Table of Contents

Python à usage scientifique

Ce document est à destination de personnes désirant découvrir la programmation ou se former à l'utilisation de python. Il est plutôt destiné à des scientifiques.

Ce cours est utilisé dans le cadre de l'offre de formation doctorale de l'école doctorale des sciences exactes et leurs applications (ED211) de l'Université de Pau et des Pays de l'Adour (UPPA) et organisé conjointement avec le pôle applications scientifiques de la direction du numérique de l'UPPA.

Le cours d'introduction au langage python est extrait du Manuel Django Carrots.

Le site de SciPy est également une source d'information pour faire des sciences avec python.

Formateurs : Patrice Bordat, Jacques Hertzberg, Vincent Le Bris, Marc Odunlami, Germain Salvato Vallverdu

---

Ressources sur python

La première source d'information est la documentation officielle de python 3 : docs.python.org/3/

Une version française de la documentation est en cours de traduction et ne demande que des bonnes volontés pour avancer.

Modules python pour les sciences

Les module suivants sont quelques modules utiles en sciences. La liste n'est bien sur pas exhaustive.

La base :

Numpy, scipy et matplotlib forment la base d'un environement de travail.

- NumPy : Numerical python : site officiel
- SciPy : Scientific python documentation officielle.
- matplotlib : Matplotlib est une bibliothèque permettant de réaliser des graphiques. documentation officille

Autres :

- Pandas : Python Data Analysis Library documentation officille
- plotly : Visualize Data, Together Create and share charts, datasets, and dashboards online site officiel
- SymPy : SymPy is a Python library for symbolic mathematics site officiel

SymPy est une large bibliothèque couvrant divers domaines : mathématiques, physiques, statistiques ...

Environnement de travail

IPython est un prompt python interactif permettant de tester ses idées et des lignes de codes avec un accès intégré à la documentation.

IPython est une partie du projet jupyter. Le notebook Jupyter est une application web qui permet de créer et de partager des documents qui contiennent des lignes de codes exécutables, des équations, de la visualisation et du texte pour les explications.

L'utilisation du notebook jupyter ou de `jupyter qtconsole` est vivement recommandées.

La communauté Python

La communauté d'utilisateurs de python est large, hétéroclite et dynamique. Il se dit du langage python *qu'il est excellent nulle part mais très bon partout*. En effet python permet de faire aussi bien des sciences que des applications web ou encore du traitement d'images ou de fichiers musicaux. L'AFPY, association francophone python a pour objectif la promotion du langage python et rassemble la communauté francophone.

Plus localement, sur Pau, plusieurs membres de l'association Paulla sont également membres de l'AFPY.

Python sur pyrene

Cette page donne quelques éléments spécifiques à l'utilisation de python sur la machine pyrene de l'Université de Pau et des Pays de l'Adour.

Pour une installation personnelle de python, reportez vous aux exécutables disponnibles sur le site officiel de python. De nombreux paquets python sont disponnibles par défaut dans les dépôts de la plupart des distributions linux. Python mais également à disposition sont propre dépôt de modules : PyPi. L'installation se fait via la commande `pip3`. L'utilisation de `python3` est recommandée.

Utilisation sur pyrene

Comme sur toute distribution linux, python est disponnible par défaut.

```
# pyuser@pyrene <~> python
Python 2.6.6 (r266:84292, Jan 22 2014, 05:06:49)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Sur pyrene, modules environnement python sont disponnibles et permettent de choisir une version.

```
# pyuser@pyrene <~> module avail python

---------- /opt/cluster/modulefiles ---------
python/2.7.9 python/3.5.1

---------- /opt/cluster/modulefilesold ----------
python/2.7.3 python/2.7.5
```

Il est recommandé de choisir python3 et donc de charger le module correspondant `python/3.5.1` :

```
# pyuser@pyrene <~> module load python/3.5.1
# pyuser@pyrene <~> python3
Python 3.5.1 (default, Jan 25 2016, 22:08:53)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-11)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Prompt amélioré

Pour utiliser python dans un prompt plus convivial, on pourra utiliser `ipython3` ou la nouvelle version du projet : `jupyter console`.

```
# pyuser@pyrene <~> ipython3
Python 3.5.1 (default, Jan 25 2016, 22:08:53)
Type "copyright", "credits" or "license" for more information.

IPython 4.0.3 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

ou

```
# pyuser@pyrene <~> jupyter console
Jupyter Console 4.1.0


In [1]:
```

Il est également possible d'utiliser un `notebook` avec la commande `jupyter notebook`.

ATTENTION : On rappelle que sur pyrene, le temps d'exécution d'un programme est limité à 20min ! Pour une utilisation plus longue du prompt ou d'un notebook et pour ne pas surcharger le serveur frontal pour des traitements lourd, il est conseiller de prendre une session interactive avec `salloc`, par exemple, pour une session de 4h :

```
# pyuser@pyrene <~> salloc --time=04:00:00
salloc: Granted job allocation 579381
srun: Job step created
# pyuser@dnas-node31 <~>
```

Apprentissage du langage

Cette première partie est dédiée à la découverte de la programmation et en particulier à l'apprentissage du langage python. Vous pourrez ensuite passer à des exercices plus avancés.

Les exercices sur le morse, l'arbre de noël et l'introduction des classes sont directement extraits du cours d'introduction au langage python du Manuel Django Carrots.

Introduction à Python

Commençons par lancer l'interpréteur Python disponnible sur pyrene.

```
# pyuser@pyrene <~> python3
Python 3.5.1 (default, Jan 25 2016, 22:08:53)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-11)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Avant d'entrer python, nous entrions nos commandes sur la ligne de commande du système d'exploitation.

L'invite de ligne de commande (aussi appelée "prompt") était `# pyuser@pyrene <~>`. Une fois la commande python entrée, l'invite de commande a changé et est désormais `>>>`. Cela signifie qu'à partir de maintenant, nous devons uniquement utiliser des commandes du langage Python.

Les commandes telles que cd ou mkdir ne fonctionneront pas. Il est temps d'apprendre un nouveau langage !

Nous ne taperons pas les signes `>>>`, l'interpréteur python le fera pour nous.

Commençons par additionner deux nombres:

```
>>> 2 + 2
4
```

Python est une super calculatrice:

```
>>> 6 * 7
42
>>> 1252 - 38 * 6
1024
>>> (3 - 5) * 7
-14
>>> 21 / 7
3.0
>>> 3**2
9
>>> 5 / 2
2.5
```

À côté des opérateurs standards : +, -, /, *; on trouve la puissance* `** et le reste de la division entière (ou module), %`.

```
>>> 4 % 2
0
>>> 5 % 2
1
```

Faites bien attention lorsque vous entrez des nombres à virgules, utilisez des points comme séparateurs, et non pas des virgules. Les virgules nous seront utiles plus tard, pour définir des listes ou des tuples, mais nous y reviendrons.

Remarque concernant pyton 2 et python 3 : La division n'est pas la même quand on passe de python2 à python3. En python 2, le signe `/` désigne une division entière, en conséquence :

```
# python 2
>>> 2 / 3
0
```

En python3, le signe `/` retourne un nombre flottant. Pour effectuer une division entière on utilisera `//`.

```
# python 3
>>> 2 / 3
0.666666666666
>>> 2 // 3
0
>>> 4 / 2
2.0
```

Le temps des présentations

Chaînes de caractères (Strings)

Les nombres ne sont pas suffisants pour communiquer de manière efficace, nous avons donc besoin d'utiliser des chaînes de caractères (aussi appelées strings).

Voici quelques exemples:

```
>>> "Bonjour tout le monde"
'Bonjour tout le monde'
>>> 'Foo Bar'
'Foo Bar'
>>> "Rock 'n' Roll"
"Rock 'n' Roll"
>>> 'Mon nom est "Camille"'
'Mon nom est "Camille"'
```

Vous pouvez également ajouter (on dit également concaténer) deux chaînes l'une à l'autre:

```
>>> 'Mon nom est ' + '"Camille"'
'Mon nom est "Camille"'
```

Ou elles peuvent être aussi multipliés par des nombres:

```
>>> 'oui ' * 3
'oui oui oui '
```

Une chaîne de caractères doit toujours commencer et se terminer par le même caractère. Il peut s'agir d'un guillemet simple ('), ou d'un guillemet double ("). Cela n'a aucun effet sur la valeur de la chaîne de caractères. Par exemple, si nous entrons "Batman", nous créons une chaîne de caractères Batman, les guillemets ne font pas partie de la chaîne de caractères, ils sont là uniquement pour indiquer qu'il s'agit d'une chaîne de caractères (string) (malheureusement, Python n'est pas suffisamment brillant pour se rendre compte de ça lui-même).

Afficher les chaînes de caractères

Mais, comment afficher ces chaînes de caractères d'une manière lisible ? Il est possible de le faire en utilisant la fonction `print()`.

```
>>> print("Bonjour tout le monde !")
Bonjour tout le monde !
```

Il est aussi possible d'afficher différentes chaînes de caractères sur une même ligne, sans avoir à les ajouter l'une à l'autre. Elles seront séparées par des espaces:

```
>>> print("Bonjour, mon nom est", "Camille")
Bonjour, mon nom est Camille
```

La fonction `print()` peut être utilisée de différentes manières, puisqu'elle peut écrire à peu près n'importe quoi. Pour l'instant, le seul type de valeurs que nous connaissons sont les nombres:

```
>>> print(1)
1
>>> print(1, 2, 3)
1 2 3
>>> print("2 + 2 =", 2 + 2)
2 + 2 = 4
```

Et voilà, nous avons terminé d'utiliser la console interactive de Python pour l'instant. Pour sortir, entrez quit():

```
>>> quit()
```

Ou tapez `Ctrl+D` pour Linux ou `Ctrl+Z` pour Windows.

Fichiers de code source Python

Jusqu'à présent nous avons exécuté du code dans l'invite de commande interactive dans laquelle nous récupérons une réponse immédiate à nos commandes. C'est un bon moyen d'apprendre et d'expérimenter les éléments du langage. C'est pourquoi on y retourne.

Notre premier programme pourrait ressembler à ça:

```
print("Hi, my name is Lucas")
```

Enregistrez ce programme dans un fichier appelé `visitingcard.py`, et lancez-le depuis l'invite de commande :

```
# pyuser@pyrene <~> python visitingcard.py
Hi, my name is Lucas
# pyuser@pyrene <~>
```

Un même programme peut contenir plusieurs commandes, chacune devant être sur une ligne séparée, par exemple:

```
print("Hi,")
print()

print("my name is Lucas")

print()
print("Bye.")
```

Nous pouvons ajouter des lignes vides où nous le souhaitons dans le fichier `visitingcard.py` pour améliorer la lisibilité. Ici, nous avons séparé l'entête du message de son contenu et de sa fin.

Envoyer du Morse grâce à Python

Essayons de créer un programme simple permettant d'envoyer des signaux en Morse.

En cas d'urgence et de panique liée à cet ordinateur, vous pouvez avoir besoin d'envoyer un signal de détresse.

Le signal de détresse, SOS (Save Our Soul) est en effet le signal le plus connu pour communiquer sa situation critique par la lumière ou par le son. Il est utilisé par les bateaux et par les naufragé(e)s en situation de détresse. Nous allons pour notre part commencer à afficher des signaux morse depuis le terminal. En commencant par notre fameux signal de détresse SOS.

Pour écrire un SOS en morse :

S.O.S va se traduire en morse par (... --- ...) soit 3 signaux courts suivis de 3 signaux longs. Le S est donc égal à "..." et le O à "---"

Nous connaissons déjà la fonction `print()` (pour afficher des choses). Nous allons donc créer un "programme" qui stocke les deux lettres et leur équivalent en morse dans des "variables".

Pour commencer, créez un fichier `morse.py` puis écrivez à l'intérieur les lignes suivantes :

```
print("...---...")
```

Lancez votre programme comme ceci:

```
# pyuser@pyrene <~> python3 morse.py
```

Vous obtenez:

```
...---...
```

Comme vous le voyez notre programme a besoin de quelques améliorations :

Si quelqu'un souhaite envoyer un autre message que "SOS", nous devons modifier le fichier `morse.py` .

Programmer c'est l'art de résoudre les problèmes, alors mettons nous au travail ! Cela va nous donner l'occasion d'apprendre de nouvelles fonctionnalités de Python.

Variables

Pour commencer nous aimerions bien rendre notre programme plus lisible, pour permettre au lecteur de savoir immédiatement quel code morse correspond à quelle lettre (... correspond à "S" et --- correspond à "O").

C'est pourquoi nous donnons des noms à ces valeurs:

```
s = "..."
o = "---"
print(s + o + s)
```

Le résultat n'a pas changé:

```
.....---...
```

Remarque : Ici on utilise l'opérateur + qui sert à concaténer (coller) deux chaines de caractères entre elles. Tapez par exemple dans votre console Python:

```
>>> "Bonjour" + " a tous et a toutes"
'Bonjour a tous et a toutes'
```

Pour mieux comprendre le fonctionnement des variables, revenons à l'invite de commande (aussi nommée "console") Python et essayons d'en créer quelques-unes :

```
>>> x = 42
>>> PI = 3.1415
>>> name = "Amelia"
>>> print("Quelques valeurs:", x, PI, name)
Quelques valeurs: 42 3.1415 Amelia
```

En programmation, le signe égal désigne une affectation et non une égalité. On dit "x reçoit 42" ou "pi reçoit 3.1415".

Une variable peut être vue comme une boite portant une étiquette :

- Elle contient quelque chose (on dit que la variable contient une valeur)
- Elle a un nom (comme l'inscription sur l'étiquette de la boite)

Deux variables (ayant des noms différent) peuvent contenir la même valeur :

```
>>> y = x
>>> print(x, y)
42 42
```

Ici les deux variables ont pour noms y et x (se sont les étiquettes sur les boites) et elles contiennent la même valeur : 42

On peut également modifier la valeur d'une variable (changer le contenu de la boite). La nouvelle valeur n'a pas besoin d'être du même type (nombre entier, nombre décimal, texte ...) que la précédente :

```
>>> x = 13
>>> print(x)
13
>>> x = "Scarab"
>>> print(x)
Scarab
```

Les variables sont indépendantes les unes des autres. Si on modifie la valeur de x, la valeur de y reste la même :

```
>>> print(y)
42
```

Nous pouvons également mettre le résultat de calculs ou d'opérations dans des variables et utiliser ensuite ces variables comme alias de la valeur dans d'autres calculs.

```
>>> s = "..."
>>> o = "---"
>>> aidez_moi = s + o + s
>>> print(aidez_moi)
...---...
```

À noter qu'une fois que la valeur est calculée, elle n'est pas modifiée :

```
>>> s = "@"
>>> print(aidez_moi)
...---...
```

Sauf si on demande à Python de la recalculer :

```
>>> aidez_moi = s + o + s
>>> print(aidez_moi)
@---@
```

Il est grand temps d'ajouter quelques commentaires à notre programme afin de faciliter la compréhension pour les lecteurs-trices (dont nous faisons parti).

Les commentaires nous permettent de rajouter du texte dans notre code Python. Les commentaires seront simplement ignorés par l'interpréteur Python lors de l'exécution du code.

En Python, un commentaire est tout ce qui se trouve entre un caractère # et la fin de la ligne:

```
# Code Morse du "S"
s = "..."

# Code Morse du "O"
o = "---"

aidez_moi = s + o + s # Code Morse pour "SOS"
print(aidez_moi)
...---...
```

Les fonctions

Notre programme n'est pas trop mal, mais si l'utilisateur(trice) souhaite pouvoir envoyer plusieurs SOS, ou bien réutiliser ce bout de programme sans dupliquer trop de lignes, il va falloir empaqueter notre fonctionnalité dans ce qu'on appelle : une fonction.

Une fonction, c'est un mini moteur, un groupe d'instructions qui prend des données en entrée, execute les instructions (calcule) en utilisant (ou pas) les données en entrée et renvoie (ou pas) un résultat en sortie.

En Python on définie une fonction comme suit:

```
def nom_de_la_fonction(argument1, argument2):
    """
    Documentation de la foncion : qu'est ce qu'elle fait, à quoi correspondent
    les arguments :

    Args:
        * argument1: bla bla
        * argument2: bla bla
    """
    # les instructions à executer
    # les instructions peuvent utiliser les arguments
    # pour retourner un résultat il faut utiliser le mot clef "return"
    # Si la fonction ne retourne rien, "return" est optionel
    return 42
```

Pour executer cette fonction (on dit "appeller" la fonction):

```
nom_de_la_fonction(argument1, argument2)
```

Pour récupérer la valeur de retour (résultat, sortie) de la fonction dans une variable:

```
ma_variable = nom_de_la_fonction(argument1, argument2)
```

Notre première fonction va se contenter d'imprimer notre signal de détresse.

On crée donc la fonction et on l'appelle à la fin du fichier:

```
def print_sos():
    """ Écrit SOS en Morse """
    s = "..."
    o = "---"
    print(s+o+s)

print_sos()
```

Note : On remarque qu'ici notre fonction ne prend aucun argument et ne renvoie aucune valeur (pas de mot clef `return`). J'ai maintenant une fonction toute simple que je peux appeler à plusieurs reprises juste en duplicant l'appelle `print_sos()`.

On peut aussi vouloir découper le signal et signifier que notre mot est terminé. On va donc ajouter une nouvelle variable "stop" pour découper le mot et ainsi savoir quand le mot est terminé:

```
def print_sos():
    """ Écrit SOS en Morse """
    s = "..."
    o = "---"
    stop = "|"
    print (s+o+s+stop)

print_sos()
```

On peut encore simplifier notre code en remarquant que s contient 3 points et o contient 3 tirets. Il se trouve qu'on peut dupliquer une chaine de caractères en utilisant la syntaxe suivante:

```
>>> "hello"*2
'hellohello'
```

On peut donc obtenir "..." en faisant "." * 3:

```
def print_sos():
    """ Écrit SOS en Morse """
    s = "." * 3
    o = "-" * 3
    stop = "|"
    print(s+o+s+stop)

print_sos()
```

Si maintenant on veut afficher plusieurs SOS, on peut écrire autant de fois que nécessaire print_sos() à la fin du fichier. Mais les informaticien(ne)s sont flemmard(e)s et la machine est là pour nous éviter de refaire la même chose et faire le travail répétitif et ennuyeux.

On a besoin de dire à la machine combien de fois on veut imprimer notre SOS. On va donc modifier la fonction et lui passer le nombre de fois que l'on veut imprimer le signal SOS en argument:

```
def print_sos(nb):
    """ Écrit SOS en Morse nb fois """
    s = "." * 3
    o = "-" * 3
    stop = "|"
    print((s+o+s+stop) * nb)

print_sos(3)
```

Ce qui donne:

```
# pyuser@pyrene <~> python3 morse.py
.........|.........|.........|
```

Pour qu'on puisse mieux lire les différents SOS et visualiser la fin de la phrase on va lui ajouter un retour à la ligne `\n` :

```
def print_sos(nb):
    """ Écrit SOS en Morse nb fois """
    s = "." * 3
    o = "-" * 3
    stop = "|"
    print((s+o+s+stop+"\n") * nb)

print_sos(3)
```

Ce qui donne lorsqu'on execute morse.py:

```
# pyuser@pyrene <~> python3 morse.py
...---...|
...---...|
...---...|
```

On a donc une fonction qui prend en entrée le nombre de fois que l'on veut emettre le signal SOS. Pour l'instant elle ne se contente que d'afficher. Si on veut rendre ce programme encore plus facile à utiliser, imaginons par exemple que nous ayons un robot qui transforme le . et le - en sons différents, ou une machine qui allume et éteigne une lampe plus ou moins longtemps. On peut vouloir que cette fonction retourne la chaine de caractère du message à transmettre sans l'afficher dans le terminal. On pourra ensuite mettre cette chaine dans une variable et la passer en argument à une autre fonction dont le rôle sera d'émettre du son ou d'allumer une lampe. On va donc demander à la fonction de retourner (via `return`) le message en morse et donc on va changer le nom de la fonction de `print_sos` à `emit_sos` :

```
def emit_sos(nb):
    """ Renvoie SOS en Morse nb fois """
    s = "." * 3
    o = "-" * 3
    stop = "|"
    return (s+o+s+stop) * nb

emit_sos(5)
```

Cette fois-ci lorsqu'on execute notre programme, plus rien n'est affiché.

Mais rassurez-vous, on peut toujours vérifier que celà marche en modifiant la dernière ligne en:

```
print(emit_sos(5))
```

En effet, `emit_sos()` retourne une chaine de caractère que `print()` va afficher.

On a donc créé une fonction réutilisable, et que l'on peut greffer à d'autre comme par exemple emettre un son ou encore allumer et éteindre un phare.

Mais avant de gérer les phares et cassez les oreilles des autres, nous allons plutôt interagir avec notre machine à sos et nous familiariser avec elle.

Un peu d'interactivité avec l'utilisateur(trice) serait le bienvenu, par exemple demander à l'utilisateur(trice) de rentrer au clavier le nombre de fois qu'il faut afficher le SOS.

Nous allons utiliser la fonction `input()` (fonctionne seulement avec Python 3, si vous utilisez Python 2 remplacez `input()` par `raw_input()`) pour ça.

La fonction `input()` laisse l'utilisateur(trice) taper un message (terminé par l'appui sur la touche Entrée) puis retourne la chaine de caractère qui a été tapée :

```
>>> input()
Bonjour a toutes et a tous
'Bonjour a toutes et a tous'
```

Le résultat peut bien sûr être stoqué dans une variable afin de l'utiliser par la suite :

```
>>> message = input()
Ceci est un test
>>> print("Vous avez tape : " + message)
Vous avez tape : Ceci est un test
```

Essayons maintenant de l'intégrer à notre machine à SOS:

```
def emit_sos(nb):
    """ Renvoie SOS en Morse nb fois """
    s = "." * 3
    o = "-" * 3
    stop = "|"
    return (s+o+s+stop) * nb

print("Entrez le nombre de SOS que vous voulez: ")
nb_sos = input()
print(emit_sos(nb_sos))
```

Voici ce que le programme donne une fois executé:

```
# pyuser@pyrene <~> python3 morse.py
Entrez le nombre de SOS que vous voulez:
5
Traceback (most recent call last):
  File "morse.py", line 9, in <module>
    print(emit_sos(nb_sos))
  File "morse.py", line 5, in emit_sos
    return (s+o+s+stop) * nb
TypeError: can't multiply sequence by non-int of type 'str'
```

Ceci est une erreur Python (on dit une exception). L'erreur vient du fait que la fonction input() retourne une chaine de caractères et non pas un nombre entier. En Python, "5" est différent de 5, le premier est une chaine de caractères et le deuxième est un entier. La fonction `type()` permet d'afficher le type d'une expression.

```
>>> type("5")
<class 'str'>
>>> type(5)
<class 'int'>
```

Pour convertir notre chaine de caractères en entier, nous allons utiliser la fonction int():

```
>>> a = "5"
>>> a
'5'
>>> int(a)
5
```

Voici le code corrigé:

```
def emit_sos(nb):
    """ Renvoie SOS en Morse nb fois """
    s = "." * 3
    o = "-" * 3
    stop = "|"
    return (s+o+s+stop) * nb

print("Entrez le nombre de SOS que vous voulez: ")
nb_sos = int(input())
print(emit_sos(nb_sos))
```

Une fois lancé :

```
# pyuser@pyrene <~> python3 morse.py
Entrez le nombre de SOS que vous voulez:
5
...---...|...---...|...---...|...---...|...---...|
```

Les conditions

En avant vers notre prochaine problématique. Maintenant on a une machine à émettre des SOS autant de fois qu'on le désire. Mais il faut faire attention, trop de SOS peuvent faire imploser le bateau ou faire sauter l'élecricité ou encore simplement rendre fou le capitaine.

On va donc prévenir le lanceur de SOS s'il dépasse la limite autorisée. On va modifier notre fichier morse.py pour ajouter des précautions d'usage :

```
def emit_sos(nb):
    """ Renvoie SOS en Morse nb fois """
    s = "." * 3
    o = "-" * 3
    stop = "|"
    return (s+o+s+stop) * nb

print("Entrez le nombre de SOS que vous voulez: ")

nb_sos = int(input())
# Et maintenant nous allons vérifier que l'utilisateur n'abuse pas en
# nombre de sos.
# En français on dit :
# si nb_sos est 0
#    pas de SOS pour toi donc
# sinon si nb_sos est plus grand que 10
#    Trop de SOS! Stoppez ça
# sinon
#    emit_sos(nb_sos)
# Et maintenant avec python :
if nb_sos == 0:
    print("Pas SOS pour toi donc.")
elif nb_sos > 10:
    print("Trop de SOS! Stoppez ca s'il vous plait! Vous allez casser la machine!")
else:
    print(emit_sos(nb_sos))
```

Maintenant l'utilisateur a peut être *VRAIMENT* un problème, il faut quand même envoyer un signal. On va donc quand même envoyer le signal mais en respectant la limite :

```
if nb_sos == 0:
    print("Pas SOS pour toi donc.")
elif nb_sos > 10:
    print("Trop de SOS! Stoppez ca s'il vous plait! Vous allez casser la machine!")
    print(emit_sos(10))
else:
    print(emit_sos(nb_sos))
```

L'utilisateur de la machine à SOS, maintenant qu'il est prévenu, peu informer de l'urgence de sa situtation en fonction du nombre de SOS qu'il envoie :

Nombre de S.O.S	Type de Signal	Signification
< 5	Avarie mineure	on rentre au port rapidemment
5 – 12	Avarie moyenne	patrouille de reconnaissance demandée
≥ 12	Avarie majeure	envoi immédiat des forces d'interventions

Exercice : Ecrire une fonction qui va afficher le type de signal en fonction du nombre de SOS envoyé (n'oubliez pas de prendre en compte le cas où il n'y aurait pas de signal).

Conditions : vrai ou faux

Une chose dont nous n'avons pas encore parlé sont les conditions. Pour les nombres, cela fonctionne exactement comme en mathématiques :

```
>>> 2 > 1
True
>>> 1 == 2
False
>>> 1 == 1.0
True
>>> 10 >= 10
True
>>> 13 <= 1 + 3
False
>>> -1 != 0
True
```

Le résultat d'une condition est toujours un booléen : `True` ou `False`.

Remarque : On différencie la condition égalité qui s'écrit deux signes égals de l'affectation d'une valeur à une variable que nous avions vu précédemment.

On peut utiliser les opérateurs `and` et `or` pour construire des conditions plus complexes:

```
>>> x = 5
>>> x < 10
True
>>> 2 * x > x
True
>>> (x < 10) and (2 * x > x)
True
>>> (x != 5) and (x != 4)
False
>>> (x != 5) and (x != 4) or (x == 5)
True
```

Indentation

Une deuxième chose à laquelle il faut faire attention en Python, c'est l'indentation du code.

Ouvrez l'interpreteur Python et entrez une combinaison simple, par exemple:

```
>>> if 2 > 1:
...
```

Pour l'instant rien ne se passe, comme le montrent les points `...` à la place des habituels chevrons `>>>`. Python s'attend à ce que nous donnions des instructions complémentaires qui devront être exécutées si la condition `2 > 1` s'avère vraie. Essayons d'afficher "OK" :

```
>>> if 2 > 1:
... print("OK")
  File "<stdin>", line 2
    print("OK")
        ^
IndentationError: expected an indented block
```

Apparemment, ça n'a pas très bien fonctionné. En fait Python doit savoir si l'instruction que nous avons entrée est une instruction à exécuter uniquement si la condition est vraie ou si c'est une instruction à exécuter sans qu'elle ne soit affectée par la condition.

C'est pourquoi nous devons indenter notre code:

```
>>> if 2 > 1:
...     print("OK")
...
OK
```

Tout ce que vous devez faire c'est ajouter un espace ou une tabulation avant votre instruction pour dire qu'elle fait partie des instructions dépendantes de l'instruction `if`. Attention, toutes les lignes à exécuter qui dépendent du `if` doivent être indentées de la même manière :

```
>>> if -1 < 0:
...  print("A")
...   print("B")
  File "<stdin>", line 3
    print("B")
    ^
IndentationError: unexpected indent

>>> if -1 < 0:
...     print("A")
...   print("B")
  File "<stdin>", line 3
    print("B")
            ^
IndentationError: unindent does not match any outer indentation level

>>> if -1 < 0:
...   print("A")
...   print("B")
...
A
B
```

Pour éviter la confusion, la plupart des développeurs Python se sont mis d'accord pour toujours utiliser quatre espaces pour chaque niveau d'indentation. Nous allons nous aussi adopter cette convention:

```
>>> if 2 > 1:
...     if 3 > 2:
...         print("OK")
...     else:
...         print("ECHEC")
...     print("FAIT")
OK
FAIT
```

Et si ce n'est pas le cas ?

On pourrait se débrouiller pour écrire un programme en utilisant uniquement des `if`.

```
if nb_sos < 5:
    print("Avarie Mineure")
if nb_sos >= 5:
    if nb_sos < 12:
        print("Avarie Moyenne")
if nb_sos >= 12:
    print("Avarie Majeure")
```

Mais en fait, on peut aussi utiliser `else` et `elif`, afin de ne pas avoir à répéter les conditions similaires et améliorer la lisibilité du code. Dans des programmes plus compliqués, il n'est parfois pas évident de reconnaître que la condition lue est la condition inverse de la précédente.

En utilisant `else`, nous avons la garantie que les instructions données seront exécutées seulement si les instructions données après le `if` n'ont pas été exécutées:

```
if nb_sos < 5:
    print("Avarie Mineure")
else:
    # Si votre programme exécute ces instructions alors vous êtes
    # certains que nb_sos >= 5 !
    if nb_sos < 12:
        print("Avarie Moyenne")
    else:
        # Ici vous pouvez être certains que nb_sos >= 12
        # nous n'avons donc pas à le vérifier.
        print("Avarie Majeure")
```

Regardez bien attentivement la manière dont le code est indenté. À chaque utilisation de else, un niveau d'indentation a été ajouté à chaque niveau du code. C'est très ennuyeux d'avoir à lire du code avec de nombreux niveaux d'indentation.

C'est pourquoi les développeurs Python on ajouté un troisième mot clé, `elif`, qui permet de vérifier directement une autre condition.

```
if n < 1:
    # Si n est inférieur à un.
    print("inferieur à un")
elif n < 2:
    # Si est supérieur ou égal à un, et inférieur à deux.
    print("entre un (compris) et deux")
elif n < 3:
    # Si n est supérieur ou égal à un,
    # que n est supérieur ou égal à deux,
    # et que n est inférieur à 3
    print("entre deux et trois")
else:
    # Si aucune des conditions précédentes n'est vérifiée
    print("supérieur ou égal à trois")
```

En résumé

Dans ce chapitre nous avons appris les bases de la syntaxe Python. Nous avons découvert comment afficher des nombres entiers et décimaux, des chaînes de caractères et nous avons découvert les tuples.

Nous avons appris à utiliser la fonction `print()`, qui affiche des informations à l'utilisateur et la fonction `input()`, qui permet de lire les entrées de ce dernier.

Nous avons vu comment l'identation pouvait être importante, notamment lors de l'utilisation des instructions `if`, `else` et `elif`.

Nous avons réalisé notre premier programme dans un fichier que nous pouvons lancer.

Notre programme pose quelques questions à l'utilisateur, calcule des informations et présente les résultats dans une forme utile à l'utilisateur.

Ça fait finalement beaucoup de choses pour un premier programme. Nous avons encore pas mal de travail mais vous pouvez être fier de ce que vous avez fait jusqu'à présent !

Un premier exercice

Utilisation de la calculatrice python.

- Au moment du départ, les divers cadrans d’une auto marquent respectivement :
  - 7h50
  - 14 739 km
  - 20 litres.
- En cours de route, l'automobiliste s'arrête pendant 30 minutes pour prendre 35 litres d'essence et se reposer.
- A l'arrivée, les compteurs indiquent :
  - 13h20
  - 15 124 km
  - 10 litres.

Questions :

1. Quelle distance l’auto a-t-elle parcourue ?
2. Durant combien de temps a-t-elle roulé ?
3. Quelle a été sa vitesse moyenne (arrêt non compris) ?
4. Quelle est la consomation pour ce trajet ?

Correction :

ATTENTION : Vous êtes sûr d'avoir suffisament essayé avant de voir la correction ?

Bien sur c'est une possibilité. Dans ce genre de problème il est particulièrement important de choisir des noms de variables adaptés.

```
>>> heure_depart = 7.0 + 50.0/60.0
>>> heure_arrivee = 13.0 + 20.0/60.0
>>> pause_temps = 30.0/60.0
>>> km_depart = 14739
>>> km_arrivee = 15124
>>> essence_depart = 20
>>> essence_arrivee = 10
>>> plein = 35
>>> distance = km_arrivee - km_depart
>>> print(distance)
385
>>> conso = essence_depart + plein - essence_arrivee >>> print(conso)
45
>>> duree = heure_arrivee - heure_depart - pause_temps >>> print(duree)
5.5
>>> vitesse = distance / duree
>>> print(vitesse)
70.0
```

Les tuples

Nous savons déjà que nous pouvons concaténer des chaînes de caractères, les multiplier par des nombres, vous allez voir qu'on peut aussi les formater. Tout d'abord, nous avons besoin de découvrir un nouveau type de données (en plus des strings et des nombres, `int` et `float` , que nous connaissons déjà).

Rappelez-vous, je vous disais que nous ne pouvions pas utiliser les virgules dans les nombres car nous en aurions besoin par la suite pour définir les tuples. Nous y voici.

```
>>> 1, 2, 3
(1, 2, 3)
>>> ("Ala", 15)
('Ala', 15)
>>> x = 1,5
>>> print(x)
(1, 5)
```

Un tuple n'est ni plus ni moins qu'une valeur contenant un groupe de valeurs. Les valeurs que nous souhaitons grouper doivent être séparées par des virgules. L'ensemble peut-être entouré de parenthèses pour rendre plus explicite le fait qu'il s'agisse bien d'un groupe, mais ce n'est pas obligatoire. Sauf pour le cas d'un groupe vide (aussi bizarre que cela puisse paraître).

```
>>> ()
()
```

Il est possible de combiner des tuples:

```
>>> names = ("Pauline", "Dupontel")
>>> details = (27, 1.70)
>>> names + details
('Pauline', 'Dupontel', 27, 1.7)
```

Un tuple peut aussi contenir un autre tuple, par exemple un point sur une carte peut-être groupé comme ceci:

```
>>> point = ("Pizzeria", (longitude, latitude))
```

Avec `longitude` et `latitude` des coordonnées géographiques.

On peut ensuite se référer aux valeurs d'un groupe en utilisant leurs positions (en commençant à zéro):

```
>>> p = (10, 15)
>>> p[0]  # première valeur
10
>>> p[1]  # deuxième valeur
15
```

Les tuples sont des listes dites immuables, qu'on ne peut modifier. On ne peut pas modifier le nombre d'éléments qu'elle contient après sa création. Nous verrons plus tard un autre type de listes qui peut être modifié à loisir.

Première série d'exercices

Savoir Lire, écrire et calculer

1. Écrire un programme qui demande à l'utilisateur la longueur et la largeur d'un rectangle et calcule l'aire.
2. Écrire un programme qui calcule le reste de la division de deux entiers demandés à l'utilisateur.
3. Écrire un programme qui calcule le périmètre d'un cercle après avoir demandé son rayon.

Faire une condition

1. Écrire un programme qui demande deux chiffres à l'utilisateur et affiche le plus grand.
2. Écrire un programme qui calcule la racine carré d'un nombre demandé à l'utilisateur.

La fonction racine carré est accessible dans le module `math`. Nous verrons plus tard comment fonctionne les modules python. Pour importer la fonction racine carré ajouter la ligne :

```
from math import sqrt
```

1. Écrire un programme qui calcule les racines d'un polynôme du second degré.

On rappelle que les solutions de $ax^2 + bx + c = 0$ sont :

$$\Delta = b^2 - 4ac$$

$$\begin{cases} \Delta > 0 & x = \dfrac{-b \pm \sqrt{\Delta}}{2a} & 2 \text{ solutions réelles} \\ \Delta = 0 & x = \dfrac{-b}{2a} & \text{une unique solution} \\ \Delta < 0 & x = \dfrac{-b \pm i\sqrt{|\Delta|}}{2a} & 2 \text{ solutions complexes} \end{cases}$$

Structure générale d'un programme python

Un fichier contenant un programme python pourrait être organisé de la façon suivante :

```
#!/usr/bin/env python
# -*- coding=utf-8 -*-

""" Présentation du code """

def fonction(argument1, argument2):
    """ description de la fonction """

    # instructions de la fonction

    # la fonction retourne éventuellement une valeur
    return "une valeur"

if __name__ == "__main__":
    # instructions à exécuter, par exemple
    #    * lecture des parametres
    #    * appel de la fonction
    fonction()
```

- La première ligne indique que le fichier doit être interprété avec python.
- La deuxième ligne indique l'encodage du fichier.
- Les instructions qui suivent `if __name__ == "__main__":` sont interprétées lors de l'exécution du programme.

Corrections

ATTENTION : Vous êtes sûr d'avoir suffisament essayé avant de voir la correction ?

Savoir Lire, écrire et calculer

1. Écrire un programme qui demande à l'utilisateur la longueur et la largeur d'un rectangle et calcule l'aire.

```
# lecture des variables
largeur = float(input("largeur = "))
print("largeur = ", largeur)

longueur = float(input("largeur = "))
print("longueur = ", longueur)

# arriche le résultats
print("Aire du rectangle = ", largeur * longueur)
```

On peut écrire une fonction qui attend la longueur et la largeur comme arguments et retourne l'aire du rectangle.

```
def aire_rectangle(largeur, longueur):
    """ Calcul de l'aire d'un rectangle """
    return largeur * longueur
```

Elle s'utilise de la façon suivante :

```
# lecture des variables
largeur = float(input("largeur = "))
print("largeur = ", largeur)

longueur = float(input("largeur = "))
print("longueur = ", longueur)

# arriche le résultats
print("Aire du rectangle = ", aire_rectangle(largeur, longueur))
```

2. Écrire un programme qui calcule le reste de la division de deux entiers demandés à l'utilisateur.

```
# lecture des valeurs
dividende = int(input("entrer le dividende : "))
print("dividende = ", dividende)

diviseur = int(input("entre le diviseur : "))
print("diviseur = ", diviseur)

reste = dividende % diviseur
print("reste = ", reste)
```

On peut écrire une fonction qui attend le dividende et le diviseur comme arguments et retourne le reste de la division entière.

```
def reste(dividende, diviseur):
    """ Calcul du reste de la division de deux entiers """
    reste = dividende % diviseur
    print("reste = ", reste)
```

3. Écrire un programme qui calcule le périmètre d'un cercle après avoir demandé son rayon.

```
from math import pi

# lecture du rayon
rayon = float(input("entrer le rayon : "))
print("rayon = ", rayon)

# affichage du résultat
print("périmètre = ", 2 * pi * rayon)
```

On peut écrire une fonction qui attend comme argument le rayon et retourne le périmètre du cercle.

```
def perimetre(rayon):
    """ Calcul du périmètre d'un cercle """
    return 2 * pi * rayon
```

Faire une condition

1. Écrire un programme qui demande deux chiffres à l'utilisateur et affiche le plus grand.

```
# lecture de x et y
x = float(input("entrer x : "))
print("x = ", x)

y = float(input("entrer y : "))
print("y = ", y)

# test entre x et y
if x > y:
    print("x est plus grand")
    print("le plus grand = ", x)
elif y > x:
    print("y est plus grand")
    print("le plus grand = ", y)
else:
    print("x et y sont égaux")
    print("x = ", x, "\t y = ", y)
```

2. Écrire un programme qui calcule la racine carré d'un nombre demandé à l'utilisateur.

```python
from math import sqrt

def racine(x):
    """ Calcul de la racine carré de x si x est positif. """

    if x >= 0:
        print("RACINE(x) = ", sqrt(x))
    else:
        print("x est négatif")

if __name__ == "__main__":
    # lecture de x
    x = float(input("entrer x : "))
    print("x = ", x)

    racine(x)
```

3. Écrire un programme qui calcule les racines d'un polynôme du second degré.

```python
from math import sqrt

def racine_trinome(a, b, c):
    """ Calcul des racines réelles d'un polynome de degré 2 """

    # Calcul du discriminent
    delta = b**2 - 4. * a * c
    print("delta = ", delta)

    # Test du discriminent
    if delta > 0:
        print("L'équation a deux solutions")
        print("x1 = ", (-b - sqrt(delta)) / (2. * a))
        print("x2 = ", (-b + sqrt(delta)) / (2. * a))
    elif delta < 0.:
        print("L'équation a deux solutions complexes")
    else:
        print("L'équation a une seule solution")
        print("x  = ", -b / (2.0 * a))

if __name__ == "__main__":
    print("On va résoudre l'équation a*x^2 + b*x + c = 0")

    # lecture des variables
    a = float(input("entrer a : "))
    print("a = ", a)

    b = float(input("entrer b : "))
    print("b = ", b)

    c = float(input("entrer c : "))
    print("c = ", c)

    racine_trinome(a, b, c)
```

L'arbre de noël

Noël arrive (dans plus ou moins longtemps c'est vrai), ce sera le temps des c adeaux et des sapins de Noël dans tous les magasins. :)

Comme exercice, je vous propose de dessiner un Arbre de Noël dans la console.

Nous allons commencer par la version la plus simple puis ajouter des fonctionnalités au fur et à mesure.

Pour démarrer, commençons par dessiner la moitié d'un Arbre de Noël :

```
print("*")
print("**")
print("***")
print("*")
print("**")
print("***")
print("****")
print("*****")
print("*")
print("**")
print("***")
print("****")
print("*****")
print("******")
print("*******")
```

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

C'est pas si mal, mais nous avons du taper beaucoup de choses. Et que se passe-t-il si je veux un arbre plus petit ? Ou un plus grand, composé de centaines d'étoiles pour l'imprimer sur un poster géant au format A0 ? Oui ça fait certainement beaucoup trop de caractères à

taper, quand bien même on multiplierait les caractères par centaines ("″ 100, et ainsi de suite). Ça ressemble au genre de tâche qu'on confierait volontiers à un programme ça, non ?

Les listes et les boucles `for`

Les boucles sont faites exactement pour ce genre d'actions répétitives. Pour rester dans l'atmosphère de Noël, imaginez un instant que vous êtes le Père Noël et que vous devez distribuer tous les cadeaux.

Comme vous le savez, les lutins ont une liste précise des enfants sages qui méritent un cadeau. La solution la plus simple pour garantir qu'un enfant ne soit pas oublié serait de prendre la liste et d'aller distribuer les cadeaux, dans l'ordre.

Outre les aspects physiques de la tâche, la procédure de distribution des cadeaux pourrait ressembler à cela:

Disons que la liste des enfants sages, contient la liste des enfants qui méritent un cadeau.

```
Pour chaque enfant (alias `child`), qui se trouve dans la liste des enfants sages:
    Distribuer un cadeau à cet enfant
```

La disposition du texte ci-dessus n'est pas une erreur, c'est en fait un programme Python déguisé:

```
children = children_who_deserve_gifts()

for child in children:
    deliver_gift(child)
    print("Cadeau distribué à :", child)
print("Tous les enfants sages ont reçus un cadeau")
```

La plupart des choses doivent vous sembler familières. On appelle deux fonctions :

- `children_who_deserve_gifts()` et `deliver_gift()`, leur fonctionnement interne est uniquement connu du Père Noël.
- Le résultats de la première peut être enregistré dans la variable `children`, afin de se rappeler par la suite à quoi corresponds cette valeur.

Le nouvel élément, c'est la boucle elle-même, qui consiste en :

- Sur la première ligne
  - Le mot clé `for`,
  - Le nom du prochain élément de la liste,

    - Le mot clé in,
    - Une liste de valeur ou une variable qui y fait référence.
- Les instructions indentées à effectuer pour chaque valeur de la liste (comme dans le cas de `if`).

Attendez, nous n'avons encore rien dit à propos des listes, mais rassurez-vous, le concept de liste en Python est très proche du concept de liste dans la vie de tous les jours. Nous pouvons simplement nous représenter une liste en Python comme nous nous représentons n'importe quelle autre liste le reste du temps (liste de courses, liste d'invités, résultats d'examens, etc...) écrite sur un papier et numérotée.

Commençons par une liste vide :

```
>>> L = []
>>> L
[]
```

Quand nous le souhaitons, nous pouvons demander le nombre d'éléments qui se trouvent dans notre liste en utilisant la fonction `len()`.

```
>>> len(L)
0
```

Essayons avec une autre liste (qui peut avoir le même nom ou pas) :

```
>>> L = ["Yara", "Pierre", "Amel"]
>>> len(L)
3
```

Comme pour le cas des tuples, les éléments consécutifs d'une liste sont séparés par des virgules. À la différence des tuples, les crochets sont obligatoires.

Pour récupérer la valeur d'un élément d'une position particulière de la liste on se sert de sa position dans la liste (en se souvenant que les index des positions commencent à 0) :

```
>>> L[0]
'Yara'
>>> L[1]
'Pierre'
>>> L[2]
'Amel'
>>> L[3]
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Pour `L[3]` on obtient une exception de type `IndexError` qui nous indique que l'élément d'indice 3 n'existe pas.

En python, les listes sont dites *dynamiques*. On peut ajouter ou supprimer des éléments.

```
>>> L = ["Yara", "Pierre", "Amel"]
>>> L.append("Rémi")
>>> print(L)
['Yara', 'Pierre', 'Amel', 'Rémi']
>>> L.pop()
'Rémi'
>>> print(L)
['Yara', 'Pierre', 'Amel']
>>> L.remove("Pierre")
>>> print(L)
['Yara', 'Amel']
```

`append()`, `pop()`, et `remove()` sont des fonctions qui agissent sur la liste. On les appelles des méthodes, mais nous verrons ça plus précisément au sujet des objets et des classes.

La boucle `for` que nous avons vu tout à l'heure, va nous servir pour exécuter une instruction sur chaque élément de la liste. Elle permet de parcourir la liste.

```
>>> for name in L:
...     print("Nom :", name)
...
Nom : Yara
Nom : Pierre
Nom : Amel
```

En passant, nous pouvons ainsi afficher la première moitié de notre Arbre de Noël :

```
>>> lst = [1, 2, 3]
>>> for n in lst:
...     print("*" * n)
...
*
**
***
```

Malheureusement, nous devons encore écrire le contenu de la liste. Ce problème peut-être résolu à l'aide de la fonction `range()`. Vous pouvez entrer `help(range)` pour apprendre à l'utiliser (touche `q` pour sortir) ou regardez ces exemples :

```
>>> list(range(2, 5, 1))
[2, 3, 4]
>>> list(range(1, 11, 2))
[1, 3, 5, 7, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(1, 2))
[1]
>>> list(range(2))
[0, 1]
```

La fonction `range()` ne crée pas directement une liste, mais retourne un générateur. Les générateurs génèrent les éléments un à un, ce qui permet de ne pas avoir à stocker l'ensemble des valeurs de la liste dans la mémoire de l'ordinateur.

Pour obtenir une liste à partir d'un générateur, on utilise la fonction `list()` (en fait il s'agit d'une classe mais nous verrons ça plus tard). Si on oublie l'appel à `list()`, le résultat ressemblera à ça :

```
>>> range(1, 4)
range(1, 4)
```

La fonction `range()` a trois formes. La plus simple, qui est la plus utilisée, permet de générer une séquence de nombres de 0 à un nombre donné. Les autres formes vous permettent de spécifier le chiffre de départ et le pas d'un nombre à l'autre de la séquence. La séquence créée n'inclut jamais la borne supérieure.

Affichons un Arbre de Noël plus grand :

```
>>> lst = list(range(1, 11))
>>> lst
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> for i in lst:
...     print("*" * i)
*
**
***
****
*****
******
*******
********
*********
**********
```

`range()` nous a épargné beaucoup de temps, on peut en gagner encore plus si on ne nomme pas la liste:

```
>>> for i in list(range(1, 5)):
...     print(i * "@")
@
@@
@@@
@@@@
```

Lorsqu'on utilise l'instruction `for`, on n'a pas besoin d'utiliser la fonction list(). `for` sait gérer le générateur retourné par `range()`. Ce qui nous permet de simplifier notre programme encore plus.

```
>>> for i in range(1, 5):
...     print(i * "@")
@
@@
@@@
@@@@
```

Rien ne nous empêche de créer une boucle dans une autre boucle, essayons ! Simplement rappelez-vous d'utiliser l'indentation appropriée et d'utiliser des noms différents, par exemple i et j, (ou mieux un nom en rapport avec le contenu de la liste) pour les éléments de chaque liste :

```
>>> for column in range(1, 3):
...     for line in range(11, 14):
...         print(column, line)
1 11
1 12
1 13
2 11
2 12
2 13
```

Nous avons une boucle intérieure allant de 11 à 13 (n'oubliez pas que, 14 n'est pas incluse lorsqu'on utilise `range()`), inclue dans une boucle extérieure qui elle va de 1 à 2 (idem, 3 n'est pas inclue).

Comme vous pouvez le voir les éléments de la boucle intérieure sont affichés deux fois, une fois pour chaque itération de la boucle extérieure.

En utilisant cette technique, on peut répéter les éléments de notre Arbre de Noël :

```
>>> for etages in range(3): # répéter 3 fois
...     for taille in range(1, 4):
...         print(taille * "*")
*
**
***
*
**
***
*
**
***
```

Avant d'aller plus loin, créez le fichier `noel.py` avec ce programme et essayez de le modifier afin que pour chaque itération de la boucle extérieure la boucle intérieure soit exécutée une fois de plus. (Que pour chaque étage on ait une branche de plus).

Vous devriez obtenir le résultat de notre demi Arbre de Noël décrit en début de chapitre. Par exemple :

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

Correction :

La principale modification concerne la boucle intérieure dont la borne supérieur doit changer à chaque nouvelle itération de la boucle extérieure.

On notera l'opérateur d'incrémentation `+=` qui permet d'ajouter une valeur à la valeur déjà contenu dans une variable :

```
>>> i = 1
>>> print(i)
1
>>> i += 1
>>> print(i)
2
>>> i += 1
>>> print(i)
3
```

Solution :

```
taille_max = 2
for etages in range(5):
    taille_max += 1
    for taille in range(1, taille_max):
        print(taille * "*")
```

Les fonctions

Nous avons déjà pu voir comment les fonctions résolvent nombre de nos problèmes. Par contre elle ne les résolvent pas tous, ou du moins pas exactement de la manière dont nous aimerions les résoudre.

Parfois, et même assez souvent nous devons résoudre nous-mêmes un problème. Ce serait donc assez cool de pouvoir créer des fonctions qui le fassent pour nous.

Voici comment nous pouvons faire en Python:

```
>>> def print_triangle(n):
...     """ Affiche un triangle avec n lignes """
...     for size in range(1, n + 1):
...         print(size * "*")
...
>>> print_triangle(3)
*
**
***
>>> print_triangle(5)
*
**
***
****
*****
```

Regardons de plus près la fonction print_triangle():

```
def print_triangle(n):
    """ Affiche un triangle avec n lignes """
    for size in range(1, n + 1):
        print(size * "*")
```

La définition d'une fonction commence toujours avec l'instruction `def`. Ensuite on donne un nom à la fonction. Entre les parenthèses, on indique quels sont les noms des arguments passés à la fonction lorsqu'elle est appelée. Les lignes suivantes définissent les instructions à exécuter lors de l'utilisation de la fonction.

Comme vu dans l'exemple, les instructions peuvent utiliser les variables passées en arguments. Le principe opératoire est le suivant, si on créé une fonction avec trois arguments :

```
>>> def foo(a, b, c):
...     print("FOO", a, b, c)
```

Lorsque vous appelez cette nouvelle fonction, vous devez spécifier une valeur pour chacun des arguments. De la même manière que ce que nous faisions pour appeler les fonctions précédentes :

```
>>> foo(1, "Ala", 2 + 3 + 4)
FOO 1 Ala 9
>>> x = 42
>>> foo(x, x + 1, x + 2)
FOO 42 43 44
```

On notera qu'un argument est simplement un alias, si on modifie la valeur liée à cet alias pour une autre valeur, les variables initiales ne sont pas modifiés, c'est la même chose pour les arguments :

```
>>> def plus_five(n):
...     """ ajoute 5 à n """
...     n = n + 5
...     print(n)
>>> x = 43
>>> plus_five(x)
48
>>> x
43
```

ça fonctionne comme pour les variables que nous avons vu précédement. Il y a seulement deux différences :

1. Premièrement, les variables correspondants aux arguments d'une fonction sont définies à chaque appel de la fonction. Python attache la valeur de la variable passée comme argument à la variable qu'il vient de créer spécifiquement pour l'appel de cette fonction.
2. Deuxièmement, les variables correspondants aux arguments ne sont pas utilisable à l'extérieur de la fonction car ils sont créé lors de l'appel de la fonction et oublié à la fin de celle-ci. C'est pourquoi, si vous essayez d'accéder à la valeur de n que nous avons définie dans notre fonction `plus_five()`, à l'extérieur du code de la fonction Python vous dit qu'elle n'est pas définie :

```
>>> n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

C'est comme ça notre cher Python fait le ménage à la fin d'un appel de fonction. On parle alors de *portée* d'une variable ou de variable *interne* à la fonction.

Mais rappelez-vous, nous avons vu une instruction `return` qui nous permet de récupérer ce qui s'est passé dans la fonction. C'est une instruction spécifique qui ne fonctionne qu'au sein d'une fonction.

Pour finir, comme dernier exemple de fonction, voici la solution au problème posé à la fin du chapitre précédent en utilisant une fonction :

```
def print_triangle(n):
    """ Affiche un triangle avec n lignes """
    for size in range(1, n + 1):
        print(size * "*")

for i in range(2, 5):
    print_triangle(i)
```

Ce qui donne à l'exécution :

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

Un Arbre de Noël entier

Le chapitre précédent était principalement de la théorie. Utilisons nos nouvelles connaissances pour terminer notre programme et afficher notre Arbre de Noël.

Voici à quoi ressemble notre programme actuel:

```
def print_triangle(n):
    """ Affiche un triangle avec n lignes """
    for size in range(1, n+1):
        print(size * "*")

for i in range(2, 5):
    print_triangle(i)
```

Comment pouvons-nous améliorer la fonction `print_triangle()`, pour afficher un Arbre de Noël entier et non juste la moitié ?

Tout d'abord, essayons de déterminer le résultat attendu en fonction de la valeur de l'argument n. Il parait naturel que n soit la largeur. Ainsi pour n = 5 on s'attendrait à:

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

Il est intéressant de noter que chaque ligne possède deux étoiles de plus que la ligne précédente. Nous pouvons donc utiliser le troisième argument de `range()` :

```
def print_segment(n):
    """ affiche un segment de l'arbre avec maximum n étoiles """
    for size in range(1, n + 1, 2):
        print(size * "*")

print_segment(5)
```

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

Ce n'est pas exactement ce à quoi on s'attendait, il y a effectivement le bon nombre d'étoiles mais on souhaiterait qu'elles soient alignées au centre.

La fonction `unicode.center()` peut nous aider. Elle s'applique à une chaine de caractères. Voyons comment elle fonctionne :

```
>>> "Bonjour".center(10)
' Bonjour  '
>>> "**".center(4)
' ** '
```

Cette fois la chaine de caractères est centrée ! Modifions donc notre fonction :

```
def print_segment(n):
    """ affiche un segment de l'arbre avec maximum n étoiles """
    for size in range(1, n + 1, 2):
        print((size * "*").center(n))

print_segment(5)
```

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

Cependant, un nouveau problème apparait :

```
def print_segment(n):
    for size in range(1, n + 1, 2):
        print((size * "*").center(n))

for i in range(3, 8, 2):
    print_segment(i)
```

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

Si nous avions un moyen de connaitre à l'avance la taille du segment le plus grand, nous pourrions ajouter un argument supplémentaire à print_segment(), pour faire le centrage sur cette largeur. Voici une solution en combinant toute les connaissances acquises :

```
def print_segment(segment_size, total_width):
    """
    affiche un segment sur une largeur de total_width avec
    au plus segment_size étoiles

    args :
        segment_size: maximum d'étoiles
        total_width: largeur du segment
    """
    for line_size in range(1, segment_size + 1, 2):
        print((line_size * "*").center(total_width))

def print_tree(size):
    """ affiche un arbre de noel de taille size """
    for segment_size in range(3, size + 1, 2):
        print_segment(segment_size, size)

print("Choisissez la taille de votre Arbre de Noël :")
tree_size = int(input())
print_tree(tree_size)
```

```
Choisissez la taille de votre Arbre de Noël :
7
   *
  ***
   *
  ***
 *****
   *
  ***
 *****
*******
```

La boucle `while`

Nous avons vu la boucle `for` permettant de répéter une opération un certains nombre de fois en parcourant une liste. Il existe une autre façon de répéter une action en python, il s'agit de la boucle `while`. En français, on dirait :

```
Tant que ma condition est vrai,
    opérations à effectuer
```

Ce qui se traduit en python par :

```
while condition:
    do_something()
```

Par exemple, nous allons demander 4 nombres entre 1 et 100 à l'utilisateur que nous enregistrerons dans une liste :

```
# compteur des nombres
n = 0
# listes pour enregistrer les valeurs
liste = list()

print("Donner moi 4 nombres entre 1 et 100:")
while n < 4:
    valeur = int(input("nombre : "))
    liste.append(valeur)
    n += 1
print("liste : ", liste)
```

Résultat :

```
Donner moi 4 nombres entre 1 et 100:
nombre : 10
nombre : 88
nombre : 17
nombre : 64
liste :  [10, 88, 17, 64]
```

Deux éléments manquent cruellement à ce petit programme :

- On n'a pas vérifié si les nombres étaient compris entre 1 et 100
- La taille de la liste (4) est imposée

Compte tenu de vos connaissances actuelles, vous pouvez modifier ce programme pour qu'il refuse d'enregistrer un nombre s'il n'est pas compris entre 1 et 100. Vous pourriez aussi écrire une fonction qui permet de choisir la taille de la liste.

En résumé

Vous savez maintenant créer une liste.

Vous savez utiliser une boucle `for` pour parcourir cette liste.

Nous avons appris à utiliser la fonction `range()`, qui retourne un générateur que l'on peut parcourir avec une boucle `for`. Nous avons vu également la fonction `len()` qui donne le nombre d'éléments d'une liste.

Nous avons manipulé encore une fois les fonctions qui permettent d'organiser le code et le rendre fonctionnel.

Nous avons vu les boucles `while`.

Vous savez presque tout ce qu'il faut pour programmer en python. Il reste une dernière chose à voir, la notion d'objet.

Deuxième série d'exercices

Répéter une action

Les deux exercices suivants consistent à calculer les valeurs d'une fonction de votre choix dans l'intervalle $x \in [-5; 5]$ avec

1. un pas de 1
2. un pas de 0.5

Vous tacherez de définir une fonction pour votre fonction mathématique et afficherez les valeurs de x et les valeurs de la fonction.

Suite de fibonacci

Calculer les termes de la suite de Fibonacci définis de la façon suivante :

$$\begin{cases} U_n &= U_{n-1} + U_{n-2} \\ U_0 &= 0 \\ U_1 &= 1 \end{cases}$$

Manipuler une liste

Ces deux exercices ont pour objectif d'introduire la manipulation des listes.

1. Écrire un programme qui crée une liste contenant les premiers entiers impairs et affiche cette liste.
2. Écrire un programme qui crée une liste contenant une série de nombres pseudo-aléatoires compris entre -1 et 1 et affiche cette liste. On utilisera pour cela la fonction `random` du module `random`.
3. Amusons nous un peu.

   Le bouclier déflecteur de l'étoile de la mort (Star Wars, épisode VI) est composé de plusieurs couches de plasma thermo-magnétique. En traversant la couche n, l'intensité du laser d'un *X-Wing Starfighter* diminue d'un facteur $\alpha(n)$ dont l'expression est :

   $$\alpha(n) = 0.83 \exp\left(-0.04982(n-1)\right)$$

- Combien de couches le laser aura-t-il traversé s'il a perdu les trois quart de son intensité ?
- Sachant que l'intensité initiale du laser est $I_o = 112358W$, combien de couches doit contenir le bouclier déflecteur pour que l'intensité du laser soit inférieure à 10 W lorsque le tir atteint sa cible.

Corrections

ATTENTION : Vous êtes sûr d'avoir suffisament essayé avant de voir la correction ?

Valeurs d'une fonction

On se donne tout d'abord une fonction, par exemple la fonction suivante sur l'intervalle $x \in [-1;3[$

$$f(x) = \frac{(4x^3-6x^2+1)\sqrt{x+1}}{3-x}$$

Écrivons cette fonction en prennant garde de rester dans le bon intervalle. Il ne faut pas oublier d'importer la fonction `sqrt` du module `math`.

```
from math import sqrt
def f(x):
    """ ma fonction """
    if -1 <= x < 3:
        return (4 * x**3 - 6 * x**2 + 1) * sqrt(x + 1)/ (3 - x)
    else:
        print("WARNING : x n'est pas dans l'intervalle de définition de f")
        return None
```

On va maintenant calculer les valeurs de la fonction par pas de 1 :

```
for x in range(6):
    print(x, f(x))
```

Ce qui donne

```
0 0.3333333333333333
1 -0.7071067811865476
2 15.588457268119894
WARNING : x n'est pas dans l'intervalle de définition de f
3 None
WARNING : x n'est pas dans l'intervalle de définition de f
4 None
WARNING : x n'est pas dans l'intervalle de définition de f
5 None
```

Il serait préférable de faire

```
for x in range(-1, 3):
    print(x, f(x))
```

```
-1 -0.0
0 0.3333333333333333
1 -0.7071067811865476
2 15.588457268119894
```

Passons maintenant à un pas de 0.5. Nous allons utiliser une boucle `while` :

```
# initialisation
x = -1
pas = 0.5
# boucle
while x < 3:
    print(x, f(x))
    x += pas
```

Ce qui donne :

```
-1 -0.0
-0.5 -0.20203050891044216
0.0 0.3333333333333333
0.5 0.0
1.0 -0.7071067811865476
1.5 1.0540925533894598
2.0 15.588457268119894
2.5 97.28309205612247
```

Suite de Fibonacci

Les termes de la suite de Fibonacci sont définis par :

$$\begin{cases} U_n & = U_{n-1} + U_{n-2} \\ U_0 & = 0 \\ U_1 & = 1 \end{cases}$$

On peut calculer les termes de cette liste avec une boucle `while` ou une boucle `for`.

```
def fibonacci(n):
    """ Calcule les n premiers éléments de la suite de fibonacci """

    # initialisation
    u = [0, 1]

    # calcul des valeurs avec une boucle while
    i = 2
    while i < n:
        u.append(u[i-1] + u[i-2])
        i += 1

    return u

print(fibonacci(10))
```

Résultat :

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

On peut faire la même chose avec une boucle `for` :

```
def fibonacci(n):
    """ Calcule les n premiers éléments de la suite de fibonacci """

    # initialisation
    u = [0, 1]

    # calcul des valeurs
    for i in range(2, n):
        u.append(u[i-1] + u[i-2])

    return u
```

Utilisation des listes

Liste contenant les premiers entiers impairs et affiche cette liste.

En fait on peut obtenir très simplement cette liste à partir de la fonction `range()`.

```
>>> impairs = list(range(1, 20, 2))
>>> print(impairs)
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

C'est également l'occasion de découvrir les *listes en compréhension* ou *list comprehension* en anglais qui permettent de construire une liste avec une boucle `for` mais sur une seule ligne. Par exemple, les lignes suivantes :

```
>>> malist = list()
>>> for i in range(5):
...     malist.append(3 * i - 1)
>>> print(malist)
[-1, 2, 5, 8, 11]
```

peuvent être remplacées par l'unique ligne suivante :

```
>>> malist = [3 * i - 1 for i in range(5)]
>>> print(malist)
[-1, 2, 5, 8, 11]
```

Une autre façon de créer une liste de nombres impairs est donc :

```
>>> impairs = [2 * i + 1 for i in range(10)]
>>> print(impairs)
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Essayons maintenant d'avoir une fonction qui renvoie le nombre souhaité de nombres impairs :

```
def get_odd(n):
    """ retourne les n premiers nombres impairs """
    return [2 * i + 1 for i in range(n)]
impairs = get_odd(5)
print(impairs)
```

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

Liste de nombres aléatoires entre -1 et 1.

Première version sans compréhension de listes.

```
from random import random
def nombre_aleatoire(n):
    """ retoune une liste de n nombres aleatoires dans l'intervalle ]-1;1[] """
    # creation de la liste
    maListe = list()

    # remplissage et affichage de la liste
    for i in range(n):
        x = 2 * random() - 1
        maListe.append(x)

    # renvoie de la liste
    return maListe

liste_aleatoire = nombre_aleatoire(10)
print(liste_aleatoire)
```

```
[-0.6046927445855892, -0.878348850701256, -0.8226590776853149,
  0.9461206300847769, 0.006834292886062965, 0.13024638273947442,
 -0.7178558215419062, 0.8398222023594819, -0.03999145945919658,
  0.3536726384249611]
```

Deuxième version avec compréhension de listes.

```
from random import random
def nombre_aleatoire(n):
    """ retoune une liste de n nombres aleatoires dans l'intervalle ]-1;1[] """
    return [2 * random() - 1 for i in range(n)]

liste_aleatoire = nombre_aleatoire(10)
print(liste_aleatoire)
```

```
[-0.8038590712419067, -0.3459227696759206, -0.44087135408816436,
  0.25538004239761736, -0.8050229532763393, 0.19489054235953884,
 -0.7613568207194763, 0.31555687563975554, 0.26278528290623804,
 -0.14809381625603346]
```

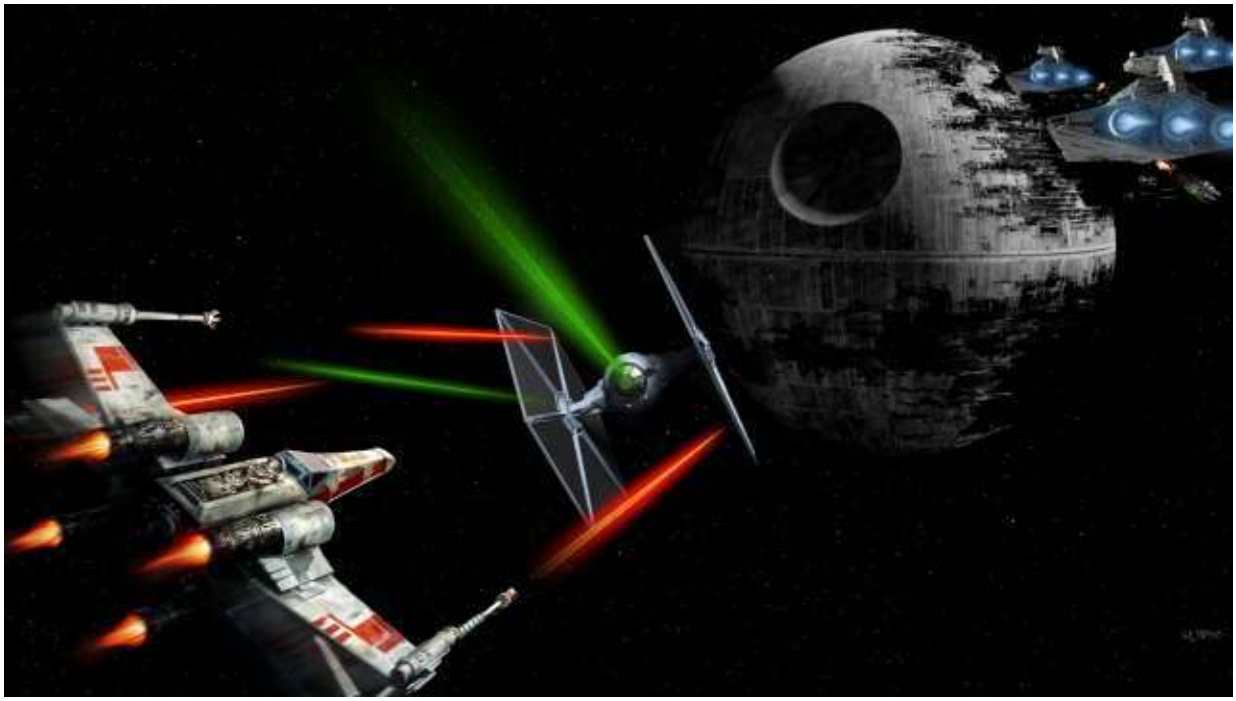
Star Wars

Rappel du facteur d'atténuation :

$$\alpha(n) = 0.83 \exp\left(-0.04982(n-1)\right)$$

```
def attenuation(n):
    """ facteur d'attenuation du laser en fonction du nombre de couche """
    return 0.83 * np.exp(-0.04982 * (n - 1))

# Calcul du nombre de couche après avoir perdu 3/4 de l'intensité initiale
# initialisation :
I = Io = 112358
n = 0
# boucle :
while I > 0.25 * Io:
    n += 1
    I = attenuation(n) * I
    print("%3d %12.2f %12.2f" % (n, I, 0.25 * Io))

print("Nombre de couches nécessaires : ", n)

# calcul du nombre de couche nécessaires pour que l'intensité chute à 10 W
# initialisation :
I = 112358
n = 0
# boucle :
while I > 10:
    n += 1
    I = attenuation(n) * I
    print("%3d %12.2f" % (n, I))

print("Nombre de couches nécessaires : ", n)
```

On notera l'utilisation d'un format dans la fonction `print()` pour améliorer l'affichage des résultats.



Objets et classes

Dans les faits, ce chapitre pourrait faire l'objet d'un cours complet. Nous allons simplement nous concentrer sur les fonctionnalités les plus simples dont vous aurez besoin pour pouvoir travailler avec python. En effet, on peut distinguer deux approches de l'objet en python :

- Le côté *utilisateur*, il faut savoir utiliser les objets
- Le côté *développeur*, pour la création de nouveaux objets

Nous allons nous focaliser sur le premier point.

Toute valeur est un objet

Toutes les choses que nous avons appelées valeur jusqu'à présent peuvent être appelé *"un objet"* dans l'univers de Python. On dit souvent qu'en Python *"tout est objet"*.

Par exemple les entiers, pour lesquels la fonction help() nous retournait des dizaines de lignes d'information à propos de `int()` sont aussi des objets.

Tout objet a une classe

Une classe est le type d'un objet (un type d'objet ?). Par analogie, on peut dire que c'est le moule qui permet de créer l'objet.

On peut tout simplement utiliser la fonction `type()` pour connaitre le type d'un objet :

```
>>> type(2)
<class 'int'>
>>> type(2.0)
<class 'float'>
>>> type("spam eggs")
<class 'str'>
>>> x = 1, 2
>>> type(x)
<class 'tuple'>
>>> type([])
<class 'list'>
```

Nous avons déjà parlé des classes que vous pouvez voir ici : `int`, `float`, `str`, `tuple`.

Quand nous utilisons des nombres dans notre programme, nous attendons qu'ils se comportent comme des nombres, et nous savons intuitivement ce qu'est un nombre. Par contre, Python doit savoir exactement ce que signifie "être un nombre".

Par exemple que se passe-t-il lorsqu'on additionne deux nombres ? Ou qu'on les divise ? La classe `int` définit tout cela et bien plus.

En utilisant la fonction `help()`, vérifiez ce que nous donne la classe `str`. Voici quelques fonctionnalités intéressantes :

```
>>> help(str.lower)
Help on method_descriptor:

lower(...)
    S.lower() -> str

    Return a copy of the string S converted to lowercase.

>>> help(str.upper)
Help on method_descriptor:

upper(...)
    S.upper() -> str

    Return a copy of S converted to uppercase.

>>> help(str.ljust)
Help on method_descriptor:

ljust(...)
    S.ljust(width[, fillchar]) -> str

    Return S left-justified in a Unicode string of length width. Padding is
    done using the specified fill character (default is a space).

>>> help(str.center)
Help on method_descriptor:

center(...)
    S.center(width[, fillchar]) -> str

    Return S centered in a string of length width. Padding is
    done using the specified fill character (default is a space)
```

Toutes ces opérations (ou méthodes) sont applicable à n'importe quelle chaîne de caractères. Pour y accéder, on ajoute un point suivi de l'appel de la fonction à appliquer :

```
>>> x = "Ala"
>>> x.upper()
'ALA'
>>> x.lower()
'ala'
>>> x.center(9)
'   Ala   '
```

Une fonction appliquée à un objet est appelée une méthode de l'objet. Il est important de comprendre le rôle du point. Dans ces situations, on lit de droite à gauche :

- méthode `upper` appliquée à x
- méthode `lower` appliquée à x
- méthode `center` appliquée à x

Le point sépare en quelque sorte un contenant et un contenu. On applique la méthode `upper` contenue dans la classe `str`.

Encore une dernière chose importante, pour créer un nouvel objet, on appelle la classe de l'objet (dans le jargon technique on dit qu'on instancie un objet). L'objet ainsi créé est appelé une instance de la classe :

```
>>> int()
0
>>> str()
''
>>> list()
[]
>>> tuple()
()
```

Une instance est donc une nouvelle valeur du type décrit par la classe.

Pour résumer, nous avons vu les classes `int()`, `str()`, `tuple()` et `list()`. Nous avons vu que pour connaitre la classe décrivant une valeur (un objet), nous pouvions regarder son type avec la fonction `type()`. Pour créer une instance de la classe (un nouvel objet), on appelle la classe de la même manière que nous appelons une fonction, en ajoutant des parenthèses (). Par exemple : `int()`.

Définir une classe

Les classes telles que `int` ou `str` font partie du langage Python et sont déjà définies, mais nous pouvons créer nos propres classes pour définir leur comportement. Cela s'appelle définir une classe.

Il est aussi facile de définir une classe que de définir une fonction. En fait une classe n'est rien de plus qu'un ensemble de fonctions, appelées méthodes. Prenons par exemple une classe `Dog` qui contient une méthode `bark` :

```
class Dog(object):

    def bark(self):
        print("Woof! Woof!")
```

Les classes commencent par le mot clé `class`, suivi du nom de la classe. L'`(object)` indique que le nouveau type `Dog` est un nouveau type de l'ensemble des classes de type `object`. Ainsi, les instances de notre classe, c'est à dire les objets créés, seront de type `Dog` mais également du type plus général des `objects`. Pour les habitués de la programmation orientée objet, cela signifie que la classe `Dog` hérite de la classe `object`, mais laissons cela de côté pour l'instant.

En fait c'est exactement pour cela qu'on dit que "tout est objet en Python". Car chaque classe est une spécialisation de la classe object de Python. C'est pourquoi quasiment chaque valeur est de type général object.

Il est important de noter que chaque fonction d'une classe doit prendre pour premier argument la valeur de l'objet duquel elle a été appelée. Nous l'appelons systématiquement `self` par convention. Dans notre exemple, nous avons une fonction appelée `bark` ("aboyer" en anglais), qui comme vous le voyez n'a qu'un seul argument, `self`. Regardons comment elle fonctionne :

```
>>> my_new_pet = Dog()
>>> my_new_pet.bark()
Woof! Woof!
```

Attributs des objets

Outre les méthodes (les fonctions définies dans une classe), les objets peuvent également avoir des attributs. Par exemple :

```
my_new_pet = Dog()
my_new_pet.name = "Snoopy"

print(my_new_pet.name)
Snoopy
```

Parfois nous souhaitons que tous les objets d'une classe aient un attribut, par exemple tous les chiens doivent avoir un nom. Nous pouvons le spécifier en créant une fonction, au nom spécial, appelée `__init__()`. Par exemple, attribuer un nom à notre chien :

```
class Dog(object):

    def __init__(self, name):
        self.name = name

    def bark(self):
        print("Woof! Woof!")
```

Dans la fonction `__init__()`, nous avons assigné une valeur à un nouvel attribut `name` de l'objet `self`. Comme expliqué précédemment, `self` est l'objet courant de la classe `Dog` que nous sommes en train de manipuler.

Remarque : On retrouve le point qui comme pour les méthodes indique que l'attribut `name` est un attribut de `self` (le nom est contenu dans la classe).

Nous pouvons maintenant utiliser cet attribut dans les autres méthodes :

```
class Dog(object):

    def __init__(self, name):
        self.name = name

    def bark(self):
        print(self.name, " Woof! Woof!")

snoopy = Dog("Snoopy")
pluto = Dog("Pluto")
print(snoopy.bark())
print(pluto.bark())
Snoopy Woof! Woof!
Pluto Woof! Woof!
```

La fonction `__init__()` est appelée durant la création de l'objet. On l'appelle constructeur, car elle aide à la création de l'objet et lui donne un état de départ.

Dans cet exemple, la fonction `__init__()` accepte deux arguments: `self` et `name`, mais quand on créé une instance de la classe `Dog`, nous ne spécifions que l'argument `name`, `self` est automatiquement spécifié par Python. Désormais, lorsque que nous instancions un nouvel objet `Dog`, celui-ci a un attribut : son nom.

Héritage

Dans le chapitre précédent, nous avons créé une classe Dog comme sous-ensemble du type object, mais ce n'est pas la seule possibilité. Nous pouvons également dire que Dog est aussi un Animal :

```
class Animal(object):
    pass

class Dog(Animal):

    def __init__(self, name):
        self.name = name

    def bark(self):
        print(self.name, " Woof! Woof!")
```

Nous avons donc une nouvelle classe `Animal`, qui hérite du type `object`. `Dog` hérite du type `Animal`. En d'autres termes :

- Tout `Animal` est un `object`
- Tout `Dog` est un `Animal`, tout `Dog` est un `object`

Ainsi nous pouvons décrire des comportements communs à tous les Animaux dans notre classe `Animal`, par exemple le fait de courir, et laisser dans la classe `Dog` des comportements plus spécifiques, comme aboyer:

```
class Animal(object):

    def run(self, distance):
        print("Run ", distance, " meters.")
```

La méthode `run` sera disponible pour tous les sous-types de `Animal` (comme les objets de type `Dog` par exemple) :

```
>>> scooby = Dog("Scooby")
>>> print(scooby.run(10))
Run 10 meters.
```

Arbre de noël

Revenons à l'arbre de noël que nous avons vu au chapitre précédent. Écrire une classe XMASTree qui pour une taille donnée et lors de l'appel de la méthode `draw()` va afficher les résultats suivants (pour les tailles 1, 2 et 3) :

```
  *
 /|\
/_|_\
  |
```

```
   *
  /|\
 /_|_\
  /|\
 / | \
/__|__\
   |
```

```
    *
   /|\
  /_|_\
   /|\
  / | \
 /__|__\
   /|\
  / | \
 /  |  \
/___|___\
    |
```

Livre de cuisine

Les documents de cette partie ont pour objectif de donner quelques exemples concrets pour effectuer certaines actions avec `python`. Les exemples reprennent :

- la lecture/écriture dans un fichier
- l'utilisation de `numpy` / `scipy`
- l'utilisation de `matplotlib` pour tracer une courbe
- l'utilisationd e `plotly` pour tracer une courbe

Lire et écrire dans un fichier

Comme il existe un objet `int` ou `float` il existe également un objet `file`. Pour lire et écrire dans un fichier il faut donc apprendre à utiliser cet objet.

La fonction `open()`

Avant de lire et écrire dans un fichier on dit qu'on "ouvre" ce fichier. Cette étape passe par la fonction `open()` qui crée un objet `file`.

Lorsqu'on travaille avec un fichier, il est préférable de s'assurer que certaines opérations sont exécutées avant et après l'ouverture. Pour ce faire on utilise un *contexte*. Il permet de séparer un bloc d'instructions donné. Python se charge alors d'exécuter des opérations de contrôle avant et après le bloc d'instructions.

Voici la syntaxe :

```
with objet as alias:
    # instructions sur alias
```

Lors de la création de alias et après l'exécution des instructions le concernant python fait le nécessaire pour nous simplifier la gestion de certaines erreurs.

On se donne un fichier `donnees.dat` :

```
# titre du fichier
1. 2.1
2. 2.9
3. 4.2
4. 5.05
5. 5.85
6. 6.95
7. 8.1
8. 9.
9. 10.2
10. 10.9
```

On va donc écrire :

```
>>> with open("donnees.dat", "r") as f:
...     contenu = f.read()
>>> print(contenu)
# titre du fichier
1. 2.1
2. 2.9
3. 4.2
4. 5.05
5. 5.85
6. 6.95
7. 8.1
8. 9.
9. 10.2
10. 10.9
```

On a utilisé la méthode `read()` pour lire tout le contenu du fichier.

La fonction `open()` :

- Le premier argument est le nom du fichier (son chemin pour être précis)
- Le second argument donne le *mode* d'accès :
  - `"r"` pour *read*, lecture du fichier
  - `"w"` pour *write*, écriture du fichier
  - `"a"` pour *append*, écriture à la fin du fichier

Lecture ligne par ligne

On va maintenant lire ligne par ligne le fichier pour récupérer dans une liste la première colonne (x) et dans une autre liste la deuxième colonne (y).

```
with open("donnees.dat", "r") as f:
    x = list()
    y = list()
    for line in f:
        if "#" in line:
            # on saute la ligne
            continue
        data = line.split()
        x.append(data[0])
        y.append(data[1])
print(x)
print(y)
```

```
['1.', '2.', '3.', '4.', '5.', '6.', '7.', '8.', '9.', '10.']
['2.1', '2.9', '4.2', '5.05', '5.85', '6.95', '8.1', '9.', '10.2', '10.9']
```

Remarques :

- La fonction `split()` découpe une chaîne de caractères suivant les espaces qu'elle contient.
- L'instruction `continue` impose à la boucle de passer à l'itération suivante sans exécuter les instructions suivantes.
- On remarquera que la liste contient des chaines de caractères qu'il faudrait convertir en nombre flottant avec la fonction `float()`.

D'autres méthodes existent : `readlines()` lit toutes les lignes et retourne une liste :

```
with open("donnees.dat", "r") as f:
    lines = f.readlines()
print(lines)
```

```
['1. 2.1\n', '2. 2.9\n', '3. 4.2\n', '4. 5.05\n', '5. 5.85\n', '6. 6.95\n',
'7. 8.1\n', '8. 9.\n', '9. 10.2\n', '10. 10.9\n']
```

On pourra se servir de `strip()` qui supprime les caractères spéciaux dans la chaine de caractères.

Nous verrons dans la partie sur numpy qu'il existe une façon très efficace de lire un fichier de ce type.

Écrire un fichier

Essayons par exemple de calculer les valeurs d'une fonction et d'écrire les résultats dans un fichier. La méthode `write()` attend une chaîne de caractères.

```
def fonction(x):
    return 2 * x**2 - 9 * x + 4

x = range(-5, 6, 1)
with open("valeurs.dat", "w") as f:
    f.write("# Un titre\n")
    for xi in x:
        valeurs = "%12.6f %12.6f\n" % (xi, fonction(xi))
        f.write(valeurs)
```

Dans la boucle, `xi` et `fonction(xi)` sont convertis en chaînes de caractères avec l'utilisation d'un format. On aurait pu simplement écrire `str(xi)` et `str(fonction(xi))` mais il faut penser à mettre un espace entre les deux. Le format `%12.6f` indique qu'on va écrire un nombre flottant, `f`, sur 12 caractères, avec 6 chiffres après la virgule.

Note : On n'oubliera pas le caractère de fin de ligne `\n`, sinon python ajoute le texte à la suite. Il ne passe pas à la ligne suivante après chaque appel à la méthode `write()`.

Autre solution : nous allons écrire toutes les valeurs dans une chaîne de caractères puis écrire cette chaîne dans le fichier.

```
def fonction(x):
    return 2 * x**2 - 9 * x + 4

x = range(-5, 6, 1)
lines = "# Un titre÷\n"
for xi in x:
    lines += "%12.6f %12.6f\n" % (xi, fonction(xi))

with open("valeurs.dat", "w") as f:
    f.write(lines)
```

Extraire des données

Le code suivant présente deux techniques pour extraire des données d'un fichier. Nous allons travailler sur le fichier soup.txt qui est affiché ci-dessous.

Combinons un test et `split()`

Premier cas simple

L'objectif est de rechercher la valeur de l'énergie écrite sur la ligne 39 du fichier `soup.txt`.

```
SCF Done:  E(RPBE-PBE) =  -548.263942119     A.U. after   14 cycles
```

Voici le code d'une fonction qui renvoie cette énergie :

```
def read_SCF(fichier):
    """ Lit la ligne SCF Done dans le fichier et retourne l'énergie """
    with open(fichier, "r") as f:
        for line in f:
            if "SCF Done:" in line:
                mots = line.split()
                energie = float(mots[4])
    return energie
```

Le principe est de chercher si `"SCF Done"` est sur la ligne courante, puis de la découper avec `split` et d'enregistrer la bonne valeur.

Supposons que cette ligne, contenant l'énergie, apparaisse plusieurs fois dans le fichier, on peut construire une liste contenant toutes les valeurs en seulement deux lignes de code :

```
with open(fichier, "r") as f:
    energies = [float(line.split()[4]) for line in f if "SCF Done:" in line]
```

Un cas plus complexe

On veut maintenant lire les valeurs de la section `excitation energies` sur les lignes 68 à 85. Par exemple, sur la ligne suivante, on cherche à lire l'énergie, la longueur d'onde et la force d'oscillateur (f) :

```
Excited State   2:      Singlet-A"     4.2831 eV  289.47 nm  f=0.0000  <S**2>=0.000
```

Voici une proposition :

```
def read_td(fichier):
    """ Lit la section excitation energies """

    # les résultats seront dans une liste
    transitions = list()

    with open(fichier, "r") as f:
        line = f.readline()
        td = False
        # lecture tant qu'on n'est pas à la fin du fichier
        while line != "":
            if "Excitation energies and oscillator strengths:" in line:
                td = True

            if td:
                if "Excited State" in line:
                    val = line.split()
                    energie = float(val[4])
                    nm = float(val[6])
                    force = float(val[8].strip("f="))
                    transitions.append((energie, nm, force))
            line = f.readline()

    return transitions

transitions = read_td("soup.txt")
print(transitions)
```

Ce qui donnera :

```
[(3.9281, 315.64, 0.0054), (4.2831, 289.47, 0.0), (6.4162, 193.24, 0.0457), (7.949, 15
5.97, 0.0013)]
```

Mieux que `split` les expressions régulières

Les expressions régulières permettent de réaliser avec peu de lignes des choses très complexes mais elles sont également difficiles à lire. Elles permettent d'identifier une chaîne de caractères suivant des critères précis. Nous allons reprendre l'exemple précédent.

Voici un outils en ligne très pratique pour tester vos expressions régulières : regex101

```
import re
def read_td(fichier):
    """ Lit la section excitation energies """

    # cette expression régulière repère tout type de nombre flottant
    float_patt = re.compile("\s*([+-]?\d+\.\d+)")

    transitions = list()

    # read in file
    with open(fichier, "r") as f:
        line = f.readline()
        td = False
        # lecture tant qu'on n'est pas à la fin du fichier
        while line != "":
            if re.search("^Excitation energies and oscillator strengths:", line):
                td = True

            if td:
                if re.search("^Excited State\s*\d", line):
                    val = [float(v) for v in float_patt.findall(line)]
                    transitions.append(tuple(val[0:3]))
            line = f.readline()
    return transitions

transitions = read_td("soup.txt")
print(transitions)
```

Commentaires :

- Il faut utiliser le module `re` pour *regular expression*
- `re.search("regex", line)` est équivalent à `if "regex" in line:`
- `float_patt.findall(line)` renvoie toutes les valeurs qui correspondent au modèle défini par `float_patt`
- `"\s*([+-]?\d+\.\d+)"` nombre à virgule avec éventuellement un signe : +0.1 ou 3.14 ou 100. ou -27.21

- Quelques éléments de syntaxe :
  - `\d` pour *digit*
  - `\s` pour *whitespace*
  - `*` caractère précédent de 0 à un nombre de fois illimité
  - `+` caractère précédent de 0 à un nombre de fois illimité
  - `?` caractère précédent 0 ou 1 fois
  - `^` début de la ligne

Quelques exemples d'expressions régulières

Le fichier `soup.txt` :

télécharger le fichier

```
Standard basis: 6-31G(d,p) (5D, 7F)
There are    36 symmetry adapted cartesian basis functions of A'  symmetry.
There are    13 symmetry adapted cartesian basis functions of A"  symmetry.
There are    33 symmetry adapted basis functions of A'  symmetry.
There are    13 symmetry adapted basis functions of A"  symmetry.
   46 basis functions,   108 primitive gaussians,    49 cartesian basis functions
   16 alpha electrons       16 beta electrons
      nuclear repulsion energy       106.6890740164 Hartrees.
NAtoms=    3 NActive=    3 NUniq=    3 SFac= 1.00D+00 NAtFMM=   60 NAOKFM=F Big=F
Integral buffers will be    131072 words long.
Regular integral format.
Two-electron integral symmetry is turned on.
One-electron integrals computed using PRISM.
NBasis=    46 RedAO= T EigKep=  3.97D-02  NBF=    33    13
NBsUse=    46 1.00D-06 EigRej= -1.00D+00 NBFU=    33    13
ExpMin= 1.17D-01 ExpMax= 2.19D+04 ExpMxC= 3.30D+03 IAcc=1 IRadAn=         1 AccDes= 0.
00D+00
Harris functional with IExCor= 1009 and IRadAn=       1 diagonalized for initial guess
.
HarFok:  IExCor= 1009 AccDes= 0.00D+00 IRadAn=         1 IDoV= 1 UseB2=F ITyADJ=14
ICtDFT=  3500011 ScaDFX=  1.000000  1.000000  1.000000  1.000000
FoFCou: FMM=F IPFlag=           0 FMFlag=      100000 FMFlg1=           0
        NFxFlg=           0 DoJE=T BraDBF=F KetDBF=T FulRan=T
        wScrn=  0.000000 ICntrl=     500 IOpCl=  0 I1Cent=   200000004 NGrid=
  0
        NMat0=    1 NMatS0=      1 NMatT0=    0 NMatD0=    1 NMtDS0=    0 NMtDT0=    0
Petite list used in FoFCou.
Initial guess orbital symmetries:
      Occupied  (A') (A') (A') (A') (A') (A') (A") (A') (A') (A')
                (A') (A') (A") (A") (A') (A')
      Virtual   (A") (A') (A') (A') (A") (A') (A') (A") (A') (A")
                (A') (A') (A") (A') (A') (A") (A') (A') (A') (A')
                (A") (A") (A') (A') (A') (A") (A") (A') (A') (A')
The electronic state of the initial guess is 1-A'.
Keep J ints in memory in symmetry-blocked form, NReq=1482832.
```

```
Requested convergence on RMS density matrix=1.00D-08 within 128 cycles.
Requested convergence on MAX density matrix=1.00D-06.
Requested convergence on             energy=1.00D-06.
No special actions if energy rises.
Integral accuracy reduced to 1.0D-05 until final iterations.
Initial convergence to 1.0D-05 achieved.  Increase integral accuracy.
SCF Done:  E(RPBE-PBE) =  -548.263942119     A.U. after   14 cycles
           NFock= 14  Conv=0.63D-08     -V/T= 2.0030
ExpMin= 1.17D-01 ExpMax= 2.19D+04 ExpMxC= 3.30D+03 IAcc=3 IRadAn=         5 AccDes= 0.
00D+00
HarFok:  IExCor=  205 AccDes= 0.00D+00 IRadAn=         5 IDoV=-2 UseB2=F ITyADJ=14
ICtDFT= 12500011 ScaDFX=  1.000000  1.000000  1.000000  1.000000
Range of M.O.s used for correlation:     8    46
NBasis=    46 NAE=    16 NBE=    16 NFC=     7 NFV=     0
NROrb=     39 NOA=     9 NOB=     9 NVA=    30 NVB=    30
Keep J ints in memory in symmetry-blocked form, NReq=1810774.
Orbital symmetries:
      Occupied  (A') (A') (A') (A') (A') (A") (A') (A') (A') (A')
                (A") (A') (A') (A") (A') (A')
      Virtual   (A") (A') (A') (A') (A") (A') (A') (A") (A") (A')
                (A') (A') (A") (A') (A") (A') (A') (A') (A') (A')
                (A") (A") (A') (A') (A') (A") (A") (A') (A') (A')
   16 initial guesses have been made.
Convergence on wavefunction:    0.001000000000000
Iteration     1 Dimension    16 NMult     0 NNew     16
CISAX will form    16 AO SS matrices at one time.
Iteration     2 Dimension    32 NMult    16 NNew     16
Iteration     3 Dimension    36 NMult    32 NNew      4
Iteration     4 Dimension    39 NMult    36 NNew      3
Iteration     5 Dimension    40 NMult    39 NNew      1
*****************************************************************************
Excited states from <AA,BB:AA,BB> singles matrix:
*****************************************************************************

Ground to excited state transition densities written to RWF  633

Excitation energies and oscillator strengths:

Excited State   1:      Singlet-A"     3.9281 eV  315.64 nm  f=0.0054  <S**2>=0.000
     16 -> 17         0.70680
This state for optimization and/or second-order correction.
Total Energy, E(TD-HF/TD-KS) =  -548.119587803
Copying the excited state density for this state as the 1-particle RhoCI density.

Excited State   2:      Singlet-A"     4.2831 eV  289.47 nm  f=0.0000  <S**2>=0.000
     15 -> 17         0.70716

Excited State   3:      Singlet-A'     6.4162 eV  193.24 nm  f=0.0457  <S**2>=0.000
     14 -> 17         0.65477
     15 -> 18         0.20991
     16 -> 19         0.19021

Excited State   4:      Singlet-A"     7.9490 eV  155.97 nm  f=0.0013  <S**2>=0.000
```

```
     13 -> 17         0.70470
SavETr:  write IOETrn=   770 NScale= 10 NData=  16 NLR=1 NState=    4 LETran=      82.

Symmetry A'   KE= 5.029717769773D+02
Symmetry A"   KE= 4.367837409063D+01
1\1\GINC-DNAS-NODE33\SP\RPBEPBE TD-FC\6-31G(d,p)\O2S1\GVALLVER\22-Sep-
2015\0\\# PBEPBE/6-31G** 5d td=(nstates=4)\\SO2 molecule\\0,1\S,0,4.99
586601,4.99305474,5.\O,0,6.44958723,4.99309717,5.\O,0,4.28472976,6.260
9081,5.\\Version=EM64L-G09RevD.01\State=1-A'\HF=-548.2639421\RMSD=6.29
7e-09\PG=CS [SG(O2S1)]\\@


HEAVEN'S NET CASTS WIDE.
THOUGH ITS MESHES ARE COARSE, NOTHING SLIPS THROUGH.

                                         -- LAO-TSU
Job cpu time:       0 days  0 hours  0 minutes  6.1 seconds.
File lengths (MBytes):  RWF=      6 Int=      0 D2E=      0 Chk=      1 Scr=      1
Normal termination of Gaussian 09 at Tue Sep 22 21:55:11 2015.
Fin  execution du job  : mar. sept. 22 21:55:11 CEST 2015
```

NumPy : une très courte introduction

Cette page donne quelques éléments sur l'utilsation des `array` du module NumPy. Bien entendu ces exemples ne sont pas exhaustifs et ne reflètent pas toute la richesse des modules NumPy et SciPy. Un tutoriel est disponnible directement sur la page de la documentation de NumPy : Quickstart tutorial.

Des liens importants :

- Le site de SciPy un point de départ pour python et les sciences
- NumPy : Numerical python
- SciPy : Scientific python

NumPy et SciPy contiennent également les fonctions ou constantes du module math de python.

Liste python ou array NumPy ?

Nous avons vu précédemment que les listes python sont *dynamiques* (leur taille n'est pas fixe) et elles peuvent contenir tout types d'éléments : entiers, flottants, chaînes de caractères ... Ce comportement donne aux listes une grande flexibilité et les rends très commode à utiliser au détriment d'une perte d'efficacité. C'est ce dernier point auquel NumPy tente de répondre avec les `array` .

Quelques spécificités des `array` :

- la taille d'un `array` est fixe
- un `array` ne contient qu'un seul type de données
- un `array` ne contient que la valeur et non l'objet en entier

Ces spécificités permettent une plus grande efficacité lors du parcours d'un `array` ou lors d'opérations impliquant des `array` .

Numpy ou Scipy ?

Avant de commencer voici un petit commentaire sur la différence entre NumPy et SciPy. Ce texte est extrait de la FAQ scipy :

Idéalement, NumPy ne devrait contenir que la classe `array`, avec les méthodes permettant de réaliser des opérations de base sur ces `array`s tandis que SciPy contiendrait toutes les fonctions numériques qui utilisent ces `array`s.

Dans un soucis de compatibilité NumPy conserve les méthodes historiquement contenues par lui-même ou ses prédécesseurs. Les versions les plus récentes de ces méthodes ou les nouveautés sont par contre présentes dans SciPy. De plus, SciPy contient entièrement NumPy.

Lequel dois-je utiliser ?

- NumPy, si vous souhaitez simplement utiliser les `array`
- SciPy, pour les méthodes les plus avancées

Les exemples

Importer les modules :

Habituellement on importe NumPy et SciPy avec des alias pour faciliter leur utilisation.

```
import numpy as np
import scipy as sp
```

Créer un `array`

```
# à partir d'une liste :
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
# avec la fonction arange
>>> r = np.arange(2, 10)
>>> r
array([2, 3, 4, 5, 6, 7, 8, 9])
>>> r = np.arange(2, 10, dtype="float64")
>>> r
array([ 2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
# avec des méthodes de numpy
>>> un = np.ones(4)
>>> un
array([ 1.,  1.,  1.,  1.])
>>> nul = np.zeros(3)
>>> nul
array([ 0.,  0.,  0.])
```

Quel type de données

Comme on l'a dit plus haut, contrairement aux listes, les `array` contiennent un seul type de données :

```
>>> a.dtype
dtype('int64')
>>> un.dtype
dtype('float64')
```

Les dimensions (forme) d'un `array`

```
>>> a = np.arange(2, 10, dtype="int")
>>> a
array([2, 3, 4, 5, 6, 7, 8, 9])
>>> a.shape
(8,)
>>> b = a.reshape(2,4)
>>> b
array([[2, 3, 4, 5],
       [6, 7, 8, 9]])
>>> b.shape
(2, 4)
```

Algèbre

Un avantage des `array` est que les opérateurs mathématiques s'appliquent à tous ses éléments. Cela permet, par exemple, d'additioner des vecteurs ou des matrices :

```
>>> a = np.ones(3)
>>> y = np.array([0, 1, 0])
>>> x = np.array([1, 0, 0])
>>> a
array([ 1.,  1.,  1.])
>>> x
array([1, 0, 0])
>>> y
array([0, 1, 0])
>>> x + y
array([1, 1, 0])
>>> a - x - y
array([ 0.,  0.,  1.])
>>> a * x
array([ 1.,  0.,  0.])
```

On peut calculer des sommes des produits scalaires, vectoriels ...

```
# produit scalaire
>>> np.dot(a, x)
1.0
>>> np.cross(x, y)
array([0, 0, 1])
# somme des éléments du array
>>> a.sum()
3.0
>>> b = np.arange(2, 10).reshape(2, 4)
>>> b
array([[2, 3, 4, 5],
       [6, 7, 8, 9]])
>>> b.sum(axis=0)
array([ 8, 10, 12, 14])
>>> b.sum(axis=1)
array([14, 30])
# produit matriciel
>>> np.dot(b, np.array([0, 1, 0, 0]))
array([3, 7])
```

Sélections de lignes, colonnes

Attention : En python les indices commencent toujours à 0.

```
>>> r = np.random.random((4, 5))
>>> r
array([[ 0.04159733,  0.66203517,  0.75505845,  0.30373412,  0.41154307],
       [ 0.19002609,  0.56761014,  0.54222685,  0.29554527,  0.05386348],
       [ 0.40999777,  0.28028108,  0.90805301,  0.67776604,  0.55878456],
       [ 0.30443061,  0.54716066,  0.03860554,  0.3677808 ,  0.39150167]])
# une valeur
>>> r[0, 0]
0.04159733383275166
>>> r[-1, -2]
0.36778080344490116
>>> r[1, 2]
0.54222685348487565
# ligne 2
>>> r[1,:]
array([ 0.19002609,  0.56761014,  0.54222685,  0.29554527,  0.05386348])
# colonne 3
>>> r[:,2]
array([ 0.75505845,  0.54222685,  0.90805301,  0.03860554])
```

Un filtre rapide avec `where`

```
>>> np.where(r > 0.5)
(array([0, 0, 1, 1, 2, 2, 2, 3]), array([1, 2, 1, 2, 2, 3, 4, 1]))
```

Ici, on récupère les indices `(ligne, colonne)` des éléments qui satisfont la condition.

```
>>> np.where(r > 0.5, r, -r)
array([[-0.04159733,  0.66203517,  0.75505845, -0.30373412, -0.41154307],
       [-0.19002609,  0.56761014,  0.54222685, -0.29554527, -0.05386348],
       [-0.40999777, -0.28028108,  0.90805301,  0.67776604,  0.55878456],
       [-0.30443061,  0.54716066, -0.03860554, -0.3677808 , -0.39150167]])
```

Dans ce second cas, on récupère la valeur de `r` si la condition est satisfaite et la valeur de `-r` sinon.

Lecture, écriture d'un fichier avec NumPy

Reprennons le fichier vu précédemment :

```
# titre du fichier
1. 2.1
2. 2.9
3. 4.2
4. 5.05
5. 5.85
6. 6.95
7. 8.1
8. 9.
9. 10.2
10. 10.9
```

La fonction `loadtxt` permet de lire facilement un fichier avec des données en colonne ou de type csv :

```
>>> data = np.loadtxt("data/donnees.dat", comments="#")
>>> print(data)
[[  1.     2.1 ]
 [  2.     2.9 ]
 [  3.     4.2 ]
 [  4.     5.05]
 [  5.     5.85]
 [  6.     6.95]
 [  7.     8.1 ]
 [  8.     9.  ]
 [  9.    10.2 ]
 [ 10.    10.9 ]]
>>> x, y = np.loadtxt("data/donnees.dat", unpack=True)
>>> print(x)
[  1.   2.   3.   4.   5.   6.   7.   8.   9.  10.]
>>> print(y)
[  2.1    2.9    4.2    5.05   5.85   6.95   8.1    9.    10.2   10.9 ]
```

On pourra utiliser les options suivantes :

- `comments="&"` , saute les lignes démarrant par `"&"`
- `usecols=(1, 3)` : lire les colonnes 1 et 3
- `skiprows=4` : saute les 4 premières lignes

Pour écrire un fichier, on pourra utiliser `np.savetxt()` , voici un exemple fonctionnel :

```
>>> header = "# titre du fichier\n"
>>> header += "# column 1 : ....\n"
>>> header += "# column 2 : ....\n"
...
>>> n = len(datas[0])
>>> X = [np.array(vec).reshape(n, 1) for vec in datas]
>>> X = np.concatenate(tuple(X), axis=1)
>>> np.savetxt("mesdonnees.dat", X, fmt="%10.5f", header=header)
```

- le premier argument est le nom du fichier
- `header` sera imprimé au début du fichier
- `fmt` est le format qui sera utilisé pour imprimer les nombres
- `X` est un `array`
  - Pour construire `X` on utilise `concatenate` permettant de combiner plusieurs `array`
  - On met `axis=1` pour ajouter successivement des colonnes
  - Les `array` doivent avoir une forme (n, 1)

Conclusion

Cette très courte introduction donne un apperçu de ce qu'on peut faire avec les `array`. L'utilisation de NumPy et SciPy dépend ensutie de ce qu'on veut en faire. De très nombreux modules et fonctions sont disponnibles en voici quelques uns :

- [SciPy] Le module `linalg` pour *linear alegebra*
- [SciPy] le module `integrate` pour l'intégration numérique
- [SciPy] le module `constante` contient de nombreuses constantes : h, c, R ...
- [NumPy] contient également une classe `np.matrix` qui fonctionne *"à la matlab"*.

Le site de SciPy regorge de ressources sur le sujet. On y trouve, entre autre, ce cours sur SciPy.

Représentation graphique avec matplotlib

Le site de SciPy, propose un cours sur `matplotlib`. On trouve de plus une gallerie bien fournie sur le site de matplotlib d'où on peut extraire de nombreux exemples :

- Le cours sur SciPy
- La galerie sur matplotlib

Cette page donne quelques bases pour démarrer :

Importation

Habituellement on importe matplotlib suivant :

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

Le module `pyplot` contient de nombreuses fonctions pour faire des graphiques diverses : `plot` (simple tracé), `pie` (camembert), `bar`, `hist` (histogramme), `polar`, `boxplot` (boîte à moustaches) ... Il existe aussi des fonctions pour représenter des données en 3D avec visualisation interactive.

Dans IPython ou Jupyter, on peut utiliser la commande magique `%pylab` pour charger `scipy`, `numpy` et `matplotlib` :

```
In [1]: %pylab --no-import-all inline
```

- `--no-import-all` évite `from numpy import *` et `from scipy import *`
- `inline` permet d'intégrer les graphiques dans le notebook ou la qtconsole

Exemples :

Ceci est un exemple pas à pas pour démarrer. Reportez-vous à la galerie pour plus d'exemples.

On va tracer la fonction :

$$S(t) = \sin(2\,t)e^{-0.04\,t}$$

Commençons par définir des fonctions

```
def signal(t):
    return np.sin(2 * t) * np.exp(-4e-2 * t)

def envelope(t):
    return np.exp(-4e-2 * t)
```

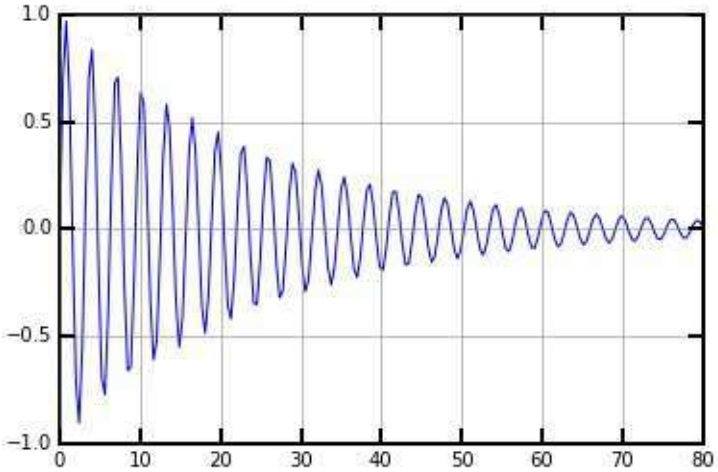
Remarque : Dans les lignes de code ci-dessous, `matplotlib` est importé en faisant :

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

Le premier graphique :

La fonction `plot` attend simplement comme premier argument l'abscisse et comme deuxième argument l'ordonnée. On utilise NumPy pour se donner une série de valeurs de t avec la fonction `np.linspace()`.

```
t = np.linspace(0, 80, 200)
plt.plot(t, signal(t))
plt.show()
# ou:
# plt.savefig("mongraph.eps")
```



Deux solutions s'offrent à nous pour affichier le graphique :

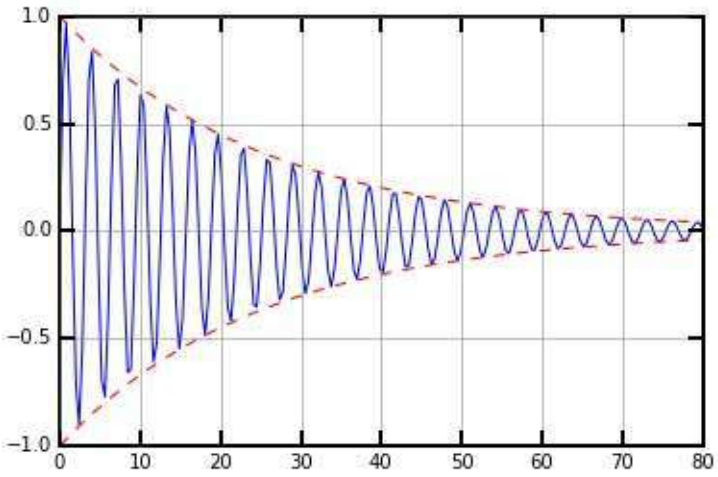
- `plt.show()` : ouvre une fenêtre intéreactive qui contient le graphique
- `plt.savefig()` : permet de sauvegarder le graphique dans un fichier
- Le comportement de `plt.show()` peut être modifié, notament avec l'option `inline` dans `IPython` ou `jupyter` qui permet d'intégrer les graphiques.

Remarque : Avec NumPy, l'appel de la fonction `signal(t)` calcule automatiquement les valeurs de la fonction pour toutes les valeurs de t.

Avec l'envelope

On ajoute l'envelope :

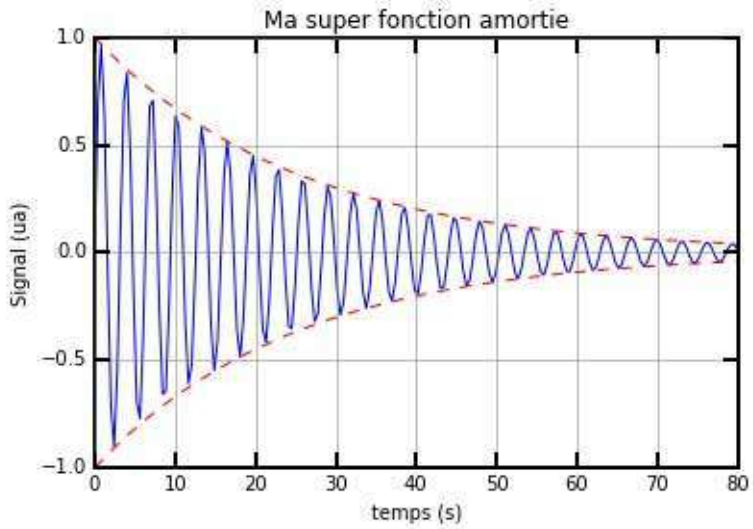
```
t = np.linspace(0, 80, 200)
plt.plot(t, signal(t))
plt.plot(t, envelope(t), color="red", linestyle="--")
plt.plot(t, -envelope(t), color="red", linestyle="--")
plt.show()
```



Ajoutons des informations

On va maintenant ajouter le nom des axes et un titre.

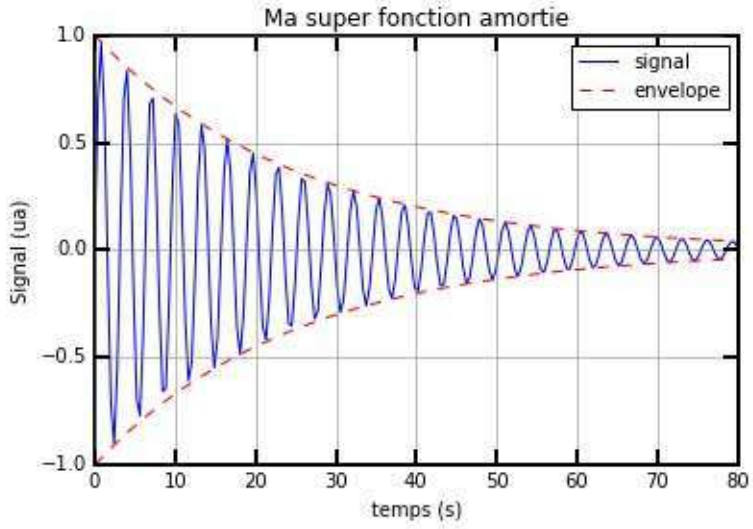
```
t = np.linspace(0, 80, 200)
plt.plot(t, signal(t))
plt.plot(t, envelope(t), color="red", linestyle="--")
plt.plot(t, -envelope(t), color="red", linestyle="--")
plt.xlabel("temps (s)")
plt.ylabel("Signal (ua)")
plt.title("Ma super fonction amortie")
plt.show()
```



Et la légende ?

On rajoute la légende.

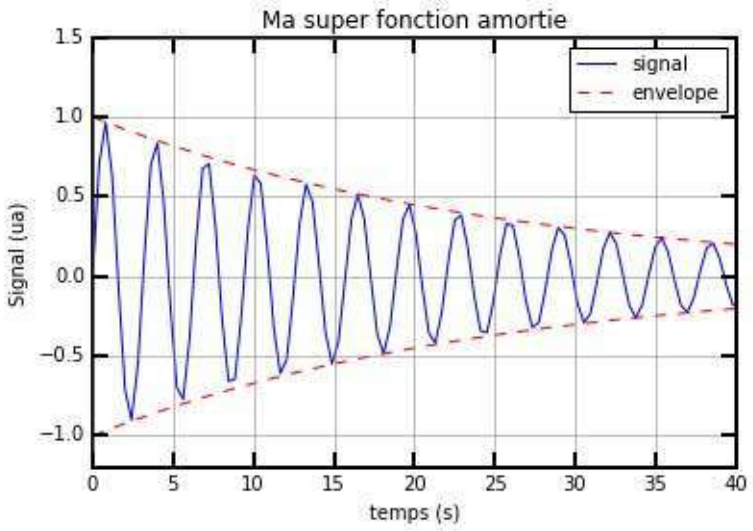
```
t = np.linspace(0, 80, 200)
plt.plot(t, signal(t), label="signal")
plt.plot(t, envelope(t), color="red", linestyle="--", label="envelope")
plt.plot(t, -envelope(t), color="red", linestyle="--")
plt.xlabel("temps (s)")
plt.ylabel("Signal (ua)")
plt.title("Ma super fonction amortie")
plt.legend()
plt.show()
```



Les axes

On ajuste l'échelle des axes :

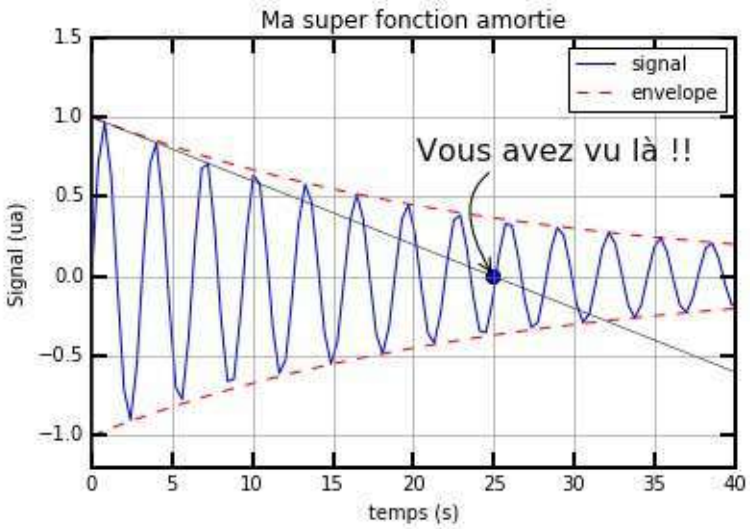
```
t = np.linspace(0, 80, 200)
plt.plot(t, signal(t), label="signal")
plt.plot(t, envelope(t), color="red", linestyle="--", label="envelope")
plt.plot(t, -envelope(t), color="red", linestyle="--")
plt.xlabel("temps (s)")
plt.ylabel("Signal (ua)")
plt.title("Ma super fonction amortie")
plt.legend()
plt.xlim((0, 40))
plt.ylim((-1.2, 1.5))
plt.show()
```



Annotations

On peut ajouter des annotations ou mettre en valeur des points.

```
t = np.linspace(0, 80, 200)
plt.plot(t, signal(t), label="signal")
plt.plot(t, envelope(t), color="red", linestyle="--", label="envelope")
plt.plot(t, -envelope(t), color="red", linestyle="--")
plt.xlabel("temps (s)")
plt.ylabel("Signal (ua)")
plt.title("Ma super fonction amortie")
plt.legend()
plt.xlim((0, 40))
plt.ylim((-1.2, 1.5))
plt.scatter([25, ], [0, ], s=50)
plt.plot(t, 1 - 4e-2 * t, color="#0C0B0E", linestyle="solid", linewidth=.5)
plt.annotate("Vous avez vu là !!", color="#0C0B0E",
             xy=(25, 0), xycoords='data',
             xytext=(-40, +60), textcoords='offset points', fontsize=16,
             arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.8"))
plt.show()
```



Conclusion

En conclusion : à vous de jouer. Les exemples ci-dessous devrait vous permettre de démarrer mais ils sont bien pauvres devant les possibilités de `matplotlib`. Rappelez-vous ce principe simple : il existe certainement dans la galerie un exemple proche de ce que vous cherchez à faire.

Voir une version dans un jupyter notebook de ces exemples.

Représentation graphique avec plotly

Qu'est ce que plotly, depuis le site de PyPi : *Python plotting library for collaborative, interactive, publication-quality graphs.*

La syntaxe est différente de celle de `matplotlib`. Mais plotly se différencie de matplotlib par un service web, un mode collaboratif et des graphiques interactifs.

Important : Par défaut plotly communique avec ses serveurs pour proposer des sercices collaboratifs et d'intégrations web. Pour éviter, si besoin, se passage par les serveurs plotly, il faudra penser à utiliser le mode `cloud`.

Les liens utiles :

- Utilisation de plotly avec python
- Doc python de référence
- Utilisation de plotly avec R
- La galerie pour trouver des exemples

Exemple

Voici un exemple qui illustre le principe :

- Chaque courbe est une `trace`
- Une figure contient des données `Data` et un `Layout`
- Les `Data` sont des listes de `trace`
- L'ensemble des objets de plotly fonctionne avec une syntaxe `clef=valeur`

L'exemple ci-dessous pourrait être utilisé dans un `jupyter notebook`. Nous allons réutiliser les fonctions précédentes :

Les fonctions

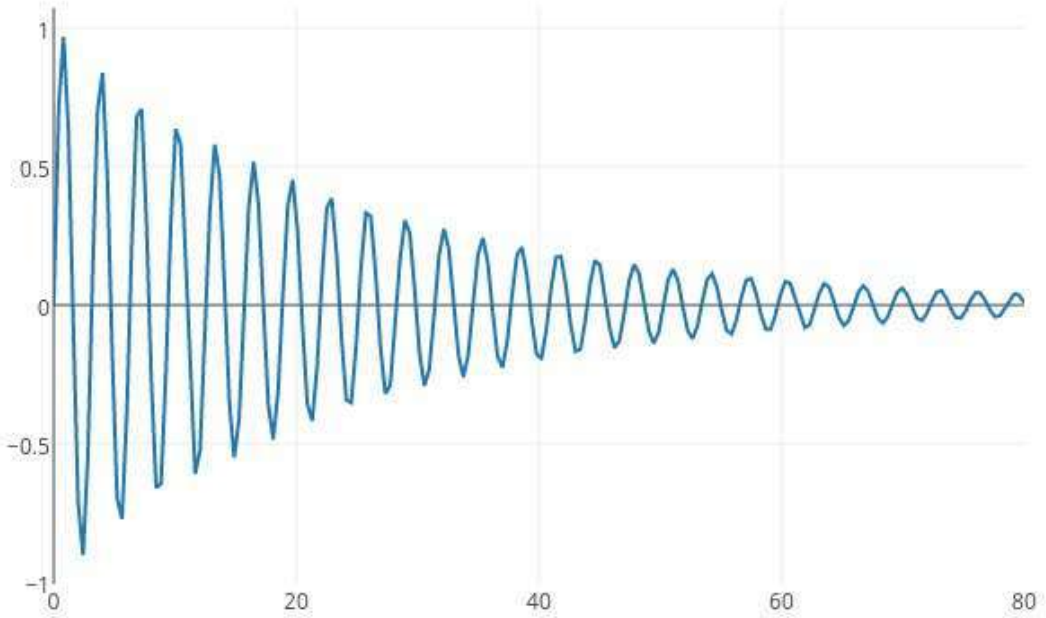
```
def signal(t):
    return np.sin(2 * t) * np.exp(-4e-2 * t)
def envelope(t):
    return np.exp(-4e-2 * t)
```

Chargement de plotly

```
import plotly
import plotly.graph_objs as go
# numpy
import numpy as np
# initialize plotly for jupyter notebook
plotly.offline.init_notebook_mode()
```

Le premier graphique :

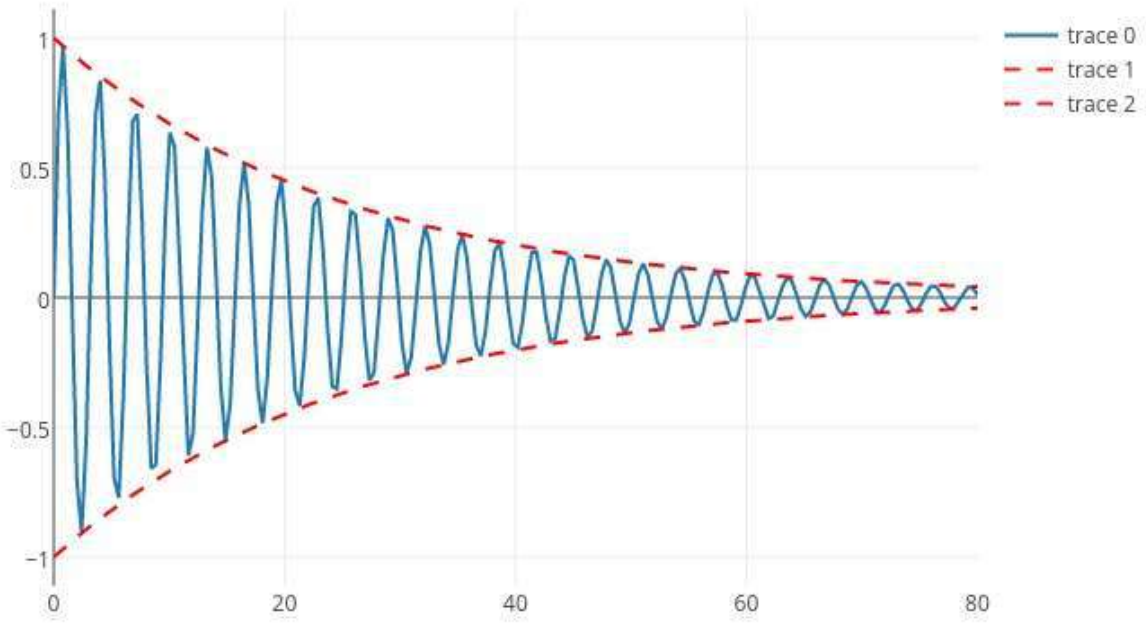
```
t = np.linspace(0, 80, 200)
trace = go.Scatter(
    x = t,
    y = signal(t),
)
data = go.Data([trace])
plotly.offline.iplot(data)
```



Avec l'envelope

On ajoute l'envelope, on ajoute des traces :

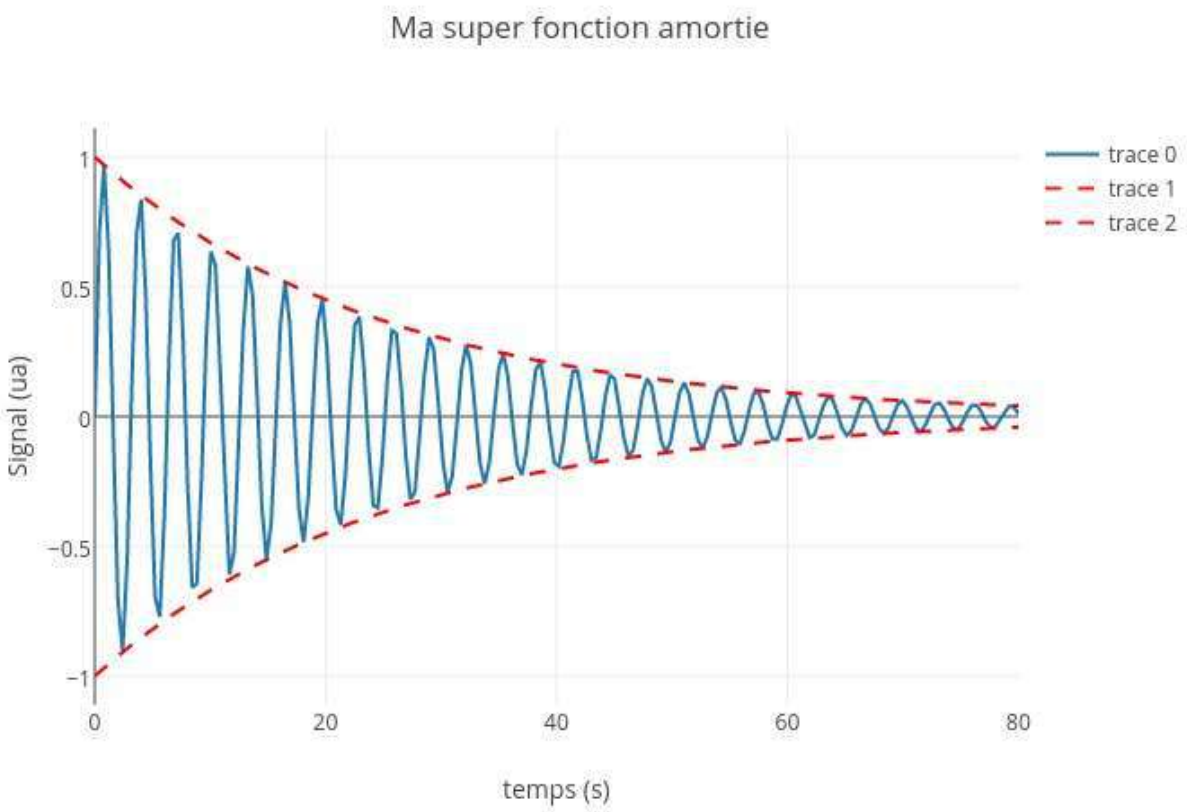
```
t = np.linspace(0, 80, 200)
trace = go.Scatter(
    x = t,
    y = signal(t)
)
envp = go.Scatter(
    x = t,
    y = envelope(t),
    line = go.Line(
        color = "red",
        dash = "dash"
    )
)
envm = go.Scatter(
    x = t,
    y = -envelope(t),
    line = go.Line(
        color = "red",
        dash = "dash"
    )
)
data = go.Data([trace, envp, envm])
plotly.offline.iplot(data)
```



Ajoutons des informations

On va maintenant ajouter le nom des axes et un titre. On ajoute un `Layout`

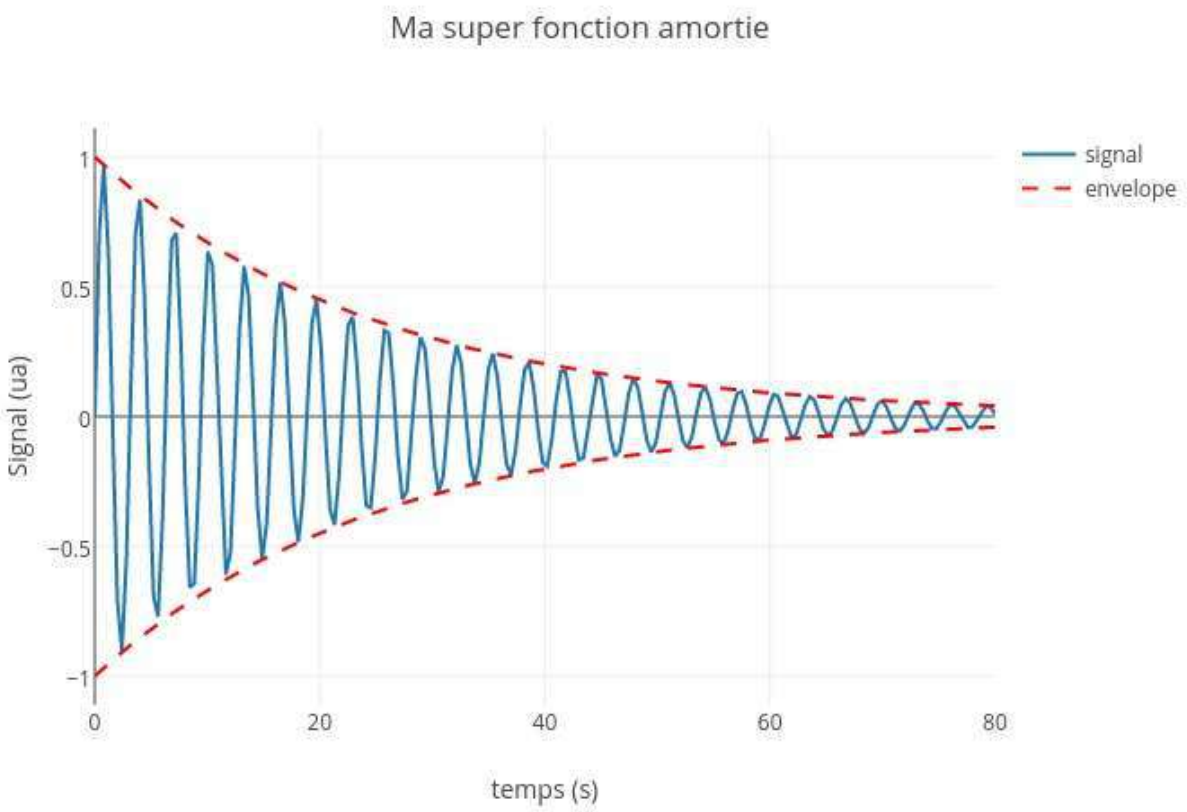
```
t = np.linspace(0, 80, 200)
# les traces
trace = go.Scatter(
    x = t,
    y = signal(t)
)
envp = go.Scatter(
    x = t,
    y = envelope(t),
    line = go.Line(
        color = "red",
        dash = "dash"
    )
)
envm = go.Scatter(
    x = t,
    y = -envelope(t),
    line = go.Line(
        color = "red",
        dash = "dash"
    )
)
# les traces sont regroupées dans un objet Data
data = go.Data([trace, envp, envm])
# le layout s'occupe de l'aspect général
layout = go.Layout(
    title = "Ma super fonction amortie",
    xaxis = go.XAxis(
        title = "temps (s)"
    ),
    yaxis = go.YAxis(
        title = "Signal (ua)"
    )
)
# la figure est un objet Data + un objet Layout
figure = go.Figure(data=data, layout=layout)
plotly.offline.iplot(figure)
```



Et la légende ?

On rajoute les noms adéquats, vous avez du remarquer que la légende est affichée par défaut.

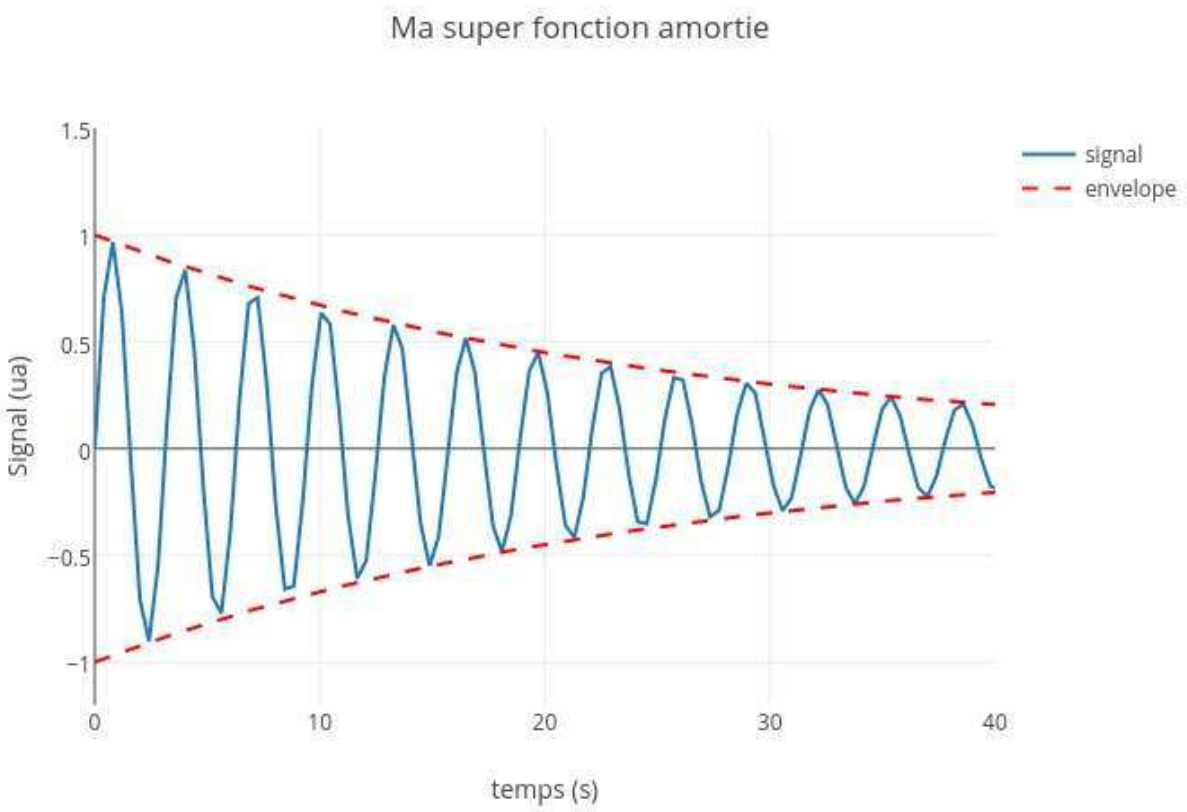
```
t = np.linspace(0, 80, 200)
trace = go.Scatter(
    x = t,
    y = signal(t),
    name = "signal"      # nom de la trace
)
envp = go.Scatter(
    x = t,
    y = envelope(t),
    name = "envelope",   # nom de la trace
    line = go.Line(
        color = "red",
        dash = "dash"
    )
)
envm = go.Scatter(
    x = t,
    y = -envelope(t),
    showlegend = False,
    line = go.Line(
        color = "red",
        dash = "dash"
    )
)
data = go.Data([trace, envp, envm])
layout = go.Layout(
    title = "Ma super fonction amortie",
    xaxis = go.XAxis(
        title = "temps (s)"
    ),
    yaxis = go.YAxis(
        title = "Signal (ua)"
    )
)
figure = go.Figure(data=data, layout=layout)
plotly.offline.iplot(figure)
```



Les axes

L'échelle des axes est définie dans le Layout via les objets XAxis et YAxis. Dans plotly chaque élément du graphique a un objet qui lui est propre qui se trouve dans `plotly.graphical_objs` .

```
t = np.linspace(0, 80, 200)
trace = go.Scatter(
    x = t,
    y = signal(t),
    name = "signal"
)
envp = go.Scatter(
    x = t,
    y = envelope(t),
    name = "envelope",
    line = go.Line(
        color = "red",
        dash = "dash"
    )
)
envm = go.Scatter(
    x = t,
    y = -envelope(t),
    showlegend = False,
    line = go.Line(
        color = "red",
        dash = "dash"
    )
)
data = go.Data([trace, envp, envm])
layout = go.Layout(
    title = "Ma super fonction amortie",
    xaxis = go.XAxis(
        title = "temps (s)",
        range = [0, 40]
    ),
    yaxis = go.YAxis(
        title = "Signal (ua)",
        range = [-1.2, 1.5]
    )
)
figure = go.Figure(data=data, layout=layout)
plotly.offline.iplot(figure)
```



Conclusion

En conclusion : à vous de jouer et comme pour matplotlib, rendez-vous sur la la galerie. Les figures ci-dessus sont statiques, pour se faire une idée du côté dynamique des graphiques fait avec `plotly`, consulter la version sour forme d'un jupyter notebook de ces exemples.

Visualisation de données avec pandas et matplotlib

Introduction de Pandas

La librairie `pandas` permet de manipuler de façon performantes et facile des données structurées, par exemple sous forme de tableau. Les objets disponnibles dans pandas permettent de trier, consolider, compléter vos données et de les exporter dans divers format (csv, latex, excel ...)

Pour commencer avec Pandas, je vous recommande la lecture de cette introduction sur la documentation officielle :

- 10 minutes to pandas.

et de cette description des divers objets de pandas :

- Intro to Data Structures

Chargement des données à partir d'un fichier CSV

Les données proviennent de http://opendata.agglo-pau.fr/index.php/fiche?idQ=27

Lors du téléchargement de l'archive, le fichier csv contient les données et un fichiers excel décrit le contenu de chaque colonne.

Ici, le fichier csv est lu directement avec la fonction `read_csv` de pandas, en précisant que le séparateur est un point virgule. Cette fonction retourne un objet `DataFrame` qui s'apparente à un tableau de données. On précise également que `LIBGEO`, le nom de la commune, sera utilisé comme index de la `DataFrame` de pandas. Chaque ligne portera alors le nom de la ville correspondante.

```
import pandas as pd
df = pd.read_csv("Evolution_et_structure_de_la_population/Evolution_structure_populati
on.csv", sep=";")
df = df.set_index("libgeo")
df
```

	ccodgeo	reg	dep	arr	cv	ze2010	id_modif_
libgeo							
ARTIGUELOUTAN	64059	72	64	643	6431	7214	ZZZZZZ
BILLERE	64129	72	64	643	6451	7214	ZZZZZZ
BIZANOS	64132	72	64	643	6447	7214	ZZZZZZ
GAN	64230	72	64	643	6446	7214	ZZZZZZ
GELOS	64237	72	64	643	6448	7214	ZZZZZZ
IDRON	64269	72	64	643	6431	7214	ZZZZZZ
JURANCON	64284	72	64	643	6446	7214	ZZZZZZ
LEE	64329	72	64	643	6431	7214	ZZZZZZ
LESCAR	64335	72	64	643	6419	7214	ZZZZZZ
LONS	64348	72	64	643	6419	7214	ZZZZZZ
MAZERES-LEZONS	64373	72	64	643	6448	7214	ZZZZZZ
OUSSE	64439	72	64	643	6431	7214	ZZZZZZ
PAU	64445	72	64	643	6499	7214	ZZZZZZ
SENDETS	64518	72	64	643	6423	7214	ZZZZZZ

14 rows × 148 columns

4 exemples de graphiques

Dans cette section sera présenté la construction de 4 graphiques en utilisant `matplotlib` comme librairie graphique et pandas pour manipuler les données.

Les données utilisées dans cet exemple sont issues de la base de données ouverte de la Communauté d'Agglomération de Pau-Pyrénées (CAPP) opendata.agglo-pau.fr. La section précédente présente comment lire ces données avec pandas.

Ce document peut être consulté sous forme de notebook.

Nous allons réaliser les 4 graphiques suivants. Avant de lire la suite, vous pouvez essayer de reproduire vous même ces graphiques :

1. Un graphique simple de type xy
    - Représenter le nombre de naissance et de décés à Pau en fonction du temps
    - (Sup) ajouter une courbe de tendance linéaire.

2. Un diagramme en barres horizontales
    - Chaque barre représente la population d'une commune en 2011
    - La colonne du tableau est `P11_POP`
    - Les communes seront classées par ordre croissant de population.
3. Une *treemap* sous forme de rectangle avec `squarify` :
    - L'aire de chaque rectangle correspond à la superficie de la commune
    - L'échelle de couleur correspond à la population de la commune
    - La population de Pau n'est pas prise en compte pour ne pas écraser l'échelle
    - Chaque rectangle est annoté par la superficie et la population.
4. Un diagramme camembert (ou pie chart)
    - Représenter la répartition en catégories socio-professionnelles pour Pau et la CAPP
    - Représenter la répartition en catégories socio-professionnelles pour Billère et la CAPP

Commençons par importer les modules python nécessaires :

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

1. Graphique xy classique

Dans ce graphique nous allons simplement représenter le nombre de naissances et de décés à Pau entre 1999 et 2011. C'est un graphique très simple à réaliser avec pandas. Nous allons voir tout d'abord comment extraire les données puis nous ferons un premier graphique basique avec la fonction `plot()` de pandas. Ensuite, nous ajouterons une courbe de tendance linéaire avec `lineregress()` de scipy.

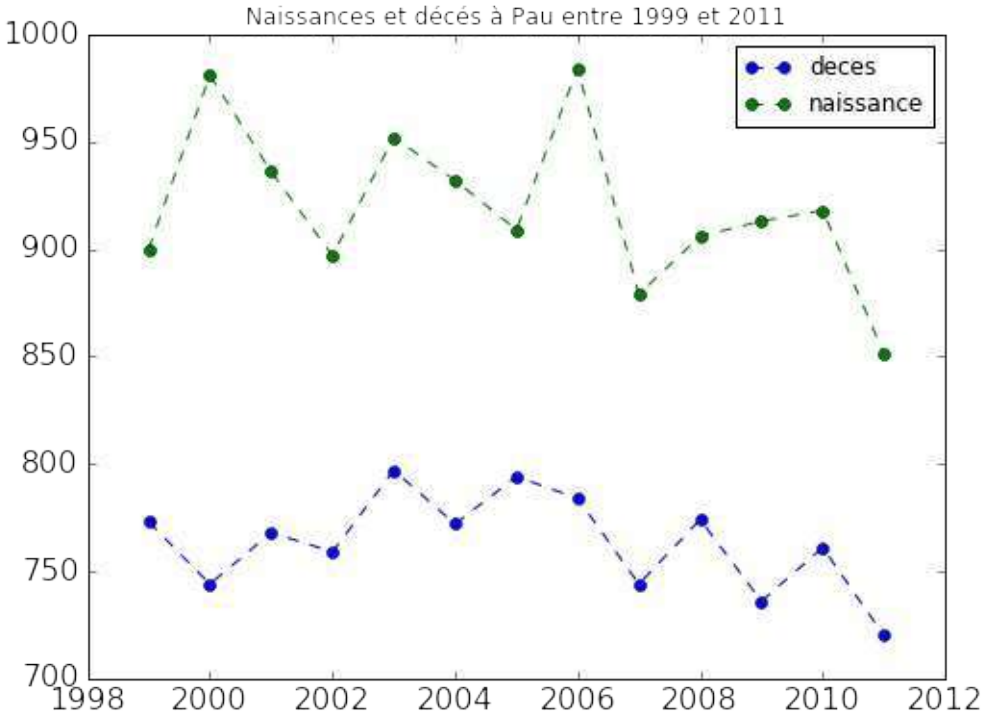
Tout d'abord nous devons extraire les colonnes `NAISXX` et `DECEXX` du tableau global pour la ligne concernant Pau.

```
# noms des colonnes:
years = [99] + list(range(0, 12))
naissance = ["nais%02d" % y for y in years]
deces = ["dece%02d" % y for y in years]

# extraction de la ligne qui concerne Pau
df_pau = df[df.index == "PAU"].squeeze()

# nouvelle DataFrame avec les deces et les naissances à Pau en fonction des années
dfxy = pd.DataFrame(
    data={
        "naissance": df_pau[naissance].values,
        "deces": df_pau[deces].values
    },
    index=[1999 + i for i in range(0, 13)]
)
dfxy
```

	deces	naissance
1999	773	900
2000	744	981
2001	768	936
2002	759	897
2003	797	952
2004	772	932
2005	794	909
2006	784	984
2007	744	879
2008	774	906
2009	736	913
2010	761	918
2011	720	851

À partir de ce type de tableau, il est très simple de tracer des courbes avec la fonction `plot()` de `pandas`. Les arguments sont du même type que ceux de matplotlib :

```
dfxy.plot(
    marker="o",
    linestyle="dashed",
    title="Naissances et décés à Pau entre 1999 et 2011",
    figsize=(8, 6),
    fontsize=16,
    xlim=(1998, 2012)
)
```

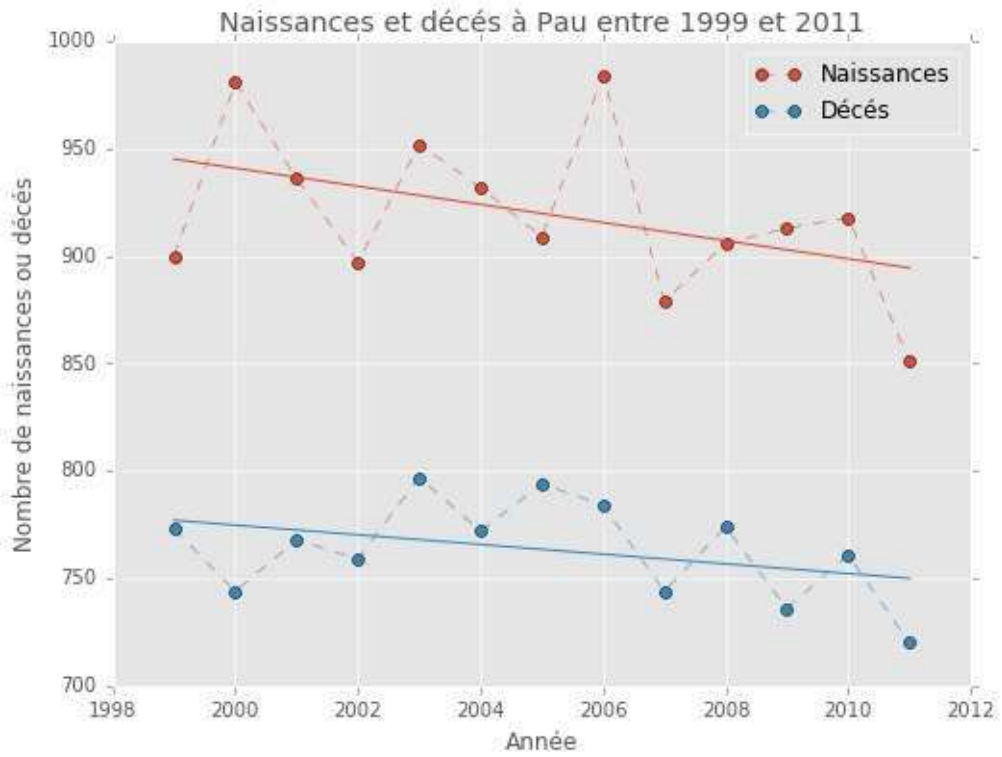


Ajoutons maintenant une courbe de tendance et ajoutons les données au tableau (bien que deux points seulement auraient suffit) :

```
from scipy.stats import linregress
# pour les naissances
p_naissance, i_naissance, *others = linregress(x=dfxy.index, y=dfxy.naissance)
dfxy["trend_naissance"] = p_naissance * dfxy.index + i_naissance
# pour les décés
p_deces, i_deces, *others  = linregress(x=dfxy.index, y=dfxy.deces)
dfxy["trend_deces"] = p_deces * dfxy.index + i_deces
```

	deces	naissance	trend_naissance	trend_deces
1999	773	900	945.164835	777.054945
2000	744	981	940.945055	774.802198
2001	768	936	936.725275	772.549451
2002	759	897	932.505495	770.296703
2003	797	952	928.285714	768.043956
2004	772	932	924.065934	765.791209
2005	794	909	919.846154	763.538462
2006	784	984	915.626374	761.285714
2007	744	879	911.406593	759.032967
2008	774	906	907.186813	756.780220
2009	736	913	902.967033	754.527473
2010	761	918	898.747253	752.274725
2011	720	851	894.527473	750.021978

```
plt.style.use('ggplot')
# figure
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111)
ax.set_title("Naissances et décés à Pau entre 1999 et 2011", color="#555555")
# plot the differents quantities
ax.plot(dfxy.index, dfxy.naissance, marker="o", label="Naissances", linestyle="--", li
newidth=.5)
ax.plot(dfxy.index, dfxy.deces, marker="o", label="Décés", linestyle="--", linewidth=.5
)
ax.plot(dfxy.index, dfxy.trend_naissance, label="", color="#E24A33")
ax.plot(dfxy.index, dfxy.trend_deces, label="", color="#348ABD")
# format and style
ax.set_xlabel("Année")
ax.set_ylabel("Nombre de naissances ou décés")
ax.legend()
fig.savefig("xy_pop.png", dpi=300)
```



2. Un diagramme en barres

Premièrement nous allons construire un graphique sous forme de barres horizontales, représentant la population dans chaque ville de l'aglomération de Pau-Pyrénées en 2011.

Nous utiliserons un diagramme en barre horizontale avec la fonction `barh` de matplotlib (exemple).

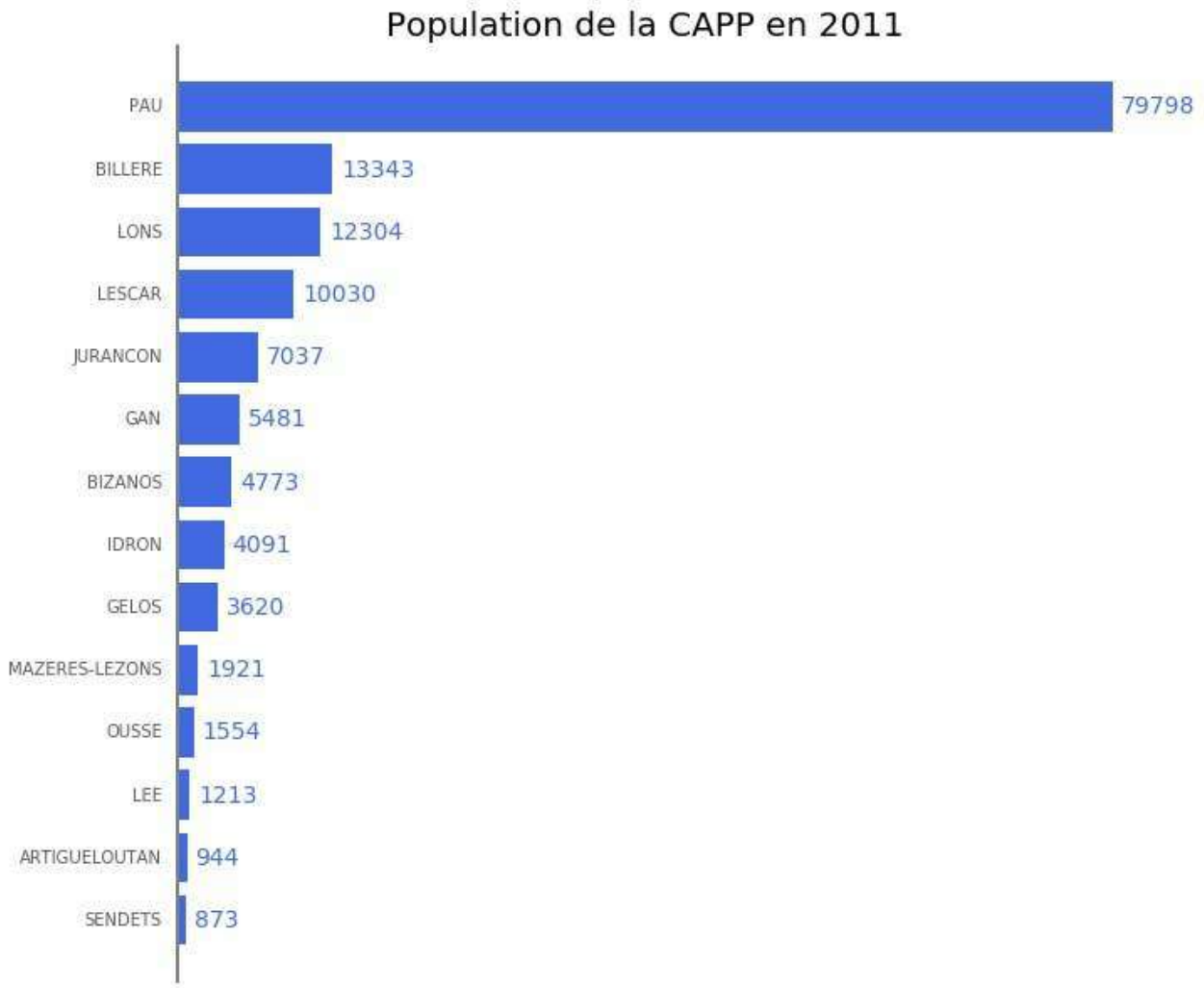
Commençons par extraire la colonne qui nous intéresse (`P11_POP`) et classons les valeurs par ordre croissant.

```
pop11 = df["p11_pop"]
pop11 = pop11.sort_values(ascending=True)
pop11
```

```
libgeo
SENDETS             873
ARTIGUELOUTAN       944
LEE                1213
OUSSE              1554
MAZERES-LEZONS     1921
GELOS              3620
IDRON              4091
BIZANOS            4773
GAN                5481
JURANCON           7037
LESCAR            10030
LONS              12304
BILLERE           13343
PAU               79798
Name: p11_pop, dtype: int64
```

Construisons maintenant le graphique avec les données ci-dessus.

```
# figure
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111)
ax.set_title("Population de la CAPP en 2011", fontsize=20)
# bar plot
ax.barh(
    # position et longueur des barres
    bottom=range(pop11.count()),
    width=pop11,
    # labels
    align="center",
    tick_label=pop11.index,
    # couleurs et traits
    color="RoyalBlue",
    linewidth=0
)
# format axis
ax.set_frame_on(False)
ax.set_xticks([])
ax.tick_params("both", length=0)
ax.vlines(0, -1, pop11.count(), color="gray", linewidth=4)
# add the value at the end of the bar
for y, pop in enumerate(pop11):
    ax.annotate(
        # texte
        "%.0f" % pop,
        # position
        xy=(pop, y),
        xytext=(5, -4),
        textcoords="offset points",
        # format
        fontsize=14,
        color="RoyalBlue"
    )
fig.savefig("barh_pop.png", dpi=300)
```



3. Réalisation d'une Treemap

Pour faire ce graphique on va utiliser la librairie `squarify`. La superficie des rectangles représentera la superficie de la commune. Une échelle de couleur donnera la population de la commune.

```
import squarify
```

Extraction de la superficie des communes et de la population en 2011. Le tableau est trié par ordre décroissant de la superficie.

```
df2 = df[["superf", "p11_pop"]]
df2 = df2.sort_values(by="superf", ascending=False)
df2
```

	superf	p11_pop
libgeo		
GAN	40	5481
PAU	32	79798
LESCAR	27	10030
JURANCON	19	7037
LONS	12	12304
GELOS	11	3620
ARTIGUELOUTAN	8	944
IDRON	8	4091
SENDETS	8	873
BILLERE	5	13343
BIZANOS	4	4773
MAZERES-LEZONS	4	1921
OUSSE	4	1554
LEE	3	1213

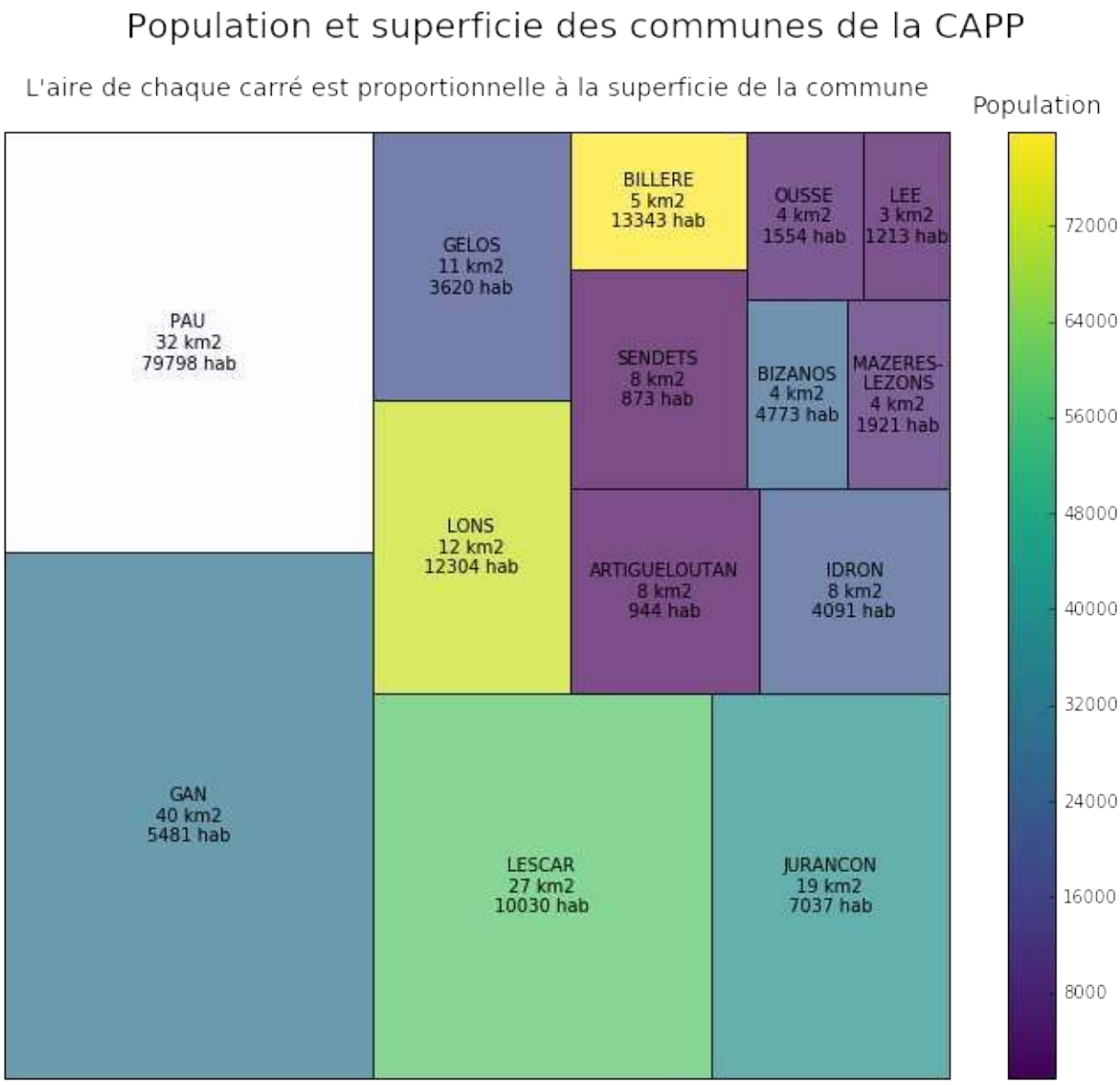
Superficie et population de Pau :

```
print("Pau: superficie = %d km2, population = %d\n" % (df2["superf"]["PAU"], df2["p11_
pop"]["PAU"]))
```

```
Pau: superficie = 32 km2, population = 79798
```

Représentation en Treemaps avec les fonctions inclues dans `squarify`.

```
x = 0.
y = 0.
width = 100.
height = 100.
cmap = matplotlib.cm.viridis

# color scale on the population
# min and max values without Pau
mini, maxi = df2.drop("PAU").p11_pop.min(), df2.drop("PAU").p11_pop.max()
norm = matplotlib.colors.Normalize(vmin=mini, vmax=maxi)
colors = [cmap(norm(value)) for value in df2.p11_pop]
colors[1] = "#FBFCFE"

# labels for squares
labels = ["%s\n%d km2\n%d hab" % (label) for label in zip(df2.index, df2.superf, df2.p
11_pop)]
labels[11] = "MAZERES-\nLEZONS\n%d km2\n%d hab" % (df2["superf"]["MAZERES-LEZONS"], df
2["p11_pop"]["MAZERES-LEZONS"])

# make plot
fig = plt.figure(figsize=(12, 10))
fig.suptitle("Population et superficie des communes de la CAPP", fontsize=20)
ax = fig.add_subplot(111, aspect="equal")
ax = squarify.plot(df2.superf, color=colors, label=labels, ax=ax, alpha=.7)
ax.set_xticks([])
ax.set_yticks([])
ax.set_title("L'aire de chaque carré est proportionnelle à la superficie de la commune
\n", fontsize=14)

# color bar
# create dummy invisible image with a color map
img = plt.imshow([df2.p11_pop], cmap=cmap)
img.set_visible(False)
fig.colorbar(img, orientation="vertical", shrink=.96)

fig.text(.76, .9, "Population", fontsize=14)
fig.text(.5, 0.1,
         "Superficie totale %d km2, Population de la CAPP : %d hab" % (df2.superf.sum(
), df2.p11_pop.sum()),
         fontsize=14,
         ha="center")
fig.text(.5, 0.07,
         "Source : http://opendata.agglo-pau.fr/",
         fontsize=14,
         ha="center")

fig.savefig("treemap_capp_pau.png", dpi=300)
```



4. Camembert ou pie chart ou wedge chart

Le fichiers de données de la CAPP contient aussi les catégorie socio-professionnelle pour chaque commune. Nous allons les représenter avec un diagramme de type camembert.

Commençons par regrouper les données dans une nouvelle table et faisons la somme sur toutes les communes.

```
columns = {
    "c10_pop15p_cs1": "agriculteurs",
    "c10_pop15p_cs2": "Artisans",
    "c10_pop15p_cs3": "Cadres",
    "c10_pop15p_cs4": "Intermédiaires",
    "c10_pop15p_cs5": "Employés",
    "c10_pop15p_cs6": "Ouvriers",
    "c10_pop15p_cs7": "Retraités",
    "c10_pop15p_cs8": "Autres"
}
df_cat = df[list(columns.keys())]
df_cat = df_cat.rename(columns=columns)
# add the sum over CAPP
df_cat.loc["CAPP"] = df_cat.sum(axis=0)
df_cat["total"] = df_cat.sum(axis=1)
df_cat
```

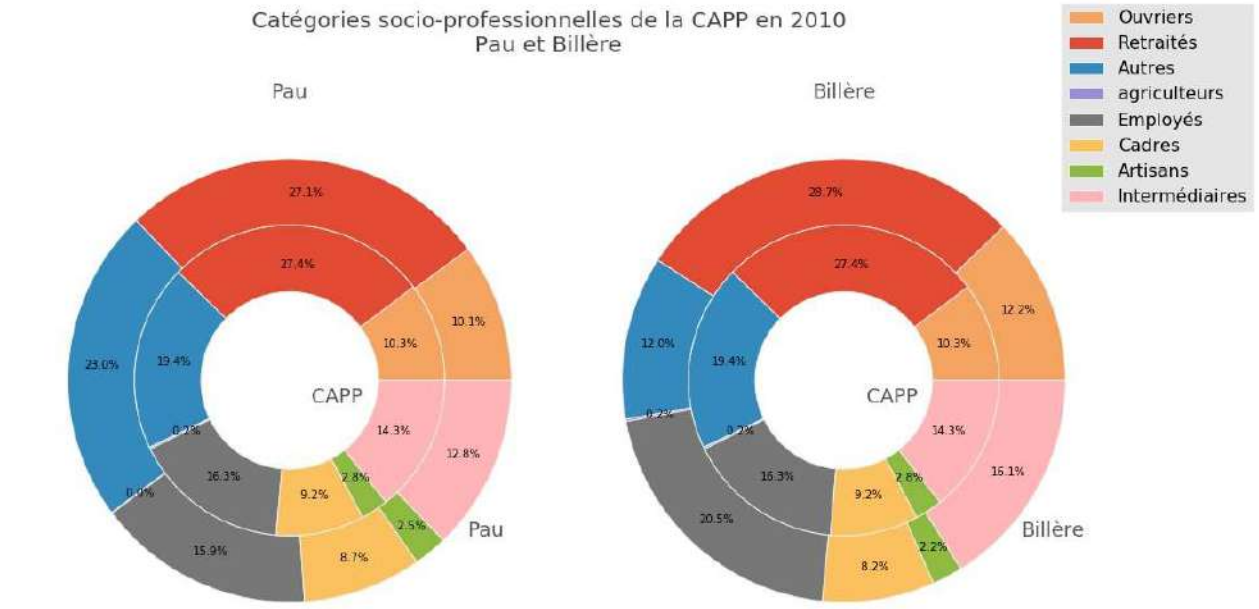
	Ouvriers	Retraités	Autres	agriculteurs	Employés
libgeo					
ARTIGUELOUTAN	72	184	80	24	140
BILLERE	1352	3178	1325	18	2270
BIZANOS	401	1337	578	0	622
GAN	406	1415	450	36	658
GELOS	293	921	491	20	498
IDRON	193	763	587	4	436
JURANCON	755	1761	900	16	932
LEE	70	199	172	4	98
LESCAR	925	1920	1408	21	1222
LONS	1028	2615	1604	37	1866
MAZERES-LEZONS	183	597	230	8	286
OUSSE	79	253	223	9	196
PAU	7087	19014	16149	35	11137
SENDETS	53	207	65	12	118
CAPP	12897	34364	24262	244	20479

Construction du graphique :

```
fig = plt.figure(figsize=(18, 10))
```

```
fig.suptitle("Catégories socio-professionnelles de la CAPP en 2010\nPau et Billère", f
ontsize=20, color="#444444")
width = .3

# color scale
colors = ["#F4A460", '#E24A33', '#348ABD', '#988ED5', '#777777', '#FBC15E', '#8EBA42',
'#FFB5B8']

# PAU et CAPP
ax1 = fig.add_subplot(121, aspect="equal")
pau_pie = ax1.pie(
    # data
    df_cat.loc["PAU"].drop("total"),
    # wedges
    colors=colors,
    radius=1,
    wedgeprops={"width": width, "linewidth": 1},
    # labels
    autopct="%4.1f%%",
    pctdistance=.85
)
capp_pie = ax1.pie(
    df_cat.loc["CAPP"].drop("total"),
    colors=colors,
    radius=1 - width,
    wedgeprops={"width": width, "linewidth": 1},
    autopct="%4.1f%%",
    pctdistance=.75
)
ax1.set_title("Pau", fontsize=18, color="#555555")
ax1.text(.1, -.1, "CAPP", fontsize=18, color='#555555')
ax1.text(.8, -.7, "Pau", size=18, color='#555555')

# Billere et CAPP
ax2 = fig.add_subplot(122, aspect="equal")
bill_pie = ax2.pie(
    df_cat.loc["BILLERE"].drop("total"),
    colors=colors,
    radius=1,
    wedgeprops={"width": width, "linewidth": 1},
    autopct="%4.1f%%",
    pctdistance=.85
)
capp_pie = ax2.pie(
    df_cat.loc["CAPP"].drop("total"),
    colors=colors,
    radius=1 - width,
    wedgeprops={"width": width, "linewidth": 1},
    autopct="%4.1f%%",
    pctdistance=.75
)
ax2.set_title("Billère", fontsize=18, color="#555555")
ax2.text(.1, -.1, "CAPP", size=18, color='#555555')
```

```
ax2.text(.8, -.7, "Billère", size=18, color='#555555')

# legende
fig.legend(pau_pie[0], df_cat.columns.values, fontsize=16)
fig.subplots_adjust(wspace=0)
fig.savefig("capp_pie.png", dpi=300)
```



Exercices

Les exercices suivants représentent de petits programmes nécessitant l'utilisation de boucles et de conditions. Ils sont classés par ordre de complexité croissante.

Consignes

- Pour chaque exercice choisir des noms de variables adaptés (lisibilité du programme)
- Il n'y a jamais trop de commentaires dans un programme, utilisez-les
- Essayer d'avoir un programme facile d'utilisation ou facile à utiliser dans un autre contexte
  - Création de fonctions
  - Documentation
  - Création de nouvelles classes
- Favoriser les fonctions déjà écrites : elles sont débeuguées et éprouvées. Ne perdez pas du temps à faire ce que quelqu'un d'autre à déjà fait.

Structure générale

Vous pouvez reprendre la structure générale d'un programme présentée lors des séries d'exercices précédentes :

```
#!/usr/bin/env python
# -*- coding=utf-8 -*-

""" Présentation du code """

def fonction(argument1, argument2):
    """ description de la fonction """

    # instructions de la fonction

    # la fonction retourne éventuellement une valeur
    return "une valeur"

if __name__ == "__main__":
    # instructions à exécuter, par exemple
    #    * lecture des parametres
    #    * appel de la fonction
    fonction()
```

Énoncés

1. Écrire un programme qui calcule la factorielle d'un entier naturel.

Rappel :

$$\begin{cases} n! = 1 \times 2 \times 3 \times \ldots \times n & \forall n \in \mathbb{N}^* \\ n! = 1 & \text{si } n = 0 \end{cases}$$

1. Écrire un programme qui calcule le produit des n premiers entiers impairs.
2. Écrire un programme qui calcule la somme des n premiers entiers naturels.
3. Une population décroit de 40% tous les 3 ans. La population étant considérée négligeable lorsqu'elle est inférieure à 0.1% de sa valeur initiale, au bout de combien d'année l'extinction est-elle atteinte ?
4. Écrire un programme qui construit une liste de nombres compris entre 0 et 100 puis cherche le minimum et le maximum dans cette liste.
   - On peut utiliser la fonction `random()` du module `random` pour construire la liste.
   - On peut aussi explorer le module `numpy.random` avec les fonctions `random()` et `randint()`, entre autres.
5. Une marche aléatoire : La marche aléatoire d'un point peut être modélisée de la façon suivante :

   $$\vec{r}(t + dt) = \vec{r}(t) + a\hat{\vec{R}}$$

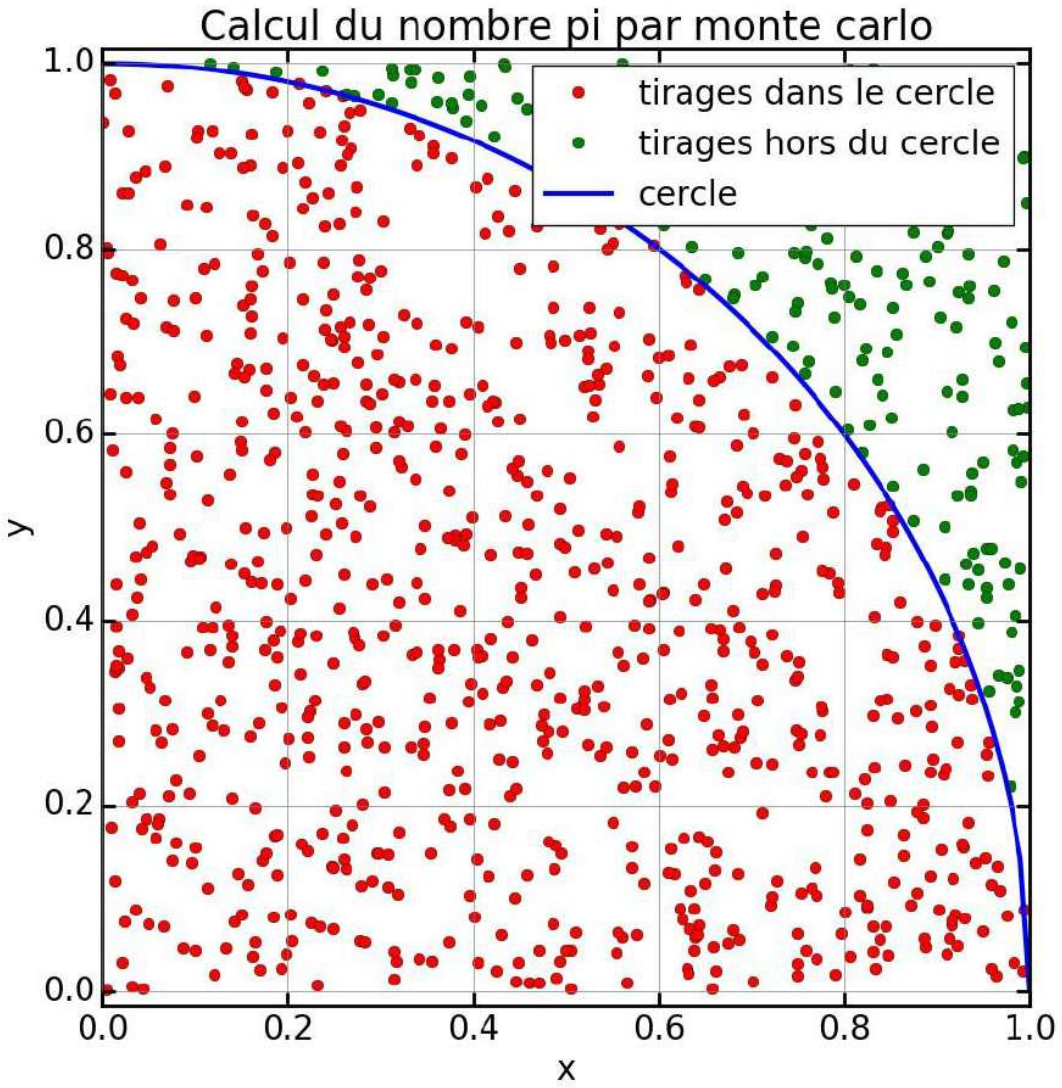
   où a désigne l'amplitude du déplacement aléatoire et $\hat{\vec{R}}$ est un vecteur aléatoire dont les composantes sont comprises entre -1 et 1.

   Écrire un programme qui met en œuvre une marche aléatoire dans un espace à deux dimensions et affiche à chaque pas de temps les coordonnées du point.

   Bonus : Utiliser le module `matplotlib` pour représenter la trajectoire.

6. Le nombre π peut être calculé par un processus dit de Monte Carlo, mettant en œuvre le tirage de nombres aléatoires. Le principe est le suivant : La probabilité pour qu'un point, de coordonnées (x, y), $x \in [0, 1]$ et $y \in [0, 1]$, choisies aléatoirement, soit dans un cercle de rayon 1 est égale au rapport de l'aire du quart de cercle de rayon 1 et l'aire de ce carré. L'aire du quart de cercle est $\pi/4$, l'aire du carré est égale à 1. Le nombre de points appartenant au quart de cercle sur le nombre total de points tirés doit donc converger vers $\pi/4$.

- Écrire un programme qui calcule le nombre π par cette méthode.
- Visualiser les points avec `matplotlib`.



7. Écrire un programme qui calcule l'intégrale d'une fonction par la méthode des trapèze.
8. Écrire un programme qui calcule l'intégrale d'une fonction par la méthode de Simpson.
9. [Numpy] Écrire un programme qui met en œuvre le procédé d'orthogonalisation de Gramm-Schmidt dont voici une brève description :

   Soit $\vec{u}$ et $\vec{v}$ deux vecteurs quelconques, le vecteur $\vec{v}'$ orthogonal au vecteur $\vec{u}$ est obtenu par :

$$\vec{v}' = \vec{v} - \frac{(\vec{u}.\vec{v})}{||\vec{u}||^2}\vec{u}$$

   - Utiliser le module `numpy` pour travailler avec des vecteurs

- La méthode `dot()` permet de calculer un produit scalaire
- Le module `numpy.linalg` contient, entre autre, une méthode `norm()`.

Corrections

ATTENTION : Vous êtes sûr d'avoir suffisamment essayé avant de regarder la correction !!

factorielle d'un entier naturel.

```
def factorielle(n):
    """ Calcul de factorielle n """

    if n == 0:
        factorielle = 1
    else:
        # initialisation
        factorielle = 1

        # calcul
        for i in range(2, n + 1):
            factorielle *= i

    return factorielle

if __name__ == "__main__":
    # lecture de n
    n = int(input("entrer n : "))
    print("n = ", n)

    # affichage du résultat
    print("{0}! = {1}".format(n, factorielle(n)))
```

La fonction factorielle est disponible dans le module `scipy` :

```
# la fonction factorial est dans le module misc de scipy
>>> from scipy.misc import factorial
>>> factorial(6)
array(720.0)
>>> factorial(3)
arrat(6.0)
# en utilisant n! = gamma(n+1) avec la fonction gamma dans le module 'special'
>>> from scipy.special import gamma
>>> gamma(4)
6.0
>>> gamma(7)
720.0
```

produit des n premiers entiers impairs

```
def prog_produit(n):
    """ Calcul du produit des n premiers entiers impairs """

    # initialisation
    produit = 1

    # calcul
    for i in range(3, n + 1):
        if i % 2 == 1:
            produit *= i

    # retour
    return produit

def prog_produit2(n):
    """ Calcul du produit des n premiers entiers impairs """

    # initialisation
    produit = 1

    # calcul
    for i in range(3, n + 1, 2):
        produit *= i

    # retour
    return produit

if __name__ == "__main__":
    # lecture de n
    n = int(input("entrer n : "))
    print("Calcul du produit des ", n, " premiers entiers impairs")
    produit = prog_produit(n)
    print("Résultat = ", produit)
```

On peut tester la performance d'un algorithme avec le module `cProfile` :

```
import cProfile
cProfile.run("prog_produit(100000)")
```

Ce qui donne :

```
4 function calls in 0.783 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1    0.783    0.783    0.783    0.783 <ipython-input-13-8309133be728>:1(prog_produit)
1    0.000    0.000    0.783    0.783 <string>:1(<module>)
1    0.000    0.000    0.783    0.783 {built-in method builtins.exec}
1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

```
import cProfile
cProfile.run("prog_produit2(100000)")
```

Ce qui donne :

```
4 function calls in 0.754 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1    0.754    0.754    0.754    0.754 <ipython-input-13-8309133be728>:16(prog_produit2
)
1    0.000    0.000    0.754    0.754 <string>:1(<module>)
1    0.000    0.000    0.754    0.754 {built-in method builtins.exec}
1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

La sortie de la fonction `cProfile.run()` indique le temps passé dans chacune des fonctions du programme. On remarque que la version 2 de notre fonction est légèrement plus efficace, en raison du test qui est absent dans celle-ci.

Somme des n premiers entiers naturels

```
def somme(n):
    """ calcul de la somme des n premiers entiers naturels """
    # initialisation
    somme = 0
    # calcul
    for i in range(n + 1):
        somme += i
    # retour
    return somme

if __name__ == "__main__":
    # lecture de n
    n = int(input("entrer n : "))
    print("Calcul de la somme des ", n," premiers entiers.")
    unesomme = somme(n)
    print("Résultat = ", unesomme)
```

Le calcul de la somme des premiers entiers peut se faire de façon analytique :

```
def somme(n):
    """ calcul de la somme des n premiers entiers naturels """
    return n * (n + 1) / 2
```

Décroissance d'une population

```
def prog_population():
    """
    Une population est réduite de 40% tous les 3 ans.
    Au bout de combien d'années la population est négligeable
    (inférireure à 0.1% de la population initiale) ?
    """

    # initialisation
    population = 100.0
    an = 0
    perte = .4
    seuil = .1 / 100. * population

    # boucle sur les années
    while population > seuil:
        an += 3
        population *= (1. - perte)
        print(an, population)

    print("Au bout de ", an, " ans, la population est négligeable")

if __name__ == "__main__":
    prog_population()
```

On peut améliorer cette fonction en ajoutant des arguments. Si on donne une valeur aux arguments dans la définition de la fonction, ils sont optionnels.

```
def prog_population(population, perte=0.4, periode=1):
    """
    Une population est réduite d'un facteur 'perte' sur une 'periode'.
    Au bout de combien d'années la population est négligeable
    (inférireure à 0.1% de la population initiale) ?

    Args:
        population (float): population initiale
        perte (float): facteur de perte
        periode (int): période en année
    """

    # initialisation
    an = 0
    seuil = .1 / 100. * population

    # boucle sur les années
    while population > seuil:
        an += periode
        population *= (1. - perte)
        print(an, population)

    print("Au bout de ", an, " ans, la population est négligeable")

if __name__ == "__main__":
    prog_population(population=12345, periode=3)
```

Ici on a passé les arguments dans une syntaxe du type `clef=valeur`. Dans ce cas, l'ordre des arguments n'a pas d'importance. Comme nous n'avons pas indiqué la valeur de perte lors de l'appel de la fonction, c'est la valeur par défaut qui est utilisée, soit 0.4.

Chercher le minimum et le maximum dans un liste

```
from random import random
from numpy.random import randint

def minimax(maListe):
    """ Recherche du maximum et du minimum dans maListe """

    # initialisation
    mini = maListe[0]
    maxi = maListe[0]

    for element in maListe:
        # recherche du minimum
        if element < mini:
            mini = element

        # recherche du maximum
        if element > maxi:
            maxi = element

    # retour du résultat
    return mini, maxi

if __name__ == "__main__":
    # nombre de points
    n = int(input("nombre de points : "))
    print("n = ", n)

    # remplissage d'une liste pseudo aleatoire de nombres entre 0 et 100
    maListe = [100. * random() for i in range(n)]
    # ou avec numpy
    # maListe = [randint(0, 100) for i in range(n)]

    mini, maxi = minimax(maListe)
    print("maximum = ", maxi)
    print("minimum = ", mini)
```

Les fonctions `min()` et `max()` existent en python. On peut écrire notre fonction plus simplement suivant :

```
def minimax2(maListe):
    """ Recherche du maximum et du minimum dans maListe """

    # initialisation
    mini = maListe[0]
    maxi = maListe[0]

    for element in maListe:
        mini = min(mini, element)
        maxi = max(maxi, element)

    # retour du résultat
    return mini, maxi
```

Bon, on vous a menti ... les fonctions `min()` et `max()` renvoient directement le minimum et le maximum dans une liste. Mais vous n'avez pas travaillé pour rien ! L'écriture de cette fonction est formatrice ! :)

```
>>> from numpy.random import randint
>>> n = 10
>>> maListe = [randint(0, 10) for i in range(n)]
>>> print(maListe)
[53, 65, 12, 38, 38, 91, 31, 46, 44, 50]
>>> max(maListe)
12
>>> min(maListe)
91
```

Une version simpliste de notre fonction pourrait alors être :

```
def minimax3(maListe):
    """ Recherche du maximum et du minimum dans maListe """
    return min(maListe), max(maListe)
```

Regardons ce que nous donne l'analyse des performances avec une liste de 1000000 valeurs :

```
>>> n = 1000000
>>> maListe = [random() for i in range(n)]
>>> cProfile.run("minimax(maListe)")
```

Pour la fonction 1

```
4 function calls in 0.086 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1    0.000    0.000    0.086    0.086 <string>:1(<module>)
1    0.086    0.086    0.086    0.086 minmax.py:9(minimax)
1    0.000    0.000    0.086    0.086 {built-in method builtins.exec}
1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Pour la fonction 2

```
2000004 function calls in 0.744 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1          0.000    0.000    0.744    0.744 <string>:1(<module>)
1          0.332    0.332    0.744    0.744 minmax.py:29(minimax2)
1          0.000    0.000    0.744    0.744 {built-in method builtins.exec}
1000000    0.206    0.000    0.206    0.000 {built-in method builtins.max}
1000000    0.207    0.000    0.207    0.000 {built-in method builtins.min}
1          0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' ob
jects}
```

On remarque ici un grand nombre d'appels aux fonctions `min()` et `max()` ce qui semble être particulièrement inefficace.

Pour la fonction 3

Sans surprise cette fonction est la plus efficace.

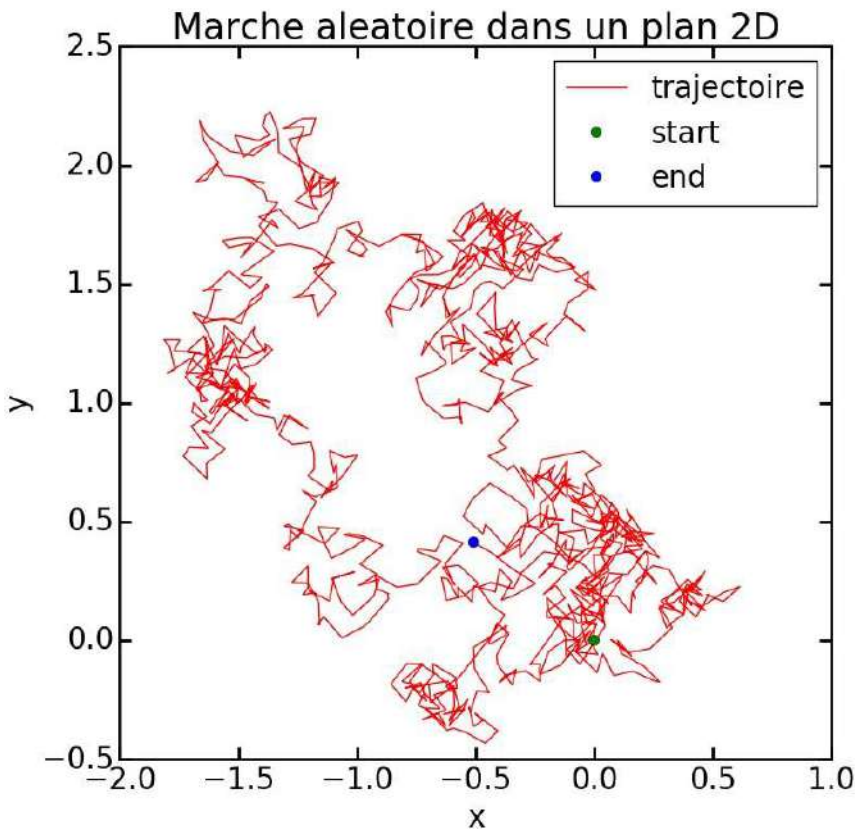
```
6 function calls in 0.051 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1    0.000    0.000    0.051    0.051 <string>:1(<module>)
1    0.000    0.000    0.051    0.051 minmax.py:44(minimax3)
1    0.000    0.000    0.051    0.051 {built-in method builtins.exec}
1    0.026    0.026    0.026    0.026 {built-in method builtins.max}
1    0.025    0.025    0.025    0.025 {built-in method builtins.min}
1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Une marche aléatoire

```
from random import random

def marche_aleatoire(npas, amplitude=0.1):
    """
    Marche aléatoire dans un plan 2D

    Args:
        npas (int): nombre de pas
        amplitude (float): amplitude du terme aléatoire
    """
    # liste pour les coordonnées
    x = list()
    y = list()

    # initialisation : condition initiale
    x.append(0.)
    y.append(0.)

    # marche aléatoire
    for i in range(npas):
        wx = 2. * random() - 1.
        wy = 2. * random() - 1.
        x.append(x[i] + amplitude * wx)
        y.append(y[i] + amplitude * wy)

if __name__ == "__main__":
    marche_aleatoire()
```

Pour faire une représentation graphique de la trajectoire avec `matplotlib` :

```
# représentation avec matplotlib
import matplotlib.pyplot as plt

# la trajectoire
plt.plot(x, y, "r-", label="trajectoire")

# points de départ et d'arrivé
plt.plot(x[0], y[0], "go", label="start")
plt.plot(x[-1], y[-1], "bo", label="end")

# format
plt.title("Marche aleatoire dans un plan 2D")
plt.axis("equal")
plt.grid(True)
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```

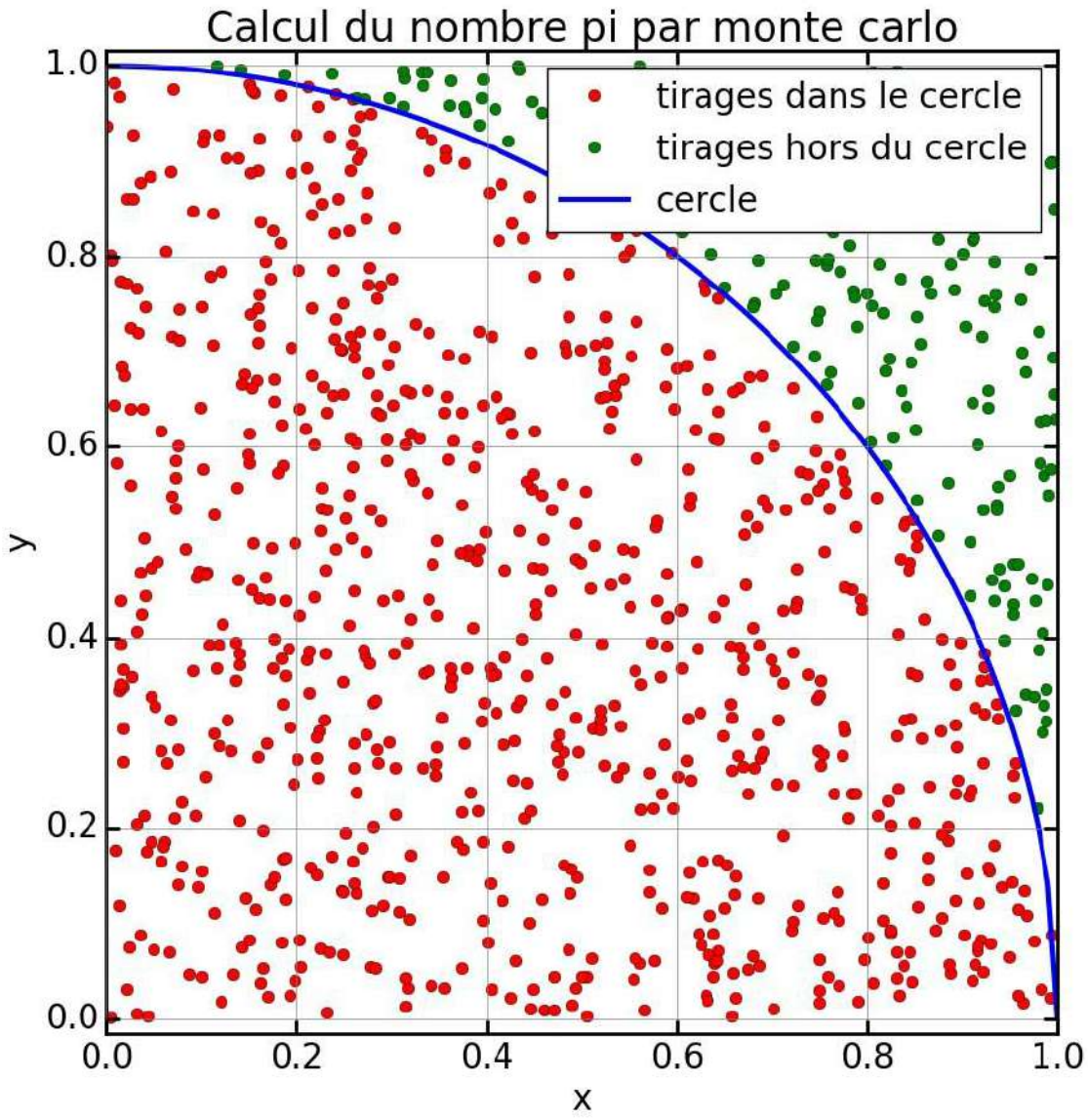


Calcul de π par Monte Carlo

```
import math
from random import random

def prog_pi(ntirage=100):
    """ calcul du nombre pi """

    # initialisation
    n = 0
    xDansCercle = list()
    yDansCercle = list()
    xHorsCercle = list()
    yHorsCercle = list()

    # calcul monte carlo
    for i in range(ntirage):
        x = random()
        y = random()

        if x**2 + y**2 < 1:
            n += 1
            xDansCercle.append(x)
            yDansCercle.append(y)
        else:
            xHorsCercle.append(x)
            yHorsCercle.append(y)

    # affichage des résultats
    piCalcule = 4 * n / ntirage
    print("pi         = ", piCalcule)
    print("% d'erreur = ", (math.pi - piCalcule) / math.pi * 100)

if __name__ == "__main__":
    ntirage = int(input("entrer le nombre de tirage : "))
    print("ntirage = ", ntirage)
    prog_pi(ntirage)
```

Représentation graphique avec `matplotlib`.

```
# representation graphique
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 10))

# points dans le cercle
plt.plot(xDansCercle, yDansCercle, "ro", label="tirages dans le cercle")

# points hors du cercle
plt.plot(xHorsCercle, yHorsCercle, "go", label="tirages hors du cercle")

# cercle
npts = 100
xcercle = [float(xi) / float(npts) for xi in range(npts + 1)]
ycercle = [math.sqrt(1.0 - xi**2) for xi in xcercle]
plt.plot(xcercle, ycercle, "b-", linewidth=3, label="cercle")

# options du graphiques
plt.title("Calcul du nombre pi par monte carlo")
plt.axis("equal")
plt.grid(True)
plt.legend()
plt.xlabel("x")
plt.ylabel("y")

# affichage
plt.show()
```



Méthode des trapèze

Valeur attendue :

$$\int_1^3 (x^2 - 3x - 6)e^{-x}dx = -2.525369$$

```
from math import exp

def trapeze(a, b, npas):
    """ integration par la methode des trapeze """

    # initialisation
    integrale = 0
    pas = (b - a) / npas

    # calcul de l'integrale
    for i in range(npas):
        x = a + i * pas
        integrale += pas * (f(x) + f(x + pas)) / 2

    return integrale

def f(x):
    """ fonction numerique utilisée """
    return (x**2 - 3. * x - 6.) * exp(-x)

if __name__ == "__main__":
    # lecture de l'intervalle
    a = float(input("entrer a : "))
    print("a = ", a)

    b = float(input("entrer b : "))
    print("b = ", b)

    # nombre de segments
    npas = int(input("entrer le nombre de pas : "))
    print("npas = ", npas)

    integrale = trapeze(a, b, npas)
    print("Résultat  = ", integrale)
```

Remarque : Voir la fonction `scipy.integrate.trapz()`

Méthode de Simpson

Valeur attendue :

$$\int_1^3 (x^2 - 3x - 6)e^{-x}dx = -2.525369$$

```
from math import exp

def simpson(a, b, npas=20):
    """ integration de simpson """

    # initialisation
    integrale = 0.0
    pas = (b - a) / float(npas)
    x = a

    # calcul de l'integrale
    while x < b:
        integrale += pas / 3 * (f(x) + 4 * f(x + pas) + f(x + 2 * pas))
        x += 2 * pas

    return integrale

def f(x):
    """ fonction numérique utilisée """
    return (x**2 - 3 * x - 6) * exp(-x)

if __name__ == "__main__":
    # lecture de l'intervalle
    a = float(input("entrer a : "))
    print("a = ", a)

    b = float(input("entrer b : "))
    print("b = ", b)

    # nombre de pas
    npas = int(input("entrer le nombre de pas : "))
    print("npas = ", npas)

    integrale = simpson(a, b, npas)

    print("Résultat  = ", integrale)
```

Remarque : Voir la fonction `scipy.integrate.simps()`

orthogonalisation de Gram-Schimdt

On va se limiter au cas où l'on a seulement deux vecteurs. Rappel de l'expression :

$$\vec{v}' = \vec{v} - \frac{(\vec{u}.\vec{v})}{||\vec{u}||^2}\vec{u}$$

Dans un premier temps nous allons utiliser les listes python.

```
def schmidt(u, v):
    """
    Procédé d'orthogonalisation de Gram-Schimdt.
    Soit u et v deux vecteurs. On cherche le vecteur vp le plus proche de v
    qui soit rthogonal à u.

    Args :
        u (list): vecteur u
        v (list): vecteur v
    """

    def scal(u, v):
        return sum([ui * vi for ui, vi in zip(u, v)])

    u2 = scal(u, u)

    # calcul du produit scalaire
    scalaire = scal(u, v)
    print("u.v = ", scalaire)

    # orthogonalisation de schmidt
    if scalaire != 0:
        vp = [vi - scalaire / u2 * ui for ui, vi in zip(u, v)]

        # verification
        print("u.vp = ", scal(u, vp))
    else:
        print("u et v sont orthogonaux.")
        vp = v

    return vp

if __name__ == "__main__":
    vp = schmidt(u=[1, 1, 1], v=[1, 2, 3])
    print("vp = ", vp)
```

Utilisons maintenant les `array` de `numpy` :

```
import numpy as np

def schmidt(u, v):
    """
    Procédé d'orthogonalisation de Gram-Schimdt.
    Soit u et v deux vecteurs. On cherche le vecteur vp le plus proche de v
    qui soit rthogonal à u.

    Args :
        u (list): vecteur u
        v (list): vecteur v
    """

    # Conersion en array numpy
    u = np.array(u)
    v = np.array(v)

    u2 = (u**2).sum()

    # calcul du produit scalaire
    scalaire = np.dot(u, v)
    print("u.v = ", scalaire)

    # orthogonalisation de schmidt
    if scalaire != 0:
        vp = v - scalaire / u2 * u

        # verification
        print("u.vp = ", np.dot(u, vp))
    else:
        print("u et v sont orthogonaux.")
        vp = v

    return vp

if __name__ == "__main__":
    vp = schmidt(u=[1, 1, 1], v=[1, 2, 3])
    print("vp = ", vp)
```

Problèmes complets

Les problèmes listés ci-dessous nécessitent l'écriture de petit programmes. Le principe est de commencé par écrire le programme directement puis de voir comment l'écrire en se servant de ce qui existe dans les bibliothèques disponibles. Cela permet de s'exercer à deux aspects importants de la programmation :

- Traduire le problème dans un langage de programmation et maîtriser la syntaxe du langage
- Utiliser des bibliothèques existantes et lire la documentation associée.

- Moindres carrés
- Traduction, transcription de l'ADN

Comment mettre en oeuvre une régression linéaire avec python ?

L'objectif de ce TP est de mettre en pratique le langage python pour réaliser une regression linéaire. L'idée est, dans un premier temps, de reprendre les éléments de base du langage (condition, boucles ...) pour créer un outil qui calcule les paramètres par la méthode des moindres carrés. Dans une deuxième partie les modules spécifiques tels que numpy ou matplotlib seront mis à profit pour réaliser la même opération avec des méthodes existantes.

Introduction

Cahier des charges

Le programme que nous allons écrire devra réaliser les opérations suivantes :

- Lire les valeurs des valeurs de x et y sur le fichier `donnees.dat` .
- Calculer les paramètres a et b de la régression linéaire par la méthode des moindres carrés et les afficher.
- (bonus) Représenter les points (x,y) et tracer la droite de régression.

Rappels mathématiques

La régression linéaire consiste à chercher les paramètres a et b définissant la droite y=ax+b qui passe au plus près d'un ensemble de points (x_k, y_k). Les paramètres a et b sont déterminés par la méthodes des moindres carrés qui consiste, dans le cas d'une régression linéaire, à minimiser la quantité :

$$Q(a,b) = \sum_{k=1}^{N} (y_k - ax_k - b)^2$$

Le minimum de $Q(a,b)$ est obtenu lorsque ses dérivées par rapport à a et b sont nulles. Il faut donc résoudre le système à deux équations deux inconnues suivant :

$$\begin{cases} \dfrac{\partial Q(a,b)}{\partial a} = 0 \\ \dfrac{\partial Q(a,b)}{\partial b} = 0 \end{cases} \Leftrightarrow \begin{cases} -2\displaystyle\sum_{k=1}^{N} x_k \left(y_k - ax_k - b\right) = 0 \\ -2\displaystyle\sum_{k=1}^{N} \left(y_k - ax_k - b\right) = 0 \end{cases}$$

Les solutions de ce système sont :

$$a = \frac{N\displaystyle\sum_{k=1}^{N} x_k y_k - \sum_{k=1}^{N} x_k \sum_{k=1}^{N} y_k}{N\displaystyle\sum_{k=1}^{N} x_k^2 - \left(\sum_{k=1}^{N} x_k\right)^2} \qquad b = \frac{\displaystyle\sum_{k=1}^{N} x_k^2 \sum_{k=1}^{N} y_k - \sum_{k=1}^{N} x_k \sum_{k=1}^{N} x_k y_k}{N\displaystyle\sum_{k=1}^{N} x_k^2 - \left(\sum_{k=1}^{N} x_k\right)^2}$$

Progression

Le programme sera écrit de plusieurs façon différentes.

1. Tous les calculs seront réalisés à la main étape par étape
2. Création d'une fonction qui réalise la régression linéaire
3. Utilisation des `array` du module NumPy
4. Utilisation des méthodes des modules NumPy/SciPy pour réaliser la régression linéaire
5. (bonus) Utilisation du module matplotlib pour représenter les points et la droite de régression.

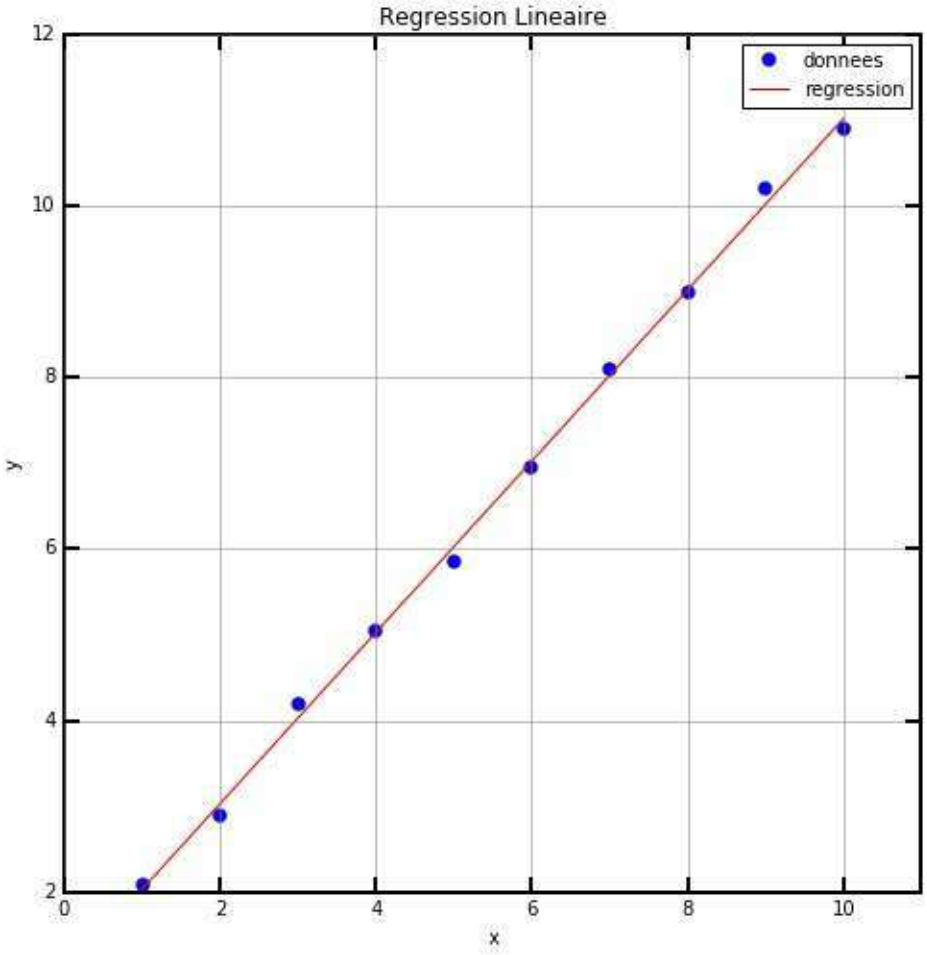
Vous pouvez utiliser les deux fichiers ci-dessous contenant un ensemble de données pour réaliser la régression linéaire :

- donnees.dat : 10 valeurs, y = x + 1
- regLinData.dat : 200 valeurs

Programmation

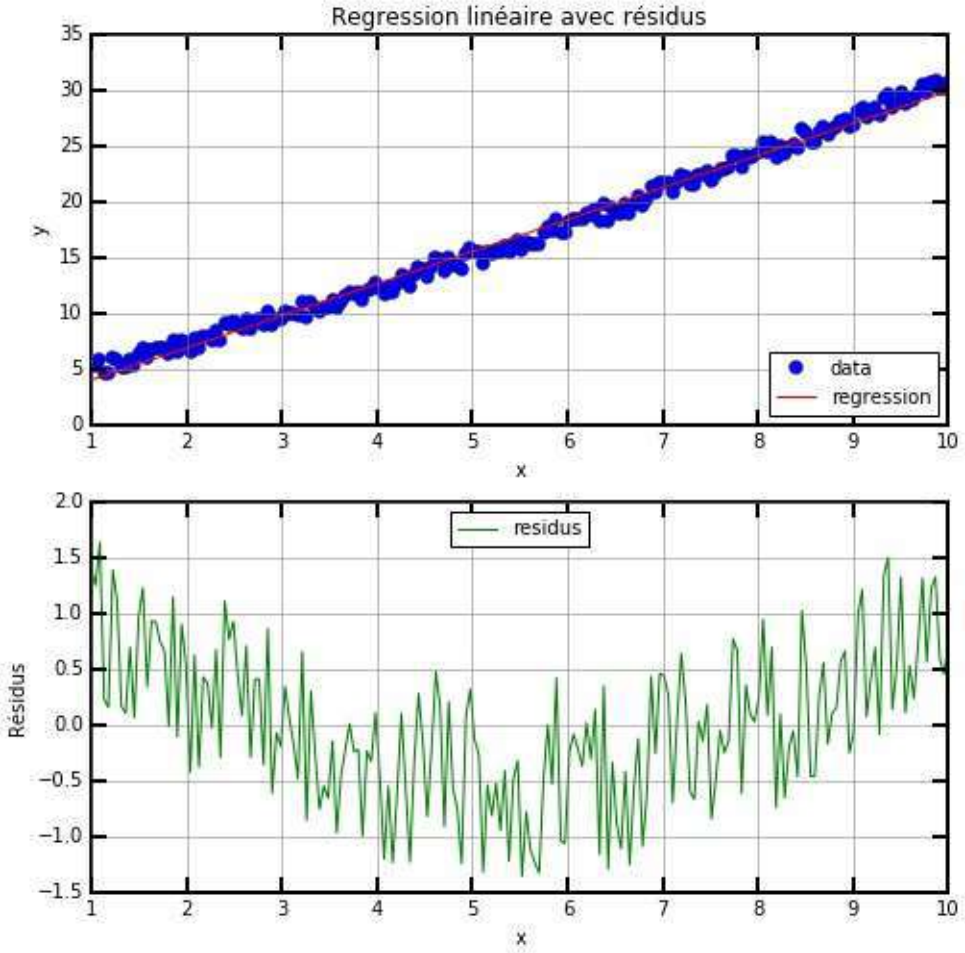
À vous de jouer !

Voici un exemple de graphique que vous pouvez obtenir à la fin de la question 5. :



Vous pouvez aussi ajouter le calcul des résidus, notament dans le cas d'un grand nombre de points. On peut alors représenter les données expérimentales et les résidus. On rappelle que les résidus sont les écarts entre les points expérimentaux et le modèle obtenu par la méthode des moindres carrés :

$$R_k = y_k - f(x_k) = y_k(a * x_k + b)$$



La correction de ce problème est disponnible sous la forme d'un notebook jupyter.

À la découverte de notre ADN

L'objectif de ce TP est de dmettre en pratique le langage python pour construire et analyser un brin d'ADN. L'idée est, dans un premier temps, de reprendre les éléments de base du langage (condition, boucles ...) pour créer des fonctions qui construise un brin d'ADN, le lisent, réalise une transcription une traduction ... puis d'écrire une classe qui réalise ces actions.

Introduction

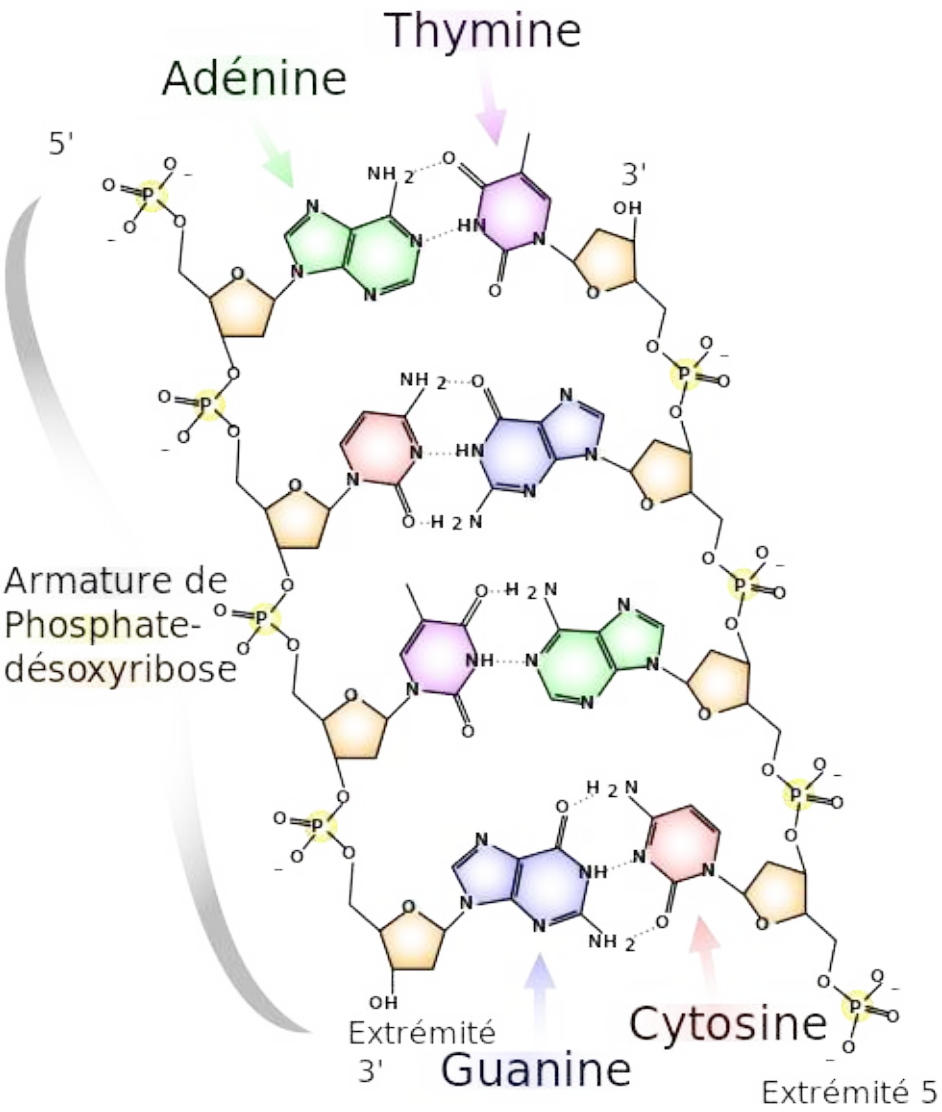
Ce n'est pas l'objet de faire ici un cours complet sur l'ADN. Vous trouverez de nombreuses choses sur le sujet. On va juste rappeler quelques éléments de base pour qu'un *non biologiste* puisse faire le TP.

ADN ?

L'ADN, pour Acide DésoxyriboNucléique, est une macromolécule constituée de deux brins qui forme une double hélice maintenue par des liaisons hydrogène. Ces brins sont formés par un enchainement de maillons appelés, nucléotides qui contiennent les *bases* de l'ADN :

- A pour Adénine
- T pour Thymine
- G pour Guanine
- C pour Cytosine

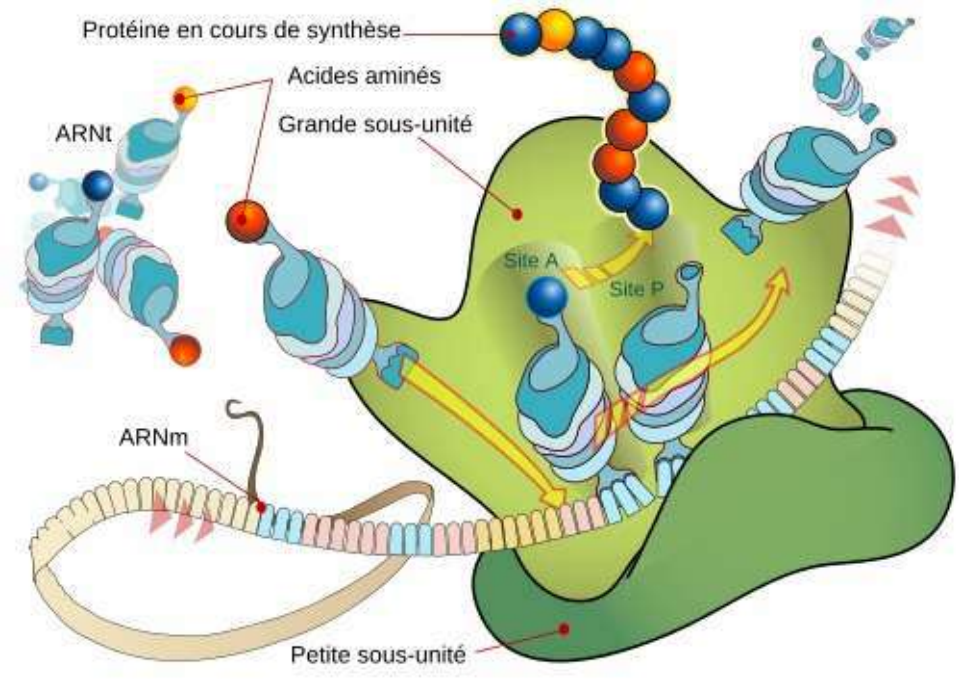
Les bases de l'ADN fonctionnent par paire, une sur chaque brin : adénine avec thymine et guanine avec cytosine.



Traduction et transcription

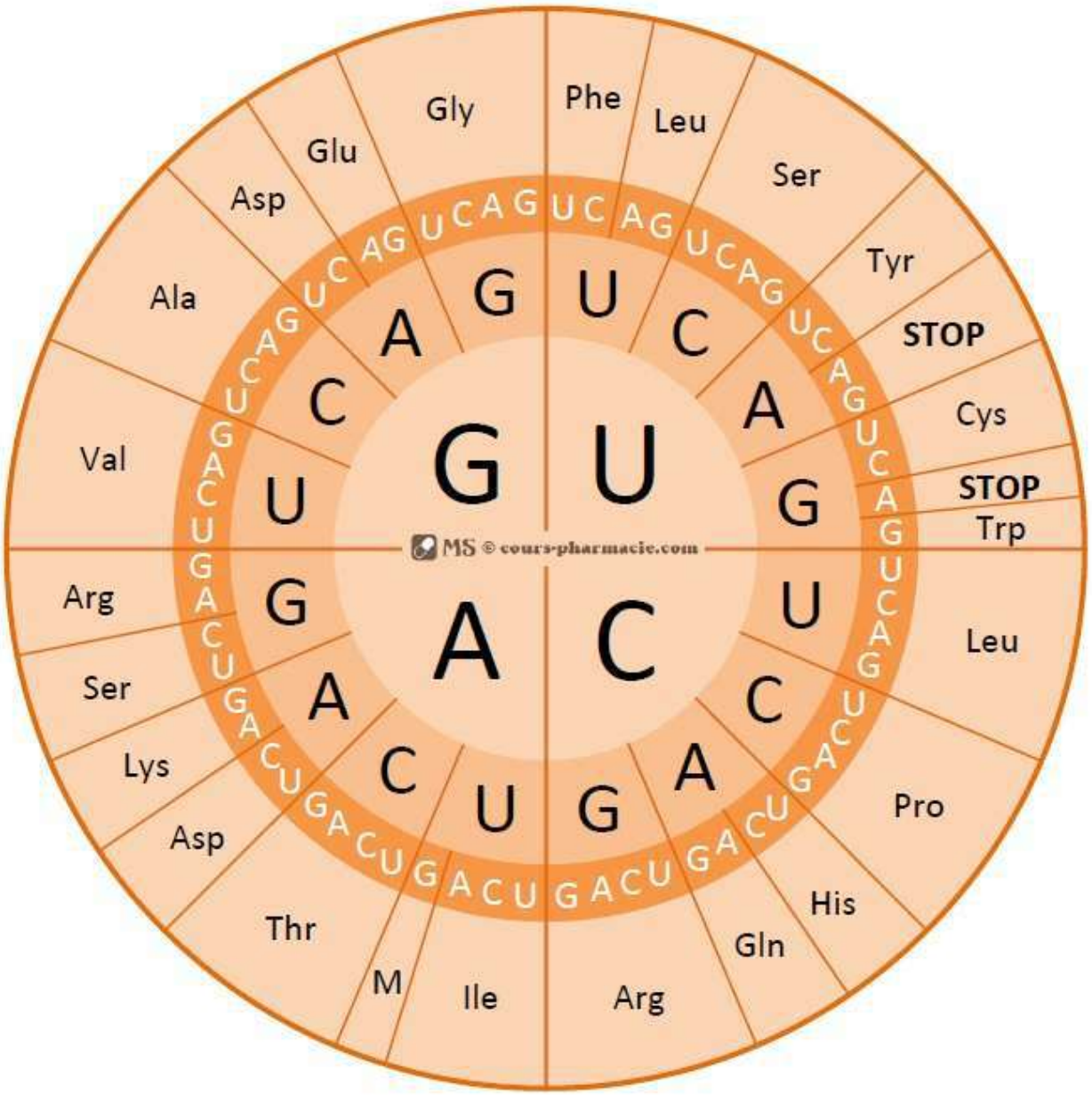
La transcription est un mécanisme qui permet de "recopier" l'ADN dans le noyau de la cellule pour former un ARN (acide ribonucléique) qui sera utilisé dans la cellule notamment lors de la traduction. L'ARN présente la même structure que l'ADN mais lors de la transicription, la thymine (T) est remplacé par l'uracile (U).

La traduction de l'ADN consiste à lire l'ARN issue de la transcription pour synthétiser une protéine avec l'aide de la machinerie cellulaire. L'ARN est découpé en codons qui sont constitués de 3 bases et correspondent à un acide aminé, c'est le code génétique. Les codons sont lus les uns à la suite des autres et la protéines est assemblée comme une chaîne (peptidique) d'acides aminés.



Correspondance codons - acides aminés

Le schéma ci-dessous vous donne la correspondance entre un codon, composé de trois bases de l'ARN et un acide aminé.



Par exemple, GUA est un codon qui code pour l'acide aminé Val, c'est à dire la Valine. On remarquera que plusieurs codons peuvent coder pour le même acide aminé ce qui limite la portée des erreurs de copies ou des mutations. On note également la présence de codons STOP indiquant la fin de la partie "codante" et qui stoppe la synthèse de la protéine.

Les dictionnaires python

Pour entrer le code génétique dans python on pourra utiliser les dictionnaires. Ces objets python sont un peu comme des listes ils contiennent plusieurs autres objets. À la différence des listes, les éléments d'un dictionnaire sont repérés par une clef et non par un indice. On peut utiliser tout type de clef : des nombres, des chaînes de caractères ou même des tuples.

Petit aperçu

```
>>> aa = dict()
>>> aa
{}
```

Une liste est délimitée par des crochets, un dictionnaire par des accolades. Par contre, comme pour les listes, La clef est donnée entre crochets :

```
>>> aa["M"] = "Met"
>>> aa["L"] = "Leu"
>>> aa["A"] = "Ala"
>>> print("dictionnaire : ", aa)
>>> print("un élément   : ", aa["L"])
dictionnaire :  {'A': 'Ala', 'L': 'Leu', 'M': 'Met'}
un élément   :  Leu
```

Remarque : Les dictionnaires ne sont pas ordonnés. On voit sur l'exemple ci-dessus que bien que la clef "A" ait été ajoutée en dernier, elle apparait en premier dans le dictionnaire.

On peut lister les `clefs` et les `valeurs` d'un dictionnaire ou les deux et les parcourir avec une boucle `for` .

Liste des clefs :

```
>>> print(aa.keys())
>>> for key in aa:
>>>     print(key)
dict_keys(['A', 'L', 'M'])
A
L
M
```

Pour les valeurs :

```
print(aa.values())
for val in aa.values():
    print(val)
dict_values(['Ala', 'Leu', 'Met'])
Ala
Leu
Met
```

On peut aussi parcourir les deux simultanément :

```
print(aa.items())
for key, val in aa.items():
    print(key, " = ", val)
dict_items([('A', 'Ala'), ('L', 'Leu'), ('M', 'Met')])
A  =  Ala
L  =  Leu
M  =  Met
```

`aa.items()` retourne une liste de tuple de la forme `(clef, valeur)`.

Petit exercice

1. Construire un dictionnaire qui met en relation le code à une lettre et le code à trois lettre des acides aminés. Voir ce tableau.
2. Construire un dictionnaire qui pour un acide aminé donné donne plusieurs informations :

```
* Code à 1 lettre,
* Code à 3 lettres,
* Polarité
* Masse
* pI
* ...
```

Conseil : Les dictionnaires sont des objets permettant de structurer des données. C'est un modèle simpliste de base de données. Il est tout à fait possible de les imbriquer.

Pour le code génétique

Pour le code génétique on peut envisager plusieurs solutions :

- la clef est un tuple correspondant au codon :

```
>>> gencode = {("A", "U", "A"): "Ile", ("U", "G", "A"): "STOP"}
>>> print(gencode)
{('U', 'G', 'A'): 'STOP', ('A', 'U', 'A'): 'Ile'}
```

- la clef est un acide aminé, la valeur est la liste des codons associés

```
>>> gencode = {"Phe": ["UUU", "UUC"], "Met": ["AUG"]}
>>> print(gencode)
{'Met': ['AUG'], 'Phe': ['UUU', 'UUC']}
```

Questions

L'idée est d'écrire dans un premier temps des fonctions qui réalisent les opérations suivantes puis d'écrire une classe qui contient les même fonctionnalités.

1. Écrire une fonction qui génère aléatoirement un brin d'ADN. On pourra choisir aléatoirement un codon STOP pour terminer le brin ou la partie codante.
2. Écrire une fonction qui écrit le brin d'ADN dans un fichier
3. Écrire une fonction qui lit un brin d'ADN dans un fichier
4. Identifier s'il s'agit d'un brin d'ADN ou d'ARN et si ce brin est valide
5. Statistique : extraire les informations suivantes d'un brin d'ADN
    - Nombre total de bases
    - Nombre de codons
    - pourcentage de chaque base dans le brin
6. Écrire une fonction qui réalise la transcription de l'ADN en ARN
7. Écrire une fonction qui traduit l'ARN et renvoie la chaîne d'acides aminés correspondante. Attention, elle doit s'arrêter au codon STOP.
8. Statistique. Extraire des statistiques sur les acides aminés
    - Le nombre d'acides aminés
    - Le pourcentage d'acide aminé polaire
    - Le nombre de chaque acide aminé différent

Programmation

À vous de jouer !

Commencez par vous exercer sur les dictionnaires avec l'exercice proposé puis vous pourrez transcrire et traduire votre ADN. Introspection garantie, à la fin du travail vous vous connaitrez sur le bout des doigts !

La correction de ce problème est disponnible sous la forme d'un notebook jupyter.