Julien Guillod

Programmation Python par la pratique

DUNOD

Table des matières

Chapitre 1

Introduction

Python est un langage de programmation phare dans le monde scientifique. Il est parfaitement adapté pour programmer des problèmes mathématiques. Cet ouvrage propose de se focaliser sur l'utilisation pratique du langage Python dans différents domaines des mathématiques : les suites, l'algèbre linéaire, l'intégration, la théorie des graphes, la recherche de zéros de fonctions, les probabilités, les statistiques, les équations différentielles, le calcul symbolique, et la théorie des nombres. À travers 40 exercices de difficulté croissante, et corrigés en détails, il permet d'avoir une bonne vision d'ensemble des possibilités d'utilisation de la programmation dans les mathématiques et d'être à même de résoudre des problèmes mathématiques complexes.

Il n'est pas nécessaire de faire les exercices dans l'ordre proposé, même si certains exercices font parfois appel à des notions vues dans des exercices précédents. Les exercices plus difficiles sont indiqués par des points d'exclamation :

— ! : plus long ou plus difficile ;

— !! : passablement long et complexe ;

— !!! : défi proposé sans correction.

Le séparateur décimal utilisé dans cet ouvrage est le point et non pas la virgule, afin de se conformer avec l'usage international employé par Python et éviter les confusions. Ces exercices servent de base aux travaux pratiques donnés à Sorbonne Université dans le cadre de la licence de mathématiques.

L'ensemble des codes sources de l'ouvrage est disponible en ligne à l'adresse : `https://python.guillod.org/`. Ce site est mis à jour régulièrement, aussi il se peut qu'il diffère du présent ouvrage dans le futur.

1.1 Remerciements

Merci à Marie Postel et Nicolas Lantos pour leurs relectures attentives de la première version de ce recueil et pour les nombreuses corrections et suggestions.

Merci également aux membres des équipes pédagogiques de Sorbonne Université ayant utilisé ces exercices pour leurs retours et contributions : Mathieu Barré, Constantin Bône, Jules Bonnard, Cédric Boutillier, Thibault Cimic, Jeanne Decayeux, Cécile Della Valle, Guillaume Duboc, Jean-Jil Duchamps, Johann Faouzi, Jean-Merwan Godon, Elise Grosjean, Cindy Guichard, Nicolas Lantos, Mathieu Mari, David Michel, Leo Miolane, Anouk Nicolopoulos, Arnaud Padrol, Diane Peurichard, Marie Postel, Xavier Poulot-Cazajous, Alexandre Rege, Othmane Safsafi, Emmanuel Schertzer, Agustín

Somacal, Didier Smets, Robin Strudel, Gauthier Tallec, Nicolas Thomas, Paul Vernhet, Jules Vidal et Raphaël Zanella.

Finalement merci aux étudiantes et aux étudiants ayant planché sur ces exercices pour leurs retours constructifs qui ont contribué à l'amélioration du présent recueil.

La lectrice attentive ou le lecteur attentif est remercié·e par avance pour tout signalement de coquilles ou autres.

1.2 Pourquoi Python ?

Python est un langage généraliste de programmation interprété qui a la particularité d'être très lisible et pragmatique. Il dispose d'une très grosse base de modules externes, notamment scientifiques, qui le rend particulièrement attractif pour programmer des problèmes mathématiques. Le fait que Python soit un langage interprété le rend plus lent que les langages compilés, il assure en revanche une grande rapidité de développement qui permet à l'humain de travailler un peu moins tandis que l'ordinateur devra travailler un peu plus. Cette particularité fait que Python est devenu l'un des principaux langages de programmation utilisés par les scientifiques.

1.3 Prérequis

Cet ouvrage n'a pas pour but premier d'exposer la syntaxe et les principes du langage Python, aussi les prérequis sont-ils d'en connaître les bases. Il existe de nombreuses ressources pour se mettre à jour en cas de besoin, par exemple :

— le cours en ligne *Python 3 : des fondamentaux aux concepts avancés du langage* d'Arnaud Legout et Thierry Parmentelat à suivre sur le site de *France Université Numérique* (`https://www.fun-mooc.fr/courses/course-v1:UCA+107001+session02/about`). Les vidéos sont également disponibles sur YouTube (`https://www.youtube.com/channel/UCIlUBOXnXjxdjmL_atU53kA`)

— l'ouvrage *Programmation en Python pour les sciences de la vie* de Patrick Fuchs et Pierre Poulain (`https://dunod.com/EAN/9782100796021`)

— le cours *1IN001* dispensé à Sorbonne Université (`http://www.licence.info.upmc.fr/lmd/licence/2020/ue/LU1IN001-2020oct/`)

Par ailleurs la réalisation des exercices demande d'avoir accès à un ordinateur ou un service en ligne disposant de Python 3.6 (ou plus récent) complété par les modules suivants : Numpy, Scipy, Sympy, Matplotlib, Numba, NetworkX et Pandas. L'emploi d'un éditeur de code permettant l'écriture en Python est aussi vivement conseillé. Il est ici suggéré d'utiliser Jupyter Lab, qui permet à la fois l'écriture des notebooks interactifs et des scripts et également l'ajout de ses propres solutions en dessous des énoncés, ce qui est très pratique. Il n'est pas indispensable d'utiliser Jupyter Lab, d'autres environnements sont aussi adaptés, notamment Spyder ou Jupyter Notebook.

Les sections suivantes décrivent comment installer et lancer l'environnement Python ou l'utiliser en ligne sans installation.

1.4 Documentation

Il n'est généralement pas utile (ni souhaitable) de connaître toutes les fonctions et subtilités du langage Python lors d'une utilisation occasionnelle. Par contre il est indispensable de savoir utiliser la documentation de manière efficace. La documentation officielle est disponible à l'adresse `https://docs.python.org/`. La langue et la version peuvent être sélectionnées en haut à gauche. Il est fortement conseillé de regarder comment la documentation est écrite et d'apprendre à l'utiliser.

1.5 Installation

Les personnes ne pouvant ou ne voulant pas installer Python peuvent directement se rendre à la section 1.6 pour des alternatives disponibles en ligne sans installation.

Il existe essentiellement quatre façons d'installer Python et les modules requis pour réaliser les exercices :

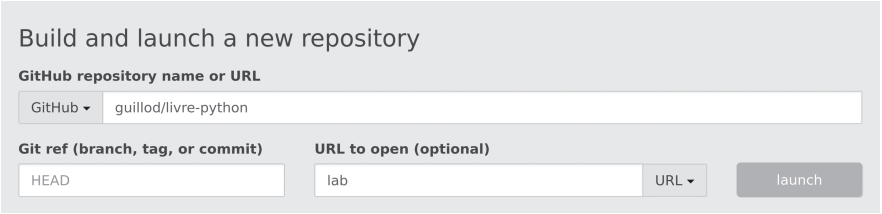
— **Anaconda** est une distribution Python complète, c'est-à-dire qu'elle installe directement une très grande quantité de modules (beaucoup plus que nécessaire pour faire les exercices suivants). L'avantage de cette installation est qu'elle est très simple ; son désavantage est qu'elle prend beaucoup d'espace disque. C'est la méthode à privilégier si vous êtes sous Windows ou MacOS et que vous n'avez pas de problème d'espace disque.

— **Miniconda** est une version légère d'Anaconda, qui installe par défaut uniquement la base. L'avantage est qu'elle prend peu d'espace disque, mais elle requiert une action supplémentaire pour installer les modules requis pour faire les exercices. C'est la méthode à privilégier si vous êtes sous Windows ou MacOS et que vous avez peu d'espace disque disponible.

— **Dépôts Linux :** la plupart des distributions Linux permettent d'installer Python et les modules de base directement à partir des dépôts de paquets qui les accompagnent. C'est la méthode privilégiée sous Linux.

— **Pip** est un gestionnaire de paquets pour Python. C'est la méthode à privilégier pour ajouter un module si Python est déjà installé par votre système d'exploitation, et que ce module n'est pas inclus dans les paquets de votre distribution. Cette méthode permet une gestion plus fine et avancée des modules installés que ce qui est proposé avec les méthodes précédentes.

Installation avec Anaconda : La façon la plus simple d'installer Python 3 et toutes les dépendances nécessaires sous Windows et MacOS est d'installer Anaconda. Le désavantage d'Anaconda est que son installation prend beaucoup d'espace disque car



Pour utiliser Jupyter Lab plutôt que Jupyter Nootebook sur GESIS, il suffit de remplacer `tree` par `lab` à la fin de l'URL.

Sinon différents services offrent la possibilité d'utiliser gratuitement Jupyter Lab après création d'un compte :

— CoCalc (`https://cocalc.com/`)

— Google Colaboratory (`https://colab.research.google.com/`)

1.7 Utilisation de Jupyter Lab

Une fois Jupyter Lab lancé, la fenêtre représentée à la Figure 1.1 doit apparaître dans un navigateur.

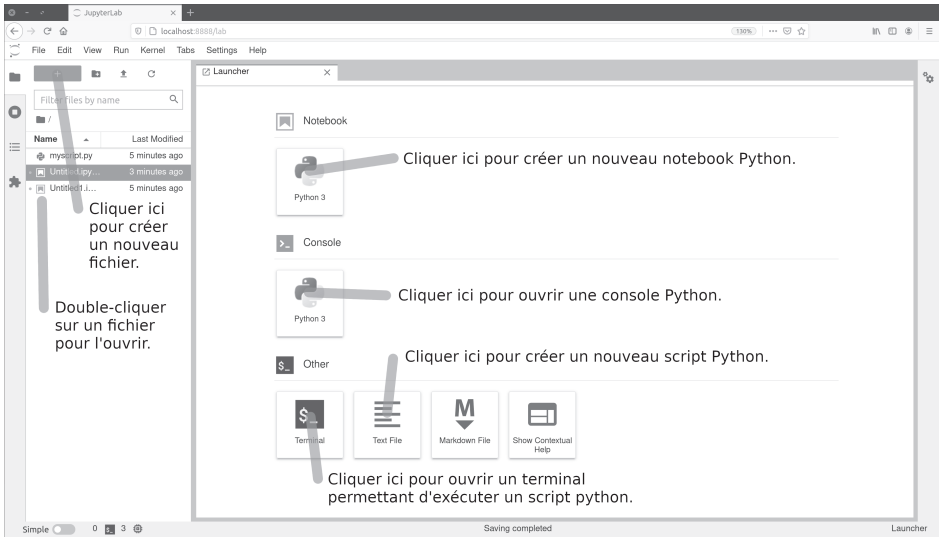


Figure 1.1 – Fenêtre de lancement de Jupyter Lab

Jupyter Lab permet essentiellement de traiter trois types de documents : les **notebooks**, les **scripts** et les **terminaux**. Un notebook est constitué de cellules qui peuvent contenir soit du code soit du texte au format Markdown. Les cellules de code peuvent être évaluées de manière interactive à la demande, ce qui permet une grande flexibilité. Les cellules de texte peuvent contenir des commentaires, des titres ou des formules LaTeX comme représenté sur la Figure 1.2.

Un script Python est simplement un fichier texte contenant des instructions Python. Il s'exécute en entier de A à Z et il n'est pas possible d'interagir interactivement avec lui pendant son exécution (à moins que cela n'ait été explicitement programmé). Pour exécuter un script Python il est nécessaire d'ouvrir un terminal.

Commandes de base :

— Créer un nouveau fichier : cliquer sur le bouton «+» situé en haut à gauche, puis choisir le type de fichier à créer.

— Renommer un fichier : cliquer avec le second bouton de la souris sur le titre du notebook (soit dans l'onglet, soit dans la liste des fichiers).

— Changer le type de cellules : menu déroulant permettant de choisir entre «Code» et «Markdown».

— Exécuter une cellule de code : combinaison des touches `SHIFT+ENTRÉE`.

— Mettre en forme une cellule de texte : combinaison des touches `SHIFT+ENTRÉE`.

— Modifier une cellule de texte : double-cliquer sur la cellule.

— Exécuter un script : taper `python nomduscript.py` dans un terminal pour exécuter le script `nomduscript.py`.

— Réorganiser les cellules : cliquer-déposer.

— Juxtaposer des onglets : cliquer-déposer.

La documentation détaillée de Jupyter Lab est disponible à l'adresse : `https://jupyterlab.readthedocs.io/`.

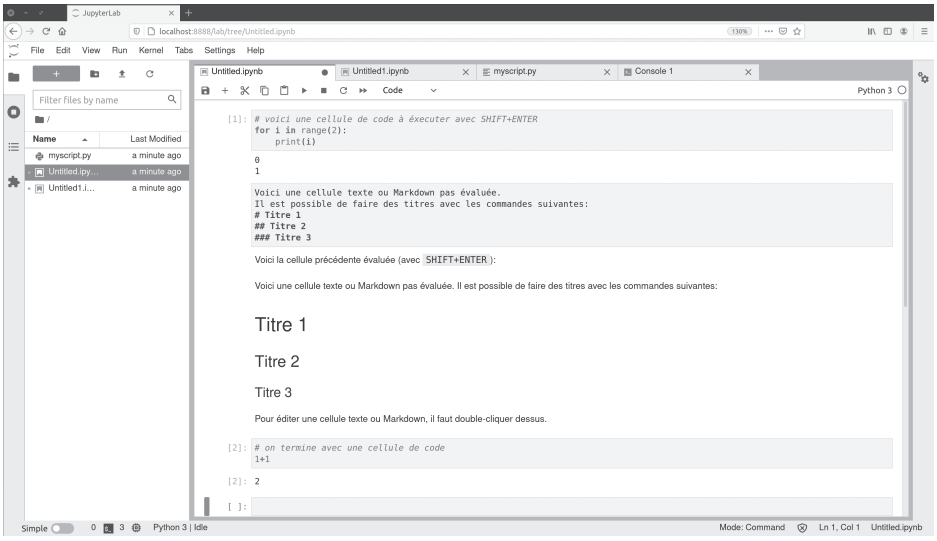


Figure 1.2 – Un notebook Jupyter est composé de plusieurs cellules ; ici une cellule de code suivie d'une cellule de texte non évaluée puis la même cellule évaluée et enfin une dernière cellule de code.

Chapitre 2

Structures de données

Pour représenter des structures de données, Python propose quatre types de base : les listes (type `list`), les tuples (type `tuple`), les ensembles (type `set`) et les dictionnaires (type `dict`). Le but de ce chapitre est de montrer les différences fondamentales entre ces structures de données et d'expliquer à quoi elles sont le plus adaptées. La documentation détaillée sur les structures de données est disponible à l'adresse : `https://docs.python.org/fr/3/tutorial/datastructures.html`.

Concepts abordés

— structures de données (liste, tuple, ensemble, dictionnaire)

— types mutable et immutable

— type hashable

— compréhensions de liste, ensemble et dictionnaire

— suites numériques

Exercices

Exercice 2.1. Listes

Une liste est une structure permettant de stocker des éléments hétérogènes :

```
list0 = [0, 5.4, "chaîne", True]
```

Les listes sont mutables, c'est-à-dire qu'il est possible d'y modifier un élément, d'en rajouter un ou d'en supprimer un sans avoir à redéfinir toute la liste.

```
list0[3] = False # remplace True par False
list0.append("nouveau") # ajoute la chaîne de caractères "nouveau"
 ↪  à la liste
list0.insert(2, 34) # insère 34 à la place 2
list0.remove(0) # enlève 0
```

En particulier, il faut faire attention en copiant une liste. Si l'on exécute le code suivant :

```
list1 = list0
list1[2] = "change"
list0
```

alors `list0` est aussi modifié et est égal à `list1`. Pour créer une vraie copie, il faut utiliser le code suivant :

```
list2 = list0.copy()
list2[2] = "rechange"
list0
```

qui ne modifie pas `list0`. À noter qu'il est possible de modifier les éléments d'une liste à l'intérieur d'une fonction :

```
def f(l):
    l[0] = 0
f(list0)
```

Enfin il est possible de créer des listes à l'aide de la compréhension de listes :

```
list1 = [2*i+1 for i in range(10)]
```

a) Chercher dans la documentation la syntaxe pour concaténer deux listes.

Indication : Voir la documentation disponible à l'adresse : `https://docs.python.org/fr/3/library/stdtypes.html#sequence-types-list-tuple-range`.

b) Chercher dans la documentation la syntaxe pour extraire une tranche d'une liste, c'est-à-dire : si `a` est par exemple une liste de longueur 10, retourner les éléments de 6 à 9.

Indication : Voir la documentation disponible à l'adresse : `https://docs.python.org/fr/3/library/stdtypes.html#sequence-types-list-tuple-range`.

c) Chercher dans la documentation la syntaxe pour retourner la longueur d'une liste.

d) Écrire une fonction `fibonacci(N)` qui retourne la liste des $N \geq 2$ premiers termes de la suite de Fibonacci définie par $u_{n+2} = u_{n+1} + u_n$ avec $u_0 = 0$ et $u_1 = 1$.

e) Écrire une fonction `pascal(N)` qui retourne la N-ième ligne du triangle de Pascal :

$$\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}$$

f) Soit les suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ définies par $u_0 = 1$, $v_0 = 1$, et

$$u_{n+1} = u_n + v_n\,, \qquad\qquad v_{n+1} = 2u_n - v_n\,,$$

pour $n \geq 0$. Calculer u_{100} et v_{100}.

Réponse : $u_{100} = v_{100} = 71789798769185258877024 9$.

g) Écrire une fonction `vk(n0,K)`, qui pour deux entiers n_0 et $K \geq 1$ calcule la suite des valeurs v_k définies par $v_0 = n_0$ et

$$v_{k+1} = \begin{cases} 3v_k + 1 & \text{si } v_k \text{ est impair,} \\ \frac{v_k}{2} & \text{si } v_k \text{ est pair,} \end{cases}$$

pour $0 \leq k < K$. Pour $K = 1\,000$ et diverses valeurs de $n_0 \in \{10, 100, 1\,000, 10\,000\}$, afficher les cinq dernières valeurs calculées, c'est-à-dire $(v_{K-4}, v_{K-3}, v_{K-2}, v_{K-1}, v_K)$.

Réponse : Les assertions suivantes sont vraies :

```
vk(10,1000) == [1, 4, 2, 1, 4]
vk(100,1000) == [2, 1, 4, 2, 1]
vk(1000,1000) == [1, 4, 2, 1, 4]
vk(10000,1000) == [4, 2, 1, 4, 2]
```

Exercice 2.2. Tuples

Les tuples permettent tout comme les listes de stocker des éléments hétérogènes :

```
tuple0 = (0, 5.4, "chaîne", True)
```

Mais au contraire des listes, les tuples ne sont pas mutables. Il n'est pas possible d'y modifier un élément, d'en rajouter un ou d'en supprimer un sans redéfinir tout le tuple. L'avantage d'un tuple sur une liste est qu'il tuple est hashable, c'est-à-dire qu'il peut être utilisé comme clef dans un dictionnaire.

Enfin il est possible d'affecter des variables à l'intérieur d'un tuple, par exemple :

```
(a,b) = (1,9)
```

Cela est en particulier très utile pour échanger deux variables sans avoir à utiliser une variable supplémentaire :

```
(a,b) = (b,a)
```

a) Vérifier qu'un tuple est bien immutable.

b) Définir une fonction `mdlast(lst,val)` ayant pour argument une liste de tuples d'entiers `lst` et un entier `val` et retourner la liste de tuples avec le dernier élément de chaque tuple remplacé par `val`. Par exemple si `lst = [(10, 20), (30, 40, 50, 60), (70, 80, 90)]` alors `mdlast(lst,100)` doit retourner `[(10, 100), (30, 40, 50, 100), (70, 80, 100)]`.

c) Comment convertir un tuple en liste et réciproquement ?

Exercice 2.3. Ensembles

Les ensembles permettent de stocker des éléments hétérogènes au sens mathématique de la théorie des ensembles :

```
set0 = {0, 5.4, "chaîne", True}
```

Il est possible de tester si un élément appartient à un ensemble :

```
if "chaîne" in set0:
    print("dedans")
```

Les ensembles sont mutables, il est donc possible de rajouter ou retirer un élément d'un ensemble :

```
set0.add(18) # ajoute 18 à l'ensemble
set0.add(0) # ajoute 0 à l'ensemble (ne fait rien car 0 est déjà
 ↪  dedans)
set0.remove("chaîne") # retire "chaîne de l'ensemble
```

En revanche les ensembles ne peuvent contenir que des éléments hashables, c'est-à-dire immutables. En particulier un ensemble ne peut pas contenir un autre ensemble :

```
set1 = {{1,2},{3},{4}}
TypeError: unhashable type: 'set'
```

À noter qu'il existe également en Python des ensembles immutables `frozenset` :

```
frozenset0 = frozenset([0, 5.4, "chaîne", True])
```

Une chaîne de caractères peut être transformée en ensemble :

```
set1 = set('abracadabra')
```

Comme pour les listes, il est possible de faire des compréhensions d'ensembles :

```
set2 = {x for x in 'abracadabra' if x not in 'abc'}
```

Dans cet exemple, les chaînes de caractères sont automatiquement transformées en ensemble. À noter que l'ensemble vide est défini par `set()`.

a) Définir une fonction `divisible(n)` qui retourne l'ensemble des nombres entiers divisibles par `n` inférieurs ou égaux à 100.

b) Chercher dans la documentation comment faire l'intersection, l'union et la différence de deux ensembles. Déterminer les nombres inférieurs ou égaux à 100 qui sont non divisibles par 2 mais divisibles par 3 et 5.

Indication : Voir la documentation de `set` *à l'adresse :* `https://docs.python.org/fr/3/library/stdtypes.html#set`.

Exercice 2.4. Dictionnaires

Les dictionnaires sont une structure permettant de stocker des éléments hétérogènes indexés par des clefs (elles aussi hétérogènes) :

```
dict0 = {"pommes": 0, "poires": 4, 12: 2}
```

Les éléments d'un dictionnaire sont accessibles par les clefs :

```
dict0["pommes"]
dict0[12]
```

Un dictionnaire peut être vu comme un tableau associatif associant à chaque clef une valeur. La liste des clefs et celle des valeurs sont accessibles respectivement avec `dict0.keys()` et `dict0.values()`. Les dictionnaires sont mutables, il est donc possible de modifier une association clef-valeur et d'en rajouter ou supprimer une :

```
dict0["pommes"] = 3 # modifie la valeur associée à pommes
dict0["oranges"] = "beaucoup" # rajoute orange comme clef avec la
 ↪  valeur "beaucoup"
del dict0["poires"] # supprime le couple clef-valeur associé à
 ↪  poires
dict0.pop("pommes") # supprime le couple clef-valeur associé à
 ↪  pommes
```

Bien qu'un dictionnaire soit mutable, les clefs qui le composent doivent être des objets hashables, c'est-à-dire immutables. Ainsi une liste ou un ensemble ne peuvent pas servir de clefs dans un dictionnaire :

```
dict0[list0] = "test"
TypeError: unhashable type: 'list'
dict0[set0] = "retest"
TypeError: unhashable type: 'set'
```

En revanche il est possible d'avoir un tuple ou un frozenset comme clef :

```
dict0[tuple0] = "test"
dict0[frozenset0] = "rest"
```

d'où l'intérêt des frozensets. Comme pour les listes et les ensembles, il est possible de faire des compréhensions de dictionnaires :

```
dict1 = {x: x**2 for x in range(5)}
```

Finalement une chose intéressante avec les dictionnaires est l'unpaking illustré par l'exemple suivant :

```
def add(a=0, b=0):
    return a + b
d = {'a': 2, 'b': 3}
add(**d)
```

a) Comment définir un dictionnaire vide ?

b) Comment concaténer plusieurs dictionnaires entre eux ?

c) On considère une liste de mots :

```
mots = ['Abricot', 'Airelle', 'Ananas', 'Banane', 'Cassis',
 ↪  'Cerise', 'Citron', 'Clémentine', 'Coing', 'Datte', 'Fraise',
 ↪  'Framboise', 'Grenade', 'Groseille', 'Kaki', 'Kiwi', 'Litchi',
 ↪  'Mandarine', 'Mangue', 'Melon', 'Mirabelle', 'Nectarine',
 ↪  'Orange', 'Pamplemousse', 'Papaye', 'Pêche', 'Poire', 'Pomme',
 ↪  'Prune', 'Raisin']
```

Écrire une fonction `position(mots, x, n)` qui retourne la liste des mots ayant le caractère `x` comme `n`-ième lettre.

Réponse : Par exemple `position(mots,'e',4)` *doit retourner la liste :*

```
['Clémentine', 'Datte', 'Groseille', 'Pêche', 'Poire', 'Pomme',
 ↪  'Prune']
```

d) En imaginant que la liste des mots soit très longue, alors à chaque évaluation de la fonction `position` l'ensemble des mots est parcouru, ce qui prend pas mal de temps. Pour améliorer cela, construire un dictionnaire `mots_dict` ayant pour clefs les tuples `(x,n)` et comme valeurs la liste des mots ayant le caractère `x` comme `n`-ième lettre, c'est-à-dire tel que `mots_dict[x,n]` retourne la même chose que `position(mots, x , n)` à l'ordre près. Ainsi la liste `mots` n'est parcourue qu'une seule fois lors de la construction du dictionnaire et ensuite l'évaluation du dictionnaire est extrêmement rapide pour n'importe quelle requête.

Solutions

Solution 2.1. Listes

a) L'opérateur `+` permet la concaténation de listes :

```
list1 = [1,3,4,"a",29]
list2 = ["e",37,2]
list1 + list2
```

b) Il faut utiliser `list` pour transformer le type `range` en liste :

```
a = list(range(10))
a[6:10]
```

c) La fonction `len` retourne la longueur d'une liste :

```
list3 = [2,5,"19",4,8,"R"]
len(list3)
```

d) L'idée est de construire une liste élément par élément à partir des deux premiers éléments :

```
def fibonacci(N):
    # initialisation
    out = [0,1]
    for i in range(2,N):
        # ajoute à la liste l'élément formé par la somme des deux
        ↪ précédents
        out.append(out[i-1]+out[i-2])
    return out
```

e) En utilisant une fonction récursive, le problème se réduit à écrire la logique permettant de passer d'une ligne à l'autre :

```
def pascal(N):
    # initialisation pour la récurrence
    if N == 1:
        return [1]
    else:
        # ligne précédente
        previous = pascal(N-1)
        # renvoie la nouvelle ligne formée à partir de la ligne
        ↪ précédente
        return [1] + [previous[i] + previous[i+1] for i in
        ↪ range(N-2)] + [1]
```

f) L'astuce suivante permet de définir les nouvelles valeurs de `u` et `v` en une seule fois, sinon il faut utiliser une variable supplémentaire pour sauvegarder une des anciennes valeurs :

```
def uv(n):
    # initialisation du couple (u,v)
    u = 1
    v = 1
    for i in range(n):
        # permet de définir les nouveaux u et v en même temps,
        u,v = u+v, 2*u-v
    return [u,v]
print(uv(100))
```