Cours de Python

https://python.sdv.univ-paris-diderot.fr/

Patrick Fuchs et **Pierre Poulain**
prénom [point] nom [arobase] univ-paris-diderot [point] fr

version du 9 novembre 2017

Université Paris Diderot-Paris 7, Paris, France

Table des matières

Chapitre 1

Introduction

1.1 Quelques mots sur l'origine de ce cours

Ce cours a été conçu à l'origine pour les étudiants débutants en programmation Python des filières de biologie et de biochimie de l'Université Paris Diderot - Paris 7 ; et plus spécialement pour les étudiants du master Biologie Informatique.

Ce cours est basé sur la version 3 de Python, version recommandée par la communauté scientifique. Des références à l'ancienne version, Python 2, seront néanmoins régulièrement apportées.

Si vous relevez des erreurs à la lecture de ce document, merci de nous les signaler.

Le cours est disponible en version HTML et PDF.

1.2 Remerciements

Un grand merci à Sander du *Centre for Molecular and Biomolecular Informatic* de Nijmegen aux Pays-Bas pour la toute première version de ce cours.

Merci également à tous les contributeurs, occasionels ou réguliers : Jennifer Becq, Virginie Martiny, Romain Laurent, Benoist Laurent, Benjamin Boyer, Hubert Santuz, Catherine Lesourd, Philippe Label, Rémi Cuchillo, Cédric Gageat, Philibert Malbranche, Mikaël Naveau, Amélie Bacle, Alexandra Moine-Franel.

Nous remercions aussi Denis Mestivier de qui nous nous sommes inspirés pour certains exercices.

Enfin, merci à vous tous, les curieux de Python, qui avez été nombreux à nous envoyer des retours sur ce cours, à nous suggérer des améliorations et à nous signaler des coquilles.

De nombreuses personnes nous ont aussi demandé les corrections des exercices. Nous ne les mettons pas sur le site afin d'éviter la tentation de les regarder trop vite, mais vous pouvez nous écrire et nous vous les enverrons.

1.3 Avant de commencer

Avant de débuter ce cours, voici quelques indications générales qui pourront vous servir pour la suite.

— Familiarisez-vous avec le site www.python.org. Il contient énormément d'informations et de liens sur Python et vous permet en outre de le télécharger pour différentes plateformes (Linux, Mac, Windows). La page d'index des modules est particulièrement utile.

— Pour aller plus loin avec Python, Gérard Swinnen a écrit un très bon livre en français intitulé *Apprendre à programmer avec Python 3* et téléchargeable gratuitement. Les éditions Eyrolles proposent également la version papier de cet ouvrage.
— Ce cours est basé sur une utilisation de Python sous Linux mais il est parfaitement transposable aux systèmes d'exploitation Windows et Mac.
— L'apprentissage d'un langage informatique comme Python va nécessiter d'écrire des lignes de codes à l'aide d'un éditeur de texte. Si vous êtes débutants, nous vous conseillons vivement d'utiliser *gedit* ou *geany*, qui sont les plus proches des éditeurs que l'on peut trouver sous Windows (*notepad*). Des éditeurs comme *emacs* et *vi* sont très puissants mais nécessitent un apprentissage particulier.

1.4 Premier contact avec Python sous Linux

Python est un langage interprété, c'est-à-dire que chaque ligne de code est lue puis interprétée afin d'être exécutée par l'ordinateur. Pour vous en rendre compte, lancez la commande :

```
python3
```

Celle-ci va lancer l'**interpréteur Python**. Vous devriez obtenir quelque chose de ce style :

```
pierre@jeera:~$ python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Le bloc `pierre@jeera:~$` représente l'invite de commande de votre *shell* sous Linux. Par la suite, cette invite de commande sera représentée simplement par le caractère `$`.

Le triple chevron `>>>` est l'invite de commande de Python (*prompt* en anglais), ce qui signifie que Python attend une commande. Tapez par exemple l'instruction

```
print("Hello world !")
```

puis validez votre commande en appuyant sur la touche `Entrée`.

Python a exécuté la commande directement et a affiché le texte `Hello world !`. Il attend ensuite votre prochaine instruction en affichant l'invite de l'interpréteur Python (`>>>`). En résumé, voici ce qui a du apparaître sur votre écran :

```
>>> print("Hello world !")
Hello world !
>>>
```

Vous pouvez refaire un nouvel essai en vous servant cette fois de l'interpréteur comme d'une calculatrice.

```
>>> 1+1
2
>>> 6*3
18
```

À ce stade, vous pouvez entrer une autre commande ou bien quitter l'interpréteur Python, soit en tapant la commande `exit()` puis en validant en appuyant sur la touche `Entrée`, soit en pressant simultanément les touches `Ctrl` et `D`.

Finalement l'interpréteur Python est un système interactif dans lequel vous pouvez entrer des commandes, que Python exécutera sous vos yeux (au moment où vous validerez la commande en tapant sur `Entrée`).

Il existe de nombreux autres langages interprétés tels que Perl ou R. Le gros avantage est que l'on peut directement tester une commande à l'aide de l'interpréteur, ce qui est très utile pour débugger (c'est-à-dire corriger les éventuelles erreurs d'un programme). Gardez bien en mémoire cette propriété de Python qui pourra parfois vous faire gagner un temps précieux !

1.5 Premier programme Python

Bien sûr, l'interpréteur présente vite des limites dès lors que l'on veut exécuter une suite d'instructions plus complexe. Comme tout langage informatique, on peut enregistrer ces instructions dans un fichier, que l'on appelle communément un script Python.

Pour reprendre l'exemple précédent, ouvrez un éditeur de texte (par exemple *gedit* ou *nedit*) et entrez le code suivant.

```
print('Hello World !')
```

Ensuite enregistrez votre fichier sous le nom *test.py*, puis quittez l'éditeur de texte. L'extension standard des scripts Python est `.py`. Pour exécuter votre script, vous avez deux moyens.

1.5.1 Appel de l'interpréteur

Donnez le nom de votre script comme argument à la commande Python :

```
$ python3 test.py
Hello World !
$
```

1.5.2 Appel direct du script

Pour appeler directement votre script Python, deux opérations sont nécessaires :

1. Précisez au *shell* la localisation de l'interpréteur Python en indiquant dans la première ligne du script :

   ```
   #! /usr/bin/env python3
   ```

   Dans notre exemple, le script *test.py* contient alors le texte suivant :

   ```
   #!/usr/bin/env python3

   print('Hello World !')
   ```

2. Rendez votre script Python exécutable en tapant :

   ```
   chmod +x test.py
   ```

Pour exécuter votre script, tapez son nom précédé des deux caractères **./** (afin de préciser au *shell* où se trouve votre script) :

```
$ ./test.py
Hello World !
$
```

1.6 Commentaires

Dans un script, tout ce qui suit le caractère `#` est ignoré par Python jusqu'à la fin de la ligne et est considéré comme un commentaire. Une exception notable est la première ligne de votre script qui peut être

```
#! /usr/bin/env python3
```

et qui a alors une signification particulière pour Python.

Les commentaires sont indispensables pour que vous puissiez annoter votre code. Il faut absolument les utiliser pour décrire les principales parties de votre code dans un langage humain.

```
#! /usr/bin/env python3

# votre premier script Python
print('Hello World !')

# d'autres commandes plus utiles pourraient suivre
```

1.7 Notion de bloc d'instructions et d'indentation

Pour terminer ce chapitre d'introduction, nous allons aborder dès maintenant les notions de **bloc d'instructions** et d'**indentation**.

En programmation, il est courant de répéter un certain nombre de choses (avec les boucles, voir chapitre 5) ou de faire des choix (avec les tests, voir chapitre 6).

Par exemple, imaginez que vous souhaitiez répéter 10 fois 3 instructions différentes, les unes à la suite des autres. Regardez l'exemple suivant en pseudo-code :

```
instruction_qui_indique_à_Python_de_répéter_10_fois_ce_qui_suit:
    sous_instruction1
    sous_instruction2
    sous_instruction3
instruction_suivante
```

La première ligne correspond à une instruction qui va indiquer à Python de répéter 10 fois d'autres instructions (il s'agit d'une boucle, on verra le nom de la commande exacte plus tard). Dans le pseudo-code ci-dessus, il y a 3 instructions à répéter, nommées `sous_instruction1` à `sous_instruction3`.

Notez bien les détails de la syntaxe. La première ligne indique que l'on veut répéter une ou plusieurs instructions, elle se termine par `:`. Ce symbole `:` indique à Python qu'il doit attendre un bloc d'instructions, c'est-à-dire un certains nombres de sous-instructions à répéter. Python reconnaît un bloc d'instructions car il est indenté. L'indentation est le décalage d'un ou plusieurs espaces ou tabulations des instructions `sous_instruction1` à `sous_instruction3`, par rapport à `instruction_qui_indique_à_Python_de_répéter_10_fois_ce_qui_suit:`.

Pratiquement, l'indentation en Python doit être homogène (soit des espaces, soit des tabulations, mais pas un mélange des deux). Une indentation avec 4 espaces est le style d'indentation le plus traditionnel et celui qui permet une bonne lisibilité du code.

Enfin, la ligne `instruction_suivante` sera exécutée une fois que le bloc d'instructions sera terminé.

Si tout cela vous semble un peu fastidieux, ne vous inquiétez pas. Vous allez comprendre tous ces détails, et surtout les acquérir, en continuant ce cours chapitre par chapitre.

1.8 Python 2 ou Python 3 ?

Ce cours est basé sur la version 3 de Python, qui est maintenant devenu un standard. La plupart des projets importants ont migré vers Python 3. Par ailleurs Python 2 cessera d'être maintenu au delà de mi-2020. Python 3 est donc la version que nous vous recommandons fortement.

Si néanmoins, vous devez un jour travailler sur un ancien programme écrit en Python 2, sachez qu'il existe quelques différences importantes entre Python 2 et Python 3. Le chapitre 19 *Pour aller plus loin* vous donnera plus de précisions.

Chapitre 2

Variables

2.1 Définition d'une variable

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

En Python, la **déclaration** d'une variable et son **initialisation** (c'est-à-dire la première valeur que l'on va stocker dedans) se font en même temps. Pour vous en convaincre, testez les instructions suivantes après avoir lancé l'interpréteur :

```
>>> x = 2
>>> x
2
```

Dans cet exemple, nous avons déclaré, puis initialisé la variable `x` avec la valeur 2. Notez bien qu'en réalité, il s'est passé plusieurs choses :

- Python a *deviné* que la variable était un entier. On dit que Python est un langage au *typage dynamique*.
- Python a alloué (*i.e.* réservé) l'espace en mémoire pour y accueillir un entier (chaque type de variable prend plus ou moins d'espace en mémoire), et a fait en sorte qu'on puisse retrouver la variable sous le nom `x`
- Python a assigné la valeur 2 à la variable `x`.

Dans certains autres langages, il faut coder ces différentes étapes une par une (en C par exemple). Python étant un langage dit de *haut niveau*, la simple instruction `x = 2` a suffi à réaliser les 3 étapes en une fois !

Ensuite, l'interpréteur nous a permis de connaître le contenu de la variable juste en tapant son nom. Retenez ceci car c'est une **spécificité de l'interpréteur Python**, très pratique pour chasser les erreurs (*debugging*) dans un programme. Par contre, la ligne d'un script Python qui contient seulement le nom d'une variable (sans aucune autre indication) n'affichera pas la valeur de la variable à l'écran (pour autant ceci reste valide et ne génèrera pas d'erreur).

Dernière chose, l'opérateur d'affectation `=` s'utilise dans un certain sens : par exemple `x = 2` signifie qu'on attribue la valeur située à droite de l'opérateur `=` (`2`) à la variable située à gauche (`x`). Certains autres langages comme **R** utilise les symboles `<-` pour rendre les choses plus explicites, par exemple `x <- 2`.

Si on a `x = y - 3`, l'opération `y - 3` est d'abord évaluée et ensuite le résultat de cette opération est affecté à la variable `x`.

2.2 Les types de variables

Le **type** d'une variable correspond à la nature de celle-ci. Les trois types principaux dont nous aurons besoin dans un premier temps sont les entiers (*integer* ou *int*), les réels (*float*) et les chaînes de caractères (*string* ou *str*). Bien sûr, il existe de nombreux autres types (par exemple, les nombres complexes), c'est d'ailleurs un des gros avantages de Python (si vous n'êtes pas effrayés, vous pouvez vous en rendre compte ici).

Dans l'exemple précédent, nous avons stocké un nombre entier (*int*) dans la variable `x`, mais il est tout a fait possible de stocker des nombres réels (*float*), des chaînes de caractères (*string* ou *str*) ou plein d'autres types de variables que nous verrons par la suite :

```
>>> y = 3.14
>>> y
3.14
>>> a = "bonjour"
>>> a
'bonjour'
>>> b = 'salut'
>>> b
'salut'
>>> c = '''girafe'''
>>> c
'girafe'
```

Vous remarquez que Python reconnaît certains types de variable automatiquement (entier, réel). Par contre, pour une chaîne de caractères, il faut l'entourer de guillemets (simples, doubles voire trois guillemets successifs simples ou doubles) afin d'indiquer à Python le début et la fin de la chaîne.

2.3 Nommage des variables

Le nom des variable en Python peut-être constitué de lettres minuscules (`a` à `z`), de lettres majuscules (`A` à `Z`), de nombres (`0` à `9`) ou du caractère souligné (`_`).

Néanmoins, un nom de variable ne doit pas débuter ni par un chiffre, ni par `_` et ne peut pas contenir de caractère accentué. Il faut absolument éviter d'utiliser un mot "réservé" par Python comme nom de variable (par exemple : `print`, `range`, `for`, `from`, etc.).

Python est sensible à la casse, ce qui signifie que les variables `TesT`, `test` ou `TEST` sont différentes. Enfin, vous ne pouvez pas utilisez d'espace dans un nom de variable.

2.4 Opérations

2.4.1 Opérations sur les types numériques

Les quatre opérations de base se font de manière simple sur les types numériques (nombres entiers et réels) :

```
>>> x = 45
>>> x + 2
47
>>> y = 2.5
```

```
>>> x + y
47.5
>>> (x * 10) / y
180.0
```

Remarquez toutefois que si vous mélangez les types entiers et réels, le résultat est renvoyé comme un réel (car ce type est plus général). Par ailleurs, l'utilisation de parenthèses permet de gérer les priorités.

L'opérateur puissance utilise le symbole `**`. Pour obtenir le reste d'une division entière (voir ici pour un petit rappel sur la division entière), on utilise le symbole modulo `%` :

```
>>> 2**3
8
>>> 5 % 4
1
>>> 8 % 4
0
```

Les symboles `+`, `-`, `*`, `/`, `**` et `%` sont appelés **opérateurs**, car ils permettent de faire des opérations sur les variables.

2.4.2 Opérations sur les chaînes de caractères

Pour les chaînes de caractères, deux opérations sont possibles, l'addition et la multiplication :

```
>>> chaine = "Salut"
>>> chaine
'Salut'
>>> chaine + " Python"
'Salut Python'
>>> chaine * 3
'SalutSalutSalut'
```

L'opérateur d'addition `+` permet de concaténer (assembler) deux chaînes de caractères et l'opérateur de multiplication `*` permet de dupliquer plusieurs fois une chaîne.

Attention : Vous voyez que les opérateurs `+` et `*` se comportent différemment selon s'il s'agit d'entiers ou de chaînes de caractères : `2 + 2` est un addition, `'2' + '2'` est une concaténation. On appelle ce comportement **surcharge des opérateurs**. Nous serons amenés à revoir cette notion dans le chapitre 18 sur les classes.

2.4.3 Opérations illicites

Attention à ne pas faire d'opération illicite car vous obtiendriez un message d'erreur :

```
>>> 'toto' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Notez que Python vous donne le maximum d'information dans son message d'erreur. Dans l'exemple précédent, il vous indique que vous ne pouvez pas mélanger des objets de type `str` (*string*, donc des chaînes de caractères) avec des objets de type `int` (donc des entiers), ce qui est assez logique.

2.5 La fonction type()

Si vous ne vous souvenez plus du type d'une variable, utilisez la fonction `type()` qui vous le rappellera.

```
>>> x = 2
>>> type(x)
<class 'int'>
>>> y = 2.0
>>> type(y)
<class 'float'>
>>> z = '2'
>>> type(z)
<class 'str'>
```

Faites bien attention, car pour Python, la valeur `2` (nombre entier) est différente de `2.0` (nombre réel), de même que `2` (nombre entier) est différent de `'2'` (chaîne de caractères). Nous verrons plus tard ce que signifie le mot *class*.

2.6 Conversion de types

Dans tout langage de programmation, on est souvent amené à convertir les types, c'est-à-dire passer d'un type numérique à une chaîne de caractères ou vice-versa. En Python, rien de plus simple avec les fonctions `int()`, `float()` et `str()`. Pour vous en convaincre, regardez ces exemples :

```
>>> i = 3
>>> str(i)
'3'
>>> i = '456'
>>> int(i)
456
>>> float(i)
456.0
>>> i = '3.1416'
>>> float(i)
3.1416
```

On verra au chapitre 7 sur les fichiers que ces conversions sont essentielles. En effet, lorsqu'on lit ou écrit des nombres dans un fichier, ils sont considérés comme du texte.

Toute conversion d'une variable d'un type en un autre est appelé *casting* en anglais, il se peut que vous croisiez ce terme si vous allez consulter d'autres ressources.

2.7 Note sur la division

Notez bien qu'en Python 3, la division de nombres entiers renvoie par défaut un nombre réel (*float*) :

```
>>> x = 3 / 4
>>> x
0.75
>>> type(x)
<class 'float'>
```

Attention, ceci n'était pas le cas en Python 2. Pour en savoir plus sur ce point, vous pouvez consulter la section *Pour aller plus loin*

2.8 Note sur le vocabulaire et la syntaxe

Nous avons vu dans ce chapitre la notion de **variable** qui est commune à tous les langages de programmation. Toutefois, Python est un langage dit **orienté objet**, il se peut que dans la suite du cours nous employions ce mot *objet* pour désigner une variable. Par exemple *variable de type entier* sera équivalent à un *objet de type entier*. Nous verrons ce que le mot *objet* signifie réellement plus tard (tout comme le mot *class*).

Par ailleurs, nous avons rencontré plusieurs fois des **fonctions** dans ce chapitre, avec `type(x)`, `int(x)`, `float(x)` et `str(x)`. Dans le chapitre 1 nous avons également vu la fonction `print()`. On reconnait qu'il s'agit d'une fonction au nom - par exemple `type` - suivi de parenthèses `()`. En Python la syntaxe générale est `fonction()`. La variable `x` entre les parenthèses est appelé **argument** que l'on passe à la fonction. Dans `type(2)` c'est l'entier `2` qui est l'argument passé à la fonction. Pour l'instant on retiendra qu'une fonction est une sorte de *boite* à qui on passe un *argument* et qui renvoie un *résultat* ou plus généralement un objet. Par exemple, la fonction `type()` renvoie le type de la variable qu'on lui a passé en argument.

Si ces notions vous font peur, ne vous inquiétez pas, au fur et à mesure que vous avancerez dans le cours tout deviendra limpide.

Chapitre 3

Affichage

Nous avons déjà vu au chapitre 1 la fonction `print()` qui permet d'afficher une chaîne de caractères. Elle permet en plus d'afficher le contenu d'une ou plusieurs variables :

```
>>> x = 32
>>> nom = 'John'
>>> print(nom , ' a ' , x , ' ans')
John  a  32  ans
```

Python a donc écrit la phrase en remplaçant les variables `x` et `nom` par leur contenu. Vous pouvez noter également que pour écrire plusieurs blocs de texte sur une seule ligne, nous avons utilisé le séparateur `,` avec la fonction `print()`. En regardant de plus près, vous vous aperçevrez que Python a automatiquement ajouté un espace à chaque fois que l'on utilisait le séparateur `,`. Par conséquent, si vous voulez mettre un seul espace entre chaque bloc, vous pouvez retirer ceux de vos chaînes de caractères :

```
>>> print(nom , 'a' , x , 'ans')
John a 32 ans
```

Pour imprimer deux chaînes de caractères l'une à côté de l'autre sans espace, vous devrez les concaténer :

```
>>> ani1 = 'chat'
>>> ani2 = 'souris'
>>> print(ani1, ani2)
chat souris
>>> print(ani1 + ani2)
chatsouris
```

3.1 Écriture formatée

La méthode `format()` permet une meilleure organisation de l'affichage des variables (nous expliquerons à la fin de ce chapitre ce que signifie le terme *méthode* en Python).

Si on reprend l'exemple précédent :

```
>>> x = 32
>>> nom = 'John'
>>> print('{} a {} ans'.format(nom, x))
John  a  32  ans
```

— Dans la chaine de caractères, les accolades vides `{}` précisent l'endroit où le contenu de la variable doit être inséré.

— Juste après la chaine de caractères, l'instruction `.format(nom, x)` indique la liste des variables à insérer, d'abord la variable `nom` puis la variable `x`. Ainsi, la méthode `.format()` agit sur la chaine de caractères à laquelle elle est attachée par le `.`.

Remarque : il est possible d'indiquer entre les accolades `{}` dans quel ordre afficher les variables, avec 0 pour la variable à afficher en premier, 1 pour la variable à afficher en second, etc. (attention, Python commence à compter à 0). Cela permet de modifier l'ordre dans lequel sont affichées les variables.

```
>>> x = 32
>>> nom = 'John'
>>> print('{0} a {1} ans'.format(nom, x))
John a 32  ans
>>> print('{1} a {0} ans'.format(nom, x))
32 a John ans
```

Imaginez maintenant que vous vouliez calculer puis afficher la proportion de GC d'un génome. La proportion de GC s'obtient comme la somme des bases Guanine (G) et Cytosine (C) divisée par le nombre total de bases (A, T, C, G) du génome considéré. Si on a, par exemple, 4500 bases G, 2575 bases C pour un total de 14800 bases, vous pourriez faire comme suit (notez bien l'utilisation des parenthèses pour gérer les priorités des opérateurs) :

```
>>> propGC = (4500 + 2575) / 14800
>>> print("La proportion de GC est", propGC)
La proportion de GC est 0.4780405405405405
```

Le résultat obtenu présente trop de décimales (seize dans le cas présent)... Pour écrire le résultat plus lisiblement, vous pouvez spécifier dans les accolades `{}` le format qui vous intéresse. Dans votre cas, vous voulez formater un réel (*float*) pour l'afficher avec deux puis trois décimales :

```
>>> print("La proportion de GC est {:.2f}".format(propGC))
Le proportion de GC est 0.48
>>> print("La proportion de GC est {:.3f}".format(propGC))
La proportion de GC est 0.478
```

Détaillons le contenu des accolades :

— Les deux points `:` indiquent que l'on veut préciser le format.
— La lettre `f` indique que l'on souhaite afficher la variable sous forme d'un réel (*float*).
— Les caractères `.2` indiquent la précision voulue, soit ici deux chiffres après la virgule.
— Notez enfin que le formatage avec `.xf` (`x` étant un entier positif) renvoie un résultat arrondi.

Il est par ailleurs possible de combiner le formatage (à droite des 2 points) ainsi que l'emplacement des variables à substituer (à gauche des 2 points), par exemple :

```
>>> print("propGC(2 décimales) = {0:.2f}, propGC(3 décimales) = {0:.3f}".format(propGC))
propGC(2 décimales) = 0.48, propGC(3 décimales) = 0.478
```

Vous remarquerez qu'on utilise ici la même variable (`propGC`) à deux endroits différents.

Vous pouvez aussi formatez des entiers avec `d`,

```
>>> nbG = 4500
>>> print("Le génome de cet exemple contient {:d} guanines".format(nbG))
Le génome de cet exemple contient 4500 guanines
```

ou mettre plusieurs nombres dans une même chaîne de caractères.

```
>>> nbG = 4500
>>> nbC = 2575
>>> print("Ce génome contient {:d} G et {:d} C, soit une prop de GC de {:.2f}" \
```

```
... .format(nbG,nbC,propGC))
Ce génome contient 4500 G et 2575 C, soit une prop de GC de 0.48
>>> percGC = propGC * 100
>>> print "Ce génome contient {:d} G et {:d} C, soit un %GC de {:.2f} %" \
... .format(nbG,nbC,percGC)
Ce génome contient 4500 G et 2575 C, soit un %GC de 47.80 %
```

Remarque : Le signe \ en fin de ligne permet de poursuivre la commande sur la ligne suivante. Cette syntaxe est pratique lorsque vous voulez taper une commande longue.

Enfin, il est possible de préciser sur combien de caractères vous voulez qu'un résultat soit écrit et comment se fait l'alignement (à gauche, à droite ou centré). Dans la portion de code suivant, le caractère ; sert de séparateur entre les instructions sur une même ligne :

```
>>> print(10) ; print(1000)
10
1000
>>> print("{:>6d}".format(10)) ; print("{:>6d}".format(1000))
    10
  1000
>>> print("{:<6d}".format(10)) ; print("{:<6d}".format(1000))
10
1000
>>> print("{:^6d}".format(10)) ; print("{:^6d}".format(1000))
  10
 1000
>>> print("{:*^6d}".format(10)) ; print("{:*^6d}".format(1000))
**10**
*1000*
>>> print("{:0>6d}".format(10)) ; print("{:0>6d}".format(1000))
000010
001000
```

Notez que > spécifie un alignement à droite, < spécifie un alignement à gauche et ^ spécifie un alignement centré. Il est également possible d'indiquer le caractère qui servira de remplissage lors des alignements (l'espace par défaut).

Ce formatage est également possible sur des chaînes de caractères, notées `s` (comme *string*) :

```
>>> print "atom HN" ; print "atom HDE1"
atom HN
atom HDE1
>>> print "atom {:>4s}".format("HN") ; print "atom {:>4s}".format("HDE1")
atom   HN
atom HDE1
```

Vous voyez tout de suite l'énorme avantage de l'écriture formatée. Cela vous permet d'écrire en colonnes parfaitement alignées. Nous verrons que ceci est très pratique si l'on veut écrire les coordonnées des atomes d'une molécule au format PDB.

Pour les réels, il est possible de combiner le nombre de chiffres après la virgule.

```
>>> print("{:7.3f}".format(propGC))
 47.804
>>> print("{:10.3f}".format(propGC))
    47.804
```

L'instruction `7.3f` signifie que l'on souhaite écrire le réel avec 3 décimales et formaté sur 7 caractères (par défaut justifiés à droite). L'instruction `10.3f` fait la même chose sur 10 caractères. Remarquez que le séparateur décimal `.` compte pour un caractère.

Enfin, si on veut écrire des accolades litérales et utiliser la méthode `format()` en même temps, il faudra doubler les accolades pour échapper au formatage.

```
>>> print("Accolades litérales {{}} et accolades pour le formatage {}".format(10))
Accolades litérales {} et accolades pour le formatage 10
```

À retenir : Comme indiqué ci-desssus, la méthode `format()` agit sur la chaine de caractères à laquelle elle est *attachée* par un `.`, et n'a rien à voir avec la fonction `print()`. Si on donne une chaine suivie d'un `.format()` à la fonction `print()`, Python évalue d'abord le formattage et c'est la chaine qui en résulte qui est affichée à l'écran. Tout comme dans `print(5*5)`, c'est d'abord la multiplication qui est évaluée puis son résultat qui est affiché à l'écran. On peut s'en rendre compte de la manière suivante dans l'interpréteur :

```
>>> "{:10.3f}".format(propGC)
'    47.804'
>>> type("{:10.3f}".format(propGC))
<class 'str'>
```

Python affiche le résultat du `format()` comme une chaine de caractères (*string*), et la fonction `type()` nous le confirme.

3.2 Ancienne méthode de formatage des chaines de caractères

Conseil : Pour les débutants, vous pouvez passer cette section.

Sur d'anciens programmes Python, il se peut que vous rencontriez l'écriture formatée dans le style suivant :

```
>>> x = 32
>>> nom = 'John'
>>> print("%s a %d ans" % (nom,x))

>>> nbG = 4500
>>> nbC = 2575
>>> propGC = (4500.0 + 2575)/14800
>>> print("On a %d G et %d C -> prop GC = %.2f" % (nbG,nbC,propGC))
On a 4500 G et 2575 C -> prop GC = 0.48
```

La syntaxe est légèrement différente. Le symbole `%` est appelé une première fois dans la chaine de caractères (dans l'exemple ci-dessus `%d`, `%d` et `%.2f`) pour :

- Désigner l'endroit où sera placé la variable dans la chaine de caractères.
- Préciser le type de variable à formater, `d` pour un entier (`i` fonctionne également) ou `f` (*float*) pour un réel.
- Eventuellement pour indiquer le format voulu. Ici `.2` signifie une précision de deux chiffres après la virgule.

Le signe `%` est rappelé une seconde fois (`% (nbG,nbC,propGC)`) pour indiquer les variables à formater.

Cette ancienne façon de formatter vous est présentée à titre d'information. Ne l'utilisez pas dans vos programmes.

3.3 Note sur le vocabulaire et la syntaxe

Revenons quelques instants sur la notion de **méthode** abordé dans ce chapitre avec `format()`. En Python, on peut finalement considérer chaque variable comme un objet sur lequel on peut appliquer des méthodes. Une méthode est simplement une fonction qui utilise et/ou agit sur l'objet lui-même, les deux étant connectés par un point `.`. La syntaxe générale est du type `objet.méthode()`.

Dans l'exemple suivant :

```
>>> "Joe a {} ans".format(20)
'Joe a 20 ans'
```

la méthode `format()` est lié à `"Joe a {} ans"` qui est un objet de type chaine de caractères (*string*). La méthode renvoie une nouvelle chaine de caractères avec le bon formatage.

Nous aurons de nombreuses occasions de revoir cette notation `objet.méthode()`.

3.4 Exercices

Conseil : utilisez l'interpréteur Python pour les exercices 2 à 4.

3.4.1 Affichage dans l'interpréteur et dans un programme

Ouvrez l'interpréteur Python et tapez `1+1`. Que se passe-t-il ? Ecrivez la même chose dans un script `test.py`. Exécutez ce script en tapant `python test.py` dans un *shell* Unix. Que se passe-t-il ? Pourquoi ? Faites en sorte d'afficher le résultat de l'addition `1+1` en éxécutant le script dans un *shell* Unix.

3.4.2 Poly-A

Générez une chaîne de caractères représentant un oligonucléotide polyA (AAAA...) de 20 bases de longueurs, sans taper littéralement toutes les bases.

3.4.3 Poly-A et poly-GC

Suivant le modèle du dessus, générez en une ligne de code un polyA de 20 bases suivi d'un polyGC régulier (GCGCGC...) de 40 bases.

3.4.4 Écriture formatée

En utilisant l'écriture formatée, affichez en une seule ligne les variables `a`, `b` et `c` dont les valeurs sont respectivement `"salut"`, `102` et `10.318`.

3.4.5 Écriture formatée 2

Dans un script `propGC.py`, calculez un pourcentage de GC avec l'instruction suivante : `percGC = ((4500 + 2575)/14800)*100`. Ensuite, affichez le contenu de la variable `percGC` à l'écran avec 0, 1, 2, 3 puis 4 décimales sous forme arrondie en utilisant `format()`. On souhaite que le programme affiche la sortie suivante :

```
Le pourcentage de GC est 48      %
Le pourcentage de GC est 47.8    %
Le pourcentage de GC est 47.80   %
Le pourcentage de GC est 47.804  %
Le pourcentage de GC est 47.8041 %
```

Chapitre 4

Listes

4.1 Définition

Une **liste** est une structure de données qui contient une série de valeurs. Python autorise la construction de liste contenant des valeurs de type différent (par exemple entier et chaîne de caractères), ce qui leur confère une grande flexibilité. Une liste est déclarée par une série de valeurs (ne pas oublier les guillemets, simples ou doubles, s'il s'agit de chaînes de caractères) séparées par des **virgules**, et le tout encadré par des **crochets**. En voici quelques exemples :

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> tailles = [5, 2.5, 1.75, 0.15]
>>> mixte = ['girafe', 5, 'souris', 0.15]
>>> animaux
['girafe', 'tigre', 'singe', 'souris']
>>> tailles
[5, 2.5, 1.75, 0.15]
>>> mixte
['girafe', 5, 'souris', 0.15]
```

Lorsque l'on affiche une liste, Python la restitue telle qu'elle a été saisie.

4.2 Utilisation

Un des gros avantages d'une liste est que vous pouvez appeler ses éléments par leur position. Ce numéro est appelé **indice** (ou index) de la liste.

```
liste  : ['girafe', 'tigre', 'singe', 'souris']
indice :        0        1        2         3
```

Soyez très **attentifs** au fait que les indices d'une liste de n éléments commence à *0* et se termine à *n-1*. Voyez l'exemple suivant :

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> animaux[0]
'girafe'
>>> animaux[1]
'tigre'
>>> animaux[3]
'souris'
```

Par conséquent, si on appelle l'élément d'indice 4 de notre liste, Python renverra un message d'erreur :

```
>>> animaux[4]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

N'oubliez pas ceci ou vous risqueriez d'obtenir des bugs inattendus !

4.3 Opération sur les listes

Tout comme les chaînes de caractères, les listes supportent l'opérateur `+` de concaténation, ainsi que l'opérateur `*` pour la duplication :

```
>>> ani1 = ['girafe','tigre']
>>> ani2 = ['singe','souris']
>>> ani1 + ani2
['girafe', 'tigre', 'singe', 'souris']
>>> ani1 * 3
['girafe', 'tigre', 'girafe', 'tigre', 'girafe', 'tigre']
```

L'opérateur `+` est très pratique pour concaténer deux listes. Vous pouvez aussi utiliser la fonction `append()` lorsque vous souhaitez ajouter un seul élément à la fin d'une liste.

Dans l'exemple suivant nous allons créer une liste vide puis lui ajouter deux éléments, d'abord avec la concaténation :

```
>>> l = [] # liste vide
>>> l
[]
>>> l = l + [15]
>>> l
[15]
>>> l = l + [-5]
>>> l
[15, -5]
```

puis avec la fonction `append()` :

```
>>> l = []
>>> l
[]
>>> l.append(15)
>>> l
[15]
>>> l.append(-5)
>>> l
[15, -5]
```

Dans l'exemple ci-dessus nous obtenons le même résultat pour faire grandir une liste en utilisant l'opérateur de concaténation ou la fonction `append()`. Nous vous conseillons néanmoins dans ce cas précis d'utiliser la fonction `append()` dont la syntaxe est plus élégante.

Nous reverrons en détail la fonction `append()` dans le chapitre *11 - Plus sur les listes*, ainsi que la notation `objet.méthode()` dans les chapitres suivants.

4.4 Indiçage négatif et tranches

La liste peut également être indexée avec des nombres négatifs selon le modèle suivant :

```
liste          : ['girafe', 'tigre', 'singe', 'souris']
indice positif :        0        1        2         3
indice négatif :       -4       -3       -2        -1
```

ou encore :

```
liste          : ['A','C','D','E','F','G','H','I','K','L']
indice positif :   0   1   2   3   4   5   6   7   8   9
indice negatif : -10  -9  -8  -7  -6  -5  -4  -3 - 2  -1
```

Les indices négatifs reviennent à compter à partir de la fin. Leur principal avantage est que vous pouvez accéder au dernier élément d'une liste à l'aide de l'indice `-1` sans pour autant connaître la longueur de cette liste. L'avant-dernier élément a lui l'indice `-2`.

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> animaux[-1]
'souris'
>>> animaux[-2]
'singe'
```

Pour accéder au premier élément de la liste, il faut par contre connaître le bon indice :

```
>>> animaux[-4]
'girafe'
```

Dans ce cas, on utiliserait plutôt `animaux[0]`.

Un autre avantage des listes est la possibilité de sélectionner une partie en utilisant un indiçage construit sur le modèle `[m:n+1]` pour récupérer tous les éléments, du émième au énième (de l'élément `m` inclus à l'élément `n+1` exclus). On dit alors qu'on récupère une **tranche** de la liste, par exemple :

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> animaux[0:2]
['girafe', 'tigre']
>>> animaux[0:3]
['girafe', 'tigre', 'singe']
>>> animaux[0:]
['girafe', 'tigre', 'singe', 'souris']
>>> animaux[:]
['girafe', 'tigre', 'singe', 'souris']
>>> animaux[1:]
['tigre', 'singe', 'souris']
>>> animaux[1:-1]
['tigre', 'singe']
```

Notez que lorsqu'aucun indice n'est indiqué à gauche ou à droite du symbole `:`, Python prend par défaut tous les éléments depuis le début ou tous les éléments jusqu'à la fin respectivement.

On peut aussi préciser le pas en ajoutant un `:` supplémentaire et en indiquant le pas par un entier.

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> animaux[0:3:2]
['girafe', 'singe']
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::1]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::2]
[0, 2, 4, 6, 8]
>>> x[::3]
[0, 3, 6, 9]
>>> x[1:6:3]
[1, 4]
```

Finalement, on voit que l'accès au contenu d'une liste avec des crochets fonctionne sur le modèle `liste[début:fin:pas]`.

4.5 Fonction len()

L'instruction `len()` vous permet de connaître la longueur d'une liste, c'est-à-dire le nombre d'éléments que contient la liste. Voici un exemple d'utilisation :

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> len(animaux)
4
>>> len([1, 2, 3, 4, 5, 6, 7, 8])
8
```

4.6 Les fonctions range() et list()

L'instruction `range()` est une fonction spéciale en Python qui va nous permettre de générer des nombres entiers compris dans un intervalle lorsqu'elle est utilisée en combinaison avec la fonction `list()` . Par exemple :

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

La commande `list(range(10))` a généré une liste contenant tous les nombres entiers de 0 inclus à 10 **exclus**. Nous verrons l'utilisation de la fonction `range()` toute seule dans le chapitre suivant sur les boucles.

Dans l'exemple ci-dessus, la fonction `range()` a pris un argument, mais elle peut également prendre deux ou trois arguments, voyez plutôt :

```
>>> list(range(0,5))
[0, 1, 2, 3, 4]
>>> list(range(15,20))
[15, 16, 17, 18, 19]
>>> list(range(0,1000,200))
[0, 200, 400, 600, 800]
>>> list(range(2,-2,-1))
[2, 1, 0, -1]
```

L'instruction `range()` fonctionne sur le modèle `range([début,] fin[, pas])`. Les arguments entre crochets sont optionnels. Pour otenir une liste, il faut l'utiliser systématiquement avec la fonction `list()`.

Enfin, prenez garde aux arguments optionnels par défaut (`0` pour `début` et `1` pour `pas`) :

```
>>> list(range(10,0))
[]
```

Ici la liste est vide car Python a pris la valeur du pas par défaut qui est de 1. Ainsi, si on commence à 10 et qu'on avance par pas de 1, on ne pourra jamais atteindre 0. Python génère ainsi une liste vide. Pour éviter ça, il faudra absolument préciser le pas de -1 pour les listes décroissantes :

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

4.7 Listes de listes

Pour finir, sachez qu'il est tout-à-fait possible de construire des listes de listes. Cette fonctionnalité peut être parfois très pratique. Par exemple :

```
>>> enclos1 = ['girafe', 4]
>>> enclos2 = ['tigre', 2]
>>> enclos3 = ['singe', 5]
>>> zoo = [enclos1, enclos2, enclos3]
>>> zoo
[['girafe', 4], ['tigre', 2], ['singe', 5]]
```

Dans cet exemple, chaque sous-liste contient une catégorie d'animal et le nombre d'animaux pour chaque catégorie.

Pour accéder à un élément de la liste, on utilise l'indiçage habituel :

```
>>> zoo[1]
['tigre', 2]
```

Pour accéder à un élément de la sous-liste, on utilise un double indiçage :

```
>>> zoo[1][0]
'tigre'
>>> zoo[1][1]
2
```

On verra un peu plus loin qu'il existe en Python les dictionnaires qui sont très pratiques pour faire ce genre de choses. On verra aussi qu'il existe un module nommé `numpy` permettant de gérer des listes ou tableaux de nombres (vecteurs et matrices), ainsi que de faire des opérations avec.

4.8 Exercices

Conseil : utilisez l'interpréteur Python.

4.8.1 Jours de la semaine

Constituez une liste `semaine` contenant les 7 jours de la semaine.

1. À partir de cette liste, comment récupérez-vous seulement les 5 premiers jours de la semaine d'une part, et ceux du week-end d'autre part (*utilisez pour cela l'indiçage*) ?
2. Cherchez un autre moyen pour arriver au même résultat (*en utilisant un autre indiçage*).
3. Trouvez deux manières pour accéder au dernier jour de la semaine.
4. Inversez les jours de la semaine en une commande.

4.8.2 Saisons

Créez 4 listes `hiver`, `printemps`, `ete` et `automne` contenant les mois correspondants à ces saisons. Créez ensuite une liste `saisons` contenant les sous-listes `hiver`, `printemps`, `ete` et `automne`. Prévoyez ce que valent les variables suivantes, puis vérifiez-le dans l'interpréteur :

1. `saisons[2]`
2. `saisons[1][0]`
3. `saisons[1:2]`
4. `saisons[:][1]`. Comment expliquez-vous ce dernier résultat ?

4.8.3 Table des 9

Affichez la table des 9 en une seule commande avec les instructions `range()` et `list()`.

4.8.4 Nombres pairs

Avec Python, répondez à la question suivante en une seule commande. Combien y a-t-il de nombres pairs dans l'intervalle [2 , 10000] inclus ?

Chapitre 5

Boucles et comparaisons

5.1 Boucles for

5.1.1 Principe

En programmation, on est souvent amené à répéter plusieurs fois une instruction. Incontournables à tout langage de programmation, les boucles vont nous aider à réaliser cette tâche de manière compacte. Imaginez par exemple que vous souhaitiez afficher les éléments d'une liste les uns après les autres. Dans l'état actuel de vos connaissances, il faudrait taper quelque chose du style :

```
animaux = ['girafe','tigre','singe','souris']
print(animaux[0])
print(animaux[1])
print(animaux[2])
print(animaux[3])
```

Si votre liste ne contient que quatre éléments, ceci est encore faisable mais imaginez qu'elle en contienne 100 voire 1000 ! Pour remédier à cela, il faut utiliser les boucles. Regardez l'exemple suivant :

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> for animal in animaux:
...     print(animal)
...
girafe
tigre
singe
souris
```

Commentons en détails ce qu'il s'est passé dans cet exemple :

La variable `animal` est appelée **variable d'itération**, elle prend successivement les différentes valeurs de la liste `animaux` à chaque itération de la boucle. On verra en section 5.1.3 que l'on peut choisir le nom que l'on veut pour cette variable. Celle-ci est créée par Python la première fois que la ligne contenant le `for` est exécutée (si elle existait déjà son contenu serait écrasé). Une fois la boucle terminée, cette variable d'itération `animal` ne sera pas détruite et ainsi contiendra la dernière valeur de la liste `animaux` (ici la chaine `souris`).

Notez bien les types de variables : `animaux` est une **liste** sur laquelle on itère, et `animal` est une **chaîne de caractères** car chaque élément de la liste est une chaîne de caractères. Nous verrons plus loin que la variable d'itération peut être de n'importe quel type selon la liste parcourue. En Python, une boucle itère toujours sur un objet dit **séquentiel** (c'est à dire un objet constitué d'autres objets) tel

qu'une liste. Nous verrons aussi plus tard d'autres objets séquentiels sur lesquels on peut itérer dans une boucle.

D'ores et déjà, remarquez avec attention le **signe deux-points** `:` à la fin de la ligne `for`. Cela signifie que la boucle `for` attend un **bloc d'instructions**, en l'occurence toutes les instructions que Python répètera à chaque itération de la boucle. On appelle ce bloc d'instructions le **corps de la boucle**. Comment indique-t-on à Python où ce bloc commence et se termine ? Cela est signalé uniquement par l'**indentation**, c'est-à-dire le décalage de la (ou des) ligne(s) du bloc d'instructions. Dans l'exemple suivant, le corps de la boucle contient deux instructions `print(animal*2)` et `print(animal)` car elles sont indentées par rapport à la ligne `for` :

```
for animal in animaux:
    print(animal)
    print(animal*2)
print("C'est fini")
```

La ligne `print("C'est fini")` ne fait pas partie du corps de la boucle car elle est au même niveau que le `for` (c'est-à-dire non indentée par rapport au `for`). Notez également que chaque instruction du corps de la boucle doit être indentée de la même manière (ici 4 espaces).

Outre une meilleure lisibilité, les `:` et l'**indentation** sont formellement requis en Python. Même si on peut indenter comme on veut en Python (plusieurs espaces ou plusieurs tabulations, mais pas une combinaison des deux), les développeurs recommandent l'utilisation de 4 espaces. Faites en sorte de configurer votre éditeur favori de façon à écrire 4 espaces lorsque vous tapez sur la touche Tab.

Si on oublie l'indentation, Python vous renvoie un message d'erreur :

```
>>> for animal in animaux:
... print(animal)
  File "<stdin>", line 2
    print(animal)
        ^
IndentationError: expected an indented block
```

Dans les exemples ci-dessus, nous avons exécuté une boucle en itérant directement sur une liste. Une tranche d'une liste étant elle même une liste, on peut également itérer dessus :

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> for animal in animaux[1:3]:
...     print(animal)
...
tigre
singe
```

On a vu que les boucles `for` pouvaient utiliser une liste contenant des chaînes de caractères, mais elles peuvent tout aussi bien utiliser des listes contenant des entiers (ou n'importe quel type de variable finalement).

```
>>> for i in [1,2,3]:
...     print(i)
...
1
2
3
```

5.1.2 Fonction range()

Python possède la fonction `range()` que nous avons rencontrée précédemment dans le chapitre sur les *Listes* et qui est aussi bien commode pour faire une boucle sur une liste d'entiers de manière automatique :

```
>>> for i in range(4):
...     print(i)
...
0
1
2
3
```

Dans cet exemple nous pouvons faire plusieurs remarques importantes :

Contrairement à la génération de liste avec `list(range(4))`, la fonction `range()` peut être utilisée telle quelle dans une boucle. Il n'est pas nécessaire de taper `for i in list(range(4)):`, même si cela fonctionnerait également.

Comment cela est-il possible ? Et bien `range()` est une fonction qui a été spécialement conçue pour cela, c'est à dire que l'on peut itérer directement dessus. Pour Python, il s'agit d'un nouveau type, par exemple dans `x = range(3)` la variable `x` est de type *range* (tout comme on avait les types *int*, *float*, *str* ou *list*) à utiliser spécialement avec les boucles.

L'instruction `list(range(4))` se contente de transformer un objet de type *range* en un objet de type *list* (si vous vous souvenez bien, il s'agit donc d'une fonction de *casting*, c'est à dire de conversion d'un type en un autre). Il n'y aucun intérêt à utiliser dans une boucle la construction `for i in list(range(4)):`. C'est même contre-productif. En effet, `range()` se contente de stocker l'entier actuel, le pas pour passer à l'entier suivant, et le dernier entier à parcourir, ce qui revient à stocker seulement 3 entiers et ce quel que soit la longueur de la séquence, même avec un `range(1000000)`. Si on utilisait `list(range(1000000))`, Python construirait d'abord une liste de 1 million d'éléments dans la mémoire puis itérerait dessus, d'où une énorme perte de temps !

5.1.3 Nommage de la variable d'itération

Dans l'exemple précédent, nous avons choisi le nom `i` pour la variable d'itération. Ceci est un standard en informatique et indique en général qu'il s'agit d'un entier (le nom `i` vient sans doute du mot indice ou *index* en anglais). Nous vous conseillons de suivre cette convention afin d'éviter les confusions, si vous itérez sur les indices vous pouvez appeler la variable d'itération `i` (par exemple dans `for i in range(4):`).

Si vous itérez sur une liste comportant des chaînes de caractères, mettez un nom explicite pour la variable d'itération. Par exemple :

```
for prenom in ['Joe', 'Bill', 'John']:
```

5.1.4 Itération sur les indices

Revenons à notre liste `animaux`. Nous allons maintenant parcourir cette liste, mais cette fois par une itération sur ses indices :

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> for i in range(4):
...     print(animaux[i])
```

```
...
girafe
tigre
singe
souris
```

La variable `i` prendra les valeurs successives 0, 1, 2 et 3 et on accèdera à chaque élément de la liste `animaux` par son indice (*i.e.* `animaux[i]`). Notez à nouveau le nom `i` de la variable d'itération car on itère sur les **indices**.

Quand utiliser l'une ou l'autre des 2 méthodes ? La plus efficace est celle qui réalise **les itérations directement sur les élements** :

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> for animal in animaux:
...     print(animal)
...
girafe
tigre
singe
souris
```

Toutefois, il se peut qu'au cours d'une boucle vous ayez besoin des indices, auquel cas vous n'avez pas le choix que d'itérer sur les indices. Par exemple :

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> for i in range(len(animaux)):
...     print("L'animal {} est un(e) {}".format(i, animaux[i]))
...
L'animal 0 est un(e) girafe
L'animal 1 est un(e) tigre
L'animal 2 est un(e) singe
L'animal 3 est un(e) souris
```

Python possède également la fonction `enumerate()` qui vous permet d'itérer sur les indices et les éléments eux-mêmes.

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> for i, animal in enumerate(animaux):
...     print("L'animal {} est un(e) {}".format(i, animal))
...
L'animal 0 est un(e) girafe
L'animal 1 est un(e) tigre
L'animal 2 est un(e) singe
L'animal 3 est un(e) souris
```

5.2 Comparaisons

Avant de passer à une autre sorte de boucles (les boucles `while`), nous abordons tout de suite les **comparaisons**. Celles-ci seront reprises dans le chapitre sur les *Tests*.

Python est capable d'effectuer toute une série de comparaisons entre le contenu de deux variables, telles que :

Syntaxe Python	Signification
`==`	égal à
`!=`	différent de
`>`	supérieur à
`>=`	supérieur ou égal à
`<`	inférieur à
`<=`	inférieur ou égal à

Observez l'exemple suivant avec des nombres entiers.

```
>>> x = 5
>>> x == 5
True
>>> x > 10
False
>>> x < 10
True
```

Python renvoie la valeur `True` si la comparaison est vraie et `False` si elle est fausse. `True` et `False` sont des booléens.

Faites bien attention à ne pas confondre l'**opérateur d'affectation** `=` qui affecte une valeur à une variable et l'**opérateur de comparaison** `==` qui compare les valeurs de deux variables.

Vous pouvez également effectuer des comparaisons sur des chaînes de caractères.

```
>>> animal = "tigre"
>>> animal == "tig"
False
>>> animal != "tig"
True
>>> animal == 'tigre'
True
```

Dans le cas des chaînes de caractères, *a priori* seuls les tests `==` et `!=` ont un sens. En fait, on peut aussi utiliser les opérateurs `<`, `>`, `<=` et `>=`. Dans ce cas l'ordre alphabétique est pris en compte, par exemple :

```
>>> "a" < "b"
True
```

`"a"` est *inférieur à* `"b"` car il est situé avant dans l'ordre alphabétique. En fait, c'est l'ordre ASCII des caractères qui est pris en compte (*i.e.* chaque caractère est affecté à un code numérique), on peut donc comparer aussi des caractères spéciaux (comme `#` ou `~`) entre eux. On peut aussi comparer des chaînes à plus d'un caractère.

```
>>> "ali" < "alo"
True
>>> "abb" < "ada"
True
```

Dans ce cas, Python compare caractère par caractère de la gauche vers la droite (le premier avec le premier, le deuxième avec le deuxième, etc). Dès qu'un caractère est différent entre l'une et l'autre des chaînes, il considère que la chaîne la plus petite est celle qui présente le caractère ayant le plus petit code ASCII (les caractères suivants de la chaîne sont ignorés dans la comparaison), comme dans l'exemple `"abb" < "ada"` ci-dessus.

5.3 Boucles while

Une autre alternative à l'instruction `for` couramment utilisée en informatique est la boucle `while`. Le principe est simple. Une série d'instructions est exécutée tant qu'une condition est vraie. Par exemple :

```
>>> i = 1
>>> while i <= 4:
...     print(i)
...     i = i + 1
...
1
2
3
4
```

Remarquez qu'il est encore une fois nécessaire d'indenter le bloc d'instructions correspondant au corps de la boucle (ici 2 instructions).

Une boucle `while` nécessite généralement **trois éléments** pour fonctionner correctement :

— l'initialisation de la variable de test avant la boucle ;
— le test de la variable associé à l'instruction `while` ;
— la mise à jour de la variable de test dans le corps de la boucle.

Faites bien attention aux tests et à l'incrémentation que vous utilisez car une erreur mène souvent à des boucles infinies, c'est-à-dire qui ne s'arrêtent jamais. Vous pouvez néanmoins toujours stopper l'exécution d'un script Python à l'aide de la combinaison de touches Ctrl-C. Par exemple :

```
i = 0
while i < 10:
    print("Le python c'est cool !")
```

Ici nous avons omis de mettre à jour la variable `i`, ainsi la boucle ne s'arrêtera jamais (sauf en pressant Ctrl-C) puisque la condition `i < 10` sera toujours vraie.

5.4 Exercices

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

5.4.1 Boucles de base

Soit la liste `['vache','souris','levure','bacterie']`. Affichez l'ensemble des éléments de cette liste (un élément par ligne) de trois manières différentes (deux avec `for` et une avec `while`).

5.4.2 Boucle et jours de la semaine

Constituez une liste `semaine` contenant les 7 jours de la semaine.

Écrivez une série d'instructions affichant les jours de la semaine (en utilisant une boucle `for`), ainsi qu'une autre série d'instructions affichant les jours du week-end (en utilisant une boucle `while`).

5.4.3 Table des 1

Avec une boucle, affichez les nombres de 1 à 10 sur une seule ligne.

5.4.4 Nombres pairs et impairs

Soit `impairs` la liste de nombres `[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]`. Écrivez un programme qui, à partir de la liste `impairs`, construit une liste `pairs` dans laquelle tous les éléments de `impairs` sont incrémentés de 1.

5.4.5 Calcul de moyenne

Voici les notes d'un étudiant `[14, 9, 6, 8, 12]`. Calculez la moyenne de ces notes. Utilisez l'écriture formatée pour afficher la valeur de la moyenne avec deux décimales.

5.4.6 Produit de nombres consécutifs

Soit la liste `X` contenant les nombres entiers de 0 à 10. Calculez le produit des nombres consécutifs deux à deux de `X` en utilisant une boucle. Exemple pour les premières itérations :

```
0
2
6
12
```

5.4.7 Triangle

Écrivez un script qui dessine un triangle comme celui-ci :

```
*
**
***
****
*****
******
*******
********
*********
**********
```

5.4.8 Triangle inversé

Écrivez un script qui dessine un triangle comme celui-ci :

```
**********
*********
********
*******
******
*****
****
```

```
***
**
*
```

5.4.9 Triangle gauche

Écrivez un script qui dessine un triangle comme celui-ci :

```
         *
        **
       ***
      ****
     *****
    ******
   *******
  ********
 *********
**********
```

5.4.10 Triangle isocèle

Écrivez un script qui dessine un triangle comme celui-ci :

```
         *
        ***
       *****
      *******
     *********
    ***********
   *************
  ***************
 *****************
*******************
```

5.4.11 Parcours de matrice

Imaginons que l'on soutaite parcourir tous les éléments d'une matrice carrée, c'est-à-dire d'une matrice qui est constituée d'autant de colonnes que de lignes.

Écrivez un script qui parcourt chaque élément de la matrice et qui affiche le numéro de ligne et de colonne.

Pour une matrice 2x2, le schéma 5.1 vous indique comment parcourir une telle matrice. L'affichage attendu est :

```
ligne colonne
   1      1
   1      2
   2      1
   2      2
```

Attention à bien respecter l'alignement des chiffres qui doivent être justifiés à droite. Testez pour une matrice 3x3, 5x5, et 10x10.

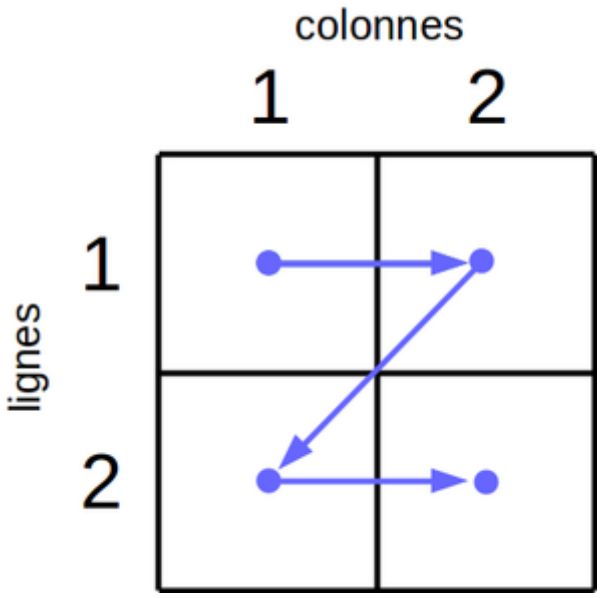


FIGURE 5.1 – Parcours d'une matrice

5.4.12 Parcours de demi-matrice sans la diagonale (exercice ++)

En se basant sur le script précédent, on souhaite réaliser le parcours d'une demi-matrice carrée sans la diagonale. On peut noter que cela donne tous les couples possibles une seule fois (1 et 2 est équivalent à 2 et 1), en excluant par ailleurs chaque élément avec lui même (1 et 1, 2 et 2, etc). Pour mieux comprendre ce qui est demandé, la figure 5.2 indique les cases à parcourir en vert :

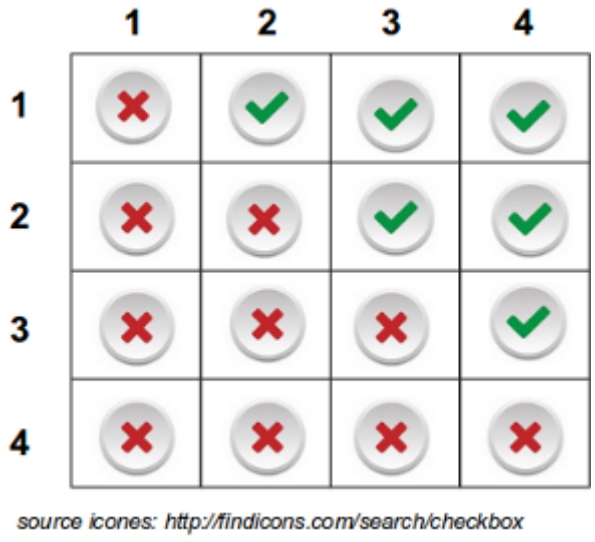


FIGURE 5.2 – Demi-matrice sans la diagonale (en vert)

Écrivez un script qui affiche le numéro de ligne et de colonne, puis la taille de la matrice `NxN` et le nombre total de cases parcourues. Par exemple pour une matrice 4x4 (N=4) :

```
ligne colonne
   1      2
   1      3
   1      4
   2      3
   2      4
```

```
   3     4
Pour une matrice 4x4, on a parcouru 6 cases
```

Testez votre script avec `N=3`, `N=4` et `N=5`.

Concevez une seconde version à partir du script précédent, où cette fois on n'affiche plus tous les couples possibles mais simplement la valeur de `N`, et le nombre de cases parcourues. Affichez cela pour des valeurs de `N` allant de 1 à 20. Pouvez-vous trouver une formule générale reliant le nombre de cases parcourues à `N` ?

5.4.13 Sauts de puce

On imagine une puce qui se déplace aléatoirement sur une ligne, en avant ou en arrière, par pas de 1 ou -1. Par exemple, si elle est à l'emplacement 0, elle peut sauter à l'emplacement 1 ou -1 ; si elle est à l'emplacement 2, elle peut sauter à l'emplacement 3 ou 1, etc. Avec une boucle `while`, simuler le mouvement de cette puce de l'emplacement 0 à l'emplacement 5 (cf schéma suivant). Combien de sauts sont nécessaires à ce parcours ? Relancez plusieurs fois le programme. Trouvez-vous le même nombre de sauts à chaque exécution ? Comparez avec votre voisin.

```
  départ     arrivée
     |          |
     v          v
...o-X-o-o-o-o-X-o...
  -1 0 1 2 3 4 5 6
```

Conseil : vous utiliserez l'instruction

```
random.choice([-1,1])
```

qui renvoie au hasard les valeurs -1 ou 1 avec la même probabilité. Avant d'utiliser cette commande vous devrez mettre en haut du script la ligne

```
import random
```

Nous reverrons la signification de cette syntaxe particulière dans le chapitre sur les modules.

5.4.14 Suite de Fibonacci (exercice +++)

La suite de Fibonacci est une suite mathématique qui porte le nom de Leonardo Fibonacci, un mathématicien italien du XIII^{e} siècle. Initialement, cette suite a été conçue pour décrire la croissance d'une population de lapins mais elle peut également être utilisée pour décrire certains motifs géométriques retrouvés dans la nature (coquillages, fleurs de tournesol. . .).

Par définition, les deux premiers termes de la suite de Fibonacci sont 0 et 1. Ensuite, le terme au rang n est la somme des nombres aux rangs $n-1$ et $n-2$. Par exemple, les 10 premiers termes de la suite de Fibonacci sont 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.

Écrivez un script qui construit la liste des 20 premiers termes de la suite de Fibonacci puis l'affiche.

Améliorez ce script en imprimant à chaque itération le rapport entre l'élément n et l'élément $n-1$. Ce rapport tend-il vers une constante ? Si oui, laquelle ?

Chapitre 6

Tests

6.1 Définition

Les **tests** sont un élément essentiel à tout langage informatique si on veut lui donner un peu de complexité car ils permettent à l'ordinateur de prendre des décisions si telle ou telle condition est vraie ou fausse. Pour cela, Python utilise l'instruction `if` ainsi qu'une comparaison que nous avons abordée au chapitre précédent. Voici un premier exemple :

```
>>> x = 2
>>> if x == 2:
...     print("Le test est vrai !")
...
Le test est vrai !
```

et un second :

```
>>> x = "souris"
>>> if x == "tigre":
...     print("Le test est vrai !")
...
```

Il y a plusieurs remarques à faire concernant ces deux exemples :

- Dans le premier exemple, le test étant vrai, l'instruction `print("Le test est vrai !")` est exécutée. Dans le second exemple, le test est faux et rien n'est affiché.
- Les blocs d'instruction dans les tests doivent forcément être indentés comme les boucles `for` et `while`. L'indentation indique la portée des instructions à exécuter si le test est vrai.
- L'instruction `if` se termine comme les instructions `for` et `while` par le caractère `:`.

6.2 Tests à plusieurs cas

Parfois, il est pratique de tester si la condition est vraie ou si elle est fausse dans une même instruction `if`. Plutôt que d'utiliser deux instructions `if`, on peut se servir des instructions `if` et `else` :

```
>>> x = 2
>>> if x == 2:
...     print("Le test est vrai !")
... else:
...     print("Le test est faux !")
...
```

```
Le test est vrai !
>>> x = 3
>>> if x == 2:
...     print("Le test est vrai !")
... else:
...     print("Le test est faux !")
...
Le test est faux !
```

On peut utiliser une série de tests dans la même instruction `if`, notamment pour tester plusieurs valeurs d'une même variable. Par exemple, on se propose de tirer au sort une base d'ADN puis d'afficher le nom de cette dernière. Dans le code suivant, nous utilisons l'intruction `random.choice(liste)` qui renvoie un élément choisi au hasard dans une `liste`. L'instruction `import random` sera vue plus tard, admettez pour le moment qu'elle est nécessaire.

```
>>> import random
>>> base = random.choice(["a", "t", "c", "g"])
>>> if base == "a":
...     print("choix d'une adénine")
... elif base == "t":
...     print("choix d'une thymine")
... elif base == "c":
...     print("choix d'une cytosine")
... elif base == "g":
...     print("choix d'une guanine")
...
choix d'une cytosine
```

Dans cet exemple, Python teste la première condition, puis, si et seulement si elle est fausse, teste la deuxième et ainsi de suite... Le code correspondant à la première condition vérifiée est exécuté puis Python sort du `if`.

Remarque : De nouveau, faites bien attention à l'indentation dans ces deux derniers exemples ! Vous devez être très rigoureux sur ce point. Pour vous en convaincre, exécutez ces deux scripts dans l'interpréteur Python :

Script 1

```
nombres = [4, 5, 6]
for nb in nombres:
    if nb == 5:
        print("Le test est vrai")
        print("car la variable nb vaut {}".format(nb))
```

Script 2

```
nombres = [4, 5, 6]
for nb in nombres:
    if nb == 5:
        print("Le test est vrai")
    print("car la variable nb vaut {}".format(nb))
```

Comment expliquez-vous ce résultat ? Observez bien l'indentation de la dernière ligne.

6.3 Tests multiples

Les tests multiples permettent de tester plusieurs conditions en même temps en utilisant des opérateurs booléens. Les deux opérateurs les plus couramment utilisés sont le **OU** et le **ET**. Voici un petit rappel du mode de fonctionnement de l'opérateurs **OU** :

Condition 1	Opérateur	Condition 2	Résultat
Vrai	OU	Vrai	Vrai
Vrai	OU	Faux	Vrai
Faux	OU	Vrai	Vrai
Faux	OU	Faux	Faux

et de l'opérateur **ET** :

Condition 1	Opérateur	Condition 2	Résultat
Vrai	ET	Vrai	Vrai
Vrai	ET	Faux	Faux
Faux	ET	Vrai	Faux
Faux	ET	Faux	Faux

En Python, on utilise le mot réservé `and` pour l'opérateur **ET** et le mot réservé `or` pour l'opérateur **OU**. Respectez bien la casse des opérateurs `and` et `or` qui, en Python, s'écrivent en minuscule. En voici un exemple d'utilisation :

```
>>> x = 2
>>> y = 2
>>> if x == 2 and y == 2:
...     print("le test est vrai")
...
le test est vrai
```

Notez que le même résulat serait obtenu en utilisant deux instructions `if` imbriquées :

```
>>> x = 2
>>> y = 2
>>> if x == 2:
...     if y == 2:
...         print("le test est vrai")
...
le test est vrai
```

Vous pouvez aussi tester directement l'effet de ces opérateurs à l'aide de `True` et `False` (attention à respecter la casse).

```
>>> True or False
True
```

Enfin, on peut utiliser l'opérateur logique de négation `not` qui inverse le résultat d'une condition :

```
>>> not True
False
>>> not False
True
```

```
>>> not (True and True)
False
```

6.4 Instructions break et continue

Ces deux instructions permet de modifier le comportement d'une boucle (`for` ou `while`) avec un test.

L'instruction `break` stoppe la boucle.

```
>>> for i in range(5):
...     if i > 2:
...         break
...     print(i)
...
0
1
2
```

L'instruction `continue` saute à l'itération suivante.

```
>>> for i in range(5):
...     if i == 2:
...         continue
...     print(i)
...
0
1
3
4
```

6.5 Tests de valeur sur des réels

Lorsque l'on souhaite tester la valeur d'une variable *float*, le premier réflexe serait d'utiliser l'opérateur d'égalité comme :

```
>>> 1/10 == 0.1
True
```

Toutefois nous vous le conseillons formellement. Pourquoi ? Python stocke les *float* sous forme de nombres flottants (d'où leur nom !), et cela mène à certaines limitations. Regardez l'exemple suivant :

```
>>> (3 - 2.7) == 0.3
False
>>> 3 - 2.7
0.2999999999999998
```

Nous voyons que le résultat de l'opération `3 - 2.7` n'est pas exactement `0.3` d'où le `False`. Pour éviter ces problèmes nous conseillons de toujours encadrer un *float* avec un niveau de précission *delta*, par exemple :

```
>>> delta = 0.0001
>>> 0.3 - delta < 3.0 - 2.7 < 0.3 + delta
True
```

Ici on teste si `3 - 2.7` est compris entre `0.3` ± `delta`.

6.6 Exercices

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

6.6.1 Jours de la semaine

Constituez une liste `semaine` contenant les sept jours de la semaine.

En utilisant une boucle, écrivez chaque jour de la semaine ainsi que les messages suivants :

— `Au travail` s'il s'agit du lundi au jeudi ;
— `Chouette c'est vendredi` s'il s'agit du vendredi ;
— `Repos ce week-end` s'il s'agit du week-end.

Ces messages ne sont que des suggestions, vous pouvez laisser libre cours votre imagination.

6.6.2 Séquence complémentaire d'un brin d'ADN

La liste ci-dessous représente la séquence d'un brin d'ADN :

```
["A","C","G","T","T","A","G","C","T","A","A","C","G"]
```

Écrivez un script qui transforme cette séquence en sa séquence complémentaire.

Rappel : la séquence complémentaire s'obtient en remplaçant A par T, T par A, C par G et G par C.

6.6.3 Minimum d'une liste

La fonction `min()` de Python, renvoie l'élément le plus petit d'une liste consistuée de valeurs numériques ou de chaînes de caractères. Sans utiliser cette fonction, écrivez un script qui détermine le plus petit élément de la liste `[8, 4, 6, 1, 5]`.

6.6.4 Fréquence des acides aminés

La liste ci-dessous représente une séquence d'acides aminés :

```
["A","R","A","W","W","A","W","A","R","W","W","R","A","G","A","R"]
```

Calculez la fréquence des acides aminés alanine (A), arginine (R), tryptophane (W) et glycine (G) dans cette séquence.

6.6.5 Notes et mention d'un étudiant

Voici les notes d'un étudiant : 14, 9, 13, 15 et 12. Écrivez un script qui affiche la note maximum (fonction `max()`), la note minimum (fonction `min()`) et qui calcule la moyenne.

Affichez la valeur de la moyenne avec deux décimales. Affichez aussi la mention obtenue sachant que la mention est passable si la moyenne est entre 10 inclus et 12 exclus, assez-bien entre 12 inclus et 14 exclus et bien au-delà de 14.

6.6.6 Nombres pairs

Construisez une boucle qui parcourt les nombres de 0 à 20 et qui affiche les nombres pairs inférieurs ou égaux à 10 d'une part, et les nombres impairs strictement supérieur à 10 d'autre part.

Pour cet exercice, vous pourrez utiliser l'opérateur modulo `%` qui renvoie le reste de la division entière entre deux nombres et dont voici quelques exemples d'utilisation :

```
>>> 4 % 3
1
>>> 5 % 3
2
>>> 4 % 2
0
>>> 5 % 2
1
>>> 6 % 2
0
>>> 7 % 2
1
```

Ainsi, vous remarquerez qu'un nombre est pair lorsque le reste de sa division entière par 2 est nul.

6.6.7 L'énigme du père Fouras

A Fort Boyard, le père Fouras nous pose l'énigme suivante :

Pour ouvrir le coffre où se trouve la clé, trouve la combinaison à trois chiffres sachant que :

- *Le nombre est inférieur à 200.*
- *Deux de ses chiffres sont identiques.*
- *La somme de ses chiffres est égale à 5.*
- *C'est un nombre pair.*

On se propose d'utiliser une méthode dite *brute force*, c'est à dire d'utiliser une boucle et à chaque itération on teste les 4 conditions. Avez-vous réussi à trouver la même solution en raisonnant ?

6.6.8 Conjecture de Syracuse (exercice +++)

La conjecture de Syracuse est une conjecture mathématique qui reste improuvée à ce jour et qui est définie de la manière suivante.

Soit un entier positif n. Si n est pair, alors le diviser par 2. Si il est impair, alors le multiplier par 3 et lui ajouter 1. En répétant cette procédure, la suite de nombres atteint la valeur 1 puis se prolonge indéfiniment par une suite de trois valeurs triviales appelée cycle trivial.

Jusqu'à présent, la conjecture de Syracuse, selon laquelle depuis n'importe quel entier positif la suite de Syracuse atteint 1, n'a pas été mise en défaut.

Par exemple, les premiers éléments de la suite de Syracuse si on prend comme point de départ 10 sont : 10, 5, 16, 8, 4, 2, 1...

Écrivez un script qui, partant d'un entier positif n (par exemple 10 ou 20), crée une liste des nombres de la suite de Syracuse. Avec différents points de départ (c'est-à-dire avec différentes valeurs de n), la conjecture de Syracuse est-elle toujours vérifiée ? Quels sont les nombres qui constituent le cycle trivial ?

Remarques

1. Pour cet exercice, vous avez besoin de faire un nombre d'itérations inconnu pour que la suite de Syracuse atteigne le chiffre 1 puis entame son cycle trivial. Vous pourrez tester votre algorithme avec un nombre arbitraire d'itérations, typiquement 20 ou 100, suivant votre nombre n de départ.
2. Un nombre est pair lorsque le reste de sa division entière (opérateur modulo `%`) par 2 est nul.

6.6.9 Attribution simple de la structure secondaire des résidus d'une protéine (exercice +++)

Les angles dièdres phi/psi d'une hélice alpha parfaite ont une valeur de -57 degrés et -47 degrés respectivement. Bien sûr, il est très rare que l'on trouve ces valeurs parfaites dans une protéine, par conséquent il est couramment accepté de tolérer une déviation de +/- 30 degrés sur celles-ci.

Vous trouverez ci-dessous une liste de listes contenant les valeurs des angles phi/psi des résidues de la première hélice de la protéine 1TFE. En utilisant cette liste, écrivez un programme qui teste, pour chaque résidu, s'il est ou pas en hélice.

```
[[48.6,53.4],[-124.9,156.7],[-66.2,-30.8],[-58.8,-43.1], \
[-73.9,-40.6],[-53.7,-37.5],[-80.6,-16.0],[-68.5,135.0], \
[-64.9,-23.5],[-66.9,-45.5],[-69.6,-41.0],[-62.7,-37.5], \
[-68.2,-38.3],[-61.2,-49.1],[-59.7,-41.1],[-63.2,-48.5], \
[-65.5,-38.5],[-64.1,-40.7],[-63.6,-40.8],[-66.4,-44.5], \
[-56.0,-52.5],[-55.4,-44.6],[-58.6,-44.0],[-77.5,-39.1], \
[-91.7,-11.9],[48.6,53.4]]
```

6.6.10 Détermination des nombres premiers inférieurs à 100 (exercice +++)

Voici un extrait de l'article sur les nombres premiers tiré de l'encyclopédie en ligne wikipédia.

Un nombre premier est un entier naturel qui admet exactement deux diviseurs distincts entiers et positifs (qui sont alors 1 et lui-même). Cette définition exclut 1, qui n'a qu'un seul diviseur entier positif. Par opposition, un nombre non nul produit de deux nombres entiers différents de 1 est dit composé. Par exemple $6 = 2 \times 3$ *est composé, tout comme* $21 = 3 \times 7$*, mais 11 est premier car 1 et 11 sont les seuls diviseurs de 11. Les nombres 0 et 1 ne sont ni premiers ni composés.*

Déterminez les nombres premiers inférieurs à 100. Combien y a-t-il de nombres premiers entre 0 et 100 ? Pour vous aider, nous vous proposons deux méthodes.

Méthode 1 (peu optimale mais assez intuitive)

Pour chaque nombre de 2 à 100, calculez le reste de la division entière (avec l'opérateur modulo `%`) depuis 1 jusqu'à lui-même. Si c'est un nombre premier, il aura exactement deux nombres pour lesquels le reste de la division entière est égal à 0 (1 et lui-même). Si ce n'est pas un nombre premier, il aura plus de deux nombres pour lesquels le reste de la division entière est égal à 0.

Méthode 2 (plus optimale et plus rapide, mais un peu plus compliquée)

Vous pouvez parcourir tous les nombres de 2 à 100 et vérifier si ceux-ci sont composés, c'est-à-dire qu'ils sont le produit de deux nombres premiers. Pratiquement, cela consiste à vérifier que le reste de la division entière (opérateur modulo `%`) entre le nombre considéré et n'importe quel nombre premier est nul. Le cas échéant, ce nombre n'est pas premier.

6.6.11 Recherche d'un nombre par dichotomie (exercice +++)

La recherche par dichotomie est une méthode qui consiste à diviser (en générale en parties égales) un problème pour en trouver la solution. À titre d'exemple, voici une discussion entre Pierre et Patrick dans laquelle Pierre essaie de deviner le nombre (compris entre 1 et 100) auquel Patrick a pensé.

— [Patrick] "C'est bon, j'ai pensé à un nombre entre 1 et 100."
— [Pierre] "OK, je vais essayer de le deviner. Est-ce que ton nombre est plus petit ou plus grand que 50 ?"
— [Patrick] "Plus grand."
— [Pierre] "Est-ce que ton nombre est plus petit, plus grand ou égal à 75 ?"
— [Patrick] "Plus grand."
— [Pierre] "Est-ce que ton nombre est plus petit, plus grand ou égal à 87 ?"
— [Patrick] "Plus petit."
— [Pierre] "Est-ce que ton nombre est plus petit, plus grand ou égal à 81 ?"
— [Patrick] "Plus petit."
— [Pierre] "Est-ce que ton nombre est plus petit, plus grand ou égal à 78 ?"
— [Patrick] "Plus grand."
— [Pierre] "Est-ce que ton nombre est plus petit, plus grand ou égal à 79" ?
— [Patrick] "Egal. C'est le nombre auquel j'avais pensé. Bravo !"

Pour arriver rapidement à deviner le nombre, l'astuce consiste à prendre à chaque fois la moitié de l'intervalle dans lequel se trouve le nombre. Voici le détail des différentes étapes :

1. le nombre se trouve entre 1 et 100, on propose 50 (100 / 2).
2. le nombre se trouve entre 50 et 100, on propose 75 (50 + (100-50)/2).
3. le nombre se trouve entre 75 et 100, on propose 87 (75 + (100-75)/2).
4. le nombre se trouve entre 75 et 87, on propose 81 (75 + (87-75)/2).
5. le nombre se trouve entre 75 et 81, on propose 78 (75 + (81-75)/2).
6. le nombre se trouve entre 78 et 81, on propose 79 (78 + (81-78)/2).

Écrivez un script qui reproduit ce jeu de devinettes. Vous pensez à un nombre entre 1 et 100 et l'ordinateur essaie de le deviner par dichotomie en vous posant des questions.

Vous utiliserez la fonction `input()` pour interagir avec l'utilisateur. Voici un exemple de son fonctionnement :

```
>>> lettre = input("Entrez une lettre : ")
Entrez une lettre : P
>>> print(lettre)
P
```

Pour vous guider, voici ce que donnerait le programme avec la conversation précédente :

```
Pensez à un nombre entre 1 et 100.
Est-ce votre nombre est plus grand, plus petit ou égal à 50 ? [+/-/=] +
Est-ce votre nombre est plus grand, plus petit ou égal à 75 ? [+/-/=] +
Est-ce votre nombre est plus grand, plus petit ou égal à 87 ? [+/-/=] -
Est-ce votre nombre est plus grand, plus petit ou égal à 81 ? [+/-/=] -
Est-ce votre nombre est plus grand, plus petit ou égal à 78 ? [+/-/=] +
Est-ce votre nombre est plus grand, plus petit ou égal à 79 ? [+/-/=] =
J'ai trouvé en 6 questions !
```

Les caractères `[+/-/=]` indiquent à l'utilisateur comment il doit interagir avec l'ordinateur, c'est-à-dire entrer soit le caractère `+` si le nombre choisi est plus grand que le nombre proposé par l'ordinateur, soit

le caractère `-` si le nombre choisi est plus petit que le nombre proposé par l'ordinateur, soit le caractère `=` si le nombre choisi est celui proposé par l'ordinateur (en appuyant ensuite sur la touché *Entrée*).

Chapitre 7

Fichiers

7.1 Lecture dans un fichier

Dans la plupart des travaux de programmation, on doit lire ou écrire dans un fichier. Python possède pour cela tout un tas d'outils qui vous simplifient la vie. Avant de passer à un exemple concret, créez un fichier dans un éditeur de texte que vous enregistrerez dans votre répertoire avec le nom `zoo.txt`, par exemple :

```
girafe
tigre
singe
souris
```

Ensuite, testez le code suivant :

```
>>> filin = open('zoo.txt', 'r')
>>> filin
<_io.TextIOWrapper name='zoo.txt' mode='r' encoding='UTF-8'>
>>> filin.readlines()
['girafe\n', 'tigre\n', 'singe\n', 'souris\n']
>>> filin.close()
>>> f.readlines()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

Il y a plusieurs commentaires à faire sur cet exemple :

- La première commande ouvre le fichier `zoo.txt` en lecture seule (ceci est indiqué avec la lettre `r`). Remarquez que le fichier n'est pas encore lu, mais simplement ouvert (*un peu comme lorsqu'on ouvre un livre, mais qu'on ne l'a pas encore lu*). Le curseur de lecture est prêt à lire le premier caractère du fichier. Lorsqu'on affiche la valeur de la variable `filin`, vous voyez que Python la considère comme un objet de type fichier ouvert : `<_io.TextIOWrapper name='zoo.txt' mode='r' encoding='UTF-8'>`.
- Ensuite, nous rencontrons à nouveau la syntaxe `objet.méthode()` (cf. Chapitre 3) : ici la méthode `readlines()` agit sur l'objet `filin` en déplaçant le curseur de lecture du début à la fin du fichier, puis elle renvoie une liste contenant les lignes du fichier (*dans notre analogie avec un livre, ceci correspondrait à lire toutes les lignes du livre*).
- Enfin, on applique la méthode `close()` sur l'objet `filin`, ce qui, vous vous en doutez, ferme le fichier (*ceci correspondrait à fermer le livre*). On pourra remarquer que la méthode `.close()`

ne renvoie rien mais modifie l'état de l'objet `filin` en fichier fermé. Ainsi, si on essaie de lire à nouveau les lignes du fichier, Python renvoie une erreur car il ne peut pas lire un fichier fermé.

Voici maintenant un exemple complet de lecture d'un fichier avec Python.

```
>>> filin = open('zoo.txt', 'r')
>>> lignes = filin.readlines()
>>> lignes
['girafe\n', 'tigre\n', 'singe\n', 'souris\n']
>>> for ligne in lignes:
...     print(ligne)
...
girafe

tigre

singe

souris

>>> filin.close()
```

Vous voyez qu'en cinq lignes de code, vous avez lu et parcouru le fichier.

Remarques :

— Notez que la liste `lignes` contient le caractère `\n` à la fin de chacun de ses éléments. Ceci correspond au saut à la ligne de chacune d'entre elles (ceci est codé par un caractère spécial que l'on représente par `\n`). Vous pourrez parfois rencontrer également la notation octale `\012`.
— Remarquez aussi que lorsque l'on affiche les différentes lignes du fichier à l'aide de la boucle `for` et de l'instruction `print()`, Python saute à chaque fois une ligne.

7.1.1 Méthode read()

Il existe d'autres méthodes que `readlines()` pour lire (et manipuler) un fichier. Par exemple, la méthode `read()` lit tout le contenu d'un fichier et renvoie une chaîne de caractères unique.

```
>>> filin = open('zoo.txt', 'r')
>>> filin.read()
'girafe\ntigre\nsinge\nsouris\n'
>>> filin.close()
```

7.1.2 Méthode readline()

La méthode `readline()` (sans `s` à la fin) lit une ligne d'un fichier et la renvoie sous forme d'une chaîne de caractères. À chaque nouvel appel de `readline()`, la ligne suivante est renvoyée. Associée à la boucle `while`, cette méthode permet de lire un fichier ligne par ligne.

```
>>> filin = open('zoo.txt', 'r')
>>> ligne = filin.readline()
>>> while ligne != "":
...     print(ligne)
...     ligne = filin.readline()
...
girafe
```

```
tigre

singe

souris

>>> filin.close()
```

7.1.3 Méthodes seek() et tell()

Les méthodes `seek()` et `tell()` permettent respectivement de se déplacer au nième caractère (plus exactement au nième octet) d'un fichier et d'afficher oĂč en est la lecture du fichier, c'est-à-dire quel caractère (ou octet) est en train d'être lu.

```
>>> filin = open('zoo.txt', 'r')
>>> filin.readline()
'girafe\n'
>>> filin.tell()
7
>>> filin.seek(0)
0
>>> filin.tell()
0
>>> filin.readline()
'girafe\n'
>>> filin.close()
```

On remarque qu'à l'ouverture d'un fichier, le tout premier caractère est considéré comme le caractère 0 (tout comme le premier élément d'une liste). La méthode `seek()` permet facilement de remonter au début du fichier lorsque l'on est arrivé à la fin ou lorsqu'on en a lu une partie. En passant, `seek()` confirme le numéro d'octet auquel il s'est déplacé en l'affichant à l'écran.

7.1.4 Itérations directement sur le fichier

Python essaie de vous faciliter la vie au maximum. Voici un moyen à la fois simple et élégant de parcourir un fichier.

```
>>> filin = open('zoo.txt', 'r')
>>> for ligne in filin:
...     print(ligne)
...
girafe

tigre

singe

souris

>>> filin.close()
```

La boucle `for` va demander à Python d'aller lire le fichier ligne par ligne. Privilégiez cette méthode par la suite.

7.2 Écriture dans un fichier

Écrire dans un fichier est aussi simple que de le lire. Voyez l'exemple suivant :

```
>>> animaux2 = ['poisson', 'abeille', 'chat']
>>> filout = open('zoo2.txt', 'w')
>>> for animal in animaux2:
...     filout.write(animal)
...
7
7
4
>>> filout.close()
```

Le contenu du fichier `zoo2.txt` est `poissonabeillechat`.

Quelques commentaires sur cet exemple :

- Après avoir initialisé la liste `animaux2`, nous avons ouvert un fichier mais cette fois-ci en mode écriture (avec le caractère `w`).
- Ensuite, on a balayé cette liste à l'aide d'une boucle. À chaque itération, nous avons écrit chaque élément de la liste dans le fichier. Remarquez à nouveau la méthode `write()` qui s'applique sur l'objet `filout`. Notez qu'à chaque utilisation de la méthode `write()`, celle-ci nous affiche le nombre d'octets écrits dans le fichier. Ceci est valable uniquement dans l'interpréteur, si vous créez un programme avec les mêmes lignes de code, ces valeurs ne s'afficheront pas à l'écran.
- Enfin, on a fermé le fichier avec la méthode `close()`.

Vous voyez qu'il est extrêmement simple en Python de lire ou d'écrire dans un fichier.

Remarque : si votre programme produit uniquement du texte, vous pouvez l'écrire sur la sortie standard (avec l'instruction `print()`). L'avantage est que dans ce cas l'utilisateur peut bénéficier de toute les potentialités d'Unix (redirection, tri, *parsing*...). S'il veut écrire le résultat du programme dans un fichier, il pourra toujours le faire en redirigeant la sortie.

7.3 Méthode optimisée d'ouverture et de fermeture de fichier

Depuis la version 2.5, Python introduit le mot-clé `with` qui permet d'ouvrir et de fermer un fichier de manière commode. Si pour une raison ou une autre l'ouverture conduit à une erreur (problème de droits, etc), l'utilisation de `with` garantit la bonne fermeture du fichier (ce qui n'est pas le cas avec l'utilisation de la méthode `open()` invoquée telle quelle). Voici un exemple :

```
>>> with open('zoo.txt', 'r') as filin:
...     for ligne in filin:
...         print(ligne)
...
girafe

tigre

singe
```

```
souris

>>>
```

Vous remarquez que `with` introduit un bloc d'indentation. C'est à l'intérieur de ce bloc que nous effectuons toutes les opérations sur le fichier. Une fois sorti, Python fermera **automatiquement** le fichier. Vous n'avez donc plus besoin d'invoquer la méthode `close()`.

Pour ceux qui veulent appronfondir, la commande with est plus générale et utilisable dans d'autres contextes (méthode compacte pour gérer les exceptions).

7.4 Note sur les retours chariots sous Unix et sous Windows

Conseil : si vous êtes débutant, vous pouvez sauter cette section.

On a vu plus haut que le caractère spécial `\n` correspondait à un retour à la ligne. C'est le standard en Unix.

Toutefois, Windows utilise deux caractères spéciaux pour le retour à la ligne : `\r` correspondant à un retour chariot (hérité des machines à écrire !) et `\n` comme en Unix.

Si vous avez commencé en Python 2, vous aurez peut-être remarqué que selon les versions la lecture de fichier supprimait parfois les `\r` et d'autres fois les laissait. Heureusement, la fonction `open()` dans Python 3 gère tout ça automatiquement et renvoie uniquement des sauts de ligne sous forme de `\n` (même si le fichier a été conçu sous Windows et qu'il contient initialement des `\r`).

7.5 Importance des conversions de types avec les fichiers

Vous avez noté que les méthodes qui lisent un fichier (par exemple `readlines()`) vous renvoient systématiquement des chaines de caractères. De même, pour écrire dans un fichier il faut passer une chaine de caractères à la méthode `.write()`. Pour ce genre de problème, il faudra utiliser les fonctions de conversions de types vues au chapitre 2 `int()`, `float()` et `str()`. Ces conversions sont essentielles lorsqu'on lit ou écrit des nombres dans un fichier. En effet, les nombres dans un fichier sont considérés comme du texte par la fonction `readlines()`, par conséquent il faut les convertir si on veut effectuer des opérations numériques dessus.

7.6 Exercices

Conseil : pour tous ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

7.6.1 Lecture et saut de ligne

Dans l'exemple `'girafe'`, `'tigre'`, etc, ci-dessus, comment expliquez-vous que Python saute une ligne à chaque itération ? Réécrivez les instructions *ad-hoc* pour que Python écrive le contenu du fichier sans sauter de ligne.

7.6.2 Écriture et saut de ligne

En reprenant le dernier exemple sur l'écriture dans un fichier, vous pouvez constater que les noms d'animaux ont été écrits les uns à la suite des autres, sans retour à la ligne. Comment expliquez-vous ce résultat ? Modifiez les instructions de manière à écrire un animal par ligne.

7.6.3 Structures secondaires

Dans cet exercice, nous allons utiliser une sortie partielle de DSSP (*Define Secondary Structure of Proteins*), qui est un logiciel d'assignation des structures secondaires des protéines. Ce fichier contient 5 colonnes correspondant respectivement au numéro de résidu, à l'acide aminé, sa structure secondaire et ses angles phi/psi.

— Téléchargez le fichier first_helix_1tfe.txt sur le site de notre cours et sauvegardez-le dans votre répertoire de travail (jetez-y un oeil en passant).
— Chargez les lignes de ce fichier en les plaçant dans une liste puis fermez le fichier.
— Écrivez chaque ligne à l'écran pour vérifier que vous avez bien chargé le fichier.
— Écrivez dans un fichier `output.txt` chacune des lignes. N'oubliez pas le retour à la ligne pour chaque acide aminé.
— Écrivez dans un fichier `output2.txt` chacune des lignes suivies du message `line checked` suivi d'un saut à la ligne.

7.6.4 Spirale (exercice +++)

On se propose de réaliser un script qui écrit dans un fichier `spirale.dat` les coordonnées cartésiennes d'une spirale. *Conseil* : si vous faites varier une variable `theta` qui représente un angle entre 0 à 2π (ou un multiple de 2π), les fonctions cosinus et sinus vous permettront d'obtenir les coordonnées des points sur un cercle.

Si vous avez réussi cette première étape, tentez la spirale en faisant en sorte que le rayon du cercle augmente au fur et à mesure que `theta` grandit.

Pour admirer votre spirale, vous pourrez utiliser le programme `xmgrace` en tapant

```
xmgrace spirale.dat
```

dans un shell bash.

Chapitre 8

Modules

8.1 Définition

Les modules sont des programmes Python qui contiennent des fonctions que l'on est amené à réutiliser souvent (on les appelle aussi bibliothèques ou *libraries*). Les développeurs de Python ont mis au point de nombreux modules qui effectuent une quantité phénoménale de tâches. Pour cette raison, prenez le réflexe de vérifier si une partie du code que vous souhaitez écrire n'existe pas déjà sous forme de module. La plupart de ces modules sont déjà installés dans les versions standards de Python. Vous pouvez accéder à une documentation exhaustive sur le site de Python. Explorez un peu ce site, la quantité de modules disponibles est impressionante (plus de 300).

8.2 Importation de modules

Jusqu'à présent, nous avons rencontré une fois cette notion de module lorsque nous avons voulu tirer un nombre aléatoire.

```
>>> import random
>>> random.randint(0,10)
4
```

Regardons de plus près cet exemple :

- L'instruction `import` permet d'accéder à toutes les fonctions du module random.
- Ensuite, nous utilisons la fonction (ou méthode) `randint(a,b)` du module `random`. Attention cette fonction renvoie un nombre entier aléatoirement tiré entre `a` inclus et `b` inclus (contrairement à la fonction `range()` par exemple). Remarquez la notation objet `random.randint()` où la fonction `randint()` peut être considérée comme une méthode de l'objet `random`.

Il existe un autre moyen d'importer une ou des fonctions d'un module :

```
>>> from random import randint
>>> randint(0,10)
7
```

À l'aide du mot-clé `from`, vous pouvez importer une fonction spécifique d'un module donné. Remarquez que dans ce cas il est inutile de répéter le nom du module, seul le nom de ladite fonction est requis.

On peut également importer toutes les fonctions d'un module :

```
>>> from random import *
>>> x = [1, 2, 3, 4]
```

```
>>> shuffle(x)
>>> x
[2, 3, 1, 4]
>>> shuffle(x)
>>> x
[4, 2, 1, 3]
>>> randint(0,50)
46
>>> uniform(0,2.5)
0.64943174760727951
```

Comme vous l'avez deviné, l'instruction `from random import *` importe toutes les fonctions du module `random`. On peut ainsi utiliser toutes ses fonctions directement, comme par exemple `shuffle()`, qui permute une liste aléatoirement.

Dans la pratique, plutôt que de charger toutes les fonctions d'un module en une seule fois (`from random import *`), nous vous conseillons de charger le module (`import random`) puis d'appeler explicitement les fonctions voulues (`random.randint(0,2)`).

Il est également possible de définir un alias (un nom plus court) pour un module :

```
>>> import random as rand
>>> rand.randint(1, 10)
6
>>> rand.uniform(1, 3)
2.643472616544236
```

Dans cet exemple, les fonctions du module `random` sont accessibles via l'alias `rand`.

Enfin, si vous voulez vider de la mémoire un module déjà chargé, vous pouvez utiliser l'instruction `del` :

```
>>> import random
>>> random.randint(0,10)
2
>>> del random
>>> random.randint(0,10)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'random' is not defined
```

Vous constatez qu'un rappel d'une fonction du module `random` après l'avoir vidé de la mémoire retourne un message d'erreur.

8.3 Obtenir de l'aide sur les modules importés

Pour obtenir de l'aide sur un module rien de plus simple, il suffit d'invoquer la commande `help()` :

```
>>> import random
>>> help(random)
...
```

On peut se déplacer dans l'aide avec les flèches ou les touches *page-up* et *page-down* (comme dans les commandes Unix `man`, `more` ou `less`). Il est aussi possible d'invoquer de l'aide sur une fonction particulière d'un module de la manière suivante `help(random.randint)`.

La commande `help()` est en fait une commande plus générale permettant d'avoir de l'aide sur n'importe quel objet chargé en mémoire.

```
>>> t = [1, 2, 3]
>>> help(t)
Help on list object:

class list(object)
 |  list() -> new list
 |  list(sequence) -> new list initialized from sequence's items
 |
 |  Methods defined here:
 |
 |  __add__(...)
 |      x.__add__(y) <==> x+y
 |
...
```

Si on veut connaître d'un seul coup d'oeil toutes les méthodes ou variables associées à un objet, on peut utiliser la fonction `dir()` :

```
>>> import random
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'SystemRandom',
'TWOPI', 'WichmannHill', '_BuiltinMethodType', '_MethodType', '__all__', '__builtins__',
'__doc__', '__file__', '__name__', '_acos', '_ceil', '_cos', '_e', '_exp', '_hexlify',
'_inst', '_log', '_pi', '_random', '_sin', '_sqrt', '_test', '_test_generator',
'_urandom', '_warn', 'betavariate', 'choice', 'expovariate', 'gammavariate', 'gauss',
'getrandbits', 'getstate', 'jumpahead', 'lognormvariate', 'normalvariate',
'paretovariate', 'randint', 'random', 'randrange', 'sample', 'seed', 'setstate',
'shuffle', 'uniform', 'vonmisesvariate', 'weibullvariate']
>>>
```

8.4 Modules courants

Il existe une série de modules que vous serez probablement amenés à utiliser si vous programmez en Python. En voici une liste non exhaustive. Pour la liste complète, reportez-vous à la page des modules sur le site de Python :

- math : fonctions et constantes mathématiques de base (sin, cos, exp, pi...).
- sys : passage d'arguments, gestion de l'entrée/sortie standard...
- os : dialogue avec le système d'exploitation (permet de sortir de Python, lancer une commande en *shell*...).
- random : génération de nombres aléatoires.
- time : permet d'accéder à l'heure de l'ordinateur et aux fonctions gérant le temps.
- calendar : fonctions de calendrier.
- profile : permet d'évaluer le temps d'exécution de chaque fonction dans un programme (*profiling* en anglais).
- urllib : permet de récupérer des données sur internet depuis Python.
- tkinter : interface python avec Tk (permet de créer des objets graphiques
- re : gestion des expressions régulières.
- pickle : écriture et lecture de structures Python (comme les dictionnaires par exemple).

Nous vous conseillons vivement d'aller surfer sur les pages de ces modules pour découvrir toutes leurs potentialités.

Vous verrez plus tard comment créer votre propres modules lorsque vous êtes amenés à réutiliser souvent vos propres fonctions.

Enfin, notez qu'il existe de nombreux autres modules qui ne sont pas installés de base dans Python mais qui sont de grand intérêt en bioinformatique (au sens large). Citons-en quelques-uns : `numpy` (notion de matrice, algèbre linéaire, transformée de Fourier), `biopython` (recherche dans les banques de données biologiques), `rpy` (dialogue R/Python)...

8.5 Module sys : passage d'arguments

Le sys contient (comme son nom l'indique) des fonctions et des variables spécifiques au système, ou plus exactement à l'interpréteur lui-même. Par exemple, il permet de gérer l'entrée (*stdin*) et la sortie standard (*stdout*). Ce module est particulièrement intéressant pour récupérer les arguments passés à un script Python lorsque celui-ci est appelé en ligne de commande. Dans cet exemple, écrivons le court script suivant que l'on enregistrera sous le nom `test.py` (n'oubliez pas de le rendre exécutable) :

```
#!/usr/bin/env python3

import sys
print(sys.argv)
```

Ensuite lancez `test.py` suivi de plusieurs arguments. Par exemple :

```
$ python3 test.py salut girafe 42
['test.py', 'salut', 'girafe', '42']
```

Dans l'exemple précédent, `poulain@cumin>` représente l'invite du *shell* Linux, `test.py` est le nom du script Python, `salut`, `girafe` et `42` sont les arguments passés au script.

La variable `sys.argv` est une liste qui représente tous les arguments de la ligne de commande, y compris le nom du script lui même qu'on peut retrouver dans `sys.argv[0]`. On peut donc accéder à chacun de ces arguments avec `sys.argv[1]`, `sys.argv[2]`...

On peut aussi utiliser la fonction `sys.exit()` pour quitter un script Python. On peut donner un argument à cette fonction (en général une chaîne de caractères) qui sera renvoyé au moment où Python quittera le script. Par exemple, si vous attendez au moins un argument en ligne de commande, vous pouvez renvoyer un message pour indiquer à l'utilisateur ce que le script attend comme argument :

```
#!/usr/bin/env python3

import sys

if len(sys.argv) != 2:
    sys.exit("ERREUR : il faut exactement un argument.")

#
# suite du script
#
```

Puis on l'exécute sans argument :

```
$ python3 test.py
ERREUR : il faut exactement un argument.
```

Notez qu'ici on vérifie que le script possède deux arguments car le nom du script lui-même est le premier argument et `file.txt` constitue le second.

8.6 Module os

Le module `os` gère l'interface avec le système d'exploitation.

`os.path.exists()` est une fonction pratique de ce module qui vérifie la présence d'un fichier sur le disque.

```
>>> import sys
>>> import os
>>> if os.path.exists("toto.pdb"):
...     print("le fichier est présent")
... else:
...     sys.exit("le fichier est absent")
...
le fichier est absent
```

Dans cet exemple, si le fichier n'est pas présent sur le disque, on quitte le programme avec la fonction `exit()` du module `sys`.

La fonction `system()` permet d'appeler n'importe quelle commande externe.

```
>>> import os
>>> os.system("ls -al")
total 5416
drwxr-xr-x 2 poulain bioinfo    4096 2010-07-21 14:33 .
drwxr-xr-x 6 poulain bioinfo    4096 2010-07-21 14:26 ..
-rw-r--r-- 1 poulain bioinfo  124335 2010-07-21 14:31 1BTA.pdb
-rw-r--r-- 1 poulain bioinfo 4706057 2010-07-21 14:31 NC_000913.fna
-rw-r--r-- 1 poulain bioinfo  233585 2010-07-21 14:30 NC_001133.fna
-rw-r--r-- 1 poulain bioinfo  463559 2010-07-21 14:33 NC_001133.gbk
0
```

La commande externe `ls -al` est introduite comme une chaîne de caractères à la fonction `system()`. Attention à ne pas faire de bétise avec cette fonction qui peut exécuter n'importe quelle commande Unix.

8.7 Exercices

Conseil : pour les trois premiers exercices, utilisez l'interpréteur Python. Pour les exercices suivants, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

8.7.1 Racine carrée

Affichez sur la même ligne les nombres de 10 à 20 (inclus) ainsi que leur racine carrée avec 3 décimales. Utilisez pour cela le module `math` avec la fonction `sqrt()` (documentation). Exemple :

```
10 3.162
11 3.317
12 3.464
13 3.606
```

...

8.7.2 Cosinus

Calculez le cosinus de pi/2 en utilisant le module `math` avec la fonction `cos()` (documentation) et la constante `pi` (documentation).

8.7.3 Liste de fichiers dans un répertoire

Affichez la liste des fichiers du répertoire courant avec le module `os`. N'utilisez pas la fonction `os.system()` mais la fonction `os.listdir()`. Lisez la documentation pour comprendre comment l'utiliser.

8.7.4 Affichage temporisé

Affichez les nombres de 1 à 10 avec 1 seconde d'intervalle. Utilisez pour cela le module `time` et sa fonction `sleep()` (documentation).

8.7.5 Séquences aléatoires de chiffres

Générez une séquence aléatoire de 6 chiffres, ceux-ci étant des entiers tirés entre 1 et 4. Utilisez le module `random` et la fonction `randint()` (documentation).

8.7.6 Séquences aléatoires de bases

Générez une séquence aléatoire de 20 bases de deux manières différentes. Utilisez le module `random` avec la fonction `randint()` (documentation) ou `choice()` (documentation).

8.7.7 Jour de naissance

Déterminez votre jour (lundi, mardi...) de naissance. Utilisez le module `calendar` avec la fonction `weekday()` (documentation).

8.7.8 Détermination du nombre pi par la méthode Monte Carlo (exercice +++)

Soit un cercle de rayon 1 (en rouge) inscrit dans un carré de coté 2 (en bleu).

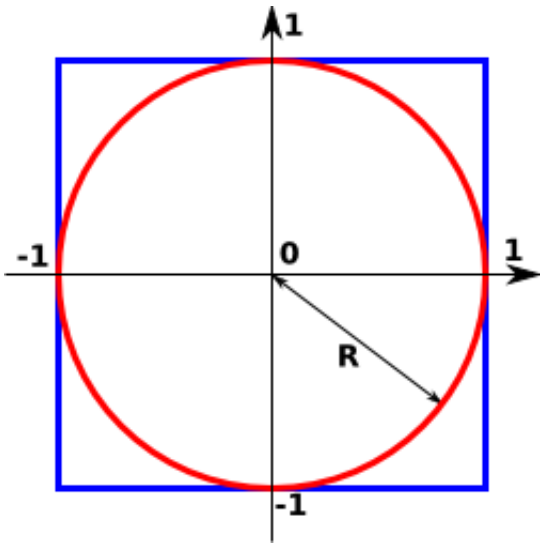
Avec $R = 1$, l'aire du carré vaut $(2R)^2$ soit 4 et l'aire du cercle vaut πR^2 soit π.

En choisissant N points aléatoires (à l'aide d'une distribution uniforme) à l'intérieur du carré, la probabilité que ces points se trouvent aussi dans le cercle est

$$p = \frac{\text{aire du cercle}}{\text{aire du carré}} = \frac{\pi}{4}$$

Soit n, le nombre points effectivement dans le cercle, il vient alors

$$p = \frac{n}{N} = \frac{\pi}{4},$$



Cercle de rayon 1 inscrit dans un carré de côté 2

d'où

$$\pi = 4 \times \frac{n}{N}.$$

Déterminez une approximation de π par cette méthode. Pour cela, pour N itérations :

- Choisissez aléatoirement les coordonnées x et y d'un point entre -1 et 1. Utilisez la fonction `uniform()` (documentation) du module `random`.
- Calculez la distance entre le centre du cercle et ce point.
- Déterminez si cette distance est inférieure au rayon du cercle, c'est-à-dire si le point est dans le cercle ou pas.
- Si le point est effectivement dans le cercle, incrémentez le compteur `n`.

Finalement calculez le rapport entre n et N et proposer une estimation de π. Quelle valeur de π obtenez-vous pour pour 100 itérations ? 1000 itérations ? 10000 itérations ? Comparez les valeurs obtenues à la valeur de π fournie par le module `math`.

On rappelle que la distance d entre deux points A et B de coordonnées respectives (x_A, y_A) et (x_B, y_B) se calcule comme :

$$d = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$$

Chapitre 9

Fonctions

9.1 Principe et généralités

En programmation, les **fonctions** sont très utiles pour réaliser plusieurs fois la même opération au sein d'un programme. Elles permettent également de rendre le code plus lisible et plus clair en le fractionnant en blocs logiques.

Vous connaissez déjà certaines fonctions Python, par exemple `math.cos(angle)` du module `math` renvoie le cosinus de la variable `angle` exprimé en radian. Vous connaissez aussi des fonctions internes à Python comme `range()` ou `len()`. Pour l'instant, une fonction est à vos yeux une sorte de *boîte noire* (cf Figure 9.1) :

— A laquelle vous passez une (ou zero ou plusieurs) valeur(s) entre parenthèses. Ces valeurs sont appelées arguments.
— Qui effectue une action. Par exemple `random.shuffle()` permute aléatoirement une liste.
— Et qui renvoie éventuellement un résultat (plus précisément un objet).

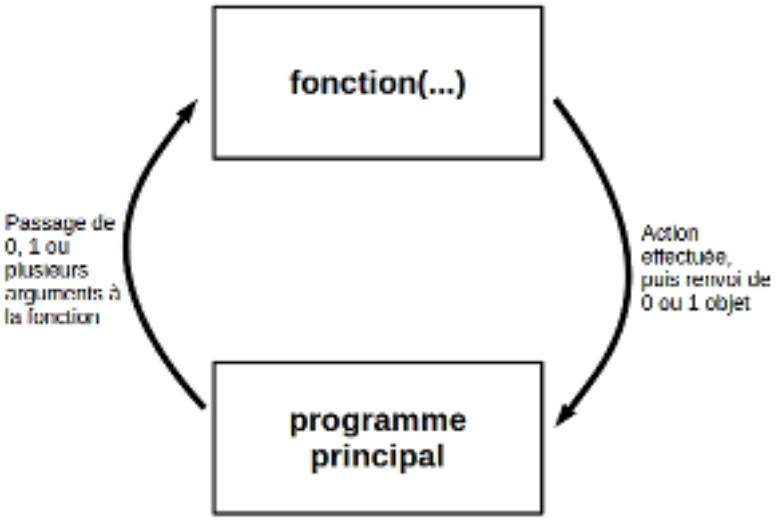


FIGURE 9.1 – Fonctionnement schématique d'une fonction

Par exemple si vous appelez la fonction `len()` en lui passant l'argument `[0, 1, 2]` (`len([0, 1, 2])`), celle-ci vous renvoie un entier indiquant la longueur de la liste passée en argument. Autre exemple, si vous appelez la méthode `liste.append(5)` (n'oubliez pas, une méthode est une **fonction** qui agit sur l'objet auquel elle est attachée), `append()` ajoute `5` à `liste` mais ne renvoie rien.

Au contraire, aux yeux du programmeur une fonction est une portion de code effectuant une suite d'instructions bien particulière. Avant de démarrer sur la syntaxe, revenons sur cette notion de *boîte noire* :

— Une fonction effectue une tâche. Pour cela, elle reçoit éventuellement des arguments et renvoie éventuellement un résultat. L'algorithme utilisé au sein de la fonction sein n'intéresse pas directement l'utilisateur. Par exemple, il est inutile de savoir comment la fonction `math.cos()` calcule un cosinus, on a juste besoin de savoir qu'il faut lui passer en argument un angle en radian et qu'elle renvoie le cosinus de cet angle. Ce qu'il se passe à l'intérieur de la fonction ne regarde que le programmeur (c'est-à-dire vous dans ce chapitre).
— Chaque fonction effectue en général une tâche **unique et précise**. Si cela se complique, il est plus judicieux d'écrire plusieurs fonctions (qui peuvent éventuellement s'appeler les unes les autres). Cette **modularité** améliore la qualité générale et la lisibilité du code. Vous verrez qu'en Python, les fonctions présentent une grande flexibilité.

Pour finir sur les généralités, vous avez vu dans la Figure 9.1 que nous avons utilisé le terme **programme principal** (*main* en anglais) pour désigner l'endroit depuis lequel on appelle une fonction (on verra plus tard que l'on peut en fait appeler une fonction de n'importe où). Le programme principal désigne le code qui est exécuté lorsqu'on lance le script Python, c'est à dire toute la suite d'instructions qui commencent à la colonne 1 du script, autrement dit toutes les instructions en dehors des fonctions. En général, dans un script Python, on écrit d'abord les fonctions puis le programme principal (donc celui-ci se situe à la fin du script). Nous aurons l'occasion de revenir sur cette notion de programme principal dans la section 9.4 de ce chapitre ainsi que dans le chapitre 12.

9.2 Définition

Pour définir une fonction, Python utilise le mot-clé `def` et si on veut que celle-ci renvoie une valeur, il faut utiliser le mot-clé `return`. Par exemple :

```
>>> def carre(x):
...     return x**2
...
>>> print(carre(2))
4
```

Notez que la syntaxe de `def` utilise les `:` comme les boucles `for`, `while` ainsi que les tests `if`, un bloc d'instrutions est donc attendu. De même que pour les boucles et les tests, l'**indentation** de ce bloc d'instructions (*i.e.* le corps de la fonction) est **obligatoire**.

Dans l'exemple précédent, nous avons passé un argument à la fonction `carre()` qui nous a retourné une valeur que nous avons affichée à l'écran. Que veut dire valeur retournée ? Et bien cela signifie que cette dernière est stockable dans une variable :

```
>>> res = carre(2)
>>> print(res)
4
```

Ici, le résultat renvoyé par la fonction est stockée dans la variable `res`. Notez qu'une fonction ne prend pas forcément un argument et ne renvoie pas forcément une valeur, par exemple :

```
>>> def hello():
...     print("bonjour")
...
>>> hello()
bonjour
```

Dans ce cas la fonction `hello()` se contente d'imprimer la chaîne de caractères `"hello"` à l'écran. Elle ne prend aucun argument et ne renvoie aucun résultat. Par conséquent, cela n'a pas de sens de vouloir récupérer dans une variable le résultat renvoyé par une telle fonction. Si on essaie tout de même, Python affecte la valeur `None` qui signifie *rien* en anglais :

```
>>> x = hello()
bonjour
>>> print(x)
None
```

Ceci n'est pas une faute car Python n'émet pas d'erreur, toutefois cela ne présente, la plupart du temps, guère d'intérêt.

9.3 Passage d'arguments

Le nombre d'argument(s) que l'on peut passer à une fonction est variable. Nous avons vu ci-dessus des fonctions auxquelles on passait 0 ou 1 argument. Dans les chapitres précédentes, vous avez vu des fonctions internes à Python qui prenaient au moins 2 arguments, pour rappel souvenez-vous de `range(1,10)` ou encore `range(1,10,2)`. Le nombre d'argument est donc laissé libre à l'initiative du programmeur qui est en train de développer une nouvelle fonction.

Une particularité des fonctions en Python est que vous n'êtes pas obligé de préciser le type des arguments que vous lui passez, dès lors que les opérations que vous effectuez avec ces arguments sont valides. Python est en effet connu comme étant un langage au *typage dynamique*, c'est-à-dire qu'il reconnaît pour vous le type des variables au moment de l'exécution, par exemple :

```
>>> def fois(x,y):
...     return x*y
...
>>> fois(2,3)
6
>>> fois(3.1415,5.23)
16.430045000000003
>>> fois('to',2)
'toto'
```

L'opérateur `*` reconnait plusieurs types (entiers, réels, chaînes de caractères), notre fonction est donc capable d'effectuer des tâches différentes ! Même si Python permet cela, méfiez-vous tout de même de cette grande flexibilité qui pourrait mener à des surprises dans vos futurs programmes. En général il est plus judicieux que chaque argument ait un type précis (*int*, *str*, *float*, etc), et pas l'un ou l'autre.

Un énorme avantage en Python est que les fonctions sont capables de renvoyer plusieurs valeurs à la fois, comme dans cette fraction de code :

```
>>> def carre_cube(x):
...     return x**2,x**3
...
>>> carre_cube(2)
(4, 8)
```

En réalité Python ne renvoie qu'un seul objet, mais celui-ci peut être séquentiel, c'est à dire contenir lui même plusieurs objets. Dans notre exemple Python renvoie un objet de type `tuple`, type que nous verrons dans le chapitre 13 (il s'agit d'une sorte de liste avec des propriétés différentes). Notre fonction pourrait tout autant renvoyer une liste :

```
>>> def carre_cube2(x):
```

```
...     return [x**2,x**3]
...
>>> carre_cube2(3)
[9, 27]
```

Renvoyer un *tuple* ou une liste de deux arguments (ou plus) est notamment très pratique en conjonction avec l'**affectation multiple**, par exemple :

```
>>> z1,z2 = carre_cube2(3)
>>> z1
9
>>> z2
27
```

Cela permet de récupérer plusieurs valeurs retournées par une fonction et les affecter à des variables différentes à la volée.

Enfin, il est possible de passer un ou plusieurs argument(s) de manière facultative et de leur attribuer une valeur par défaut :

```
>>> def useless_fct(x=1):
...     return x
...
>>> useless_fct()
1
>>> useless_fct(10)
10
```

Notez que si on passe plusieurs arguments à une fonction, le ou les arguments facultatifs doivent être situés après les arguments obligatoires. Il faut donc écrire `def fct(x, y, z=1):`.

9.4 Variables locales et variables globales

Lorsqu'on manipule les fonctions il est essentiel de bien comprendre comment se comportent les variables. Une variable est dite *locale* lorsqu'elle est créée dans une fonction, car elle n'existera et ne sera visible que lors de l'exécution de la dite fonction. Une variable est dite *globale* lorsqu'elle est créée dans le programme principal ; elle sera visible partout dans le programme.

Ceci ne vous parait pas clair ? Nous allons suivre un exemple simple illustrant notre propos et qui vous permettra de saisir aisément ces concepts. Regardez le code suivant :

```
# definition d'une fonction carre()
def carre(x):
    y = x**2
    return y

# programme principal
z = 5
resultat = carre(5)
print(resultat)
```

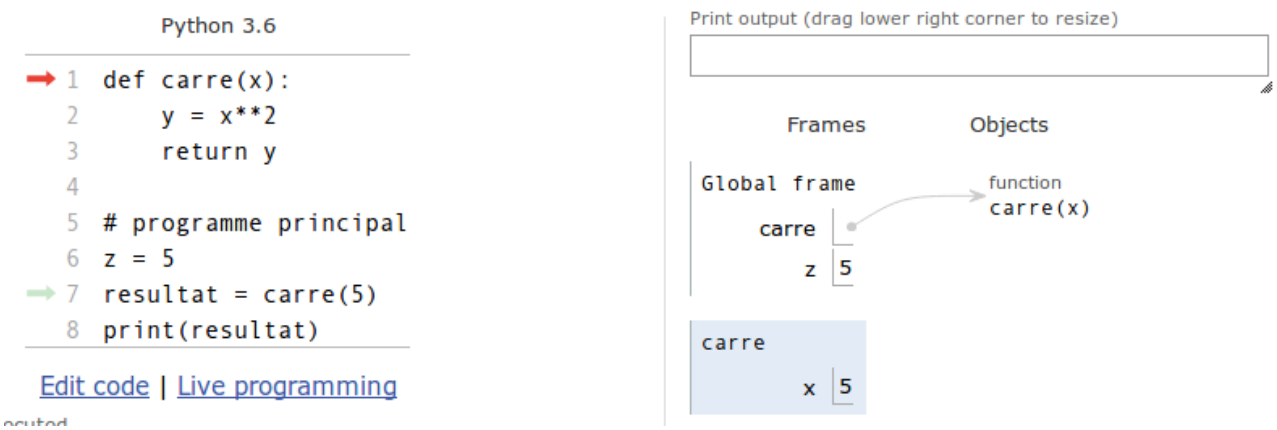
Pour la suite nous allons utilisé l'excellent site pythontutor qui permet de visualiser l'état des variables au fur et à mesure de l'exécution d'un code Python (avant de continuer nous vous conseillons de prendre 5 minutes pour tester ce site). Regardons maintenant ce qu'il se passe dans le code ci-dessus, étape par étape :

— Etape 1 : Python est prêt à lire la première ligne de code.

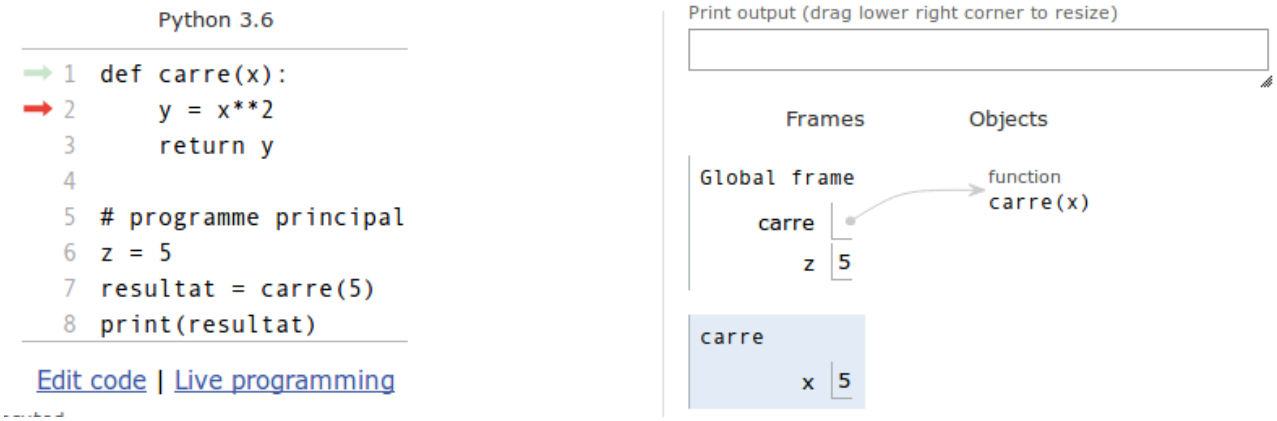
— Etape 2 : Python met en mémoire la fonction `carre()` (notez qu'il ne l'exécute pas !). La fonction est mise dans une case de la mémoire nommée *global frame* qui est l'espace mémoire du programme principal. Y seront stockées toutes les variables *globales* créées dans le programme. Python est maintenant prêt à exécuter le programme principal.

— Etape 3 : Python lit et met en mémoire la variable `z`. Celle-ci étant créée dans le programme principal, il s'agira d'une variable *globale*. Ainsi, elle sera également stockée dans le *global frame*.

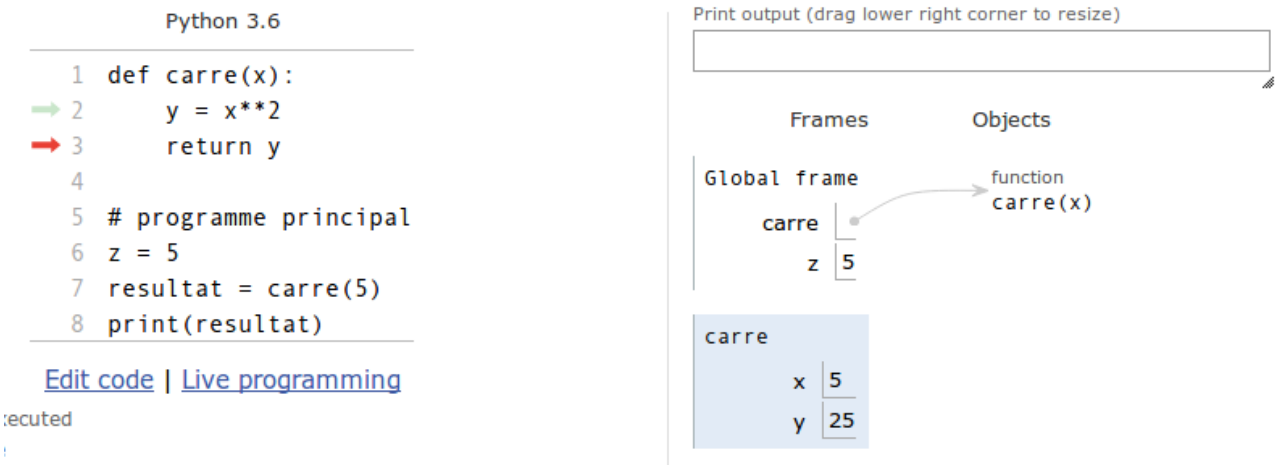
— Etape 4 : La fonction `carre()` est appelée et on lui passe en argument l'entier `5`. La fonction rentre alors en exécution et un nouveau cadre bleu est créé dans lequel *pythontutor* va nous indiquer toutes les variables *locales* à la fonction. Notez bien que la variable passée en argument, qui s'appelle `x` dans la fonction, est créée en tant que variable *locale*. On pourra aussi remarquer que les variables *globales* situées dans le *global frame* sont toujours là.



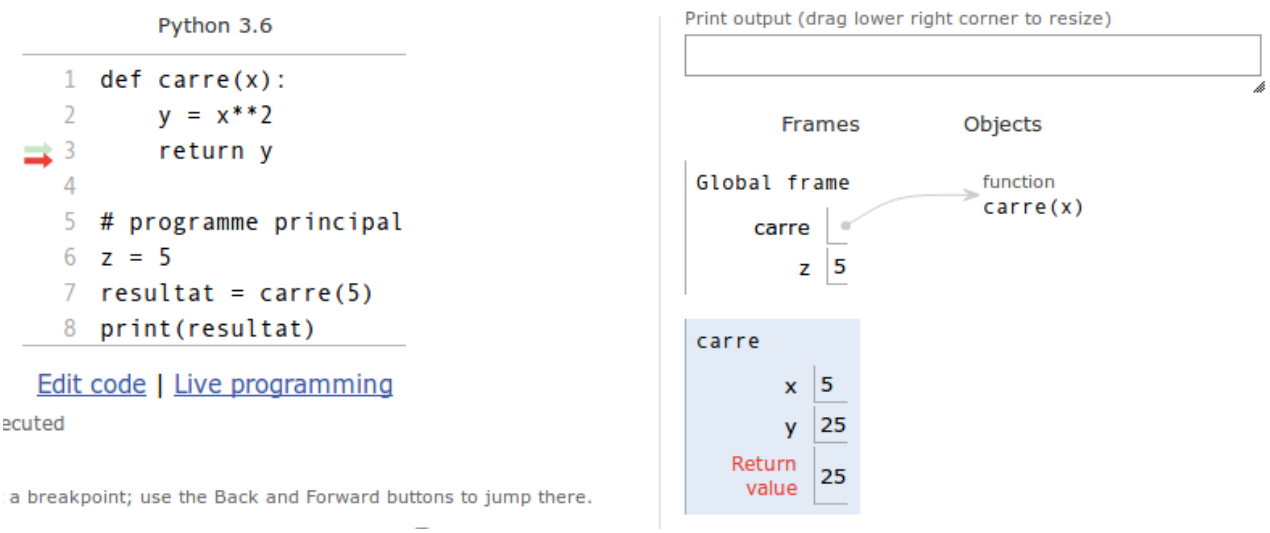
— Etape 5 : Python est maintenant prêt à exécuter chaque ligne de code de la fonction.



— Etape 6 : La variable `y` est créée dans la fonction. Celle-ci est donc stockée en tant que variable *locale* à la fonction.

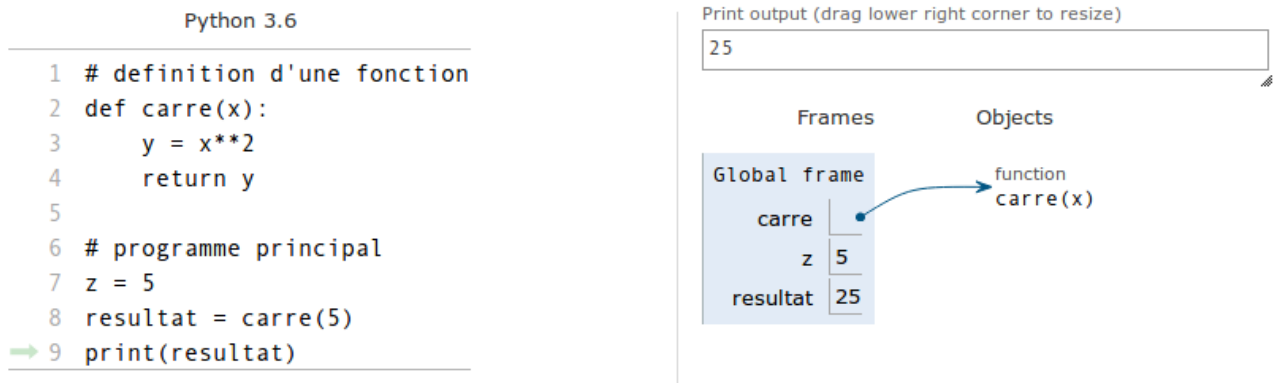


— Etape 7 : Python s'apprête à retourner la variable *locale* `y` au programme principal (*pythontutor* nous indique le contenu de la valeur retournée).



— Etape 8 : Python quitte la fonction et la valeur retournée par celle-ci est affectée à la variable *globale* `resultat`. Notez bien que lorsque Python quitte la fonction, **l'espace des variables allouées à la fonction est détruit**. Ainsi toutes les variables créées dans la fonction n'existent plus. On comprend pourquoi elles portent le nom de *locales* puisqu'elles n'existent que lorsque la fonction est en exécution.

— Etape 9 : Python affiche le contenu de la variable `resultat` et l'exécution est terminée.



Nous espérons que cet exemple guidé facilitera la compréhension des concepts de variables locales et globales. Cela viendra aussi avec la pratique. Nous irons un peu plus loin sur les fonctions dans le chapitre 12. D'ici là essayez de vous entrainer au maximum avec les fonctions. C'est un concept ardu, mais il est impératif de savoir les maitriser.

Enfin, comme vous avez vu, *pythontutor* nous a grandement aidés à comprendre ce qu'il se passait.

N'hésitez pas à l'utiliser sur des exemples ponctuels, celui-ci pourra vous aider à comprendre lorsqu'un code ne fait pas ce que vous attendez.

9.5 Exercices

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

9.5.1 Fonctions et pythontutor

Reprenez l'exemple de la section 9.4 à l'aide du site pythontutor. Regardez ensuite le code suivant et tentez de prédire sa sortie :

```
def calc_factorielle(n):
    fact = 1
    for i in range(2,n+1):
        fact = fact * i
    return fact

# prog principal
nb = 4
factorielle_nb = calc_factorielle(nb)
print("{}! = {}".format(nb,factorielle_nb))
nb2 = 10
print("{}! = {}".format(nb2,calc_factorielle(nb2)))
```

Testez ensuite cette portion de code avec *pythontutor* en cherchant à bien comprendre chaque étape. Avez-vous réussi à prédire la sortie correctement ?

9.5.2 Fonction puissance

La fonction `math.pow(x,y)` permet de calculer x^y. Reprogrammer une fonction `calc_puissance(x,y)` qui renvoie x^y en utilisant l'opérateur `**`. Dans le programme principal, calculez et affichez à l'écran 2^i avec i variant de 0 à 20. On veut que le résultat soit présenté avec le formatage suivant :

```
2^ 0 =       1
2^ 1 =       2
2^ 2 =       4
[...]
2^20 = 1048576
```

9.5.3 Fonction pyramide

Reprenez l'exercice 5.4.10 qui dessine un triangle isocele. Concevez une fonction `gen_pyramide()` à laquelle vous passez un nombre entier `N` et qui renvoie une pyramide de N lignes (i.e. triangle isocèle) sous forme de chaîne de caractères. Le programme principal demandera à l'utilisateur le nombre de lignes souhaitées (pensez à la fonction `input()` !) et affichera la pyramide à l'écran.

9.5.4 Fonction nombre premier

Reprenez l'exercice 6.6.10 sur les nombres premiers. Concevez une fonction `is_prime()` qui prend en argument un ombre entier positif `n` (supérieur à 2) et qui renvoie un booléen `True` si n est premier, et `False` si n n'est pas premier. Déterminez tous les nombres premiers de 2 à 100. On souhaite avoir une sortie comme ça :

```
  2 est premier
  3 est premier
  4 n'est pas premier
[...]
 10 n'est pas premier
[...]
100 n'est pas premier
```

9.5.5 Fonction complement

Reprenez l'exercice 6.6.2. Créez une fonction `complement()` qui prend une liste de bases et qui renvoie la séquence complémentaire d'une séquence d'ADN sous forme de liste.

Dans le programme principal, à partir d'une séquence d'ADN `seq = ['A', 'T', 'C', 'G', 'A', 'T', 'C', 'G', 'A', 'T', 'C', 'G', 'C', 'T', 'G', 'C', 'T', 'A', 'G', 'C']` sur le brin principal, affichez `seq` et sa séquence complémentaire (en utilisant votre fonction `complement()`). On souhaite avoir une sortie comme suit :

```
brin direct        : 5'-XXXXXXXXX-3'
brin complémentaire: 3'-XXXXXXXXX-5'
```

Les 5' et 3' indique le sens de lecture, et les `X` représentent les bases `A`, `T`, `G` ou `C`.

9.5.6 Fonction distance

Créez une fonction `calc_distance3D()` qui calcule une distance euclidienne en 3 dimensions entre deux atomes. Testez votre fonction sur les 2 points `A(0,0,0)` et `B(1,1,1)`. Trouvez-vous bien $\sqrt{3}$?

On rappelle que la distance euclidienne d entre deux points A et B de coordonnées cartésiennes respectives (x_A, y_A, z_A) et (x_B, y_B, z_B) se calcule comme suit :

$$d = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2 + (z_B - z_A)^2}$$

9.5.7 Fonctions distribution et stat

Créez une fonction `gen_distrib()` qui prend en argument les 3 entiers `debut`, `fin` et `n`. La fonction renverra une liste de n nombres réels aléatoires entre *debut* et *fin*. Cette liste suivra une distribution uniforme (utilisez pour cela la fonction `uniform()` du module `random`). Créez une autre fonction `calc_stat()` qui prend en argument une liste de nombres réels et qui renvoie une liste de 4 éléments contenant respectivement le minimum, le maximum, la moyenne et la médiane de la liste. Dans le programme principal, on souhaite générer 20 listes aléatoires de 100 nombres réels entre 0 et 100 et afficher les statistiques pour chacune d'entre elles (minimum, maximum, médiane et moyenne). On souhaite avoir une sortie comme suit :

```
Liste  1 : min = X.XXXX ; max = X.XXXX ; mediane = X.XXXX ; moyenne = X.XXXX
[...]
```

```
Liste 20 : min = X.XXXX ; max = X.XXXX ; mediane = X.XXXX ; moyenne = X.XXXX
```

Pour chaque ligne, on aura les statistiques (min, max, médiane, moyenne) associées à chacune des 20 listes. Les écarts sur les statistiques entre les différentes listes sont-ils importants ?

Relancez avec des listes de 1000 éléments, puis 10000 éléments. Les écarts changent-ils quand le nombre d'éléments par liste augmente ?

9.5.8 Fonction distance à l'origine

En reprenant votre fonction `calc_distance3D()`, faites-en une version 2D (`calc_distance2D()`). Ecrire une autre fonction `calc_dist2ori()` à laquelle vous passez en argument deux listes de nombres réels `list_x` et `list_y` représenant les coordonnées en 2D d'une fonction mathématique (par exemple x et $sin(x)$). Cette fonction renverra une liste de nombres réels représentant la distance entre chaque point de la fonction et l'origine (de coordonnées $(0, 0)$). Votre programme devra générer un fichier `sin2ori.dat` qui contiendra deux colonnes : la première représente les x, la deuxième la distance entre chaque point de la fonction $sin(x)$ à l'origine. On souhaite faire ce calcul entre $-\pi$ et π par pas de 0.1. Vous pourrez admirer votre résultat avec la commande :

```
xmgrace sin2ori.dat
```

9.5.9 Fonction aire sous la courbe (exercice ++++)

La méthode des trapèzes permet de calculer numériquement l'intégrale d'une fonction. Elle consiste à évaluer l'aire sous une fonction en évaluant l'aire de trapèzes successifs. On souhaite créer une fonction `calc_aire()` qui prend en argument deux listes de *floats* `list_x` et `list_y` représentant les coordonnées d'une fonction (par exemple x et $sin(x)$) et qui renvoie l'aire sous la courbe. On calculera les intégrales suivantes :

- $\int_0^1 x\,dx$
- $\int_0^1 \sqrt{x}\,dx$
- $\int_{-\pi}^{+\pi} sin(x)\,dx$
- $\int_{-\pi}^{+\pi} cos(x)\,dx$

Chapitre 10

Plus sur les chaînes de caractères

10.1 Préambule

Nous avons déjà abordé les chaînes de caractères dans le chapitre *variables* et *affichage*. Ici nous allons un peu plus loin notamment avec les méthodes associées aux chaînes de caractères.

10.2 Chaînes de caractères et listes

Les chaînes de caractères peuvent être considérées comme des listes (un peu particulières).

```
>>> animaux = "girafe tigre"
>>> animaux
'girafe tigre'
>>> len(animaux)
12
>>> animaux[3]
'a'
```

Nous pouvons donc utiliser certaines propriétés des listes comme les tranches :

```
>>> animaux = "girafe tigre"
>>> animaux[0:4]
'gira'
>>> animaux[9:]
'gre'
>>> animaux[:-2]
'girafe tig'
```

Mais *a contrario* des listes, les chaînes de caractères présentent toutefois une différence notable, ce sont **des listes non modifiables**. Une fois définie, vous ne pouvez plus modifier un de ses éléments. Le cas échéant, Python renvoie un message d'erreur :

```
>>> animaux = "girafe tigre"
>>> animaux[4]
'f'
>>> animaux[4] = "F"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Par conséquent, si vous voulez modifier une chaîne, vous êtes obligés d'en construire une nouvelle. Pour cela, n'oubliez pas que les opérateurs de concaténation (+) et de duplication (*) (voir le chapitre *variables*) peuvent vous aider. Vous pouvez également générer une liste, qui elle est modifiable, puis revenir à une chaîne de caractères.

10.3 Caractères spéciaux

Il existe certains caractères spéciaux comme `\n` que nous avons déjà vu (pour le retour à la ligne). Le caractère `\t` vous permet d'écrire une tabulation. Si vous voulez écrire un guillemet simple ou double (et que celui-ci ne soit pas confondus avec les guillemets de déclaration de la chaîne de caractères), vous pouvez utiliser `\'` ou `\"` ou utiliser respectivement des guillements doubles ou simple pour déclarer votre chaîne de caractères.

```
>>> print("Un retour à la ligne\npuis une tabulation\t, puis un guillemet\"")
Un retour à la ligne
puis une tabulation     , puis un guillemet"
>>> print('J\'affiche un guillemet simple')
J'affiche un guillemet simple
>>> print("Un brin d'ADN")
Un brin d'ADN
>>> print('Python est un "super" langage de programmation')
Python est un "super" langage de programmation
```

Lorsqu'on souhaite écrire un texte sur plusieurs lignes, il est très commode d'utiliser les guillemets triples permettant de conserver le formatage (notamment les retours à la ligne) :

```
>>> x = '''souris
... chat
... abeille'''
>>> x
'souris\nchat\nabeille'
>>> print(x)
souris
chat
abeille
```

10.4 Méthodes associées aux chaînes de caractères

Voici quelques méthodes spécifiques aux objets de type `string` :

```
>>> x = "girafe"
>>> x.upper()
'GIRAFE'
>>> x
'girafe'
>>> 'TIGRE'.lower()
'tigre'
```

Les fonctions `lower()` et `upper()` renvoient un texte en minuscule et en majuscule respectivement. On remarque que l'utilisation de ces fonctions n'altèrent pas la chaîne de départ mais renvoie la chaîne transformée.

Pour mettre en majuscule la première lettre seulement, vous pouvez faire :

```
>>> x[0].upper() + x[1:]
'Girafe'
```

ou encore plus simple avec la fonction Python adéquate :

```
>>> x.capitalize()
'Girafe'
```

Il existe une méthode associée aux chaînes de caractères qui est particulièrement pratique, la fonction `split()` :

```
>>> animaux = "girafe tigre singe"
>>> animaux.split()
['girafe', 'tigre', 'singe']
>>> for animal in animaux.split():
...     print animal
...
girafe
tigre
singe
```

La fonction `split()` découpe la ligne en plusieurs éléments appelés *champs*, en utilisant comme séparateur les espaces ou les tabulations. Il est possible de modifier le séparateur de champs, par exemple :

```
>>> animaux = "girafe:tigre:singe"
>>> animaux.split(":")
['girafe', 'tigre', 'singe']
```

La fonction `find()` recherche une chaîne de caractères passée en argument.

```
>>> animal = "girafe"
>>> animal.find('i')
1
>>> animal.find('afe')
3
>>> animal.find('tig')
-1
```

Si l'élément recherché est trouvé, alors l'indice du début de l'élément dans la chaîne de caractères est renvoyé. Si l'élément n'est pas trouvé, alors la valeur `-1` est renvoyée.

Si l'élément recherché est trouvé plusieurs fois, seul l'indice de la première occurence est retourné :

```
>>> animaux = "girafe tigre"
>>> animaux.find("i")
1
```

On trouve aussi la fonction `replace()`, qui serait l'équivalent de la fonction de substitution de la commande Unix *sed* :

```
>>> animaux = "girafe tigre"
>>> animaux.replace("tigre", "singe")
'girafe singe'
>>> animaux.replace("i", "o")
'gorafe togre'
```

Enfin, la fonction `count()` compte le nombre d'occurences d'une chaîne de caractères passée en argument :

```
>>> animaux = "girafe tigre"
>>> animaux.count("i")
2
>>> animaux.count("z")
0
>>> animaux.count("tigre")
1
```

10.5 Conversion d'une liste de chaînes de caractères en une chaîne de caractères

On a vu dans le chapitre 2 la conversion des types simples (entier, réel et chaines de caractères) en un autre avec les fonctions `int()`, `float()` et `str()`. La conversion d'une liste de chaînes de caractères en une chaîne de caractères est un peu particulière puisqu'elle fait appelle à la fonction `join()`.

```
>>> seq = ["A", "T", "G", "A", "T"]
>>> seq
['A', 'T', 'G', 'A', 'T']
>>> "-".join(seq)
'A-T-G-A-T'
>>> " ".join(seq)
'A T G A T'
>>> "".join(seq)
'ATGAT'
```

Les éléments de la liste initiale sont concaténés les uns à la suite des autres et intercalés par un séparateur qui peut être n'importe quelle chaîne de caractères. Ici, nous avons utilisé un tiret, un espace et rien.

Attention, la fonction `join()` ne s'applique qu'à une liste de chaînes de caractères.

```
>>> maliste = ["A", 5, "G"]
>>> " ".join(maliste)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 1: expected string, int found
```

Nous espérons qu'après ce petit tour d'horizon vous serez convaincu de la richesse des méthodes associées aux chaînes de caractères. Pour avoir une liste exhaustive de l'ensemble des méthodes associées à une variable particulière, vous pouvez utiliser la fonction `dir()`.

```
>>> dir(animaux)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__ge__',
'__getattribute__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__',
'__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
'__setattr__', '__str__', 'capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition',
'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill']
```

Pour l'instant vous pouvez ignorer les méthodes qui commencent et qui se terminent par deux tirets bas (*underscores*) `__`.

Vous pouvez ensuite accéder à l'aide et à la documentation d'une fonction particulière avec `help()` :

```
>>> help(animaux.split)
Help on built-in function split:

split(...)
    S.split([sep [,maxsplit]]) -> list of strings

    Return a list of the words in the string S, using sep as the
    delimiter string.  If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
    whitespace string is a separator.
(END)
```

10.6 Exercices

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

10.6.1 Parcours d'une liste de chaînes de caractères

Soit la liste `['girafe', 'tigre', 'singe', 'souris']`. Avec une boucle, affichez chaque élément ainsi que sa taille (nombre de caractères).

10.6.2 Fréquence des bases dans une séquence nucléique

Soit la séquence nucléique `ATATACGGATCGGCTGTTGCCTGCGTAGTAGCGT`. On souhaite calculer la fréquence de chaque base A, T, C et G dans cette séquence et afficher le résultat à l'écran. Créez pour cela une fonction `calc_composition()` à laquelle vous passez votre séquence nucléique sous forme de chaîne de caractères et qui renvoie une liste de 4 nombres réels (*float*) indiquant respectivement la fréquence en bases `A`, `T`, `G` et `C`.

10.6.3 Conversion des acides aminés du code à trois lettres au code à une lettre

Soit la séquence protéique `ALA GLY GLU ARG TRP TYR SER GLY ALA TRP`. Transformez cette séquence en une chaîne de caractères en utilisant le code à une lettre pour les acides aminés.

Rappel de la nomenclature des acides aminés :

Acide aminé	Code 3-lettres	Code 1-lettre
Alanine	Ala	A
Arginine	Arg	R
Asparagine	Asn	N
Aspartate	Asp	D
Cystéine	Cys	C
Glutamate	Glu	E
Glutamine	Gln	Q
Glycine	Gly	G

Acide aminé	Code 3-lettres	Code 1-lettre
Histidine	His	H
Isoleucine	Ile	I
Leucine	Leu	L
Lysine	Lys	K
Méthionine	Met	M
Phénylalanine	Phe	F
Proline	Pro	P
Sérine	Ser	S
Thréonine	Thr	T
Tryptophane	Trp	W
Tyrosine	Tyr	Y
Valine	Val	V

10.6.4 Distance de Hamming

La distance de Hamming mesure la différence entre deux séquences de même taille en sommant le nombre de positions qui, pour chaque séquence, ne correspondent pas au même acide aminé.

Écrivez la fonction `hamming()` qui prend en argument deux chaînes de caractères et qui renvoie la distance de Hamming entre ces deux chaînes de caractères.

Calculez la distance de Hamming entre les séquences `AGWPSGGASAGLAIL` et `IGWPSAGASAGLWIL`, puis entre les séquences `ATTCATACGTTACGATT` et `ATACTTACGTAACCATT`.

10.6.5 Palindrome

Un palindrome est un mot ou une phrase dont l'ordre des lettres reste le même si on le lit de gauche à droite ou de droite à gauche. Par exemple, *ressasser* et *Engage le jeu que je le gagne* sont des palindromes.

Écrivez la fonction `palindrome()` qui prend en argument une chaîne de caractères et qui affiche `xxx est un palindrome` si la chaîne de caractères est un palindrome et `xxx n'est pas un palindrome` sinon (bien sur, `xxx` est ici le palindrome en question). Pensez à vous débarasser au préalable des majuscules et des espaces.

Testez ensuite si les expressions suivantes sont des palindromes :

— Radar
— Never odd or even
— Karine alla en Iran
— Un roc si biscornu

10.6.6 Mot composable

Un mot est composable à partir d'une séquence de lettres si la séquence contient toutes les lettres du mot. Comme au Scrabble, chaque lettre de la séquence ne peut être utilisée qu'une seule fois. Par exemple, `coucou` est composable à partir de *uocuoceokzefhu*.

Écrivez la fonction `composable()` qui prendre en argument un mot (chaîne de carctères) et une séquence de lettre (chaîne de caractères) et qui affiche `Le mot xxx est composable à partie de`

`yyy` si le mot (`xxx`) est composable à partir de la séquence de lettres (`yyy`) et `Le mot xxx n'est pas composable à partir de yyy` sinon.

Testez la fonction avec différents mots et séquences.

10.6.7 Alphabet et pangramme

Les codes ASCII des lettres minuscules de l'alphabet vont de 97 (lettre 'a') à 122 (lettre 'z'). La fonction `chr()` prend en argument un code ASCII sous forme d'une entier et renvoie le caractère correspondant. Ainsi `chr(97)` renvoie `'a'`, `chr(98)` renvoie `'b'` et ainsi de suite.

Écrivez la fonction `get_alphabet()` qui utilise une boucle et la fonction `chr()` et qui renvoie une chaîne de caractères contenant toutes les lettres de l'alphabet.

Un pangramme est une phrase comportant au moins une fois chaque lettre de l'alphabet. Par exemple, "Portez ce vieux whisky au juge blond qui fume" est un pangramme.

Écrivez la fonction `pangramme()` qui utilise la fonction `get_alphabet()` précédente, qui prend en argument une chaîne de caractère (`xxx`) et qui renvoie `xxx est un pangramme` si cette chaîne de caractères est un pangramme et `xxx n'est pas un pangramme` sinon. Pensez à vous débarasser des majuscules le cas échéant.

Testez ensuite si les expressions suivantes sont des pangrammes :

— "Portez ce vieux whisky au juge blond qui fume"
— "Monsieur Jack vous dactylographiez bien mieux que votre ami Wolf",
— "Buvez de ce whisky que le patron juge fameux".

10.6.8 Affichage des carbones alpha d'une structure de protéine

Téléchargez le fichier `1bta.pdb` qui correspond à la structure tridimensionnelle de la protéine barstar sur le site de la PDB (lien direct vers le fichier).

Écrivez la fonction `trouve_calpha()` qui prend en argument le nom d'un fichier PDB (sous forme de chaîne de caractères), qui sélectionne uniquement les lignes contenant des carbones alpha, qui stocke ces lignes et les renvoie sous forme de liste.

En utilisant la fonction `trouve_calpha()`, affichez à l'écran les carbones alpha des deux premiers résidus.

Mémo sur le format PDB

Dans un fichier au format PDB, les atomes qui compose une macromolécules sont présentés sous cette forme :

```
ATOM    681  N   GLY A  43     -13.262  -2.363  -1.452  1.00  0.27           N
ATOM    682  CA  GLY A  43     -14.258  -3.210  -0.729  1.00  0.31           C
ATOM    683  C   GLY A  43     -13.627  -3.900   0.484  1.00  0.27           C
ATOM    684  O   GLY A  43     -13.610  -5.112   0.574  1.00  0.32           O
ATOM    685  H   GLY A  43     -12.322  -2.332  -1.164  1.00  0.37           H
ATOM    686  HA2 GLY A  43     -15.075  -2.588  -0.396  1.00  0.39           H
ATOM    687  HA3 GLY A  43     -14.638  -3.963  -1.404  1.00  0.37           H
ATOM    688  N   TRP A  44     -13.125  -3.148   1.425  1.00  0.25           N
ATOM    689  CA  TRP A  44     -12.519  -3.775   2.630  1.00  0.23           C
ATOM    690  C   TRP A  44     -11.944  -2.689   3.540  1.00  0.23           C
ATOM    691  O   TRP A  44     -12.415  -2.466   4.637  1.00  0.27           O
```

Pour sélectionner un carbone alpha, le champ délimité entre les caractères 1 et 6 doit correspondre à `ATOM` et celui délimité entre les caractères 13 et 16 doit correspondre à `CA`.

Les coordonnées tridimensionelles (x, y et z) des atomes se trouvent dans les champs délimités par respectivement les caractères 31 à 38, 39 à 46 et 47 à 54.

N'oubliez pas que Python commence à compter à 0.

10.6.9 Calcul des distances entre les carbones alpha consécutifs d'une structure de protéine

En utilisant la fonction `trouve_calpha()` précédente, calculez la distance inter-atomique entre les carbones alpha des deux premiers résidus (avec deux chiffres après la virgule).

Écrivez ensuite la fonction `calcule_distance()` qui prend en argument la liste renvoyée par la fonction `trouve_calpha()` précédente, qui calcule les distance inter-atomiques entre carbones alpha consécutifs et affiche ces distances sous la forme :

```
numero_calpha_1 numero_calpha_2 distance
```

La distance sera affichée avec deux chiffres après la virgule. Voici un exemple avec les premiers carbones alpha :

```
1 2 3.80
2 3 3.80
3 4 3.83
4 5 3.82
```

On rappelle que la distance d entre deux points A et B de coordonnées respectives (x_A, y_A, z_A) et (x_B, y_B, z_B) se calcule comme :

$$d = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2 + (z_B - z_A)^2}$$

Modifiez maintenant la fonction `calcule_distance()` pour qu'elle affiche à la fin la moyenne des distances.

La distance inter carbones alpha dans les protéines est très stable et de l'ordre de 3,8 angströms. Observez avec attention les valeurs que vous avez calculées pour la protéine 1BTA. Expliquez la valeur surprenante que vous devriez avoir obtenue.

Chapitre 11

Plus sur les listes

11.1 Propriétés des listes

Comme pour les chaînes de caractères, les listes possèdent de nombreuses **méthodes** (on rappelle, une *méthode* est une fonction qui agit sur l'objet à laquelle elle est attachée par un `.`) qui leur sont propres et qui peuvent se révéler très pratiques. Observez les exemples suivants :

— `append()` que nous avons déjà vue au chapitre 4 et qui permet d'ajouter un élément à la fin d'une liste existante.
```
>>> l = [1,2,3]
>>> l.append(5)
>>> l
[1, 2, 3, 5]
```
qui est équivalent à
```
>>> l = [1,2,3]
>>> l = l + [5]
>>> l
[1, 2, 3, 5]
```
Conseil : préférez la version avec `append()` qui est plus compacte et facile à lire.

— `insert()` pour insérer un objet dans une liste avec un indice déterminé.
```
>>> l.insert(2,-15)
>>> l
[1, 2, -15, 3, 5]
```

— `del` pour supprimer un élément d'une liste à une indice déterminé.
```
>>> del l[1]
>>> l
[1, -15, 3, 5]
```
Remarque : contrairement aux autre méthodes associées aux listes, `del` est une fonction générale de Python (utilisable pour d'autres objets que les listes), et celle-ci ne prend pas de parenthèse.

— `remove()` pour supprimer un élément d'une liste à partir de sa valeur.
```
>>> l.remove(5)
>>> l
[1, -15, 3]
```

— `sort()` pour trier une liste.
```
>>> l.sort()
>>> l
[-15, 1, 3]
```

— `reverse()` pour inverser une liste.
```
>>> l.reverse()
```

```
>>> l
[3, 1, -15]
```

— `count()` pour compter le nombre d'éléments (passé en argument) dans une liste.

```
>>> l=[1, 2, 4, 3, 1, 1]
>>> l.count(1)
3
>>> l.count(4)
1
>>> l.count(23)
0
```

Remarque 1 : attention, dans de nombreux exemples de méthodes ci-dessus, une liste remaniée (par exemple `l.sort()`) n'est pas renvoyée (la liste est modifiée de manière interne, mais l'appel de la méthode ne renvoie rien, c'est à dire, pas d'objet récupérable dans une variable) ! Il s'agit d'un exemple d'utilisation de méthode (donc de fonction) qui fait une action mais qui ne renvoie rien. Pensez-y dans vos utilisations futures des listes.

Remarque 2 : attention, certaines fonctions ci-dessus décalent les indices d'une liste (par exemple `insert()`, `del` etc).

La méthode `append()`, que nous avons déjà vue, est particulièrement pratique car elle permet de construire une liste au fur et à mesure des itérations d'une boucle. Pour cela, nous vous rappelons qu'il est commode de définir préalablement une liste vide de la forme `maliste = []`. Voici un exemple où une chaîne de caractères est convertie en liste :

```
>>> seq = 'CAAAGGTAACGC'
>>> seq_list = []
>>> seq_list
[]
>>> for base in seq:
...   seq_list.append(base)
...
>>> seq_list
['C', 'A', 'A', 'A', 'G', 'G', 'T', 'A', 'A', 'C', 'G', 'C']
```

Remarquez que vous pouvez directement utiliser la fonction `list()` qui prend n'importe quel objet séquentiel (liste, chaîne de caractères, *tuples* [que nous verrons au chapitre 13], etc.) et qui renvoie une liste :

```
>>> seq = 'CAAAGGTAACGC'
>>> list(seq)
['C', 'A', 'A', 'A', 'G', 'G', 'T', 'A', 'A', 'C', 'G', 'C']
```

Cette méthode est certes plus simple, mais il arrive parfois que l'on doive utiliser les boucles tout de même, comme lorqu'on lit un fichier. On rappelle l'action `list(seq)` convertit un objet de type chaîne de caractères en un objet de type liste (il s'agit donc de `casting`). De même que `list(range(10))` convertissait un objet de type *range* en un objet de type liste.

Enfin, si vous voulez avoir une liste exhaustive des méthodes disponibles, nous rappellons que vous pouvez utiliser la fonction `dir(l)` (`l` étant un objet de type *list*).

11.2 Test d'appartenance

L'opérateur `in` permet de tester si un élément fait partie d'une liste.

```
liste = [1, 3, 5, 7, 9]
```

```
>>> 3 in liste
True
>>> 4 in liste
False
>>> 3 not in liste
False
>>> 4 not in liste
True
```

La variation avec `not` permet, *a contrario*, de vérifier qu'un élement n'est pas dans une liste.

11.3 Copie de listes

Il est très important de savoir que l'affectation d'une liste (à partir d'une liste préexistante) crée en réalité une **référence** et non une **copie** :

```
>>> x = [1,2,3]
>>> y = x
>>> y
[1, 2, 3]
>>> x[1] = -15
>>> y
[1, -15, 3]
```

Vous voyez que la modification de `x` modifie `y` aussi ! Pour comprendre ce qu'il se passe nous allons de nouveau utiliser le site pythontutor sur cet exemple :

Techniquement, Python utilise des pointeurs (comme dans le langage C) vers les mêmes objets. *Pythontutor* nous l'illustre en mettant des flèches, et on voit bien que `x` et `y` pointent vers la même liste. Donc si on modifie `x`, `y` est modifiée pareillement. Rappelez-vous de ceci dans vos futurs programmes car cela pourrait avoir des effets désastreux !

Pour éviter le problème, il va falloir créer une copie explicite de la liste initiale. Regardez cet exemple :

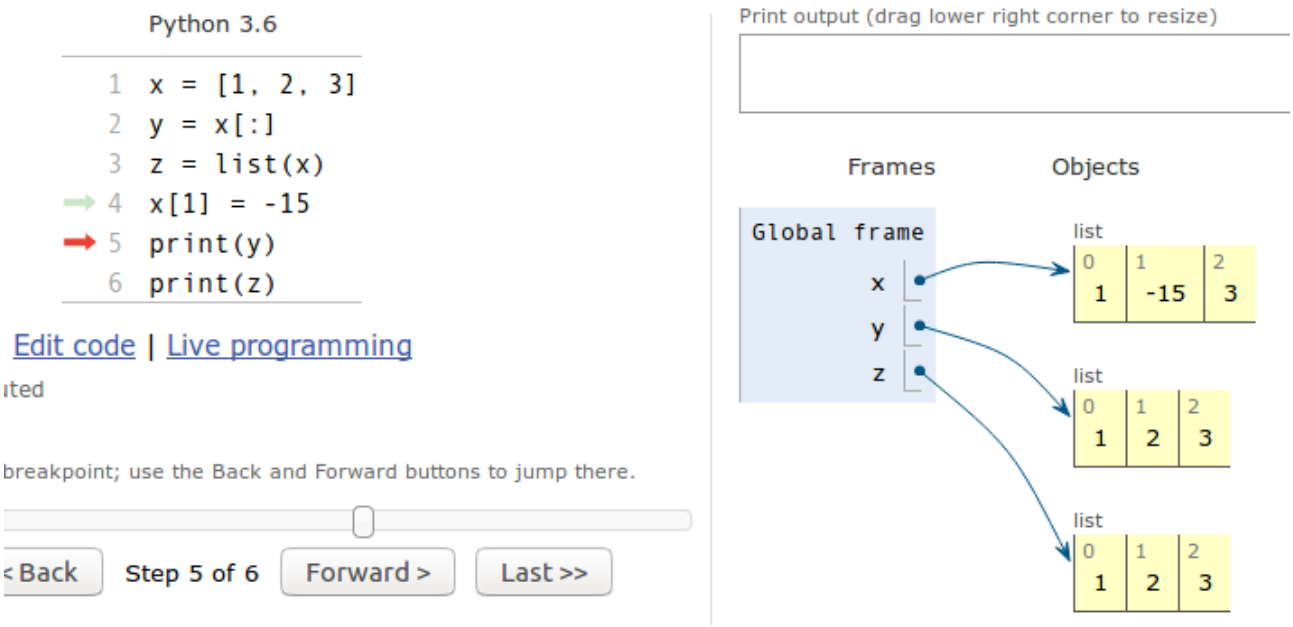
```
>>> x = [1,2,3]
>>> y = x[:]
>>> x[1] = -15
>>> y
[1, 2, 3]
```

Dans l'exemple précédent, `x[:]` a créé une copie *à la volée* de la liste `x`. Vous pouvez utiliser aussi la fonction `list()` qui renvoie explicitement une liste :

```
>>> x = [1,2,3]
```

```
>>> y = list(x)
>>> x[1] = -15
>>> y
[1, 2, 3]
```

Si on regarde à nouveau dans *pythontutor*, on voit bien que l'utilisation des tranches `[:]` ou de la fonction `list()` créee des copies explicites, c'est à dire que chaque "flèche" pointe vers des listes différentes, indépendantes les unes des autres.



Attention, les deux techniques précédentes ne fonctionnent que pour les listes à une dimension, autrement dit les listes qui ne contienent pas elles-mêmes d'autres listes.

```
>>> x = [[1,2],[3,4]]
>>> x
[[1, 2], [3, 4]]
>>> y = x[:]
>>> y[1][1] = 55
>>> y
[[1, 2], [3, 55]]
>>> x
[[1, 2], [3, 55]]
>>> y = list(x)
>>> y[1][1] = 77
>>> y
[[1, 2], [3, 77]]
>>> x
[[1, 2], [3, 77]]
```

La méthode de copie qui **fonctionne à tous les coups** consiste à appeler la fonction `deepcopy()` du module `copy`.

```
>>> import copy
>>> x = [[1,2],[3,4]]
>>> x
[[1, 2], [3, 4]]
>>> y = copy.deepcopy(x)
>>> y[1][1] = 99
```

```
>>> y
[[1, 2], [3, 99]]
>>> x
[[1, 2], [3, 4]]
```

11.4 Exercices

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

11.4.1 Tri de liste

Soit la liste de nombres `[8, 3, 12.5, 45, 25.5, 52, 1]`. Triez les nombres de cette liste par ordre croissant, sans utiliser la fonction `sort()` (les fonctions/méthodes `min()`, `append()` et `remove()` vous seront utiles).

11.4.2 Séquence nucléique aléatoire

Générez aléatoirement une séquence nucléique de 20 bases en utilisant une liste et la méthode `append()` (chaque base est équiprobable). Votre programme contiendra une fonction `gen_seq_alea()` à laquelle on passe un entier positif `n` représentant le nombre de nucléotides. La fonction renverra la séquence aléatoire sous forme de liste.

11.4.3 Séquence nucléique complémentaire

Reprenez l'exercice 9.5.5. Transformez la séquence nucléique `TCTGTTAACCATCCACTTCG` en sa séquence complémentaire inverse en utilisant toutes un maximum de méthodes associées aux listes. N'oubliez pas que la séquence complémentaire doit être inversée, pensez aux méthodes des listes !

11.4.4 Doublons

Soit la liste de nombres `[5, 1, 1, 2, 5, 6, 3, 4, 4, 4, 2]`. Enlevez les doublons de cette liste, triez-là et affichez-là.

11.4.5 Séquence nucléique aléatoire 2

Générez aléatoirement une séquence nucléique de 50 bases contenant 10 % de A, 50 % de G, 30 % de T et 10 % de C. Votre programme contiendra une fonction `gen_seq_alea2()` à qui on passe 4 *floats* représentant les pourcentages de chaque nucléotide. La fonction renverra la séquence sous forme de liste.

11.4.6 Triangle de Pascal (Exercice +++)

Voici le début du triangle de Pascal :

```
1
11
121
1331
```

```
14641
...
```

Comprenez comment une ligne est construite à partir de la précente. A partir de l'ordre 1 (ligne 2, 11), générez l'ordre suivant (121). Vous pouvez utiliser une liste préalablement générée avec range(). Généralisez à l'aide d'une boucle. Ecrivez dans un fichier pascal.out les lignes du triangle de Pascal de l'ordre 1 jusqu'à l'ordre 10.

Chapitre 12

Plus sur les fonctions

Avant d'attaquer ce chapitre, nous vous conseillons de relire le chapitre 9 et de bien en assimiler toutes les notions (et aussi en faire les exercices). Nous avons vu dans ce chapitre 9 le concept puissant et incontournable que représentent les **fonctions**. Nous vous avons également introduit la notion de variables **locales** et **globales**. Dans ce chapitre, nous allons aller un peu plus loin sur la visibilité de ces variables dans et hors des fonction, et aussi voir ce qu'il se passe lorsque ces variables sont des listes. Attention, la plupart des lignes de code ci-dessous sont données à titre d'exemple pour bien comprendre ce qu'il se passe, mais nombre d'entre elles sont des aberrations de programmation. Ainsi nous ferons un récapitulatif des bonnes pratiques à la fin du chapitre. Enfin, nous vous conseillons de tester tous les exemples ci-dessous avec l'excellent site pythontutor afin de suivre l'état des variables lors de l'exécution des exemples.

12.1 Appel d'une fonction dans une fonction

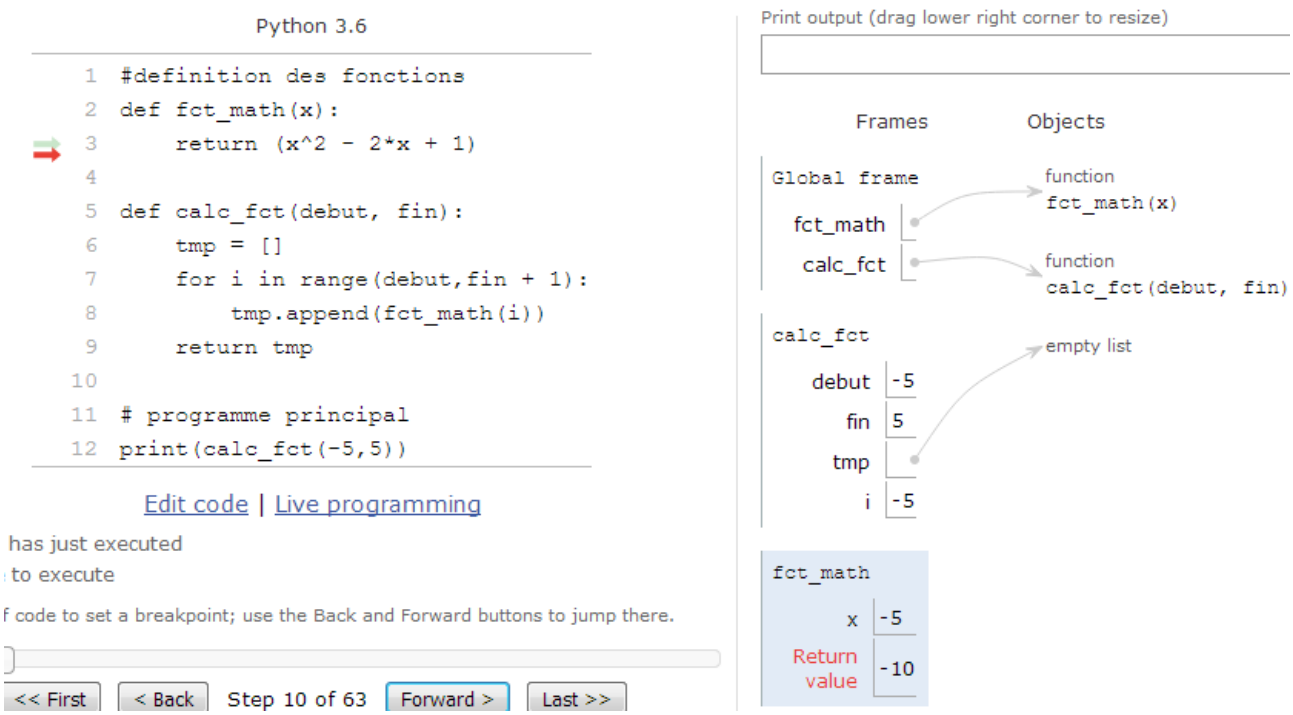
Dans le chapitre 9 nous avons vu des fonctions qui étaient appelées depuis le programme principal. Il est en fait possible d'appeler une fonction depuis une autre fonction (et plus généralement on peut appeler une fonction de n'importe où si tant est qu'elle est visible par Python, *i.e.* chargée dans la mémoire). Regardez cet exemple :

```
#definition des fonctions
def fct_math(x):
    return (x**2 - 2*x + 1)

def calc_fct(debut, fin):
    tmp = []
    for i in range(debut,fin + 1):
        tmp.append(fct_math(i))
    return tmp

# programme principal
print(calc_fct(-5,5))
```

Dans cet exemple, nous appelons depuis le programme principal la fonction `calc_fct()`, puis à l'intérieur de celle-ci nous appelons l'autre fonction `fct_math()`. Regardons ce que *pythontutor* nous montre lorsque la fonction `fct_math()` est en exécution la première fois :



Nous voyons que l'espace mémoire alloué à `fct_math()` est bleuté, nous indiquant qu'elle est en cours d'exécution. La fonction appelante `calc_fct()` est toujours là (sur un fond blanc) car son exécution n'est pas terminée. Les variables *locales* d'une fonction ne seront détruites que lorsque l'exécution de celle-ci sera terminée. Dans notre exemple, les variables *locales* de `calc_fct()` ne seront détruites que lorsque la boucle sera terminée et que la liste `tmp` sera retournée au programme principal. En passant, la fonction `calc_fct()` appelle la fonction `fct_math()` à chaque itération de la boucle.

Ainsi le programmeur est libre de faire tous les appels qu'il souhaite. Une fonction peut appeler une autre fonction, cette dernière peut appeler une autre fonction et ainsi de suite (et autant de fois qu'on le veut). Une fonction peut même s'appeler elle-même, cela s'appelle une fonction **récursive** (nous en dirons quelques mots dans le chapitre 19). Attention toutefois à retrouver vos petits si vous vous perdez dans les appels successifs !

12.2 Portée des variables

Il est très important lorsque l'on manipule des fonctions de connaître la portée des variables. On a vu que les variables créées au sein d'une fonction ne sont pas visibles à l'extérieur de celle-ci car elles étaient **locales** à la fonction. Observez le code suivant :

```
>>> def mafonction():
...     x = 2
...     print('x vaut {} dans la fonction'.format(x))
...
>>> mafonction()
x vaut 2 dans la fonction
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Lorsque Python exécute le code de la fonction, il connait le contenu de la variable `x`. Par contre, de

retour dans le module principal (dans notre cas, il s'agit de l'interpréteur Python), il ne la connaît plus d'où le message d'erreur.

De même, une variable passée en argument est considérée comme **locale** lorsqu'on arrive dans la fonction :

```
>>> def mafonction(x):
...     print('x vaut {} dans la fonction'.format(x))
...
>>> mafonction(2)
x vaut 2 dans la fonction
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'x' is not defined
```

Deuxièmement, lorsqu'une variable déclarée à la racine du module (c'est comme cela que l'on appelle un programme Python), elle est visible dans tout le module. On a vu qu'on parlait de variable **globale**

```
>>> def mafonction():
...     print(x)
...
>>> x = 3
>>> mafonction()
3
>>> print(x)
3
```

Dans ce cas, la variable `x` est visible dans le module principal et dans toutes les fonctions du module. Toutefois, Python ne permet pas la modification d'une variable globale dans une fonction :

```
>>> def mafonction():
...     x = x + 1
...
>>> x=1
>>> mafonction()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fct
UnboundLocalError: local variable 'x' referenced before assignment
```

L'erreur renvoyée montre que Python pense que `x` est une variable locale qui n'a pas été encore assignée. Si on veut vraiment modifier une variable globale dans une fonction, il faut utiliser le mot-clé `global` :

```
>>> def mafonction():
...     global x
...     x = x + 1
...
>>> x=1
>>> mafonction()
>>> x
2
```

Dans ce dernier cas, le mot-clé `global` a forcé la variable `x` a être globale plutôt que locale au sein de la fonction.

12.3 Portée des listes

Préambule : Attention, les exemples de cette section sont donnés à titre indicatif, et il ne faut pas s'en inspirer !

Soyez extrêmement attentifs avec les types modifiables (tels que les listes) car vous pouvez les changer au sein d'une fonction :

```
>>> def mafonction():
...     liste[1] = -127
...
>>> liste = [1,2,3]
>>> mafonction()
>>> liste
[1, -127, 3]
```

De même que si vous passez une liste en argument, elle est tout autant modifiable au sein de la fonction :

```
>>> def mafonction(x):
...     x[1] = -15
...
>>> y = [1,2,3]
>>> mafonction(y)
>>> y
[1, -15, 3]
```

Si vous voulez éviter ce problème, utilisez des tuples, Python renverra une erreur puisque ces derniers sont non modifiables ! Une autre solution pour éviter la modification d'une liste lorsqu'elle est passée en tant qu'argument, est de la passer explicitement (comme nous l'avons fait pour l'affectation) afin qu'elle reste intacte dans le programme principal.

```
>>> def mafonction(x):
...     x[1] = -15
...
>>> y = [1,2,3]
>>> mafonction(y[:])
>>> y
[1, 2, 3]
>>> mafonction(list(y))
>>> y
[1, 2, 3]
```

Dans ces deux derniers exemples, une copie de `y` est créée à la volée lorsqu'on appelle la fonction, ainsi la liste `y` du module principal reste intacte.

12.4 Règle LGI

Lorsque Python rencontre une variable, il va traiter la résolution de son nom avec des priorités particulières : d'abord il va regarder si la variable est **locale**, puis si elle n'existe pas localement, il vérifiera si elle est **globale** et enfin si elle n'est pas globale, il testera si elle est **interne** (par exemple la fonction `len()` est considérée comme une fonction interne à Python, *i.e.* elle existe à chaque fois que vous lancez Python). On appelle cette règle la règle **LGI** pour locale, globale, interne. En voici un exemple :

```
>>> def mafonction():
...     x = 4
...     print('Dans la fonction x vaut {}'.format(x))
...
>>> x = -15
>>> mafonction()
Dans la fonction x vaut  4
>>> print('Dans le module principal x vaut {}'.format(x))
Dans le module principal x vaut -15
```

Vous voyez que dans la fonction, `x` a pris la valeur qui lui était définie localement en priorité sur sa valeur définie dans le module principal.

Conseil : même si Python peut reconnaître une variable ayant le même nom que ses fonctions ou variables internes, évitez de les utiliser car ceci rendra votre code confus !

12.5 Recommandations

Dans ce chapitre nous avons *joué* avec les fonctions (et les listes) afin de vous montrer comment Python réagit. Toutefois, notez bien que **l'utilisation de variables globales est à banir définitivement de votre pratique de la programmation**. Parfois on se dit, je n'ai pas le temps, je préfère créer une variable globale visible partout dans le programme (donc dans toutes les fonctions), car *"ça va plus vite, c'est plus simple"*. C'est un très mauvais calcul, ne serait ce que parce-que vos fonctions ne seront pas réutilisables dans un autre contexte si elles utilisent des variables globales ! Ensuite, arriverez-vous à vous relire dans 6 mois ? Et si vous donnez votre code à lire à quelqu'un ? Il existe de nombreuses autres raisons que nous ne développerons pas ici, mais libre à vous de consulter de la documentation externe.

Heureusement, Python est orienté objet et cela permet *"d'encapsuler"* des variables dans des objets et de s'affranchir définitivement des variables globales (nous verrons cela dans le chapitre 18). En attendant, et si vous souhaitez ne pas aller plus loin sur les notions d'objet (on peut tout à fait *pythonner* sans cela), retenez la chose suivante avec les fonctions et les variables globales :

Plutôt que d'utiliser des variables globales, passez vos variables explicitement aux fonctions en tant qu'argument(s).

Nous espérons que vous maitrisez maintenant les fonctions sous tous leurs angles. Comme dit en introduction du chapitre 9, elles sont incontournables et tout programmeur se doit de les maitriser. Voici les derniers conseils que nous pouvons vous donnez :

— Lorsque vous attaquez un nouveau projet de programmation complexe, posez-vous la question : *"Comment pourriez-vous décomposer en blocs chaque tache à effectuer, chaque bloc pouvant être une fonction"*. Si une fonction s'avère trop complexe, vous pouvez également la décomposer en d'autres fonctions.
— Au risque de nous répéter, forcez vous à utiliser les fonctions en permanence, pratiquez, pratiquez... et pratiquez encore !

12.6 Exercices

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

12.6.1 Prédire la sortie

— Prédisez le comportement de ce code sans le recopier dans un script ni dans l'interpréteur Python :

```
def hello(prenom):
  print("Bonjour {}".format(prenom))

hello("Patrick")
print(x)
```

— Prédisez le comportement de ce code sans le recopier dans un script ni dans l'interpréteur Python :

```
x = 10

def hello(prenom):
  print("Bonjour {}".format(prenom))

hello("Patrick")
print(x)
```

— Prédisez le comportement de ce code sans le recopier dans un script ni dans l'interpréteur Python :

```
x = 10

def hello(prenom):
  print("Bonjour {}".format(prenom))
  print(x)

hello("Patrick")
print(x)
```

— Prédisez le comportement de ce code sans le recopier dans un script ni dans l'interpréteur Python :

```
x = 10

def hello(prenom):
  x = 42
  print("Bonjour {}".format(prenom))
  print(x)

hello("Patrick")
print(x)
```

Chapitre 13

Dictionnaires et tuples

13.1 Dictionnaires

Les **dictionnaires** se révèlent très pratiques lorsque vous devez manipuler des structures complexes à décrire et que les listes présentent leurs limites. Les dictionnaires sont des collections non ordonnées d'objets, c'est à dire qu'il n'y a pas de notion d'ordre (*i.e.* pas d'indice). On accède aux **valeurs** d'un dictionnaire par des **clés**. Ceci semble un peu confus ? Regardez l'exemple suivant :

```
>>> ani1 = {}
>>> ani1['nom'] = 'girafe'
>>> ani1['taille'] = 5.0
>>> ani1['poids'] = 1100
>>> ani1
{'nom': 'girafe', 'poids': 1100, 'taille': 5.0}
>>> ani1['taille']
5.0
```

En premier, on définit un dictionnaire vide avec les symboles `{}` (tout comme on peut le faire pour les listes avec `[]`). Ensuite, on remplit le dictionnaire avec différentes clés auxquelles on affecte des valeurs (une par clé). Vous pouvez mettre autant de clés que vous voulez dans un dictionnaire (tout comme vous pouvez ajouter autant d'éléments que vous voulez dans une liste). Pour récupérer la valeur d'une clé donnée, il suffit d'utiliser une syntaxe du style `dictionnaire['cle']`.

13.1.1 Méthodes keys() et values()

Les méthodes `keys()` et `values()` renvoient, comme vous vous en doutez, les clés et les valeurs d'un dictionnaire (sous forme de liste) :

```
>>> ani1.keys()
['nom', 'poids', 'taille']
>>> ani1.values()
['girafe', 1100, 5.0]
```

On peut aussi initialiser toutes les clés d'un dictionnaire en une seule opération :

```
>>> ani2 = {'nom':'singe', 'poids':70, 'taille':1.75}
```

13.1.2 Liste de dictionnaires

En créant une liste de dictionnaires possédant les mêmes clés, on obtient une structure qui ressemble à une base de données :

```
>>> animaux = [ani1, ani2]
>>> animaux
[{'nom': 'girafe', 'poids': 1100, 'taille': 5.0}, {'nom': 'singe', 'poids': 70, 'taille': 1.75
>>>
>>> for ani in animaux:
...     print(ani['nom'])
...
girafe
singe
```

13.1.3 Existence d'une clef

Enfin, pour vérifier si une clé existe, vous pouvez utiliser la propriété `has_key()` :

```
>>> if ani2.has_key('poids'):
...     print("La clef 'poids' existe pour ani2")
...
La clef 'poids' existe pour ani2
```

Python permet même de simplifier encore les choses :

```
>>> if "poids" in ani2:
...     print("La clef 'poids' existe pour ani2")
...
La clef 'poids' existe pour ani2
```

Vous voyez que les dictionnaires permettent de gérer des structures complexes de manière plus explicite que les listes.

13.2 Tuples

Les **tuples** correspondent aux listes à la différence qu'ils sont **non modifiables**. On a vu à la section précédente que les listes pouvaient être modifiées par des références ; les tuples vous permettent de vous affranchir de ce problème puisqu'ils sont non modifiables. Pratiquement, ils utilisent les parenthèses au lieu des crochets :

```
>>> x = (1,2,3)
>>> x
(1, 2, 3)
>>> x[2]
3
>>> x[0:2]
(1, 2)
>>> x[2] = 15
Traceback (innermost last):
File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

L'affectation et l'indiçage fonctionne comme avec les listes, mais si l'on essaie de modifier un des éléments du tuple, Python renvoie un message d'erreur. Si vous voulez ajouter un élément (ou le modifier), vous devez créer un autre tuple :

```
>>> x = (1,2,3)
>>> x + (2,)
(1, 2, 3, 2)
```

Remarque : pour utiliser un tuple d'un seul élément, vous devez utiliser une syntaxe avec une virgule `(element,)`, ceci pour éviter une ambiguïté avec une simple expression.

Autre particularité des tuples, il est possible d'en créer de nouveaux sans les parenthèses, dès lors que ceci ne pose pas d'ambiguïté avec une autre expression :

```
>>> x = (1,2,3)
>>> x
(1, 2, 3)
>>> x = 1,2,3
>>> x
(1, 2, 3)
```

Toutefois, nous vous conseillons d'utiliser systématiquement les parenthèses afin d'éviter les confusions.

Enfin, on peut utiliser la fonction `tuple(sequence)` qui fonctionne exactement comme la fonction `list()`, c'est à dire qu'elle prend en argument un objet séquentiel et renvoie le tuple correspondant (opération de *casting*) :

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> tuple(range(10))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> tuple("ATGCCGCGAT")
('A', 'T', 'G', 'C', 'C', 'G', 'C', 'G', 'A', 'T')
```

Remarque : les listes, dictionnaires, tuples sont des objets qui peuvent contenir des collections d'autres objets. On peut donc construire des listes qui contiennent des dictionnaires, des tuples ou d'autres listes, mais aussi des dictionnaires contenant des tuples, des listes, etc.

Pratiquement, nous avons déjà croisé les *tuples* avec la fonction `enumerate()` (cf. chapitre 5) et aussi avec les fonctions lorsqu'on voulait renvoyer plusieurs valeurs (cf. chapitre 9, par exemple dans `return x,y`, `x,y` est un tuple).

13.3 Exercices

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

13.3.1 Composition en acides aminés

En utilisant un dictionnaire, déterminez le nombre d'occurences de chaque acide aminé dans la séquence `AGWPSGGASAGLAILWGASAIMPGALW`. Le dictionnaire ne doit contenir que les acides aminés présents dans la séquence.

13.3.2 Mots de 2 lettres

Soit la séquence nucléotidique suivante :

```
ACCTAGCCATGTAGAATCGCCTAGGCTTTAGCTAGCTCTAGCTAGCTG
```

En utilisant un dictionnaire, faites un programme qui répertorie tous les mots de 2 lettres qui existent dans la sequence (`AA`, `AC`, `AG`, `AT`, etc.) ainsi que leur nombre d'occurences puis qui les affiche à l'écran.

13.3.3 Mots de 3 et 4 lettres

Faites de même avec des mots de 3 et 4 lettres.

13.3.4 Mots de 2 lettres de *Saccharomyces cerevisiae*

En vous basant sur les scripts précédents, extrayez les mots de 2 lettres et leur occurence sur le génome du chromosome I de la levure du boulanger *Saccharomyces cerevisiae* (fichier `NC_001133.fna` : lien GenBank, lien local). Attention, le génome complet est fourni au format fasta.

13.3.5 Mots de *n* lettres et fichiers genbank

Créez un script `extract-words.py` qui prend en arguments un fichier genbank suivi d'un entier compris entre 1 et 4. Ce script doit extraire du fichier genbank tous les mots (ainsi que leur nombre d'occurences) du nombre de lettres passées en option.

13.3.6 Mots de *n* lettres du génome d'*E. Coli*

Appliquez ce script sur le génome d'*Escherichia coli* : (fichier `NC_000913.fna` : lien GenBank, lien local). Attention, le génome complet est fourni au format fasta. Cette méthode vous paraît-elle efficace sur un génome assez gros comme celui d'*E. Coli* ? Comment pourrait-on en améliorer la rapidité ?

13.3.7 Dictionnaire et carbone alpha

À partir du fichier PDB 1BTA, construisez un dictionnaire qui contient 4 clés se référant au premier carbone alpha : le numéro du résidu, puis les coordonnées atomiques x, y et z.

13.3.8 Dictionnaire et PDB

Sur le même modèle que ci-dessus, créez une liste de dictionnaires pour chacun des carbones alpha de la protéine.

13.3.9 Barycentre d'une protéine

À l'aide de cette liste, calculez les coordonnées x, y et z du barycentre de ces carbones alpha.

Chapitre 14

Création de modules

14.1 Création

Vous pouvez créez votres propres modules très simplement en Python. Il vous suffit d'écrire un ensemble de fonctions (et/ou de variables) dans un fichier, puis d'enregistrer celui-ci avec une extension `.py` (comme n'importe quel script Python). A titre d'exemple, voici le contenu du module `message.py`.

```
"""Module inutile qui affiche des messages :-)
"""

def Bonjour(nom):
    """Affiche bonjour !
    """
    return "bonjour " + nom

def Ciao(nom):
    """Affiche ciao !
    """
    return "ciao " + nom

def Hello(nom):
    """Affiche hello !
    """
    return "hello " + nom + " !"

date=16092008
```

14.2 Utilisation

Pour appeler une fonction ou une variable de ce module, il faut que le fichier `message.py` soit dans le répertoire courant (dans lequel on travaille) ou bien dans un répertoire indiqué par la variable d'environnement Unix `PYTHONPATH` (on rappelle en Unix il faut taper la commande suivante : `export PYTHONPATH=/chemin/vers/mes/super/modules/python`). Ensuite, il suffit d'importer le module et toutes ses fonctions (et variables) vous sont alors accessibles.

La première fois qu'un module est importé, Python crée un fichier avec une extension `·pyc` (ici `message.pyc`) qui contient le [bytecode]{https ://docs.python.org/3.6/glossary.html)(code précompilé) du module.

L'appel du module se fait avec la commande `import message`. Notez que le fichier est bien enregistré avec une extension `verb=.py` et pourtant on ne la précise pas lorsqu'on importe le module. Ensuite on peut utiliser les fonctions comme avec un module classique.

```
>>> import message
>>> message.Hello("Joe")
'hello Joe !'
>>> message.Ciao("Bill")
'ciao Bill'
>>> message.Bonjour("Monsieur")
'bonjour Monsieur'
>>>  message.date
16092008
```

Les commentaires (entre triple guillemets) situés en début de module et sous chaque fonction permettent de fournir de l'aide invoquée ensuite par la commande `help()` :

```
>>> help(message)
NAME
    message - Module inutile qui affiche des messages :-)

FILE
    /home/cumin/poulain/message.py

FUNCTIONS
    Hello(nom)
        Affiche hello !

    Bonjour(nom)
        Affiche bonjour !

    Ciao(nom)
        Affiche ciao !

DATA
    date = 16092008
```

Remarques :

- Les fonctions dans un module peuvent s'appeler les unes les autres.
- Les fonctions dans un module peuvent appeler des fonctions situées dans un autre module s'il a été préalablement importé avec la commande `import`.

Vous voyez que les modules sont d'une simplicité enfantine à créer. Si vous avez des fonctions que vous serez amenés à utiliser souvent, n'hésitez plus !

14.3 Exercices

Conseil : pour cet exercice, écrivez un script dans un fichier, puis exécutez-le dans un *shell*.

14.3.1 Module ADN

Reprenez l'ensemble des fonctions qui gèrent le traitement de séquences nucléiques et incluez-les dans un module *adn.py*. Testez-les au fur et à mesure.

Chapitre 15

Expressions régulières et parsing

Le module re vous permet d'utiliser des expressions régulières au sein de Python. Les expressions régulières sont aussi appelées en anglais *regular expressions* ou en plus court *regex*. Elles sont incontournables en bioinformatique lorsque vous voulez récupérer des informations dans un fichier.

Cette action de recherche de données dans un fichier est appelée plus généralement *parsing* (qui signifie littéralement "analyse syntaxique"). Le *parsing* fait partie du travail quotidien du bioinformaticien, il est sans arrêt en train de "fouiller" dans des fichiers pour en extraire des informations d'intérêt comme par exemple récupérer les coordonnées 3D des atomes d'une protéines dans un fichier PDB ou alors extraire les gènes d'un fichier genbank.

Dans ce chapitre, nous ne ferons que quelques rappels sur les expressions régulières. Pour un documentation plus complète, référez-vous à la page d'aide des expressions régulières sur le site officiel de Python.

15.1 Définition et syntaxe

Une expression régulière est une suite de caractères qui a pour but de décrire un fragment de texte. Elle est constitué de deux types de caractères :

— Les caractères dits *normaux*.
— Les *métacaractères* ayant une signification particulière, par exemple le caractère `^` signifie début de ligne et non pas le caractère "chapeau" littéral.

Certains programmes Unix comme `egrep`, `sed` ou encore `awk` savent interpréter les expressions régulières. Tous ces programmes fonctionnent généralement selon le schéma suivant :

— Le programme lit un fichier ligne par ligne.
— Pour chaque ligne lue, si l'expression régulière passée en argument est présente alors le programme effectue une action.

Par exemple, pour le programme `egrep` :

```
$ egrep "^DEF" herp_virus.gbk
DEFINITION  Human herpesvirus 2, complete genome.
```

Ici, `egrep` renvoie toutes les lignes du fichier genbank du virus de l'herpès (`herp_virus.gbk`) qui correspondent à l'expression régulière `^DEF` (c'est-à-dire le mot `DEF` en début de ligne).

Avant de voir comment Python gère les expressions régulières, voici quelques éléments de syntaxe des métacaractères :

`^` Début de chaîne de caractères ou de ligne.
Exemple : l'expression `^ATG` correspond à la chaîne de caractères `ATGCGT` mais pas à la chaîne `CCATGTT`.

`$` Fin de chaîne de caractères ou de ligne.
Exemple : l'expression `ATG$` correspond à la chaîne de caractères `TGCATG` mais pas avec la chaîne `CCATGTT`.

`.` N'importe quel caractère (mais un caractère quand même).
Exemple : l'expression `A.G` correspond à `ATG`, `AtG`, `A4G`, mais aussi à `A-G` ou à `A G`.

`[ABC]` Le caractère A ou B ou C (un seul caractère).
Exemple : l'expression `T[ABC]G` correspond à `TAG`, `TBG` ou `TCG`, mais pas à `TG`.

`[A-Z]` N'importe quelle lettre majuscule.
Exemple : l'expression `C[A-Z]T` correspond à `CAT`, `CBT`, `CCT`...

`[a-z]` N'importe quelle lettre minuscule.

`[0-9]` N'importe quel chiffre.

`[A-Za-z0-9]` N'importe quel caractère alphanumérique.

`[^AB]` N'importe quel caractère sauf A et B.
Exemple : l'expression `CG[^AB]T` correspond à `CG9T`, `CGCT`... mais pas à `CGAT` ni à `CGBT`.

`\` Caractère d'échappement (pour protéger certains caractères).
Exemple : l'expression `\+` désigne le caractère `+` sans autre signification particulière. L'expression `A\.G` correspond à `A.G` et non pas à `A` suivi de n'importe quel caractère, suivi de `G`.

`*` 0 à n fois le caractère précédent ou l'expression entre parenthèses précédente.
Exemple : l'expression `A(CG)*T` correspond à `AT`, `ACGT`, `ACGCGT`...

`+` 1 à n fois le caractère précédent ou l'expression entre parenthèses précédente.
Exemple : l'expression `A(CG)+T` correspond à `ACGT`, `ACGCGT`... mais pas à `AT`.

`?` 0 à 1 fois le caractère précédent ou l'expression entre parenthèses précédente.
Exemple : l'expression `A(CG)?T` correspond à `AT` ou `ACGT`.

`{n}` n fois le caractère précédent ou l'expression entre parenthèses précédente.
Exemple : l'expression `A(CG){2}T` correspond à `ACGCGT` mais pas à `ACGT`, `ACGCGCGT` ou `ACGCG`.

`{n,m}` n à m fois le caractère précédent ou l'expression entre parenthèses précédente.
Exemple : l'expression `A(C){2,4}T` correspond à `ACCT`, `ACCCT` et `ACCCCT` mais pas à `ACT`, `ACCCCCT` ou `ACCC`.

`{n,}` Au moins n fois le caractère précédent ou l'expression entre parenthèses précédente.
Exemple : l'expression `A(C){2,}T` correspond à `ACCT`, `ACCCT` et `ACCCCT` mais pas à `ACT` ou `ACCC`.

`{,m}` Au plus m fois le caractère précédent ou l'expression entre parenthèses précédente.
Exemple : l'expression `A(C){,2}T` correspond à `AT`, `ACT` et `ACCT` mais pas à `ACCCT` ou `ACC`.

`(CG|TT)` Les chaînes de caractères `CG` ou `TT`.
Exemple : l'expression `A(CG|TT)C` correspond à `ACGC` ou `ATTC`.

Les métacaractères sont nombreux et leur signification est parfois difficile à maîtriser. N'hésitez pas à tester vos expressions régulières sur des exemples simples.

15.2 Module re et fonction search

Dans le module `re`, la fonction `search()` permet de rechercher un motif (*pattern*) au sein d'une chaîne de caractères avec une syntaxe de la forme `search(motif, chaine)`. Si `motif` existe dans `chaine`,

Python renvoie un objet du type `SRE_Match`. Sans entrer dans les détails propres au langage orienté objet, si on utilise cet objet dans un test, il sera considéré comme vrai. Regardez cet exemple dans lequel on va rechercher le motif `tigre` dans la chaîne de caractères `"girafe tigre singe"` :

```
>>> import re
>>> animaux = "girafe tigre singe"
>>> re.search('tigre', animaux)
<_sre.SRE_Match object at 0x7fefdaefe2a0>
>>> if re.search('tigre', animaux):
...     print "OK"
...
OK
```

15.2.1 Fonction match()

Il existe aussi la fonction `match()` dans le module `re` qui fonctionne sur le modèle de `search()`. La différence est qu'elle renvoie un object du type `SRE_Match` seulement lorsque l'expression régulière correspond (*match*) au début de la chaîne de caractères (à partir du premier caractère).

```
>>> animaux = "girafe tigre singe"
>>> re.search('tigre', animaux)
<_sre.SRE_Match object at 0x7fefdaefe718>
>>> re.match('tigre', animaux)
```

Nous vous recommandons plutôt l'usage de la fonction `search()`. Si vous souhaitez avoir une correspondance avec le début de la chaîne, vous pouvez toujours utiliser l'accroche de début de ligne `^`.

15.2.2 Compilation d'expressions régulières

Lorsqu'on a besoin de tester la même expression régulière sur plusieurs milliers de chaînes de caractères, il est pratique de préalablement compiler l'expression régulière à l'aide de la fonction `compile()` qui renvoie un objet de type `SRE_Pattern` :

```
>>> regex = re.compile("^tigre")
>>> regex
<_sre.SRE_Pattern object at 0x7fefdafd0df0>
```

On peut alors utiliser directement cet objet avec la méthode `search()` :

```
>>> animaux = "girafe tigre singe"
>>> regex.search(animaux)
>>> animaux = "tigre singe"
>>> regex.search(animaux)
<_sre.SRE_Match object at 0x7fefdaefe718>
>>> animaux = "singe tigre"
>>> regex.search(animaux)
```

15.2.3 Groupes

L'intérêt de l'objet de type `SRE_Match` renvoyé par Python lorsqu'une expression régulière trouve une correspondance dans une chaîne de caractères est de pouvoir ensuite récupérer certaines zones précises :

```
>>> regex = re.compile('([0-9]+)\.([0-9]+)')
```

Dans cet exemple, on recherche un nombre :

— qui débute par un ou plusieurs chiffres `[0-9]+`,
— suivi d'un point `\.` (le point a d'habitude une signification de métacaractère, donc il faut l'échapper avec `\` pour qu'il retrouve sa signification de point),
— et qui se termine encore par un ou plusieurs chiffres `[0-9]+`.

Les parenthèses dans l'expression régulière permettent de créer des groupes (`[0-9]+` deux fois) qui seront récupérés ultérieurement par la fonction `group()`.

```
>>> resultat = regex.search("pi vaut 3.14")
>>> resultat.group(0)
'3.14'
>>> resultat.group(1)
'3'
>>> resultat.group(2)
'14'
>>> resultat.start()
8
>>> resultat.end()
12
```

La totalité de la correspondance est donné par `group(0)`, le premier élément entre parenthèse est donné par `group(1)` et le second par `group(2)`.

Les fonctions `start()` et `end()` donnent respectivement la position de début et de fin de la zone qui correspond à l'expression régulière. Notez que la fonction `search()` ne renvoie que la première zone qui correspond à l'expression régulière, même s'il en existe plusieurs :

```
>>> resultat = regex.search("pi vaut 3.14 et e vaut 2.72")
>>> resultat.group(0)
'3.14'
```

15.2.4 Fonction findall()

Pour récupérer chaque zone, vous pouvez utiliser la fonction `findall()` qui renvoie une liste des éléments en correspondance.

```
>>> regex = re.compile('[0-9]+\.[0-9]+')
>>> resultat = regex.findall("pi vaut 3.14 et e vaut 2.72")
>>> resultat
['3.14', '2.72']
```

L'utilisation des groupes entre parenthèse est également possible et ceux-ci sont automatiquement renvoyés sous la forme de tuples.

```
>>> regex = re.compile('([0-9]+)\.([0-9]+)')
>>> resultat = regex.findall("pi vaut 3.14 et e vaut 2.72")
>>> resultat
[('3', '14'), ('2', '72')]
```

15.2.5 Fonction sub()

Enfin, la fonction `sub()` permet d'effectuer des remplacements assez puissants. Par défaut la fonction `sub(chaine1,chaine2)` remplace toutes les occurrences trouvées par l'expression régulière dans

`chaine2` par `chaine1`. Si vous souhaitez ne remplacer que les n premières occurrences, utilisez l'argument `count=n` :

```
>>> regex = re.compile('[0-9]+\.[0-9]+')
>>> regex.sub('quelque chose',"pi vaut 3.14 et e vaut 2.72")
'pi vaut quelque chose et e vaut quelque chose'
>>> regex.sub('quelque chose',"pi vaut 3.14 et e vaut 2.72", count=1)
'pi vaut quelque chose et e vaut 2.72'
```

Nous espérons que vous êtes convaincus de la puissance du module `re` et des expressions régulières, alors à vos expressions régulières !

15.3 Exercices : extraction des gènes d'un fichier gbk

Pour les exercices suivants, vous utiliserez le module d'expressions régulières `re` et le fichier genbank du chromosome I de la levure du boulanger *Saccharomyces cerevisiae*. Vous pouvez télécharger ce fichier :

— soit via le lien direct NC_001133.gbk ;
— soit directement sur la page Saccharomyces cerevisiae S288c chromosome I, complete sequence sur le site du NCBI, puis en cliquant sur *Send to*, puis *Complete Record*, puis *Choose Destination : File*, puis *Format : GenBank (full)* et enfin sur le bouton *Create File*.

La documentation du format de fichier GenBank est disponible sur le site du NCBI : GenBank Flat File Format.

15.3.1 Lecture du fichier

Créez un script `genbank2fasta.py` et écrivez la fonction `lit_fichier()` qui prend en argument le nom du fichier et qui renvoie le contenu du fichier sous forme d'une liste de lignes, chaque ligne étant elle-même une chaîne de caractères.

Testez cette fonction avec le fichier GenBank `NC_001133.gbk` et affichez le nombre de lignes lues.

15.3.2 Extraction du nom de l'organisme

Dans le même script, ajoutez la fonction `extrait_organisme()` qui prend en argument le contenu du fichier précédemment obtenu avec la fonction `lit_fichier()` (sous la forme d'une liste de lignes) et qui renvoie le nom de l'organisme. Utilisez de préférence une expression régulière.

Testez cette fonction avec le fichier GenBank `NC_001133.gbk` et affichez le nom de l'organisme.

15.3.3 Recherche des gènes

Dans le fichier GenBank, les gènes sens sont notés de cette manière :

```
     gene            58..272
```

ou

```
    gene            <2480..>2707
```

et les gènes antisens de cette façon :

```
     gene            complement(55979..56935)
```

ou

```
     gene            complement(<13363..>13743)
```

Les valeurs numériques séparées par `..` indiquent la position du gène dans le génome (numéro de la première base, numéro de la dernière base).

Remarque : le symbole `<` indique un gène partiel sur l'extrémité 5', c'est-à-dire que le codon START correspondant est incomplet. Respectivement, le symbole `>` désigne un gène partiel sur l'extrémité 3', c'est-à-dire que le codon STOP correspondant est incomplet. Pour plus de détails, consultez la documentation du NCBI sur les délimitations des gènes.

Repérez ces différens gènes dans le fichier `NC_001133.gbk`. Construisez deux expressions régulières pour extraire du fichier GenBank les gènes sens et les gènes antisens.

Modifiez ces expressions régulières pour que les numéros de la première et de la dernière base puissent être facilement extraits.

Dans le même script `genbank2fasta.py`, ajoutez la fonction `recherche_genes()` qui prend en argument le contenu du fichier (sous la forme d'une liste de lignes) et qui renvoie la liste des gènes.

Chaque gène sera lui-même une liste contenant le numéro de la première base, le numéro de la dernière base et une chaîne de caractère `"sens"` pour un gène sens et `"antisens"` pour un gène antisens.

Testez cette fonction avec le fichier GenBank `NC_001133.gbk` et affichez le nombre de gènes trouvés, ainsi que le nombre de gènes sens et antisens.

15.3.4 Extraction de la séquence nucléique du génome

La taille du génome est indiqué sur la première ligne d'un fichier GenBank. Trouvez la taille du génome stocké dans le fichier `NC_001133.gbk`.

Dans un fichier GenBank, la séquence du génome se trouve entre les lignes

`ORIGIN`

et

`//`

Trouvez dans le fichier `NC_001133.gbk` la première et dernière ligne de la séquence du génome.

Construisez une expression régulière pour extraire du fichier GenBank les lignes correspondantes à la séquence du génome.

Modifiez ces expressions régulières pour que la séquence puisse être facilement extraite.

Toujours dans le même script, ajoutez la fonction `extrait_sequence()` qui prend en argument le contenu du fichier (sous la forme de liste de lignes) et qui renvoie la séquence nucléique du génome (dans une chaine de caractères). La séquence ne devra pas contenir d'espaces.

Testez cette fonction avec le fichier GenBank `NC_001133.gbk` et affichez le nombre de bases de la séquence extraite. Vérifiez que vous n'avez pas fait d'erreur en comparant la taille de la séquence extraite avec celle que vous avez trouvée dans le fichier GenBank.

15.3.5 Construction d'une séquence complémentaire inverse

Toujours dans le même script, ajoutez la fonction `construit_comp_inverse()` qui prend en argument une séquence d'ADN sous forme de chaîne de caractères et qui renvoie la séquence complémentaire inverse (également sous la forme d'une chaîne de caractères).

On rappelle que construire la séquence complémentaire inverse d'une séquence d'ADN consiste à :

— Prendre la séquence complémentaire. C'est-à-dire à remplacer la base `a` par la base `t`, `t` par `a`, `c` par `g` et `g` par `c`.
— Prendre l'inverse. C'est-à-dire à que la première base de la séquence complémentaire devient la dernière base et réciproquement, la dernière base devient la première.

Pour vous faciliter le travail, ne travaillez que sur des séquences en minuscule.

Testez cette fonction avec les séquences `atcg`, `AATTCCGG` et `gattaca`.

15.3.6 Ècriture d'un fichier fasta

Toujours dans le même script, ajoutez la fonction `ecrit_fasta()` qui prend en argument un nom de fichier (sous forme de chaîne de caractères), un commentaire (sous forme de châine de caractères) et une séquence (sous forme de chaîne de caractères) et qui écrit un fichier fasta. La séquence sera écrire sur des lignes ne dépassant pas 80 caractères.

Pour rappel, un fichier fasta suit le format suivant :

```
>commentaire
sequence sur une ligne de 80 caractères maxi
suite de la séquence ......................
suite de la séquence ......................
...
```

Testez cette fonction avec :

— nom de fichier : `test.fasta`
— commentaire : `mon commentaire`
— séquence : `atcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcg`

15.3.7 Extraction des gènes

Toujours dans le même script, ajoutez la fonction `extrait_genes()` qui prend en argument la liste des gènes, la séquence nucléotidique complète (sous forme d'une chaîne de caractères) et le nom de l'organisme (sous forme d'une chaîne de caractères) et qui pour chaque gène :

— extrait la séquence du gène dans la séquence complète ;
— prend la séquence complémentaire inverse (avec la fonction `construit_comp_inverse()` si le gène est antisens ;
— enregistre le gène dans un fichier au format fasta (avec la fonction `ecrit_fasta()`) ;
— affiche à l'écran le numéro du gène et le nom du fichier fasta créé.

La première ligne des fichiers fasta sera de la forme :

`>nom-organisme|numéro-du-gène|début|fin|sens ou antisens`

Le numéro du gène sera un numéro consécutif depuis le premier gène jusqu'au dernier. Il n'y aura pas de différence de numérotation entre les gènes sens et les gènes antisens.

Testez cette fonction avec le fichier GenBank `NC_001133.gbk`.

15.3.8 Assemblage du script final

Pour terminer, modifiez le script `genbank2fasta.py` de façon à ce que le fichier GenBank à analyser (dans cet exemple `NC_001133.gbk`), soit entré comme argument du script.

Vous afficherez un message d'erreur si :

— le script `genbank2fasta.py` est utilisé sans argument,
— le fichier fourni en argument n'existe pas.

Pour vous aider, n'hésitez pas à jeter un oeil aux descrptions des modules `sys` et `os` dans le chapitre 8 sur les modules.

Testez votre script ainsi finalisé.

Bravo, si vous êtes arrivés jusqu'à cette étape. Vous avez maintenant une bonne idée de comment écrire un script Python.

Chapitre 16

Autres modules d'intérêt

en construction

Chapitre 17

Modules d'intérêt en bioinformatique

Nous allons découvrir dans cette section quelques modules très importants en bioinformatique. Le premier `numpy` permet notamment de manipuler des vecteurs et des matrices en Python. Le module `biopython` permet de travailler sur des données biologiques, comme des séquences (nucléiques et protéiques) ou des structures (fichiers PDB). Le module `matplotlib` permet de dessiner des graphiques depuis Python.

Ces modules ne sont pas fournis avec la distribution Python de base (contrairement à tous les autres modules vus précédemment). Nous ne nous étendrons pas sur la manière de les installer. Consultez pour cela la documentation sur les sites internet des modules en question. Sachez cependant que ces modules existent dans la plupart des distributions Linux récentes.

Dans ce chapitre, nous vous montrererons quelques exemples d'utlisation de ces modules pour vous convaincre de leur pertinence.

17.1 Module numpy

Le module numpy est incontournable en bioinformatique. Il permet d'effectuer des calculs sur des vecteurs ou des matrices, élément par élément, via un nouveau type d'objet appelé *array*. Ce module contient des fonctions de base pour faire de l'algèbre linéaire, des transformées de Fourier ou encore des tirages de nombre aléatoire plus sophistiqués qu'avec le module `random`. Vous pouvez télécharger `numpy` à cette adresse. Notez qu'il existe un autre module scipy que nous n'aborderons pas dans ce cours. `scipy` est lui même basé sur `numpy`, mais il en étend considérablement les possibilités de ce dernier (par exemple : statistiques, optimisation, intégration numérique, traitement du signal, traitement d'image, algorithmes génétiques, etc).

On charge le module `numpy` avec la commande

```
>>> import numpy
```

On peut également définir un raccourci pour `numpy` :

```
>>> import numpy as np
```

17.1.1 Objets de type array

Les objets de type *array* correspondent à des tableaux à une ou plusieurs dimensions et permettent d'effectuer du calcul vectoriel. La fonction `array()` permet la conversion d'un objet séquentiel (type liste ou tuple) en un objet de type *array*. Voici un exemple simple de conversion d'une liste à une dimension en objet *array* :

```
>>> import numpy as np
>>> a = [1,2,3]
>>> np.array(a)
array([1, 2, 3])
>>> b = np.array(a)
>>> type(b)
<type 'numpy.ndarray'>
>>> b
array([1, 2, 3])
```

Nous avons converti la liste `a` en *array*, mais cela aurait donné le même résultat si on avait converti le tuple `(1,2,3)`. Par ailleurs, vous voyez que lorsqu'on demande à Python le contenu d'un objet *array*, les symboles `([` et `])` sont utilisés pour le distinguer d'une liste (délimitée par les caractères `[` et `]`) ou d'un tuple (délimité par les caractères `(` et `)`).

Notez qu'un objet *array* ne peut contenir que des valeurs numériques. Vous ne pouvez pas, par exemple, convertir une liste contenant des chaînes de caractères en objet de type *array*.

Contrairement à la fonction `range()`, la fonction `arange()` permet de construire un *array* à une dimension de manière simple.

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(10.0)
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> np.arange(10,0,-1)
array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])
```

Un des avantages de la fonction `arange()` est qu'elle permet de générer des objets *array* qui contiennent des entiers ou de réels selon l'argument qu'on lui passe.

La différence fondamentale entre un objet *array* à une dimension et une liste (ou un tuple) est que celui-ci est considéré comme un **vecteur**. Par conséquent on peut effectuer des opérations **élément par élément** dessus, ce qui est bien commode lorsqu'on analyse de grandes quantités de données. Regardez ces exemples :

```
>>> v = np.arange(4)
>>> v
array([0, 1, 2, 3])
>>> v + 1
array([1, 2, 3, 4])
>>> v + 0.1
array([ 0.1,  1.1,  2.1,  3.1])
>>> v * 2
array([0, 2, 4, 6])
>>> v * v
array([0, 1, 4, 9])
```

Notez bien sur le dernier exemple de multiplication que l'*array* final correspond à la multiplication **élément par élément** des deux *array* initiaux. Avec les listes, ces opérations n'auraient été possibles qu'en utilisant des boucles ! Nous vous encourageons donc à utiliser dorénavant les objets *array* lorsque vous aurez besoin de faire des opérations élément par élément.

17.1.2 array et dimensions

Il est aussi possible de construire des objets *array* à deux dimensions, il suffit de passer en argument une liste de listes à la fonction `array()` :

```
>>> w = np.array([[1,2],[3,4],[5,6]])
>>> w
array([[1, 2],
     [3, 4],
     [5, 6]])
```

On peut aussi créer des tableaux à trois dimensions en passant à la fonction `array()` une liste de listes de listes :

```
>>> x = np.array([[[1,2],[2,3]],[[4,5],[5,6]]])
>>> x
array([[[1, 2],
      [2, 3]],

     [[4, 5],
      [5, 6]]])
```

La fonction `array()` peut créer des tableaux à n'importe quel nombre de dimensions. Toutefois ça devient vite compliqué lorsqu'on dépasse trois dimensions. Retenez qu'un objet *array* à une dimension peut être considéré comme un **vecteur** et un *array* à deux dimensions comme une **matrice**.

La méthode `.shape` renvoie les dimensions d'un *array* alors que `.size` renvoie le nombre d'éléments contenus dans l'`array`.

```
>>> v.shape
(4,)
>>> v.size
4
>>> w.shape
(3, 2)
>>> w.size
6
>>> x.shape
(2, 2, 2)
>>> x.size
8
```

Et la fonction `reshape()` permet de modifier les dimensions d'un *array* :

```
>>> a = np.arange(0, 6)
>>> a
array([0, 1, 2, 3, 4, 5])
>>> a.shape
(6,)
>>> b = t.reshape((2, 3))
>>> b
array([[0, 1, 2],
     [3, 4, 5]])
>>> b.shape
(2, 3)
```

Notez que `a.reshape((2, 3))` n'est pas la même chose que `a.reshape((3, 2))` :

```
>>> c = a.reshape((3, 2))
>>> c
array([[0, 1],
     [2, 3],
     [4, 5]])
>>> c.shape
(3, 2)
```

La fonction `reshape()` attend que les nouvelles dimensions soient `compatibles` avec la dimension initiale de l'objet *array*, c'est-à-dire que le nombre d'éléments contenus dans les différents *array* soit le même. Dans nos exemples précédents, 6 = 2 x 3 = 3 x 3.

Si les nouvelles dimensions ne sont pas compatibles avec les dimensions initiales, `reshape()` génère une erreur.

```
>>> a = np.arange(0, 6)
>>> a.shape
(6,)
>>> d = a.reshape((3, 4))
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: cannot reshape array of size 6 into shape (3,4)
```

La fonction `resize()` par contre ne déclenche pas d'erreur dans une telle situation et ajoute des 0 ou coupe la liste initiale jusqu'à ce que le nouvel *array* soit rempli.

```
>>> a = np.arange(0, 6)
>>> a.shape
(6,)
>>> a.resize((3,3))
>>> a.shape
(3, 3)
>>> a
array([[0, 1, 2],
     [3, 4, 5],
     [0, 0, 0]])


>>> b = np.arange(0, 10)
>>> b.shape
(10,)
>>> b.resize((2,3))
>>> b.shape
(2, 3)
>>> b
array([[0, 1, 2],
     [3, 4, 5]])
```

À noter qu'il existe aussi la fonction `np.resize()` qui dans le cas d'un nouvel *array* plus grand que l'*array* initial, va répéter l'*array* initial :

```
>>> a = np.arange(0, 6)
>>> a.shape
(6,)
>>> c = np.resize(a, (3, 5))
>>> c.shape
```

```
(3, 5)
>>> c
array([[0, 1, 2, 3, 4],
     [5, 0, 1, 2, 3],
     [4, 5, 0, 1, 2]])
```

17.1.3 Indices

Pour récupérer un ou plusieurs élément(s) d'un objet *array*, vous pouvez utiliser l'indiçage ou les tranchage, de la même manière que pour les listes.

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[5:]
array([5, 6, 7, 8, 9])
>>> a[::2]
array([0, 2, 4, 6, 8])
>>> a[1]
1
```

Dans le cas d'un objet *array* à deux dimensions, vous pouvez récupérer une ligne (d'indice `m`), une colonne (d'indice `n`) ou bien un seul élément.

```
>>> a = np.array([[1,2],[3,4]])
>>> a
array([[1, 2],
     [3, 4]])
>>> a[:,0]
array([1, 3])
>>> a[0,:]
array([1, 2])
>>> a[1,1]
4
```

La syntaxe `a[m,:]` récupère la ligne `m-1`, et `a[:,n]` récupère la colonne `n-1`. Les tranches sont évidemment aussi utilisables sur un tableau à deux dimensions.

17.1.4 Construction automatique de matrices

Il peut être parfois pénible de construire une matrice (*array* à deux dimensions) à l'aide d'une liste de listes. Le module `numpy` contient quelques fonctions commodes pour construire des matrices à partir de rien. Les fonctions `zeros()` et `ones()`permettent de construire des objets *array* contenant des 0 ou de 1, respectivement. Il suffit de leur passer un tuple indiquant les dimensions voulues.

```
>>> np.zeros((3,3))
array([[ 0.,  0.,  0.],
     [ 0.,  0.,  0.],
     [ 0.,  0.,  0.]])
>>> np.zeros((3,3), int)
array([[0, 0, 0],
     [0, 0, 0],
     [0, 0, 0]])
>>> np.ones((3,3))
```

```
array([[ 1.,  1.,  1.],
     [ 1.,  1.,  1.],
     [ 1.,  1.,  1.]])
```

Par défaut, les fonctions `zeros()`et `ones()` génèrent des réels, mais vous pouvez demander des entiers en passant l'option `int` en second argument.

Enfin, si voulez construire une matrice avec autre chose que des 0 ou des 1, vous avez à votre disposition la fonction `full()` :

```
>>> np.full((2,3), 7, int)
array([[7, 7, 7],
       [7, 7, 7]])
>>> np.full((2,3), 7, float)
array([[ 7.,  7.,  7.],
       [ 7.,  7.,  7.]])
```

Nous construisons ainsi une matrice constituée de 2 lignes et 3 colonnes et qui ne contient que le chiffre 7. Des entiers (`int`) dans le premier cas et des réels (`float`) dans le second.

17.1.5 Un peu d'algèbre linéaire

Après avoir manipulé les objets *array* comme des vecteurs et des matrices, voici quelques fonctions pour faire de l'algèbre linéaire.

La fonction `transpose()` renvoie la transposée d'un *array*. Par exemple pour une matrice :

```
>>> a
array([[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]])
>>> np.transpose(a)
array([[1, 4, 7],
     [2, 5, 8],
     [3, 6, 9]])
```

La fonction `dot()` vous permet de faire une multiplication de matrices.

```
>>> a = np.resize(numpy.arange(4),(2,2))
>>> a
array([[0, 1],
     [2, 3]])
>>> np.dot(a,a)
array([[ 2,  3],
     [ 6, 11]])
>>> a * a
array([[0, 1],
     [4, 9]])
```

Notez bien que `dot(a,a)` renvoie le **produit matriciel** entre deux matrices, alors que `a * a` renvoie le produit **élément par élément**.

Remarque : Dans numpy, il existe également des objets de type *matrix* pour lesquels les multiplications de matrices sont différents, mais nous ne les aborderons pas ici.

Pour toutes les opérations suivantes, il faudra utiliser des fonctions dans le sous-module `numpy.linalg`. La fonction `inv()` renvoie l'inverse d'une matrice carrée, `det()` son déterminant, `eig()` ses vecteurs et

valeurs propres.

```
>>> a
array([[0, 1],
     [2, 3]])
>>> np.linalg.inv(a)
array([[-1.5,  0.5],
     [ 1. ,  0. ]])
>>> np.linalg.det(a)
-2.0
>>> np.linalg.eig(a)
(array([-0.56155281,  3.56155281]), array([[-0.87192821, -0.27032301],
     [ 0.48963374, -0.96276969]]))
>>> np.linalg.eig(a)[0]
array([-0.56155281,  3.56155281])
>>> np.linalg.eig(a)[1]
array([[-0.87192821, -0.27032301],
     [ 0.48963374, -0.96276969]])
```

Notez que la fonction `eig()` renvoie un tuple dont le premier élément correspond aux valeurs propres et le second élément aux vecteurs propres.

17.1.6 Un peu de transformée de Fourier

La transformée de Fourier est très utilisée pour l'analyse de signaux, notamment lorsqu'on souhaite extraire des périodicités au sein d'un signal bruité. Le module `numpy` possède la fonction `fft()` (dans le sous-module `fft`) permettant de calculer des transformées de Fourier.

Voici un petit exemple sur la fonction cosinus de laquelle on souhaite extraire la période à l'aide de la fonction `fft()` :

```
# 1) on définit la fonction y = cos(x)
import numpy as np
debut = -2 * np.pi
fin = 2 * np.pi
pas = 0.1
x = np.arange(debut,fin,pas)
y = np.cos(x)

# 2) on calcule la transformée de Fourier (TF) de la fonction cosinus
TF = np.fft.fft(y)
ABSTF = np.abs(TF)
# abcisse du spectre en radian^-1
pas_xABSTF = 1/(fin-debut)
x_ABSTF = np.arange(0,pas_xABSTF * len(ABSTF),pas_xABSTF)
```

Plusieurs commentaires sur cet exemple :

— Vous constatez que `numpy` redéfinit certaines fonctions ou constantes mathématiques de base, comme pi (nombre π), `cos()` (fonction cosinus) ou `abs()` (valeur absolue, ou module d'un complexe). Ceci est bien pratique car nous n'avons pas à appeler ces fonctions ou constantes depuis le module `math`, le code en est ainsi plus lisible.
— Dans la partie 1, on définit le vecteur `x` représentant un angle allant de -2π à 2π radians par pas de 0,1 et le vecteur `y` comme le cosinus de `x`.

— Dans la partie 2, on calcule la transformée de Fourier avec la fonction `fft()` qui renvoie un vecteur (objet *array* à une dimension) de nombres complexes. Eh oui, le module `numpy` gère aussi les nombres complexes ! On extrait ensuite le module du résultat précédent avec la fonction `abs()`.
— La variable `x_ABSTFL` représente l'abscisse du spectre (en radian^{-1}).
— La variable `ABSTF` contient le spectre lui même. L'analyse de ce dernier nous donne un pic à 0,15 radian^{-1}, ce qui correspond bien à 2π (plutôt bon signe de retrouver ce résultat). Le graphe de ce spectre est présenté dans la partie dédiée à `matplotlib`).

Notez que tout au long de cette partie, nous avons toujours utilisé la syntaxe `np.fonction()` pour bien vous montrer quelles étaient les fonctions propres à `numpy` (dont le nom est ici raccourcis par `np`).

17.2 Module biopython

Le module biopython propose de nombreuses fonctionnalités très utiles en bioinformatique. Le tutoriel est particulièrement bien fait, n'hésitez pas à le consulter.

Voici quelques exemples d'utilisation.

Définition d'une séquence :

```
>>> import Bio
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> ADN = Seq("ATATCGGCTATAGCATGCA", IUPAC.unambiguous_dna)
>>> ADN
Seq('ATATCGGCTATAGCATGCA', IUPACUnambiguousDNA())
```

L'expression `IUPAC.unambiguous_dna` signifie que la séquence entrée est bien une séquence d'ADN.

Obtention de la séquence complémentaire et complémentaire inverse :

```
>>> ADN.complement()
Seq('TATAGCCGATATCGTACGT', IUPACUnambiguousDNA())
>>> ADN.reverse_complement()
Seq('TGCATGCTATAGCCGATAT', IUPACUnambiguousDNA())
```

Traduction en séquence protéique :

```
>>> ADN.translate()
Seq('ISAIAC', IUPACProtein())
```

17.3 Module matplotlib

Le module matplotlib permet de générer des graphiques depuis Python. Il est l'outil complémentaire de `numpy` et `scipy` lorsqu'on veut faire de l'analyse de données.

17.3.1 Réprésentation sous forme de points

Dans cet exemple, nous considérons l'évolution de la concentration d'un produit dans le sang (exprimé en mg/L) en fonction du temps (exprimé en heure).

Nous avons les résultats suivants :

Temps (h)	Concentration (mg/L)
1	3.5
2	5.8
3	9.1
4	11.8
6	17.5
7	21.3
9	26.8

Nous allons maintenant représenter l'évolution de la concentration en fonction du temps :

```
import matplotlib.pyplot as plt

temps = [1, 2, 3, 4, 6, 7, 9]
concentration = [5.5, 7.2, 11.8, 13.6, 19.1, 21.7, 29.4]

plt.scatter(temps, concentration, marker='o', color = 'blue')

plt.xlabel("Temps (h)")
plt.ylabel("Concentration (mg/L)")
plt.title("Concentration de produit en fonction du temps")
plt.show()
```

Vous devriez obtenir une fenêtre graphique **interactive** qui vous permet de manipuler le graphe (se déplacer, zoomer, enregistrer comme image, etc.) comme la figure 17.1.

Revenons maintenant sur le code. Tout d'abord, vous voyez qu'on importe le sous-module `pyplot` du module `matplotlib` et qu'on lui donne un nom court `plt` pour l'utiliser plus rapidement ensuite.

— Dans un premier temps, nous définisson les variables `temps` et `concentration` comme des listes. Les deux listes doivent avoir la même longueur (7 éléments dans le cas présent).
— La fonction `scatter()` permet de représenter des points sous forme de nuage de points. Les deux premiers arguments correspondent aux valeurs en abscisse et en ordonnée des points expérimentaux, fournis sous forme de listes. Nous avons ensuite des aguments facultatifs comme le symbole (`marker`) et la couleur (`color`).
— Les fonctions `xlabel()` et `ylabel()` sont utiles pour donner un nom aux axes.
— La fonction `title()` permet de définir le titre du graphique.
— Enfin, la fonction `show()` affiche le graphique généré à l'écran.

17.3.2 Réprésentation sous forme de courbe

Nous savons par ailleurs que l'évolution de la concentration du produit en fonction du temps peut-être modélisée par la fonction $f(x) = 2 + 3 \times x$. Nous allons réprésenter cette fonction avec les points expérimentaux et sauvegarder le graphique obtenu sous forme d'une image :

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

temps = [1, 2, 3, 4, 6, 7, 9]
concentration = [5.5, 7.2, 11.8, 13.6, 19.1, 21.7, 29.4]

plt.scatter(temps, concentration, marker='o', color = 'blue')
```

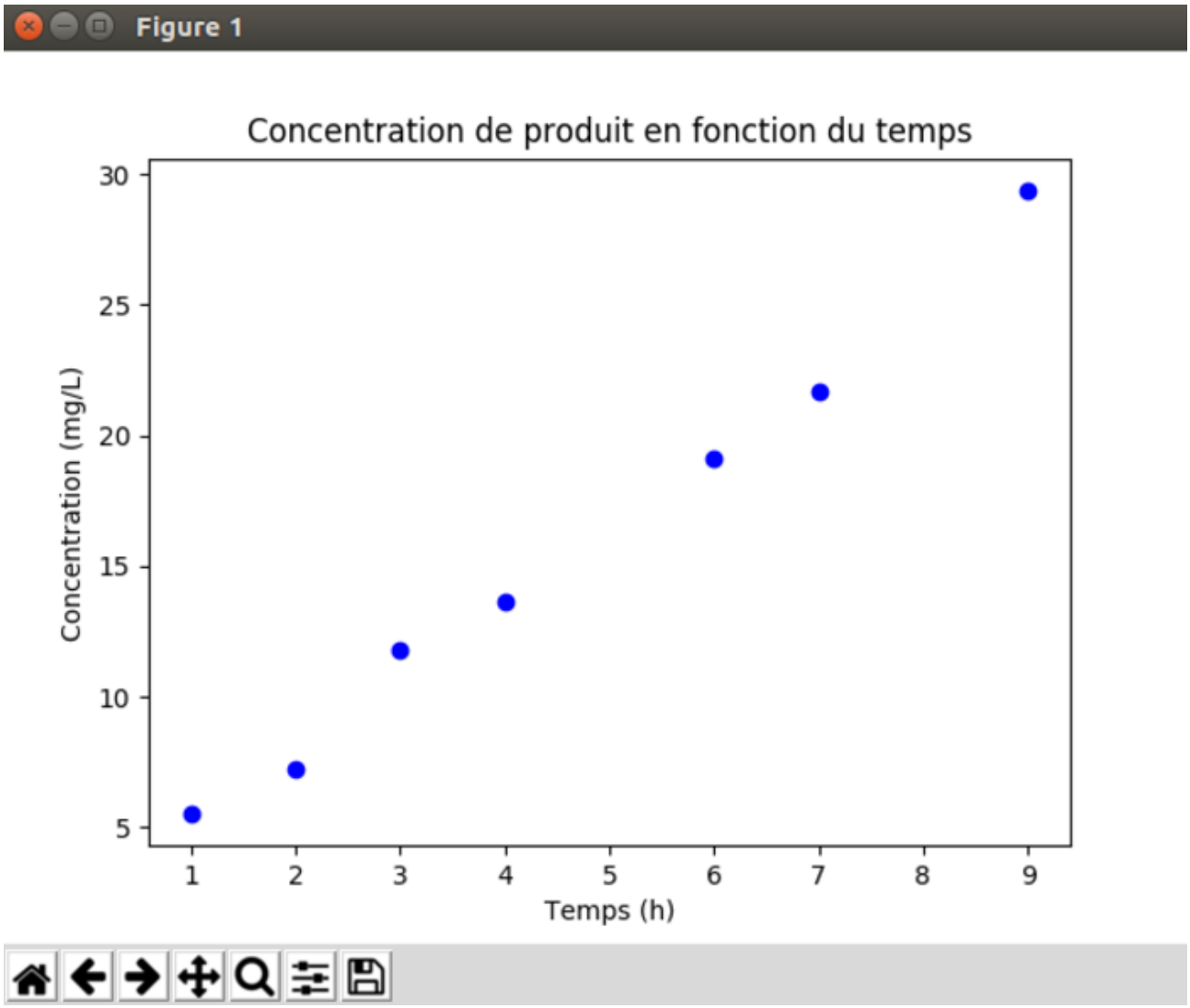


FIGURE 17.1 – Fenêtre interactive de matplotlib

```
plt.xlabel("Temps (h)")
plt.ylabel("Concentration (mg/L)")
plt.title("Concentration de produit en fonction du temps")

x = np.linspace( min(temps), max(temps), 50)
y = 2 + 3 * x
plt.plot(x, y, color='green', ls="--")
plt.grid()

plt.savefig('concentration_vs_temps.png', bbox_inches='tight', dpi=200)
```

Le résultat est représenté sur la figure 17.2.

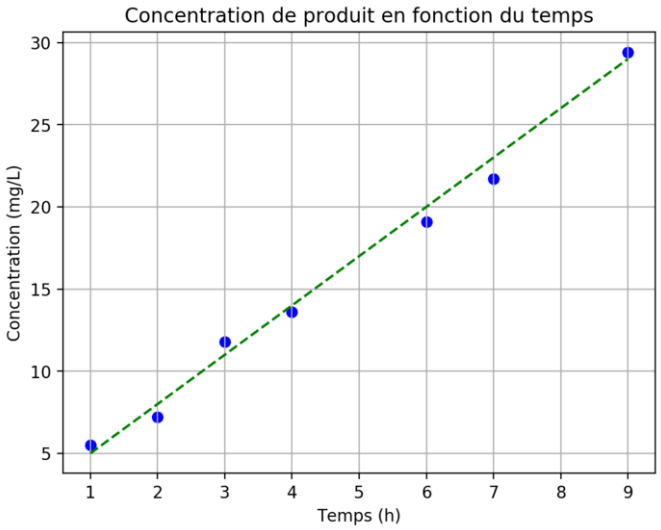


FIGURE 17.2 – Concentration du produit en fonction du temps

Nous avons ajouté les étapes suivantes :

— Nous chargeons le module `numpy` sous le nom `np`.
— Nous créons la variable `x` avec la fonction `linspace()` du module `numpy` qui crée une liste de valeurs régulièrements espacées entre deux bornes, ici le minimum (`min(temps)`) et le maximum (`max(temps)`) de la variable `temps`. Dans notre exemple, nous générons une liste de 50 valeurs. La variable `x` ainsi créée est du type *array*.
— Nous construisons ensuite la variable `y` à partir de la formule modélisant l'évolution de la concentration en fonction du temps. Cette manipulation n'est possible que parce que `x` est du type *array*. Cela ne fonctionnerait pas avec une liste classique.
— La fonction `plot()` permet de construire une courbe à partir des coordonnées en abscisse et en ordonnées des points à représenter. Nous avons ensuite des aguments facultatifs comme le style de la ligne (`ls`) et la couleur (`color`).
— La fonction `grid()` affiche une grille.
— Enfin, le fonction `savefig()` permet d'enregistrer le graphique produit sous la forme d'une image au format png. Des arguments optionels definissent la manière de générer les marges autour du graphique (`bbox_inches`) et la résolution de l'image (`dpi`).

17.3.3 Réprésentation sous forme de barres

Nous souhaitons maintenant représenter graphiquement la distribution des différentes bases dans une séquence nucléique.

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

# 1 : calcul de la distribution des bases
sequence = "ACGATCATAGCGAGCTACGTAGAA"
bases = ["A", "C", "G", "T"]
distribution = []
for base in bases:
    distribution.append(sequence.count(base))

# 2 : position des barres
x = np.arange(len(bases))

# 3 : représentation des barres
# et de l'axe des abscisses
plt.bar(x, distribution)
plt.xticks(x, bases)

# 4 : légendes des axes et titre du graphique
plt.xlabel("Bases")
plt.ylabel("Nombre")
plt.title("Distribution des bases\n dans la séquence {}".format(sequence))

# 5 : sauvegarde du graphique
plt.savefig('distribution_bases.png', bbox_inches='tight', dpi=200)
```

On obtient alors le graphique de la figure 17.3.

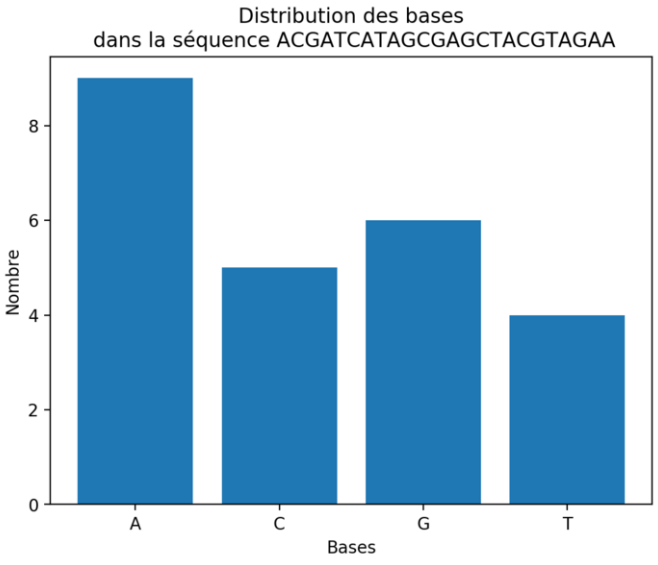


FIGURE 17.3 – Distribution des bases

Prenons le temps d'examiner les différentes étapes du script précédent :

1. Nous définissons les variables (`sequence` et `bases`) que nous allons utiliser. Nous calculons ensuite la distribution des différentes bases dans la séquence. Nous utilisons pour cela la fonction `count()` qui renvoie le nombre de fois qu'une sous-chaîne de caractères se trouve dans une autre.
2. Nous définissons la positions en abscisse des barres. Dans notre exemple, la variable `x` vaut `array([0, 1, 2, 3])`.

3. Nous construisons la réprésentation graphique sous forme de barres avec la fonction `bar()` qui prend en argument la position des barres (`x`) et leurs hauteurs (`distribution`). La fonction `xtics()` redéfinit les étiquettes (c'est-à-dire le nom des bases) correspondantes aux différents positions des barres.
4. Nous définissions les légendes des axes et le titre du graphique. Nous insérons un retour à la ligne `\n` dans le titre pour que cela soit plus esthétique.
5. Enfin, nous enregistrons le graphique généré au format png.

Voilà, nous espèrons que ce petit exemple vous aura convaincu de l'utilité du module `matplotlib`. Sachez qu'il peut faire bien plus, par exemple générer des histogrammes ou toutes sortes de graphiques utiles en analyse de données.

17.4 Exercices

Cet exercice présente le calcul de la distance entre deux atomes carbones alpha consécutifs de la barstar. Il demande quelques notions d'Unix et nécessite le module `numpy` de Python.

17.4.1 Extraction des coordonnées atomiques

Téléchargez le fichier `1bta.pdb` qui correspond à la structure tridimensionnelle de la protéine barstar sur le site de la PDB (lien direct vers le fichier).

Voici le code pour extraire les coordonnées atomiques des carbones alpha de la barstar :

```
with open("1bta.pdb", "r") as fichier_pdb, open("1bta_ca.txt", "w") as fichier_CA:
    for ligne in fichier_pdb:
          champs = ligne.split()
          if champs[0] == "ATOM" and champs[2] == "CA":
                fichier_CA.write("{} {} {} ".format(champs[6], champs[7], champs[8]))
```

Notez la structure de la forme

```
with open("1bta.pdb", "r") as fichier_pdb, open("1bta_CA.txt", "w") as fichier_CA:
```

qui permet d'ouvrir deux fichiers simultanéments. Ici, le fichier `1bta.pdb` est ouvert en lecture (`r`) et le fichier `1bta_ca.txt` est ouvert en écriture (`w`).

Pour chaque ligne du fichier pdb, nous allons séparer le contenu de la ligne en champs et lorsque le premier champs vaut `ATOM` et le troisième vaut `CA`, alors nous écrivons les coordonnées de l'atome (7e, 8e et 9e champs) dans le fichier `1bta_ca.txt`. Les coordonnées sont toutes enregistrées sur une seul ligne, les unes après les autres.

Le même résultat peut être obtenu avec une seule commande Unix. Il faut par contre s'assurer que le séparateur décimal est bien le point (au lieu de la virgule, par défaut sur les systèmes d'exploitation français), et au préalable redéfinir la variable `LC_NUMERIC` en bash :

```
export LC_NUMERIC=C
```

La commande à utiliser est ensuite :

```
awk '$1=="ATOM" && $3=="CA" {printf "%.3f %.3f %.3f ", $7, $8, $9}' 1bta.pdb > 1bta_ca.txt
```

17.4.2 Lecture des coordonnées

Ouvrez le fichier `1bta_ca.txt` avec Python et créez une liste contenant toutes les coordonnées sous forme de réels avec les fonctions `split()` et `float()`.

Affichez à l'écran le nombre de total de coordonées.

17.4.3 Utilisation de numpy

En ouvrant dans un éditeur de texte le fichier `1bta.pdb`, trouvez le nombre d'acides aminés qui constituent la barstar.

Avec la fonction `array()` du module `numpy`, convertissez la liste de coordonnées en `array`. Avec la fonction `reshape()` de `numpy`, construisez ensuite une matrice à deux dimensions contenant les coordonnées des carbones alpha de la barstar. Affichez les dimensions de cette matrice.

17.4.4 Calcul de la distance

Créez maintenant une matrice qui contient les coordonnées des `n-1` premiers carbones alpha et une autre qui contient les coordonnées des `n-1` derniers carbones alpha. Affichez les dimensions des matrices pour vérification.

En utilisant les opérateurs mathématiques habituels (`-`, `+`, `**2`) et les fonctions `sqrt()` et `sum()` du module `numpy`, calculez la distance entre les atomes `n` et `n+1`.

Pour chaque atome, affichez le numéro de l'atome et la distances entre carbones alpha consécutifs avec un chiffres après la virgule. Repérez la valeur surprenante.

Chapitre 18

Avoir la classe avec les objets

Une classe permet de définir des objets qui sont des représentants (des **instances**) de cette classe. Les objets peuvent posséder des **attributs** (variables associées aux objets) et des **méthodes** (qui peuvent être vues comme des fonctions associées aux objets).

18.1 Construction d'une classe

Exemple de la classe `Rectangle` :

```
class Rectangle:
    """
    Ceci est la classe Rectangle
    """

    def __init__(self, long=0.0, larg=0.0, coul="blanc"):
        """
        Initialisation d'un objet.

        Définition des attributs avec des valeurs par défaut.
        """
        self.longueur = long
        self.largeur = larg
        self.couleur = coul


    def calcule_surface(self):
        """
        Méthode qui calcule la surface.
        """
        return self.longueur * self.largeur


    def change_carre(self, cote):
        """
        Méthode qui transforme un rectangle en carré.
        """
        self.longueur = cote
        self.largeur = cote
```

Dans cet exemple, `longueur`, `largeur` et `couleur` sont des attributs alors que `calcule_surface()` et `change_carre()` sont des méthodes. Tous les attributs et toutes les méthodes se réfèrent toujours à `self` qui désigne l'objet lui même. Attention, les méthodes prennent au moins `self` comme argument.

18.2 Utilisation de la classe Rectangle

Création d'un objet `Rectangle` avec les paramètres par défaut :

```
rect1 = Rectangle()
print("longueur {} / largeur {} / couleur {}".format( rect1.longueur, rect1.largeur, rect1.cou
print("surface = {:.2f} m2".format( rect1.calcule_surface() ))
```

Résultat :

```
longueur 0.0 / largeur 0.0 / couleur blanc
surface = 0.00 m2
```

18.3 Modification du rectangle en carré

```
rect1.change_carre(30)
rect1.calcule_surface()
```

Résultat :

```
surface = 900.00 m2
```

18.4 Création d'un objet Rectangle avec des paramètres imposés

```
rect2 = Rectangle(2, 3, "rouge")
print("longueur {} / largeur {} / couleur {}".format( rect2.longueur, rect2.largeur, rect2.cou
print("surface = {:.2f} m2".format( rect2.calcule_surface() ))
```

Résultat :

```
longueur 2 / largeur 3 / couleur rouge
surface = 6.00 m2
```

18.5 Exercices

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

18.5.1 Classe Rectangle

Entraînez-vous avec la classe `Rectangle`. Créez la méthode `calcule_perimetre()` qui calcule le périmètre d'un objet rectangle.

18.5.2 Classe Atome

Créez une nouvelle classe `Atome` avec les attributs `x`, `y`, `z` (qui contiennent les coordonnées atomiques) et la méthode `calcul_distance()` qui calcule la distance entre deux atomes. Testez cette classe sur plusieurs exemples.

18.5.3 Classe Atome améliorée

Améliorez la classe `Atome` en lui ajoutant un nouvel attribut `masse` qui correspond à la masse atomique et `numero_atomique`) et une nouvelles méthodes`calcule_centre_masse`.

Chapitre 19

Pour aller plus loin

19.1 Shebang et /usr/bin/env python3

Lorsque vous programmez sur un sytème Unix, le shebang correspond aux caractères `#!` qui se trouve au début de la première ligne d'un script. Le shebang est suivi du chemin complet du programme qui interprète le script.

En Python, on trouve souvent la notation

```
#! /usr/bin/python3
```

Cependant, l'exécutable `python3` ne se trouve pas toujours dans le répertoire `/usr/bin`. Pour maximiser la portabilité de votre script Python sur plusieurs systèmes Unix, utilisez plutôt cette notation :

```
#! /usr/bin/env python3
```

Dans le cas présent, on appelle le programme d'environnement `env` (qui se situe toujours dans le répertoire `/usr/bin`) pour lui demander où se trouve l'exécutable `python3`.

19.2 Differences Python 2 et Python 3

19.2.1 Interpréteur par défaut

Par défault sous Ubuntu 16.04, l'exécutable `python` va lancer Python 2 :

```
$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Pensez à explicitement préciser la version de Python qui vous intéresse, ici Python 3 :

```
$ python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

19.2.2 La fonction print()

La fonction `print()` en Python 2 s'utilise sans parenthèse. Par exemple :

```
>>> print 12
12
>>> print "girafe"
girafe
```

Par contre en Python 3, si vous n'utilisez pas de parenthèse, Python vous renverra une erreur :

```
>>> print 12
  File "<stdin>", line 1
    print 12
          ^
SyntaxError: Missing parentheses in call to 'print'
```

19.2.3 Division d'entiers

En Python 3, la division de deux entiers, se fait *naturellement*, c'est-à-dire que l'opérateur `/` renvoie systématiquement un *float*. Par exemple :

```
>>> 3 / 4
0.75
```

Il est également possible de réaliser une division entière avec l'opérateur `//` :

```
>>> 3 // 4
0
```

La division entière renvoie finalement la partie entière du nombre `0.75`, c'est à dire `0`.

Attention ! En Python 2, la division de deux entiers avec l'opérateur `/` correspond à la division entière, c'est-à-dire le résultat arrondi à l'entier inférieur. Par exemple :

```
>>> 3 / 5
0
>>> 4 / 3
1
```

Faites très attention à cet aspect si vous programmez encore en Python 2, c'est une source d'erreur récurrente.

19.2.4 La fonction range()

En Python 3, la fonction `range()` est un générateur, c'est-à-dire que cette fonction va itérer sur le nombre entier donner en argument. On ne peut pas l'utiliser seule :

```
>>> range(3)
range(0, 3)
```

Lorsque qu'on l'utilise dans une boucle `for`, `range(3)` va produire successivement les nombres `0`, `1` puis `2`. Par exemple :

```
>>> for i in range(3):
...     print(i)
...
0
```

```
1
2
```

En Python 2, la fonction `range()` renvoie une liste. Par exemple :

```
>>> range(3)
[0, 1, 2]
>>> range(2, 6)
[2, 3, 4, 5]
```

La création de liste avec `range()` était pratique mais très peu efficace en mémoire lorsque l'argument de `range()` était un grand nombre.

D'ailleurs la fonction `xrange()` est disponible en Python 2 pour faire la même chose que la fonction `range()` en Python 3. Attention, ne vous mélangez pas les pinceaux !

```
>>> range(3)
[0, 1, 2]
>>> xrange(3)
xrange(3)
```

Remarque : pour générer une liste d'entiers avec la fonction `range()` en Python 3, vous avez vu dans le chapitre sur les *Listes* qu'il suffit de l'associer avec la fonction `list()`. Par exemple :

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

19.2.5 Encodage et utf-8

En Python 3, vous pouvez utiliser des caractères accentués dans les commentaires ou dans les chaîne de caractères.

Ce n'est pas le cas en Python 2. Si un caractère accentué est présent dans votre code, cela occasionnera une erreur de ce type lors de l'exécution de votre script :

```
SyntaxError: Non-ASCII character '\xc2' in file xxxx on line yyyy, but no encoding declared; s
```

Pour éviter ce genre de désagrément, ajoutez la ligne suivante à la deuxième ligne (la position est importante) de votre script :

```
# coding: utf-8
```

En résumé, tous vos scripts en Python 2 devraient ainsi débuter par les lignes :

```
#! /usr/bin/env python2
# coding: utf-8
```

Remarque : l'encodage utf-8 peut aussi être déclaré de cette manière :

```
# -*- coding: utf-8 -*-
```

mais c'est un peu plus long à écrire.

19.3 Liste de compréhension

Une manière originale et très puissante de générer des listes est la compréhension de listes. Pour plus de détails, consultez à ce sujet le site de Python et celui de Wikipédia.

Voici quelques exemples.

19.3.1 Nombres pairs compris entre 0 et 30

```
>>> print([i for i in range() if i%2 == 0])
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]
```

19.3.2 Jeu sur la casse des mots d'une phrase

```
>>> message = "C'est sympa la BioInfo"
>>> msg_lst = message.split()
>>> print([[m.upper(), m.lower(), len(m)] for m in msg_lst])
[["C'EST", "c'est", 5], ['SYMPA', 'sympa', 5], ['LA', 'la', 2], ['BIOINFO', 'bioinfo', 7]]
```

19.3.3 Formatage d'une séquence avec 60 caractères par ligne

```
# exemple d'une séquence de 150 alanines
>>> seq = "A"*150
>>> width= 60
>>> print("\n".join( [seq[i:i+width] for i in range(0,len(seq),width)] ))
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

19.3.4 Formatage fasta d'une séquence (avec la ligne de commentaire)

Exemple d'une séquence constituée de 150 alanines :

```
>>> com = "séquence de 150 alanines"
>>> seq = "A"*150
>>> width = 60
>>> print(">"+com+"\n"+"\n".join( [seq[i:i+width] for i in range(0,len(seq),width)] ))
>séquence de 150 alanines
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

19.3.5 Sélection des carbones alpha dans un fichier pdb

Exemple avec la structure de la barstar :

```
>>> with open("1BTA.pdb", "r") as fichier:
...     CA_lines = [l for l in fichier if l.split()[0] == "ATOM" and l.split()[2] == "CA"]
...
>>> print(len(CA_lines))
89
```

19.4 Gestion des erreurs

La gestion des erreurs permet d'éviter que votre programme plante en prévoyant vous même les sources d'erreurs éventuelles.

Voici un exemple dans lequel on demande à l'utilisateur d'entrer un nombre entier, puis on affiche ce nombre.

```
>>> nb = int(input("Entrez un nombre entier : "))
Entrez un nombre entier : 23
>>> print(nb)
23
```

La fonction `input()` demande à l'utilisateur de saisir une chaîne de caractères. Cette chaîne de caractères est ensuite transformée en nombre entier avec la fonction `int()`.

Si l'utilisateur ne rentre pas un nombre, voici ce qui se passe :

```
>>> nb = int(input("Entrez un nombre entier : "))
Entrez un nombre entier : ATCG
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'ATCG'
```

L'erreur provient de la fonction `int()` qui n'a pas pu convertir la chaîne de caractères `"ATCG"` en nombre entier, ce qui est parfaitement normal.

Le jeu d'instructions `try` / `except` permet de tester l'exécution d'une commande et d'intervenir en cas d'erreur.

```
>>> try:
...     nb = int(input("Entrez un nombre entier : "))
... except:
...     print("Vous n'avez pas entré un nombre entier !")
...
Entrez un nombre entier : ATCG
Vous n'avez pas entré un nombre entier !
```

Dans cette exemple, l'erreur renvoyée par la fonction `int()` (qui ne peut pas convertir `"ATCG"` en nombre entier) est interceptée et déclenche l'affichage du message d'avertissement.

On peut ainsi redemander sans cesse un nombre entier à l'utilisateur, jusqu'à ce que celui-ci en rentre bien un.

```
>>> while True:
...     try:
...             nb = int(input("Entrez un nombre entier : "))
...             print("Le nombre est", nb)
...             break
...     except:
...             print("Vous n'avez pas entré un nombre entier !")
...             print("Essayez encore")
...
Entrez un nombre entier : ATCG
Vous n'avez pas entré un nombre entier !
Essayez encore
Entrez un nombre entier : toto
Vous n'avez pas entré un nombre entier !
Essayez encore
Entrez un nombre entier : 3.2
Vous n'avez pas entré un nombre entier !
Essayez encore
```

```
Entrez un nombre : 55
Le nombre est 55
```

Notez que dans cet exemple, l'instruction `while 1` est une boucle infinie (car la condition `True` est toujours vérifiée) dont l'arrêt est forcé par la commande `break` lorsque l'utilisateur a effectivement bien rentré un nombre entier.

La gestion des erreurs est très utile dès lors que des données extérieures entrent dans un programme Python, que ce soit directement par l'utilisateur (avec la fonction `input()`) ou par des fichiers.

Vous pouvez par exemple vérifier qu'un fichier a bien été ouvert.

```
>>> nom = "toto.pdb"
>>> try:
...     with open(nom, "r") as fichier:
...         for ligne in fichier:
...             print(ligne)
... except:
...     print("Impossible d'ouvrir le fichier", nom)
```

Si une erreur est déclenchée, c'est sans doute que le fichier n'existe pas à l'emplacement indiqué sur le disque ou que vous n'avez pas les droits pour le lire.

Il est également possible de spécifier le type d'erreur à gérer. Le premier exemple que nous avons étudié peut s'écrire :

```
>>> try:
...     nb = int(input("Entrez un nombre entier : "))
... except ValueError:
...     print("Vous n'avez pas entré un nombre entier !")
...
Entrez un nombre entier : ATCG
Vous n'avez pas entré un nombre entier !
```

Ici, on intercepte une erreur de type `ValueError`, ce qui correspond bien à un problème de conversion avec `int()`. Il existe d'autres types d'erreurs comme `RuntimeError`, `TypeError`, `NameError`, `IOError`, etc.

Enfin, on peut aussi être très précis dans le message d'erreur. Observez la fonction `download_page()` qui, avec le module `urllib`, télécharge un fichier sur internet.

```
import urllib.request

def download_page(address):
    error = ""
    page = ""
    try:
        data = urllib.request.urlopen(address)
        page = data.read()
    except IOError as e:
        if hasattr(e, 'reason'):
            error =  "Cannot reach web server: " + str(e.reason)
        if hasattr(e, 'code'):
            error = "Server failed {:d}".format(e.code)
    return page, error

data, error = download_page("https://files.rcsb.org/download/1BTA.pdb")
```

```
if error:
    print("Erreur rencontrée : {}".format(error))
else:
    with open("proteine.pdb", "w") as prot:
        prot.write(data.decode('utf-8'))
    print("Protéine enregistrée")
```

La variable `e` est une instance (un représentant) de l'erreur de type `IOError`. Certains de ces attributs sont testés avec la fonction `hasattr()` pour ainsi affiner le message renvoyé (ici contenu dans la variable `error`).

Si tout se passe bien, la page est téléchargée et stockée dans la variable `data`, puis ensuite enregistrée sur le disque.

19.5 Sauvegardez votre historique de commandes

Vous pouvez sauvegarder l'historique des commandes utilisées dans l'interpréteur Python avec le module `readline`.

```
>>> print("hello")
hello
>>> a = 22
>>> a = a + 11
>>> print(a)
33
>>> import readline
>>> readline.write_history_file()
```

Quittez Python. L'historique de toutes vos commandes est dans votre répertoire personnel, dans le fichier `.history`.

Relancez l'interpréteur Python.

```
>>> import readline
>>> readline.read_history_file()
```

Vous pouvez accéder aux commandes de la session précédente avec la flèche du haut de votre clavier. D'abord les commandes `readline.read_history_file()` et `import readline` de la session actuelle, puis `print(a)`, `a = a + 11`, `a = 22`...