réaliser ces actions. Elles permettent une relecture du code plus aisée et surtout un gain de temps non négligeable d'écriture.

4.1. Définition et appel

La définition et l'appel des fonctions sont quelques-unes des instructions de base du noyau fonctionnel du langage Python. La définition d'une fonction se fait de manière simple avec le mot-clé `def` :

```
def maFonction(argument1,argument2,...):
    séquence d'instructions
    return expression ou liste d'expressions
```

Le nom de la fonction est une suite de caractères, qui sont soit alphanumériques (avec ou sans majuscules), soit le caractère underscore _. On retrouve les mêmes règles de nommage que pour les variables. Les noms choisis ne doivent pas correspondre aux mots-clés réservés sous Python ou à une fonction déjà existante.

Il est possible de passer autant d'arguments que l'on souhaite. Il suffit de les séparer par des virgules. Le type de ces arguments est quelconque.

Une fois la fonction définie, pour l'utiliser dans un programme, on tape simplement le nom de la fonction avec des valeurs pour les arguments entre parenthèses.

```
maFonction(1.2,a) #si a est une variable connue
```

La fonction définie ci-dessous permet de calculer le temps que met un point matériel à retomber au niveau du sol lorsqu'il est lancé verticalement du sol avec une vitesse initiale v_0 :

```
def tempsVol(v0):
    g = 9.81
    return 2 * v0 / g

>>> tempsVol(11)
2.2426095820591234
```

Supposons que l'on veuille généraliser ce calcul à tout astre. Il est alors nécessaire de passer la valeur de g en argument. Cet argument peut être facultatif et sa valeur par défaut est celle de l'accélération de la pesanteur au niveau du sol sur Terre :

```
def tempsVol(v0, g=9.81):
    return 2 * v0 / g

>>> tempsVol(11)
2.2426095820591234
>>> tempsVol(11, 1.622)
13.563501849568434
```

Remarque. Documentation des fonctions

Afin d'améliorer la lisibilité de vos programmes, il est nécessaire de documenter vos fonctions. Pour une fonction de plusieurs arguments, il faut documenter le rôle de chaque argument ainsi que la valeur de retour de la fonction. En Python, cela se fait simplement à l'aide des triples quotes `'''` :

```
def tempsVol(v0, g=9.81):
    ''' v0 : vitesse initiale du mobile
        g : accélération de la pesanteur au niveau du sol
        retourne le temps de vol pour un mobile
        lancé verticalement depuis le sol
    '''
    return 2*v0/g
```

```
>>> help(tempsVol)
Help on function tempsVol in module __main__:

tempsVol(v0, g=9.81)
    v0 : vitesse initiale du mobile
    g : accélération de la pesanteur au niveau du sol
    retourne le temps de vol pour un mobile
    lancé verticalement depuis le sol
```

L'instruction `return` est optionnelle. Par exemple, la fonction peut se limiter à un affichage (comme la fonction `print`) ou à la modification d'une liste. Par défaut, la fonction renverra alors l'objet `None` :

Python

```
def affiche(texte1,texte2):
    print(texte1+' '+texte2)

def modifie(liste):
    if len(liste) != 0:
        liste[0] = 2
```

Interpréteur

```
>>> affiche('Python','est facile')
Python est facile
>>> l=[1,2,3,4]
>>> test = modifie(l)
>>> l
[2, 2, 3, 4]
>>> type(test)
<class 'NoneType'>
```



Attention ! `return` dans une boucle

La présence du mot-clé `return` dans une fonction en interrompt l'exécution. Tout ce qui se trouve après sera du code mort. Par exemple, si l'instruction `return` est rencontrée lors de l'évaluation d'une boucle, la boucle se termine et la fonction renvoie la valeur après le `return`.

Il y a plusieurs méthodes pour passer des variables en paramètres dans une fonction, dont le *passage par valeur* et le *passage par référence*. Dans le cas du passage par valeur, la valeur de l'expression passée en paramètre est copiée dans une variable locale. C'est cette variable qui est utilisée pour faire les calculs dans la fonction appelée. La seconde méthode consiste non plus à passer la valeur des variables comme paramètre, mais à passer une référence à la variable. Il n'y a donc plus de copie, plus de variable locale. Toute modification du paramètre dans l'appel de la fonction entraîne la modification de la variable passée en paramètre.

Les passages par référence sont plus rapides et plus économes en mémoire que les passages par valeur, puisqu'il n'y a ni création ni initialisation de variables locales. Cependant, les passages par valeurs permettent d'éviter de détruire par mégarde les variables passées en paramètres.

En Python, tous les passages se font par référence, mais cela ne veut pas dire que les variables passées en paramètre sont modifiables. Cela dépend de leur caractère immuable ou mutable.

Dans les deux programmes suivants, la variable `a` n'est pas modifiée (à gauche) puis modifiée (à droite) :

Passage par référence d'un objet immuable

```
def f(a):
    print(a)
    a = 5
    print(a)
a = 3
f(a)
print(a)

3   # premier print de f
5   # second print de f
3   # valeur de a après la fonction
```

Passage par référence d'un objet mutable

```
def f(a):
    print(a)
    a[0] = 5
    print(a)
a = [3]
f(a)
print(a)

[3]   # premier print de f
[5]   # second print de f
[5]   # valeur de a après la fonction
```

La différence de comportement vient du fait que les valeurs sont passées par référence et qu'un entier est immuable, alors qu'une liste est variable. Cela peut néanmoins s'avérer déroutant.

Remarque. Affectation du retour d'une fonction dans une variable
Le résultat d'une fonction peut être affecté à une variable.

```
def g(a):
    b = [i for i in range(a)]
    return a, b

X = g(10)

x, y = g(10)

print(X)
(10, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]) # contenu de X

print(x,y)
10 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]    # contenu de x, y
```

La fonction retourne ici un `tuple`. Dans l'exemple, `X` contient le `tuple` complet. Mais il est également possible de dépaqueter directement le `tuple` dans plusieurs variables, ici dans `x` et `y`.

4.2. Fonctions anonymes

Les fonctions lambda sont des fonctions dites anonymes définies sur une seule ligne. L'intérêt principal que nous pouvons en tirer sera la concision de l'écriture sur une seule ligne pour des fonctions simples, mais également une utilisation plus compacte lorsqu'il s'agit de passer une fonction en paramètre d'une autre fonction.
Par exemple, on crée une fonction anonyme à laquelle, on applique 4.

```
>>> (lambda x : 3*x)(4)
12
```

Cette notation est à rapprocher de la notation mathématique $x \mapsto 3x$.
La fonction lambda peut aussi être affectée à une variable qui sera ainsi une fonction. Il peut aussi y avoir plusieurs arguments.

```
>>> f = lambda a, b:2 * a + 3 * b
>>> f(1, 2)
8
>>> f(-3, 2)
0
```

Il est également possible de réaliser une condition. Par exemple, la définition de la fonction rampe qui vaut 0 pour t négatif et $a * t$ pour t positif avec a un argument optionnel pris égal à 1 par défaut.

```
>>> rampe = lambda t, a = 1: 0 if t<0 else a*t
>>> rampe(-8)
0
>>> rampe(8)
8
>>> rampe(8,2)
16
```

Illustrons pour finir l'utilisation du passage d'une fonction dans un paramètre en écrivant une fonction qui, étant donné une fonction f, renvoie $x \mapsto (f(x))^2$.

```
def carre(f):
    def cf(x):
        return f(x)**2
    return cf

puissance_4 = carre(lambda x:x**2)

>>> puissance_4(5)
625
```

FICHE SYNTHÈSE

La notion d'algorithme Un algorithme est la description non ambiguë en un nombre fini d'étapes d'un calcul ou de la résolution d'un problème. Il n'est pas nécessaire d'avoir un ordinateur pour suivre les instructions d'un algorithme (algorithme d'Euclide, par exemple), mais un ordinateur permet de réaliser en un temps raisonnable un algorithme nécessitant un grand nombre d'étapes ou des calculs complexes.

Les types de données Les valeurs possibles prises par les données dépendent du type de la donnée. Le type d'une donnée indique la manière dont la valeur sera représentée en mémoire. Les opérateurs usuels sur les types présents dans Python sont rappelés ci-dessous.

- Les **entiers** pour représenter les entiers relatifs : type **int**. Division entière : `a//b`. Reste division entière (modulo) : `a%b`.
- Les **flottants** pour représenter les nombres réels : type **float**. Puissance : `x**y`.
- Les **séquences** pour représenter un ensemble de valeurs dans des conteneurs : type **list** `a=[1,2.5,3]`, `b=[[1,3],4.5,8,9]` ou **tuple** `b=(1,3.6,8)`. Les commandes usuelles communes à tous les conteneurs sont :
  - créer un conteneur vide : `a=[]` pour une liste ; `a=()` pour un tuple ;
  - donner le nombre d'éléments dans le conteneur : `len(a)` ;
  - accéder aux éléments du conteneur : `a[i]` pour accéder à l'élément d'indice i (les indices commencent à 0) ;
  - extraire un sous-ensemble du conteneur (tronçonnage) : `a[:]` pour accéder à toute la liste, `a[1:3]` donne `[a[1],a[2]]`, `a[debut,fin,pas]` renvoie la liste des éléments d'indice `debut` à `fin-1` par incrément de `pas`, `a[::-1]` inverse l'ordre des éléments ;
  - concaténer des conteneurs : `[1]+[2]` donne `[1,2]` ;
  - répéter des conteneurs : `[1,2]*3` donne `[1,2,1,2,1,2]`.

  Pour les listes uniquement, on peut ajouter un élément en fin de liste : `a.append(3)` ; on peut aussi supprimer un élément d'indice i : `a.pop(i)`.
  Un conteneur particulier pour obtenir une séquence de nombres : `range([debut,]fin[,pas])` : liste d'entiers de `debut` (0 par défaut) à `fin-1`, par incrément de `pas` (1 par défaut).
- Les **chaînes de caractères** type **str**, proche du type **list**, mais non mutable (comme les tuples, ce type ne peut pas être modifié sans faire une recopie) : `txt="petit texte"`. On retrouve la concaténation, la répétition et des méthodes spécifiques disponibles dans la documentation.
- Les **booléens** type **bool**. 2 valeurs : `True` ou `False`. Opérateurs de comparaison : `==` (égalité), `!=` (différence), `>, >=, <, <=`. Opérateurs logiques : `or, and, not`.
- **Opérations sur les types.** Il est possible de réaliser des opérations sur les types :
  - obtention du type d'une variable : `type(objet)` retourne le type de l'objet ;
  - conversion de type : `int(4.9)` donne 4, `float("4.2")` donne 4.2 (transforme une chaîne de caractères en flottant).

Les instructions Les instructions existant dans la plupart des langages de programmation sont listées ci-dessous.

- La **déclaration et affectation de variable** entraîne l'allocation d'un espace mémoire pour stocker une valeur en mémoire : `a = 3`.
- La **séquence d'instructions** permet de réaliser une suite d'instructions. La marque de début et de fin de séquence dépend du langage utilisé (indentation en Python).

- L'**instruction conditionnelle** permet, en fonction de la valeur d'un booléen, de réaliser une séquence précise d'instructions.

```
if condition:
  bloc1
else:
  bloc2
```

```
if condition1:
  bloc1
elif condition2:
  bloc2
else:
  bloc3
```

- La **boucle inconditionnelle** permet de réaliser une séquence d'instructions un nombre de fois prédéterminé.

```
for  i in range(10):
  bloc

# avec a une séquence quelconque
for t in a:
  bloc
```

```
for i,t in enumerate(a)
  # t prend la valeur des éléments
  # i prend la valeur de l'indice.
  # On a donc : t=a[i] (on économise 1 ligne)
  bloc
```

  La fonction `range(deb, fin, pas)` permet de créer des séquences débutant à `deb`, jusqu'à `fin-1` par pas de `pas`.

- La **boucle conditionnelle** permet de réaliser une séquence d'instructions tant qu'un booléen est évalué comme vrai (condition d'arrêt).

```
while condition :
  bloc
```

La définition et l'appel de fonctions L'utilisation de fonctions permet d'améliorer la lisibilité générale et la maintenance du code. Ce sont les briques de base de tout algorithme évolué. Il est possible de les voir comme des « boîtes » qui prennent des paramètres en entrée et qui renvoient une valeur.

Il est nécessaire d'écrire une fonction dès qu'une séquence d'instructions se retrouve plusieurs fois dans un code. L'autre avantage de la décomposition d'un problème complexe en fonctions simples est la modularité que cela entraîne : certaines fonctions peuvent être réutilisées dans d'autres problèmes. Cela permet un gain de temps et d'énergie.

Attention toutefois à ne pas confondre la définition d'une fonction (mot-clé `def` en Python) avec l'appel d'une fonction qui renvoie effectivement une valeur (éventuellement vide).

Définition standard avec arguments optionnels et valeur par défaut :

```
def identifiant(argument1,argument_optionnel=1.5):
  """ Description de la fonction"""
  bloc d instructions
  return valeur1,valeur2
  # La valeur None est retournée si le mot-clé return n'est pas utilisé.
```

Une fonction peut avoir plusieurs instructions return. La fonction s'arrête au premier return rencontré.

Les variables utilisées dans une fonction sont **locales**. Une variable définie hors d'une fonction est **globale**. Attention, les listes passées en arguments sont des variables globales (donc modifiables par le corps de fonction).



RETROUVEZ ICI LA SYNTHÈSE DE CE CHAPITRE À TÉLÉCHARGER

www.lienmini.fr/212990-SYNTH-2



EXERCICES

Vrai ou Faux

		Vrai	Faux
a)	Soit la liste `a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`. La commande `a[1:5]` renvoie `[2, 3, 4, 5, 6]`.	☐	☐
b)	Soit la liste `a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`. La commande `a[2:5] = [10,11,12]` renvoie `[1,2,10,11,12,6,7,8,9,10]`.	☐	☐

c)	À la fin des instructions suivantes, la variable `b` vaut 1.
	`n = 10`
	`if n < 15:`
	`  b = 1`
	`if n < 20:`
	`  b = 2`
	`else:`
	`  b = 3`
		☐	☐

d)	À la fin des instructions suivantes, la variable `n` vaut 20.
	`i = 0`
	`n = 0`
	`while i < 10:`
	`  n = n + 2`
		☐	☐

e)	Avec la définition suivante de la fonction `f`, `f(3)` renvoie 9.
	`def f(x):`
	`  y = 3*x`
		☐	☐

f)	Avec la définition suivante de la fonction `f`, `f(3)` renvoie 3.
	`def f(x):`
	`  for i in range(1,101):`
	`  y = i*x`
	`  return y`
		☐	☐

g)	Soit la chaîne `chaine = 'Le soleil brille mais il fait froid.'`. `chaine[:16]` renvoie `'Le soleil brille'`.	☐	☐
h)	La commande `chaine[30:] = 'chaud'` permet de modifier la chaîne de caractères.	☐	☐

Exercices d'application

Exercices sur la manipulation des nombres

●●○ **EXERCICE 1** **Multiples de 3 et 5**

Les entiers naturels strictement inférieurs à 10 qui sont multiples de 3 et 5 sont 3, 5, 6, 9. Leur somme est égale à 23.

Écrire une suite d'instructions qui affiche la somme des entiers naturels, multiples de 3 et 5, strictement inférieurs à 1 000.

●●○ **EXERCICE 2** **Somme des chiffres d'une puissance**

$2^{15} = 32\,768$ et la somme des chiffres de cette puissance est $3+2+7+6+8=26$. Quelle est la somme des chiffres du nombre 2^{50} ?

On pourra utiliser les fonctions `str` *et* `int`.

●●○ **EXERCICE 3** **Une première suite**

Soit a un réel non nul et A un réel positif. Soit la suite u définie par

$$\begin{cases} u_0 & = a \\ u_{n+1} & = \frac{1}{2}\left(u_n + \frac{A}{u_n}\right), \quad \forall n > 0. \end{cases}$$

1) Écrire une fonction `suite` qui prend en argument un entier n et les réels A et a, et qui renvoie le nombre u_n.
2) Pour $A=4$, $a=1$, calculer les 10 premiers termes de la suite. Faire de même pour $A=2$ et $a=10$. Conjecturer la limite de la suite pour un réel A quelconque.
3) Proposer une modification du programme pour arrêter la boucle itérative dès que la distance entre deux termes consécutifs de la suite est inférieure à 0,001.

●●○ **EXERCICE 4** **Triplet pythagoricien**

1) Écrire une fonction `pythagoricien` qui renvoie `True` s'il est possible de construire un triangle rectangle de côtés de longueurs entières a, b et c, en supposant que c correspond à lhypoténuse. Un tel triplet est alors appelé un triplet pythagoricien. La fonction renverra `False` sinon.
   Exemple : $3^2+4^2 = 9+16 = 25 = 5^2$ donc $(3,4,5)$ est un triplet pythagoricien.
2) Déterminer le seul triplet pythagoricien tel que $a+b+c = 1\,000$.

●●○ **EXERCICE 5** **Problème de Josephus**

Soit n et p deux entiers naturels non nuls ; n joueurs sont assis en cercle et numérotés de 1 à n. En partant du joueur 1 (inclus), on compte p joueurs et on élimine le joueur sur lequel on tombe. Puis on recommence à partir du joueur éliminé jusqu'à ce qu'il ne reste plus qu'un joueur, qui sera le gagnant du jeu.

Par exemple avec $p=3$ et $n=10$, on élimine le joueur n° 3, puis le n° 6, puis le n° 9, puis le n° 2, etc.

1) Programmer une fonction `Josephus(n, p)` qui prend en paramètre n et p et qui renvoie le numéro du gagnant (on rappelle que le premier joueur est numéroté 1).
   En Python, l'instruction `del maliste[i]` *permet de supprimer l'élément d'indice* `i` *de la liste* `maliste`.
2) Programmer une fonction `choixp(n)` qui renvoie, si elle existe, une valeur de p assurant au joueur 1 de gagner le jeu pour un nombre initial de n joueurs.

Exercices sur la manipulation des listes

●●○ **EXERCICE 6** **Minimum et maximum d'une liste**

1) Écrire une fonction `mini(L)` qui renvoie le minimum de la liste L.
2) Écrire une fonction `maxi(L)` qui renvoie le maximum de la liste L.

Pour tester votre programme, vous pouvez utiliser une liste L définie de manière aléatoire. Les lignes Python suivantes permettent de définir une telle liste :

```
import random
L = [random.randint(0,100) for i in range(1000)]
```

●●○ **EXERCICE 7** **Moyenne d'une liste**

Écrire une fonction `moyenne(L)` qui renvoie la moyenne des éléments de la liste `L`.

●●○ **EXERCICE 8** **Éléments communs entre deux listes**

Écrire une fonction `communs(L1, L2)` qui, étant donné deux listes, renvoie la liste des éléments qui sont communs à ces deux listes. Vous veillerez à ce que les éléments de la liste retournée soient deux à deux distincts.

●●○ **EXERCICE 9** **Facteurs premiers**

La décomposition en facteurs premiers consiste à écrire un nombre entier comme produit de nombres premiers. Les nombres premiers qui interviennent dans ce produit sont appelés facteurs premiers.

Exemple : $13\,195 = 5 \times 7 \times 13 \times 29$.

1) Écrire une fonction `facteurs_premiers(n)` qui renvoie la liste contenant les facteurs premiers du nombre entier n.
2) Tester la fonction en donnant à n la valeur $360 = 2 \times 2 \times 2 \times 3 \times 3 \times 5$ ou $1\,010\,021 = 17 \times 19 \times 53 \times 59$.

Exercices sur la manipulation des chaînes de caractères

●●○ **EXERCICE 10** **Inversion de chaîne**

Soit une chaîne de caractères `mot`.

1) Écrire une fonction `inverse_mot` qui renvoie l'inverse d'un mot ; `inverse_mot("bonjour")` renverra par exemple `"ruojnob"`.
2) Écrire une fonction `palindrome(mot)` qui teste si la chaîne de caractères `mot` est un palindrome (l'inverse est égal au mot). Par exemple, les mots « rotor » et « laval » sont des palindromes. La fonction renverra `True` si la chaîne est un palindrome et `False` sinon. Les espaces ne sont pas comptés pour tester une phrase (exemple : « engage le jeu que je le gagne »).

●●○ **EXERCICE 11** **Majuscules dans des chaînes**

Écrire une fonction `majuscule(prenoms)` qui prend en argument une liste de prénoms et qui la modifie pour remplacer chaque prénom par sa version avec une majuscule au début.

Par exemple, après les instructions

```
prenoms = ["jean", "Yacine", "andrew", "jerome", "li"]
majuscule(prenoms)
```

la liste `prenoms` contiendra `["Jean", "Yacine", "Andrew", "Jerome", "Li"]`.

On pourra pour cela introduire les variables

```
minuscules="abcdefghijklmnopqrstuvwxyz"
majuscules="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

●●○ **EXERCICE 12** **Suite de Conway**

La suite « audio-active » a été inventée en 1986 par le mathématicien John Conway. Chaque terme de la suite se détermine en énonçant à haute voix les chiffres du terme précédent. Ainsi, si on commence par `"1"`, le terme suivant sera `"11"`, puis `"21"`, ensuite `"1211"`, etc.

1) Écrire une fonction `AudioActive(init)` qui convertit le nombre `init` en un entier qui est son successeur dans la suite selon le principe énoncé précédemment.

2) Écrire une fonction `SuiteConway(init, n)` qui renvoie la liste des $n+1$ premiers termes de la suite de Conway commençant par `init` (qui sera un nombre dont les chiffres sont uniquement des 1 et de 2).
3) On note L_n le nombre de chiffres du n-ième terme de la suite. Écrire une fonction `rapport_suite(n)` qui renvoie la liste des rapports $\frac{L_{i+1}}{L_i}$ pour $0 \leq i \leq n-1$. Vérifier que ce rapport semble converger vers une limite indépendante du terme de départ.

Exercices sur la manipulation des fonctions

●●○ **EXERCICE 13** **Image**

Écrire une fonction `map_liste(f, liste)` qui, étant donné une fonction `f` et une liste `liste`, retourne, sans modifier `liste`, la liste constituée des images des éléments de `liste` par `f` sans en changer l'ordre.
Par exemple,

```
>>> liste1 = [2, 4, 5]
>>> def carre(x):
...     return x**2
>>> map_liste(carre, [2, 4, 5])
[4, 16, 25]
```

La fonction `map` est déjà implémentée en Python. Il convient bien sûr de ne pas l'utiliser ici.

●●○ **EXERCICE 14** **Filtrage**

Écrire une fonction `filtre(liste, predicat)` qui, étant donné une liste `liste` et une fonction booléenne `predicat`, renvoie la liste des éléments de `liste` qui satisfont `predicat`.
Par exemple,

```
>>> liste1 = [2, 4, 5]
>>> def est_pair(x):
...     return (x % 2 == 0)
>>> filtre(est_pair, [2, 4, 5])
[2, 4]
```

La fonction `filter` est déjà implémentée en Python. Il convient bien sûr de ne pas l'utiliser ici.

●●○ **EXERCICE 15** **Composée**

1) Écrire une fonction `abs_fct(f)` qui, étant donné une fonction `f`, renvoie sa valeur absolue.
2) Écrire une fonction `compose(f, g)` qui, pour deux fonctions `f` et `g`, renvoie leur composée $f \circ g$.

Exercices d'approfondissement

●○○ **EXERCICE A** **Suite de Syracuse**

La suite de Syracuse est une suite d'entiers positifs définie par un entier initial $u_0 > 0$ et la relation de récurrence suivante pour $n \geq 0$:

- si u_n est pair, alors $u_{n+1} = u_n/2$;
- si u_n est impair, alors $u_{n+1} = 3u_n + 1$.

La *conjecture de Syracuse*, non démontrée à ce jour mais vérifiée numériquement pour un très grand nombre d'entiers u_0 (jusqu'à des valeurs au-delà de 10^{18}), prétend que, pour tout entier u_0, il existe un rang N (dépendant de u_0) tel que $u_N = 1$.

On définit la fonction `syracuse(u0, N)` qui, étant donné un entier initial `u0`, renvoie la liste des u_i pour i variant de 0 à N.

1) En calculant à la main les différents termes de la suite, que vaut `syracuse(20, 10)` ? Si la conjecture de Syracuse est vraie, que semble être le comportement asymptotique de la suite ?
2) Écrire une fonction `syracuse(u0,N)`, qui prend en entrée deux entiers strictement positifs `u0` et `N`, et renvoie un tableau de taille `N+1` contenant les valeurs $(u_n)_{0\leq n\leq N}$ de la suite de Syracuse définie par la valeur initiale `u0`.
   Pour tester la parité d'un entier `a`, on pourra utiliser la commande `a%2` en Python.
3) Écrire une fonction `syracuseN(u0)`, qui prend en entrée la valeur initiale u_0 et renvoie le premier indice tel que $u_n = 1$.
   On supposera vraie la conjecture de Syracuse évoquée plus haut, qui affirme donc que cet entier n existe pour tout entier u_0.

●○○ EXERCICE B Diviseurs non triviaux [1]

1) On considère le code de la fonction `d` suivante :

```
L =[]
for nombre in range(1, n+1):
    if n % nombre == 0:
        L.append(nombre)
return L
```

   Quel est le résultat de l'appel `d(4)` ? Puis de l'appel `d(10)` ? Que fait la fonction `d` ?
2) Un diviseur non trivial d'un entier n est un diviseur de n différent de 1 et de n. Écrire une fonction `DNT` prenant en argument un entier et renvoyant la liste des diviseurs non triviaux de cet entier.
3) Écrire une fonction `sommeCarresDNT` prenant en argument un entier et renvoyant la somme des carrés des diviseurs non triviaux de cet entier.
4) Écrire la suite des instructions permettant d'afficher tous les nombres entiers inférieurs à 1 000 et égaux à la somme des carrés de leurs diviseurs non triviaux. Que peut-on conjecturer ?

●●○ EXERCICE C Comptage des occurrences dans une liste

Les listes sont un des objets de base dont la manipulation doit être maîtrisée.

Soit une liste L remplie aléatoirement avec des entiers :

```
from numpy.random import randint
#liste de taille 1 000 prenant des nombres aléatoires entre 0 et 999 999 999
L = randint(1000000000, size = 1000)
```

Pour les questions suivantes, écrire une suite d'instructions qui permettent d'y répondre.

1) Extraire le 10e élément de la liste L.
2) Extraire la sous-liste composée du 4e au 20e élément inclus de la liste L.
3) Écrire une fonction `maxi` qui prend en argument une liste et renvoie son plus grand élément. Comparer votre résultat à celui renvoyé par la fonction `max` de Python.

Remarque. Derrière les fonctions natives de Python, il faut bien avoir à l'esprit qu'il y a un algorithme implanté qui aura un coût (une complexité).

4) Écrire une fonction `maxi2` qui prend en argument une liste et renvoie la valeur de son deuxième plus grand élément. La valeur retournée sera différente de celle du maximum, sauf dans le cas d'une liste dont tous les termes sont égaux. Pour vérifier votre résultat, vous pouvez trier la liste et afficher l'avant dernier élément :

```
L.sort()
print(L[len(L)-2])
```

[1] Inspiré de E3A 2015.

On cherche maintenant à déterminer le nombre d'occurrences de chaque valeur de la liste `L`. Pour cela, on souhaite retourner une liste `R` dans laquelle `R[i]` vaudra le nombre d'occurrences de la valeur i dans la liste `L`.

On testera l'algorithme pour la liste `L` définie par

```
from numpy.random import randint
#liste de taille 10 000 prenant des nombres aléatoires entre 0 et 100
L = randint(100, size = 10000)
```

5) Écrire une fonction `occurrences(liste, maximum)` qui, étant donné une liste `liste` dont les valeurs sont comprises entre 0 et `maximum` renvoie une la liste des occurrences de chacun des entiers compris entre 0 et `maximum`.
6) Écrire une fonction `occurrences_max(liste, maximum)` qui, étant donné une liste `liste` dont les valeurs sont comprises entre 0 et `maximum`, renvoie la valeur de `liste` qui admet la plus grande occurrence.

La création de la liste des occurrences peut être améliorée. D'une part, il est possible que certaines valeurs n'apparaissent pas dans la liste, et on stocke donc inutilement le nombre de leurs occurrences (qui est nul). D'autre part, nous venons de voir qu'il était nécessaire de connaître la valeur maximale des éléments de la liste. Enfin, la liste des occurrences ne fonctionne simplement que pour des listes d'entiers. Pour éviter ces deux écueils, nous allons utiliser un dictionnaire. Un dictionnaire est un ensemble dont les éléments sont accessibles par une clé.

```
test = {}  #initialisation du dictionnaire
test['cle'] = [0, 1, 2] #création de la clé "cle" avec une liste
test[10] = 1 #création de la clé 10 avec la valeur 1
'cle' in test #renvoie True car la clé cle existe
3 in test #renvoie False car la clé 3 n'existe pas
test['cle'] #renvoie [0, 1, 2]
```

Ce type *dictionnaire* sera étudié plus en détail en 2e année.

7) Écrire une fonction `occurrences_dico(liste)` permettant de créer le dictionnaire ayant pour clés les valeurs de la liste `liste` et pour valeurs le nombre d'occurrences de ces valeurs.

Il est possible de parcourir les clés d'un dictionnaire *via* les instructions suivantes :

```
for cle in dictionnaire:
    print("Pour la clé", cle, "la valeur associée est :", dictionnaire[cle])
```

8) Écrire une fonction `occurrences_dico_max(liste)` permettant de connaître la (ou les) valeur(s) de la liste `liste` ayant le plus d'occurrences.

●●○ EXERCICE D Recherche d'une valeur dans une liste triée

Dans le cadre de la gestion d'un très grand nombre de données (base de données, gestion de stock, carnet d'adresses...), on peut être amené à faire des recherches régulières sur celles-ci. Pour faire une recherche efficace, une possibilité est de trier les données selon un critère afin de pouvoir effectuer des recherches plus efficaces par la suite.

Recherche naïve

Soit une liste déjà triée L de taille n. On initialise cette liste avec le module `random`.

```
from numpy.random import randint
L = randint(1000000000, size = 1000)
L.sort() #permet de trier la liste par ordre croissant
```

Cela permet de créer une liste de taille 1 000 en prenant aléatoirement des entiers naturels jusqu'à 1 000 000 000.

On commence par réaliser une algorithme naïf qui parcourt l'ensemble des valeurs de la liste pour vérifier si un entier donné est présent.

1) Proposer une fonction `recherche_naive(L, val)` prenant en argument une liste triée `L` et une valeur `val` à rechercher dans celle-ci qui renvoie `True` si la valeur est présente et `False` sinon.

Quelques optimisations peuvent être réalisées. En effet, la liste étant triée :

- on peut vérifier si la valeur à chercher n'est pas strictement inférieure au premier terme de la liste ;
- on peut vérifier si la valeur à chercher n'est pas strictement supérieure au dernier terme de la liste ;
- on peut sortir de la boucle de recherche dès que la valeur de la liste est strictement supérieure à la valeur recherchée.

2) Mettre en place ces optimisations sans utiliser le mot-clé `break`.

Une compétence importante en vérification des programmes est d'être capable de choisir un jeu de tests permettant de vérifier que l'algorithme fonctionne pour tous les cas fréquents et notamment les cas limites.

C'est le rôle des tests dit unitaires réalisés dans tout développement de logiciels.

3) Proposer un jeu de tests permettant de vérifier que votre programme optimisé fonctionne bien. Tester votre programme à l'aide de ces différents tests et le modifier au besoin.

On souhaite déterminer le temps de calcul pour une liste de taille plus significative de 10 millions de valeurs.

Le code suivant permet d'afficher le temps de calcul de la fonction :

```
import time
tic = time.time() #mesure un temps de référence en secondes
recherche_naive(L, valeur)
print(time.time() - tic)
```

4) Mesurer le temps de calcul de la fonction naïve et le comparer à la recherche interne de Python : `valeur in L`.
5) Donner la complexité de la recherche naïve en fonction de la taille de la liste. Pour cela, compter le nombre de comparaisons effectuées en fonction de la taille de la liste.

Remarque. On parle de complexité linéaire si le nombre de comparaisons est proportionnel à la taille de la liste, de complexité quadratique si le nombre de comparaisons est proportionnel à la taille de la liste au carré. Cet aspect de l'informatique sera détaillé dans un prochain chapitre.

Recherche par dichotomie

La méthode par dichotomie fait partie des algorithmes basés sur le principe « diviser pour régner ».

La liste étant triée, l'idée est de tester si le terme du milieu de la liste est inférieur strictement, supérieur strictement ou égal à la valeur recherchée. S'il est égal, on a trouvé la valeur et l'algorithme renvoie `True`, sinon, s'il est strictement inférieur, on cherche la valeur dans la demi-liste de gauche, et sinon dans la demi-liste de droite.

On itère jusqu'à ce que la liste de recherche soit vide.

On constate que l'intervalle de recherche est divisé par deux à chaque itération, ce qui permet de converger très vite.

6) Proposer une fonction `recherche_dicho(L, val)` basée sur cette méthode. Utiliser une boucle `while` en réfléchissant bien à la condition d'arrêt.
7) Définir un jeu de tests pour valider votre fonction.
8) Mesurer le temps de calcul de la fonction et le comparer aux temps précédents.
9) Modifier la fonction précédente pour qu'elle renvoie également le nombre de comparaisons réalisées.

On va maintenant tracer l'évolution du nombre de comparaisons réalisées en fonction de la taille de la liste pour des tailles variant de 10 à 1 000 000.

10) Mettre en place une série d'instructions permettant de déterminer les nombres de comparaisons en fonction de la taille de la liste `L` et les stocker dans une liste. Veiller à répartir de manière homogène les différentes tailles de la liste variant de 10 à 1 000 000.

Pour faire une représentation graphique d'une liste en fonction d'une autre, il faut exécuter les commandes suivantes :

```
import matplotlib.pyplot as plt
plt.plot(liste_x, liste_y)
plt.show()
```

11) Réaliser le tracé. Proposer l'expression d'une fonction pouvant interpoler la courbe précédente.

On peut remarquer que, dans le pire des cas, si la valeur n'est pas dans la liste, l'algorithme va découper l'intervalle en deux jusqu'à atteindre un intervalle de taille nulle.

En notant N la taille de la liste, le nombre de divisions de l'intervalle n est tel que $2^n \leq N$.

12) Donner la valeur maximale de n en fonction de N. Tracer cette fonction en la superposant à la courbe précédente. Conclure.

●●○ EXERCICE E Autour des chaînes de caractères

Dès que l'on utilise un traitement de texte pour rédiger un courrier, un rapport ou autre, un certain nombre de fonctionnalités sur la manipulation des chaînes de caractères sont proposées : la fonction de recherche d'un mot, la fonction rechercher-remplacer, etc.

Soit le texte suivant [2] :

```
texte = "Dans ce but, l'ingénieur Murchison, accouru à San Francisco, fit établir
d'énormes grappins d'un système automatique qui ne devaient plus lâcher le projectile,
s'ils parvenaient à le saisir dans leurs pinces puissantes. Il fit aussi préparer des
scaphandres qui, sous leur enveloppe imperméable et résistante, permettaient aux
plongeurs de reconnaître le fond de la mer. Il embarqua également à bord de la
Susquehanna des appareils à air comprimé, très ingénieusement imaginés. C'étaient
de véritables chambres, percées de hublots, et que l'eau, introduite dans certains
compartiments, pouvait entraîner à de grandes profondeurs. Ces appareils existaient
à San-Francisco, où ils avaient servi à la construction d'une digue sous-marine. Et
c'était fort heureux, car le temps eût manqué pour les construire."
```

1) Écrire une fonction `compte_caractere(texte)` qui renvoie le nombre de caractères présents dans `texte`, espaces non compris.
2) Écrire une fonction `compte_mot(texte)` qui renvoie le nombre de mots du texte.
3) Écrire une fonction `recherche_lettre(texte, lettre)` qui renvoie le booléen `True` si la lettre est présente et `False` sinon. On n'utilisera pas la solution proposée en Python : `lettre in texte`.
4) Écrire une fonction `recherche_index_lettre(texte, lettre)` qui renvoie l'index de la première occurrence de lettre dans le texte et -1 sinon. On n'utilisera pas la solution interne à Python : `texte.index(lettre)`.

Remarque. Il est « dangereux » en Python de faire retourner -1 si la lettre n'est pas présente, car `texte[-1]` existe et correspond au dernier caractère. C'est la raison pour laquelle la fonction interne de Python renvoie un message d'erreur si la lettre n'est pas présente dans le texte. On pourrait renvoyer `None`.

5) Écrire une fonction `recherche_index_lettre2(texte, lettre, ind)` qui renvoie l'index de la première occurrence de lettre dans le texte à partir de l'index `ind` et -1 si la lettre n'est pas présente dans le texte après l'index `ind`.
6) Écrire une fonction `recherche_mot_plus_long(texte)` qui renvoie le mot le plus long. Pour simplifier, on supposera que la ponctuation fait partie des mots.
7) Écrire une fonction `recherche_motif(texte, motif)` qui renvoie la liste contenant les indices de début des motifs trouvés dans le texte sachant que les motifs ne peuvent pas se superposer.
Par exemple, la recherche de "cece" dans la chaine "cecececе" ne renverra que les indices 0 et 4.

[2] Extrait de *Autour de la lune* de Jules Verne.

8) Tester votre fonction pour le motif `"ce"` avec le texte proposé en début d'énoncé.
   Pour tester si votre fonction est correcte, il faut tester si le motif est trouvé en début de texte, fin de texte, à l'intérieur d'un mot, si des motifs se suivent, etc. Tester votre fonction sur le texte `"ce abcdecegh ce abcecedfh ce"`, toujours pour le motif `"ce"`.
   Votre fonction doit renvoyer `[0, 8, 13, 18, 20, 26]`.
9) Écrire une fonction `recherche_remplace_motif(texte, ancien_motif, nouveau_motif)` qui renvoie le texte modifié en remplaçant toutes les occurrences `ancien_motif` par `nouveau_motif`.
10) Tester votre fonction en remplaçant `"appareil"` par `"dispositif"` dans le texte proposé en début d'énoncé.

●●○ EXERCICE F Autour des nombres premiers

On rappelle qu'un nombre entier $n \geqslant 2$ est *premier* s'il n'est divisible que par 1 et par lui-même. La suite des nombres premiers commence donc par $2, 3, 5, 7, 11\ldots$ et joue un rôle considérable dans plusieurs branches des mathématiques. En théorie des nombres, bien sûr, mais aussi en analyse (conjecture sur les zéros de la fonction ζ de Riemann, par exemple). Les nombres premiers sont également un ingrédient essentiel de la cryptographie moderne.

Dans cet exercice, nous explorons numériquement trois énoncés mathématiques liés aux nombres premiers : le théorème des nombres premiers (qui décrit la densité asymptotique des nombres premiers), la conjecture de Golbach (formulée en 1742 et non résolue à ce jour) et la conjecture de Pólya (formulée en 1919).

Théorème des nombres premiers

La méthode du crible d'Ératosthène est une méthode de recherche de nombres premiers, qui consiste à écrire tous les nombres entiers inférieurs à n et à rayer successivement les multiples des nombres entiers à partir de 2 (les nombres non rayés à la fin sont exactement les nombres premiers).

1) Écrire une fonction `test_premier(n)` qui prend un entier $n \geq 1$ et renvoie une liste `t` de booléens (True ou 1,False ou 0) telle que `t[k]` est vrai si et seulement si le nombre k est premier. On traduira pour cela sur le tableau `t` la méthode du crible d'Ératosthène.
   Vérifier la fonction `test_premier` pour $n = 20$ et estimer l'ordre de grandeur du n maximal atteignable en un temps raisonnable (quelques secondes).
2) Analyser, en les justifiant, les différentes optimisations réalisées dans le code Python suivant :

```
def test_premier(n):
    t = [1] * (n+1)
    t[0] = 0
    t[1] = 0
    for k in range(2, int(n**(1/2))+1):
        if t[k] == 1 :
            for j in range(k*k, n+1, k):
                t[j] = 0
    return t
```

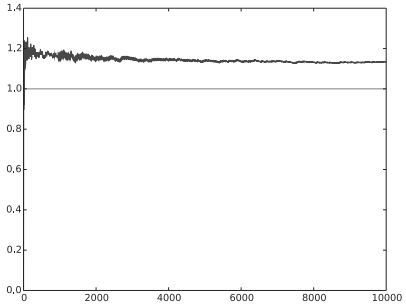
Dans toute la suite, c'est cette implémentation de la fonction `test_premier` qui sera utilisée.

3) Estimer à nouveau, pour cette nouvelle version, l'ordre de grandeur du n maximal atteignable en un temps raisonnable (quelques secondes).
4) Écrire une fonction `premiers(n)` qui prend en entrée un entier $n \geq 1$ et renvoie une liste contenant la suite des nombres premiers inférieurs ou égaux à n. Vérifier la fonction pour $n = 40$.
5) On note $\pi(n)$ le nombre d'entiers premiers inférieurs ou égaux à n (on a ainsi $\pi(5) = 3$ et $\pi(10) = 4$, par exemple). Écrire une fonction `compte_premiers(n)` qui prend en entrée un entier $n \geq 1$ et renvoie un tableau `P` tel que `P(k)`$=\pi(k)$ pour tout $k \leq n$. Vérifier la fonction pour $n = 10$.

Le *théorème des nombres premiers*, conjecturé par Gauss et Legendre à la fin du XVIIIe siècle, mais démontré seulement un siècle plus tard par Hadamard et La Vallée Poussin, établit que

$$\pi(n) \underset{n\to\infty}{\sim} \frac{n}{\ln(n)}.$$

On peut superposer sur un graphe la fonction $k \mapsto \pi(k)\ln(k)/k$, qui a donc pour limite 1 en $+\infty$. On observe que la convergence de cette fonction vers 1 est très lente.



Conjecture de Golbach

La *conjecture de Goldbach*, énoncée en 1742 et toujours non démontrée à ce jour, prétend que tout entier pair supérieur ou égal à 4 peut s'écrire comme la somme de deux nombres premiers. Il s'agit de l'une des conjectures les plus célèbres (et probablement de l'une des plus difficiles) de la théorie des nombres, et aucune preuve ni contre-exemple n'ont été trouvés à ce jour. Néanmoins, cette question a connu des développements récents, puisqu'en 2013, le mathématicien Harald Helfgott a démontré la version faible de cette conjecture : « Tout nombre impair supérieur ou égal à 7 peut s'écrire comme somme de 3 nombres premiers. »

6) Écrire une fonction `goldbach_ok(n)` qui prend en entrée un entier $n \geq 1$ et renvoie un vecteur `T` de booléens tel que pour $k \leq n$, `T(k)` est vrai si, et seulement si, l'entier $2k$ peut s'écrire comme la somme de deux nombres premiers. Pour coder cette fonction, on utilisera la liste `p = premiers(2*n)` de la question 3 de cet exercice et on listera l'ensemble des sommes de nombres premiers du type `S(i, j) = p(i) + p(j)` avec $i \geq j$. Ainsi, $T(k)$ est vrai si, et seulement si, l'un au moins des termes de `S` vaut $2k$). À l'aide de la fonction `goldbach_ok`, tester la conjecture de Goldbach jusqu'à l'entier le plus grand possible en vous limitant à un temps raisonnable le temps de calcul.
7) Plus généralement, écrire une fonction `comete_goldbach(n)` qui prend en entrée un entier $n \geq 1$ et renvoie un vecteur `C` tel que, pour tout $k \leq n$, `C(k)` soit égal au nombre de façons d'écrire l'entier $2k$ comme somme de 2 nombres premiers. Tracer ensuite à l'aide de la commande suivante, mais sans relier les points, le graphe de la fonction $k \mapsto C(k)$, appelé « comète de Goldbach » pour $k = 2\,000$.

```
import matplotlib.pyplot as plt
C = comete_goldbach(2000)
plt.plot(C, '.', ms=1); plt.show()
```

Conjecture de Pólya

Soit $N(k)$ le nombre de termes intervenant (en comptant les répétitions) dans la décomposition en facteurs premiers d'un entier $k \geq 1$ (on a ainsi, par exemple, $N(1) = 0$, $N(6) = 2$, $N(18) = 3$ et bien sûr $N(p) = 1$ pour tout nombre premier p). On définit alors la fonction de Liouville :

$$L(k) = \sum_{i=1}^{k} (-1)^{N(i)},$$

de telle sorte que $L(k) > 0$ si, et seulement si, parmi tous les entiers $i \in \{1, 2, \ldots, k\}$, une majorité (au sens strict) se décompose avec un nombre pair de facteurs premiers. La conjecture de Pólya prétend que cette situation n'arrive jamais après $k = 1$, autrement dit que $L(k) \leq 0$ pour tout entier $k > 1$.

8) Écrire tout d'abord, en utilisant la méthode du crible d'Ératosthène et en vous inspirant des codes de la question 2, une fonction `plus_petit_facteur(n)` qui prend en entrée un entier $n \geq 1$ et renvoie un vecteur `f` tel que, pour tout $k \leq n$, `f[k]` soit égal au plus petit facteur premier de k (avec la convention `f[1] = 1`).
9) Écrire une fonction `nombre_facteurs(n)` qui prend en entrée un entier $n \geq 1$ et renvoie un vecteur `N` tel que, pour tout $k \leq n$, `N[k]` soit égal au nombre de facteurs premiers de k. On pourra utiliser la définition suivante de N par récurrence :

$$N[1] = 0 \qquad \text{et} \qquad \forall k \geq 2,\ N[k] = N[k/f[k]] + 1,$$

où $f[k]$ est le plus petit facteur premier de k introduit dans la question précédente.
10) Écrire ensuite une fonction `Liouville(n)` qui renvoie le tableau des $L(k)$ (pour $k \leq n$) définis ci-dessus. Tracer le graphe de L pour $n = 100, 1\,000, 10\,000 \ldots$ et observer l'allure chaotique qui semble se répéter à différentes échelles. En optimisant le code pour aller plus vite, on pourrait déterminer l'entier k permettant de montrer que la conjecture est fausse.

```
import matplotlib.pyplot as plt
L = Liouville(100)
plt.plot(L, "b")
```

●●○ **EXERCICE G** **Simulation d'emprunts**

Lors d'un achat immobilier, il est important de déterminer la façon dont pourra être remboursé un crédit immobilier. En fonction de la somme empruntée, sur combien d'années s'étalera le prêt ? Quelles seront les mensualités de remboursement du crédit ? Quel taux pourra-t-on obtenir ? Il existe de nombreux simulateurs en ligne permettant de faire ces différents calculs. Nous allons voir ici quels sont les algorithmes en place derrière ces simulateurs.

On note S la somme empruntée et t_a le taux annuel (le taux mensuel t_m est calculé simplement par la formule $t_m = t_a/12$). On note n le nombre de mois pour rembourser la somme empruntée et m la mensualité constante. Chaque mois, on rembourse une partie du capital emprunté ainsi que des intérêts proportionnels au capital restant dû. C'est là toute la difficulté de ce type d'emprunt. En notant I_i le montant de l'intérêt chaque mois, le capital restant dû le mois i est $C_i = C_{i-1} - (m - I_i)$, avec $C_0 = S$ et $I_i = C_{i-1} \cdot t_m$.

1) Écrire une fonction `emprunt` qui, étant donné le capital initial, le taux annuel et les mensualités, renvoie une liste contenant, pour chaque mois, un tuple constitué du capital restant dû et des intérêts du mois. On testera le programme avec $S = 400\,000$, $t_a = 0{,}02$ et $m = 2\,000$.

La fonction `round` permet d'arrondir un nombre en précisant le nombre de décimales.

```
>>> round(45.2365, 2)
45.24
```

2) Modifier la fonction précédente pour qu'elle renvoie un tuple constitué de la liste calculée à la question précédente, du nombre de mois nécessaires pour rembourser le prêt (somme empruntée et intérêts) et du total des intérêts versés. Améliorer la fonction en arrondissant les intérêts au centime d'euro et en gérant la dernière mensualité différemment des autres.

Dans le cas où le nombre de mois n est fixé, on peut calculer la mensualité par la formule

$$m = \frac{(1+t_m)^n}{(1+t_m)^n - 1} \cdot S \cdot t_m.$$

Toutefois, la mensualité est arrondie au centime d'euro supérieur. La dernière mensualité est donc ainsi réduite.

La fonction `ceil` du module `math` permet de renvoyer la partie entière supérieure d'un flottant.

3) Dans le cas où le nombre de mois n est fixé (la somme empruntée S est également fixée), écrire une fonction `mensualite` qui calcule la mensualité et l'arrondir au centime d'euro supérieur. Calculer la dernière mensualité et indiquer le total des intérêts.

●●○ EXERCICE H Chiffrer/déchiffrer avec César

Dans un monde de communication, la cryptographie devient incontournable lorsque les informations à échanger sont confidentielles. Autrefois réservée aux utilisations militaires, la cryptographie est aujourd'hui très présente dans les échanges bancaires, les informations médicales, les communications…

Cet exercice et le suivant proposent de réaliser des algorithmes de cryptage et de décryptage de texte suivant diverses méthodes en analysant les faiblesses des chiffrements les plus simples.

Les méthodes de chiffrement que nous allons utiliser sont des méthodes par décalage des lettres. Le principe consiste à décaler de k places chacune des lettres du message à chiffrer et à remplacer ainsi la lettre indicée n par la lettre indicée $n+k$ modulo 26. Pour simplifier le problème, nous ne chiffrerons que des textes écrits en minuscules pour lesquels tous les caractères spéciaux (accents, espaces, sauts de ligne…) ont été supprimés.

L'objectif est de déchiffrer les messages contenus dans les fichiers `mystere1.txt` et `mystere2.txt`.

Manipulation de listes

Commençons par écrire deux fonctions sur les listes d'entiers qui seront utiles par la suite.

1) Écrire une fonction `translate(liste, k)` qui, étant donné une liste `liste` d'entiers compris entre 0 et 25 et un entier `k`, renvoie la liste contenant les entiers de `liste` translatés de `k` modulo 26. Ainsi, le i-ème élément de la nouvelle liste sera `liste[i] % 26`.
2) Écrire une fonction `occurrence(liste)` qui, étant donné une liste `liste` d'entiers compris entre 0 et 25, renvoie une liste `occ` de taille 26 telle que `occ[i]` représente le nombre d'occurrences de l'entier `i` dans la liste `liste`.

César : chiffrement/déchiffrement

Le chiffrement de César consiste à décaler chacune des lettres du message à chiffrer par une lettre à distance fixe. Ainsi, si la distance est fixée à 2, la lettre `a` est remplacée par la lettre `c`, la lettre `b` par la lettre `d`…

La commande `ord` permet de transformer un caractère en un entier (son code `Utf-8`) et la commande `chr` permet de transformer un entier en le caractère correspondant. On supposera dans toute la suite que le texte à chiffrer/déchiffrer est constitué uniquement de lettres minuscules non accentuées.

On rappelle que la fonction `list` transforme une chaîne de caractères en une liste de caractères correspondants. De manière bijective, `"".join(liste)` transforme la liste de caractères `liste` en une chaîne ou mot correspondant.

3) Écrire une fonction `char_to_int(liste)` qui transforme une liste de caractères minuscules en une liste d'entiers compris entre 0 et 25. Par exemple, `"a"` sera renvoyé sur `0`, `"b"` sera renvoyé sur `1`, etc. On pourra remarquer que `ord("a")` renvoie 97 et `ord("z")` renvoie 122.
4) Écrire une fonction `int_to_char(liste)` qui transforme une liste d'entiers compris entre 0 et 25 en la liste de caractères correspondants.
5) Écrire une fonction `chiffre_cesar(texte, k)` qui, étant donné un texte (une chaîne de caractères), renvoie la texte chiffré en décalant chacun des caractères de k éléments (modulo 26). Utiliser les fonctions définies précédemment.

Déchiffrer un message crypté avec le chiffre de César est relativement simple. En effet, la lettre `e` est la lettre la plus présente dans la langue française. Ainsi, il suffit de chercher la lettre la plus présente dans le message crypté et supposer que cette lettre est initialement un `e`. On en déduit la valeur du décalage, puis le message initial.

6) Écrire une fonction `dechiffre_cesar1(texte)` qui renvoie le texte clair à partir du texte chiffré `texte`.

PRÉPAS SCIENTIFIQUES

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

CONFORME AU NOUVEAU PROGRAMME

Informatique TRONC COMMUN

L'ouvrage indispensable pour réussir les concours et faire la différence

➜ **TOUT LE COURS**
Pour maîtriser **l'intégralité du programme**, avec toutes les commandes Python et les axes de programmation majeurs à retenir

➜ **LES MÉTHODES PAS À PAS**
Pour acquérir **les techniques et astuces** de résolution des exercices

➜ **LES COMPÉTENCES NUMÉRIQUES**
Toutes **les simulations et résolutions numériques** avec Python

➜ **UN ENTRAÎNEMENT PROGRESSIF**
Pour vous préparer et vous tester efficacement avec **plus de 200 exercices inédits, adaptés ou issus de sujets de concours** : vrai/faux, applications, exercices d'approfondissement et sujets type concours

➜ **TOUS LES CORRIGÉS DÉTAILLÉS**
Pour comprendre **les étapes de résolution des exercices** et acquérir **les bons réflexes**

OFFERT EN LIGNE

+ **100 flashcards interactives** pour mémoriser autrement
+ Toutes les **synthèses à télécharger** pour des révisions nomades
+ Tous les **scripts Python interactifs** pour s'entraîner à coder

Des auteurs au cœur de l'enseignement et des attentes des élèves en CPGE

Dans la même collection :



Retrouvez notre collection complète ici :

ISBN : 978-2-311-21299-0