1 Introduction

Python est un langage de programmation et permet donc de créer ses propres fonctions ou programmes. Cependant ce langage ne serait pas très utile s'il fallait soi-même reconstruire toutes les fonctions utilisées couramment en programmation. Le succès de Python parmi les informaticiens a eu pour conséquence la mutualisation du travail de chacun et la mise à disposition d'un grand nombre de fonctions nouvelles regroupées dans des modules.

1.1 Où trouver un module ?

• Python propose près de deux cents modules, chacun susceptible de contenir jusqu'à plusieurs dizaines de fonctions. Il est donc exclu de les connaître tous. Une documentation très précise, contenant les codes sources et des liens pour télécharger ces modules est disponible sur le site officiel de Python :

https ://docs.python.org/3/library/index.html

• À cette adresse, vous pouvez télécharger et installer sur votre ordinateur de nouveaux modules. Ils se présentent sous la forme de fichiers contenant la programmation en Python des nouvelles fonctions disponibles.

• Cependant la plupart des modules que vous allez utiliser cette année sont déjà présents dans l'environnement Pyzo.

Remarque : Dès qu'un fichier contient la définition de plusieurs fonctions, il porte le nom de module. Un ensemble de plusieurs modules est appelé une bibliothèque. En toute rigueur, on devrait donc parler des bibliothèques : **math, numpy...** au lieu d'utiliser le terme module. Le terme anglais correspondant est library.

1.2 Comment importer un module ?

• Il y a plusieurs façons de faire appel à un module au cours de notre utilisation de Python. Prenons l'exemple du module **math** dans lequel se trouvent certaines constantes et fonctions mathématiques.

```
>>>import math
#on charge le module math, on peut utiliser à présent toutes les fonctions du module math ainsi :
>>>math.cos(14)
0.13673721820783322
```

• On peut aussi utiliser un alias pour raccourcir les notations, c'est la méthode que nous adopterons cette année :

```
>>>import math as m
#on charge le module math, on peut utiliser à présent toutes les fonctions du module math ainsi :
>>>m.sqrt(2)
1.4142135623730951
```

• On peut aussi importer une seule fonction d'un module :

```
>>>from math import sqrt #on importe que la fonction racine carrée
>>>sqrt(3)
1.7320508075688772
```

• Une dernière option consiste à importer toutes les fonctions du module **math** sous leur nom d'origine. Cette option semble au premier abord la plus attrayante mais présente un inconvénient majeur lorsque plusieurs fonctions de différents modules portent le même nom. C'est alors les fonctions du dernier module ayant été importé qui vont primer.

```
>>>from math import *
#on charge le module math, on peut utiliser à présent toutes les fonctions du module math ainsi :
>>>log(3) #correspond à ln(3)
1.0986122886681098
```

1.3 Comment connaître le contenu d'un module ?

• La commande *dir* permet d'avoir accès à la liste des objets présents dans un module.

```
>>>import time as t #on importe le module time
>>>dir(t)
['_STRUCT_TM_ITEMS', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'altzone', 'asctime',
    'clock', 'ctime', 'daylight', 'get_clock_info', 'gmtime', 'localtime', 'mktime', 'monotonic', '
    perf_counter', 'process_time', 'sleep', 'strftime', 'strptime', 'struct_time', 'time', 'timezone', '
    tzname', 'tzset']
# on obtient la liste des fonctions de ce module
```

• La commande *help* permet également d'avoir accès à tous les détails relatifs à chacune de ces fonctions. Au fur et à mesure vous connaîtrez par coeur le nom des fonctions que vous utilisez fréquemment ainsi que les paramètres de ces fonctions.

```
>>>help(t.localtime)
Help on built-in function localtime in module time:

localtime (...)
    localtime([seconds]) -> (tm_year,tm_mon,tm_mday,tm_hour,tm_min,
                             tm_sec,tm_wday,tm_yday,tm_isdst)

    Convert seconds since the Epoch to a time tuple expressing local time.
    When 'seconds' is not passed in, convert the current time instead.

#il n'est pas toujours simple de s'y retrouver, le mieux est d'essayer la fonction :

>>>t.localtime()
time.struct_time(tm_year=2021, tm_mon=10, tm_mday=2, tm_hour=13, tm_min=56, tm_sec=21, tm_wday
    =5, tm_yday=275, tm_isdst=1)
```

1.4 Créer son propre module

• Il est possible de créer votre propre module de fonctions. Il suffit de créer un fichier .py à enregistrer dans votre répertoire de travail personnel dans lequel vous allez écrire vos fonctions.

• Vous pouvez à présent importer votre module avec les mêmes commandes que n'importe quel autre module. On veillera, dans le cas d'un module de fonctions, à donner un nom différent au module de ceux des fonctions qui le composent.

• Si toutefois l'importation ne fonctionne pas, il vous faudra parfois spécifier le chemin à suivre pour récupérer votre module. Par exemple, si je souhaite utiliser la fonction *tri_bulles* du module *tri* :

```
from os import chdir
chdir("/Users/lionelchaussade/Desktop/MPSI(11-21)/MPSI(21-22)/ITC/Cours/Cours2 (boucles
    imbriquées)")
#le chemin où se trouve ce fichier sur mon ordinateur
import tri as t #le chemin précisé est automatiquement suivi
t.tri_bulles([2,3,1])
```

• Une autre possibilité si jamais votre module n'est pas trouvé est d'utiliser la commande *run file as script* au lieu de *execute file* dans le menu *run*.

2 Quelques modules intéressants

Voyons à présent les principaux modules que nous utiliserons cette année, vous ne devez pas apprendre par coeur les fonctions les composant mais vous devez savoir comment vous renseigner sur une fonction et comprendre sa documentation.

2.1 Le module math

Ce module contient les fonctions mathématiques et les constantes usuelles.
Voici une liste non exhaustive de fonctions présentes dans ce module.

```
>>>import math as m

>>>m.sqrt(2) #racine carrée
1.4142135623730951
>>>m.pi
3.141592653589793
>>>m.e
2.718281828459045
>>>m.cos(m.pi)
-1.0
>>>m.sin(m.pi)
1.2246467991473532e-16 #toujours des erreurs d'arrondis
>>>m.exp(2)
7.38905609893065
>>>m.floor(4.3) #la partie entière
4
>>>m.fabs(-3.6) #la valeur absolue
3.6
>>>m.factorial(8) #factorielle
40320
>>>m.log(4) #la fonction ln
1.3862943611198906
>>>m.pow(2,m.sqrt(2)) # pow(a,b) renvoie a^b
2.665144142690225
```

Les fonctions *Arccos*, *Arcsin*, *Arctan*, *ch*, *sh*, *th* sont aussi présentes dans ce module.

2.2 Le module random

Le module **random** propose diverses fonctions permettant de générer des nombres pseudo-aléatoires. Il apparaît difficile d'écrire un algorithme qui soit réellement non déterministe, c'est-à-dire qui produise un résultat totalement imprévisible. Il existe cependant des techniques mathématiques permettant de simuler plus ou moins bien le hasard. Voici quelques fonctions de ce module, vous trouverez les autres en fouillant la documentation du module.

```
>>>import random as rd

>>>rd.randint(3,9) #randint(a,b) choisit au hasard un entier entre a (compris) et b (compris).
6

>>>rd.random() #choisit aléatoirement un nombre flottant dans [0,1]
0.2533938036000326

>>>rd.choice(L) #choisit aléatoirement un élément de la liste L
```

2.3 Le module time

Le module **time** permet notamment de mesurer la durée d'exécution d'un programme. La fonction la plus utile est la fonction *time*(), voici un exemple d'utilisation :

```
import time as t

def fact(n):
    a = t.time() #on sauvegarde le temps avant l'éxécution de la fonction
    p = 1
    for k in range(2, n + 1):
        p = p * k #la fonction calcule la factorielle de n
    b = t.time() #le temps à la fin du calcul
    return(b - a) #b-a sera alors le temps mis pour le calcul
```

2.4 Le module numpy

• Le module **numpy** permet de créer et de manipuler des tableaux de nombres et des matrices et de leur appliquer des opérations mathématiques courantes. Contrairement aux listes qui peuvent contenir des éléments de types différents, un tableau est composé de nombres uniquement. L'avantage des tableaux par rapport aux listes est que l'on peut effectuer avec les tableaux de nombreuses opérations : +, ∗, ∗∗, cos, racine carrée... Voici quelques exemples :

```
>>>import numpy as np
>>>A = np.array([1, 2, 3]) # création d'un tableau numpy
>>>B = np.array([4, 5, 3])
>>>A + B
array([5, 7, 6]) #on somme les éléments à la même place
>>>A ** B #les éléments de A à la puissance ceux de B
array([ 1, 32, 27])
>>>np.exp(A) #l'exponentielle des éléments de A
array([ 2.71828183, 7.3890561 , 20.08553692])
```

• Il est également possible de créer des tableaux multidimensionnels, comme des matrices :

```
>>>A = np.array([[1, 2, 4], [4,  5,  3]])
>>>A
array ([[1,  2,  4],
        [4,  5,  3]])  #Python présente le résultat sous la forme d'une matrice
>>>A[1, 2] # renvoie l'élément de la ligne 1 et de la colonne 2 sachant que l'on numérote à partir de 0
3
```

• Voici quelques fonctions utiles du module numpy :

```
>>>import numpy as np

>>>np.linspace(a, b, n) # crée un tableau numpy de n valeurs régulièrement espacées entre a et b (inclus)

>>>np.zeros(p) #crée un tableau de taille p rempli de zéros

>>>np.zeros((p, q)) #crée un tableau à p lignes et q colonnes rempli de 0

>>>np.array(L) #transforme la liste L en tableau numpy
```

• Dans ce module, il y a également de nombreuses opérations portant sur les matrices (inverse, produit, déterminant, résolution de systèmes...).

2.5 Divers

Il existe de très nombreux modules, que nous rencontrerons parfois cette année, dont nous ne détaillerons pas les fonctions qui y figurent ici, on peut citer :

• le module **turtle** qui permet de dessiner en programmant

• le module **pygame** qui permet de créer des jeux

• le module **sys** qui permet d'avoir accès et de modifier certains paramètres de Python

• le module **fractions** qui permet de calculer sur les nombres rationnels en évitant les erreurs d'arrondis.

3 Graphiques en Python

3.1 Graphe d'une fonction

• Pour tracer une courbe en Python, il y a plusieurs étapes à effectuer :

- ▶ définir la fonction que l'on veut tracer,
- ▶ donner la liste des abscisses,
- ▶ donner la listes des ordonnées correspodantes,
- ▶ construire la figure,
- ▶ afficher la figure.

• En effet, tracer une courbe revient à relier des points du plan. Plus le nombre de points pris en abscisses est important plus la courbe sera tracée avec précision. Voici un exemple où l'on trace une fonction sur l'intervalle $[10, 11]$.

```
1  import matplotlib.pyplot as plt #le module qui sert pour le tracé de courbes
2  import math as m
3
4  def f(x): # on définit la fonction que l'on veut tracer
5      return(m.cos(x ** 2) - 5 * x + 1)
6
7  X = [10 + i * 0.1 for i in range(11)] # on crée une liste de 10 abscisses entre 10 et 11
8  Y = [f(u) for u in X] # la liste des ordonnées correspondantes, c'est-à-dire les valeurs prises par la
       fonction
9
10 plt.plot(X,Y) # la figure est construite
11 plt.show() # on affiche la figure
```

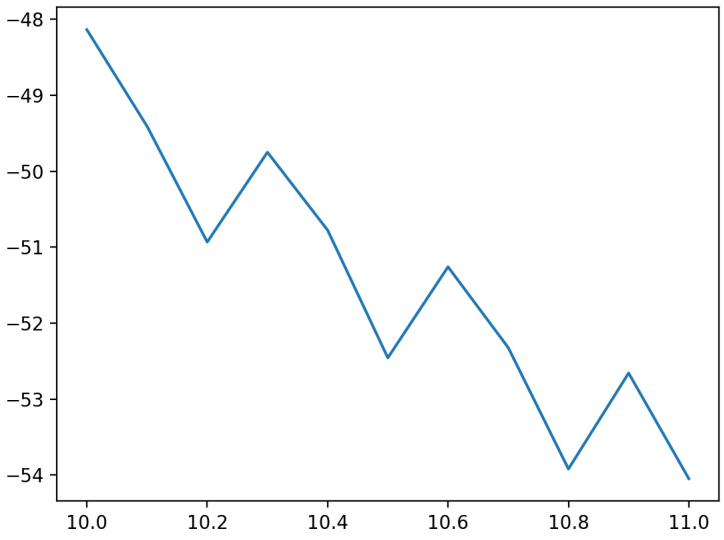
• À ce moment-là, le programme est interrompu jusqu'à ce que l'on referme la fenêtre graphique. C'est pour cela, qu'en général, l'instruction *plt.show*() est la dernière instruction du programme.

• La façon normale de procéder par étapes est donc la suivante : on exécute un programme terminé par *plt.show*(), on observe le graphique, on referme la fenêtre graphique, on modifie le programme, on le réexécute...

• La fenêtre graphique contient des boutons pour déplacer le graphique et zoomer.

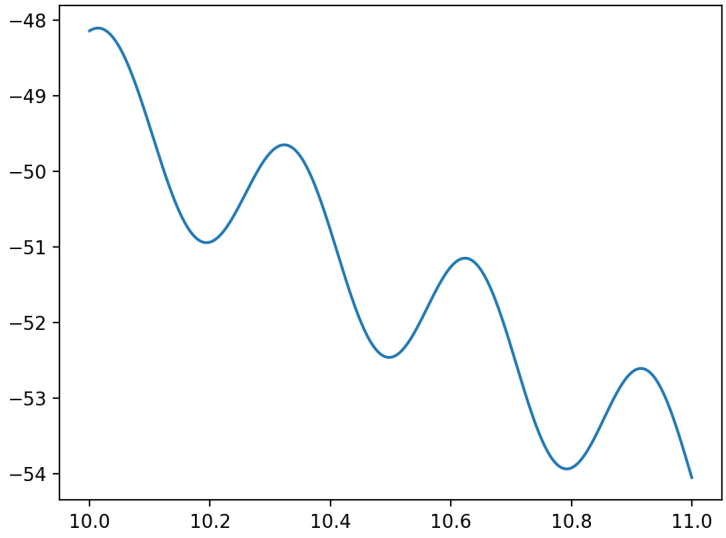
• On peut enregistrer le graphique dans un fichier avec le bouton d'enregistrement dans la fenêtre graphique.

• Voici ce que l'on obtient ici :



C'est un peu saccadé par nous n'avons pris que 10 points en abscisses, on peut changer cela en prenant par exemple 10 000 réels en abscisse et en remplaçant la ligne 7 par :

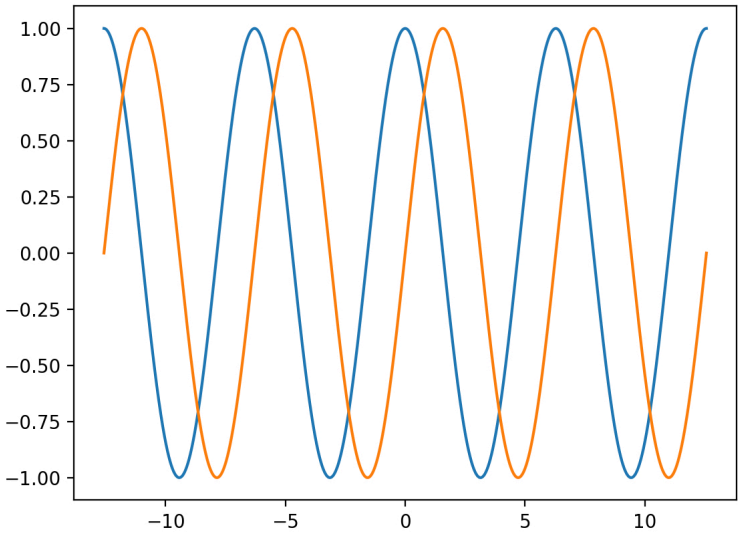
```
1  X = [10 + i * 0.0001 for i in range(10001)] # on crée une liste de 10000 abscisses entre 10 et 11
```



• Il peut être intéressant d'utiliser le module numpy pour créer rapidement des listes d'abscisses et tracer des fonctions simples. Voici un exemple :

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

X = np.linspace(−4 * np.pi, 4 * np.pi, 100000) # tableau numpy de 100000 valeurs entre −4pi et 4pi
Y = np.cos(X) #on crée le tableau des cosinus de ces valeurs
Z = np.sin(X) #on crée le tableau des sinus de ces valeurs
plt.plot(X, Y)
plt.plot(X, Z) #on construit les deux courbes
plt.show() #on affiche simultanément ces courbes
```



• Voici quelques commandes qui permettent de mettre en forme un graphique, elle ne sont bien sûr pas à connaître par coeur :

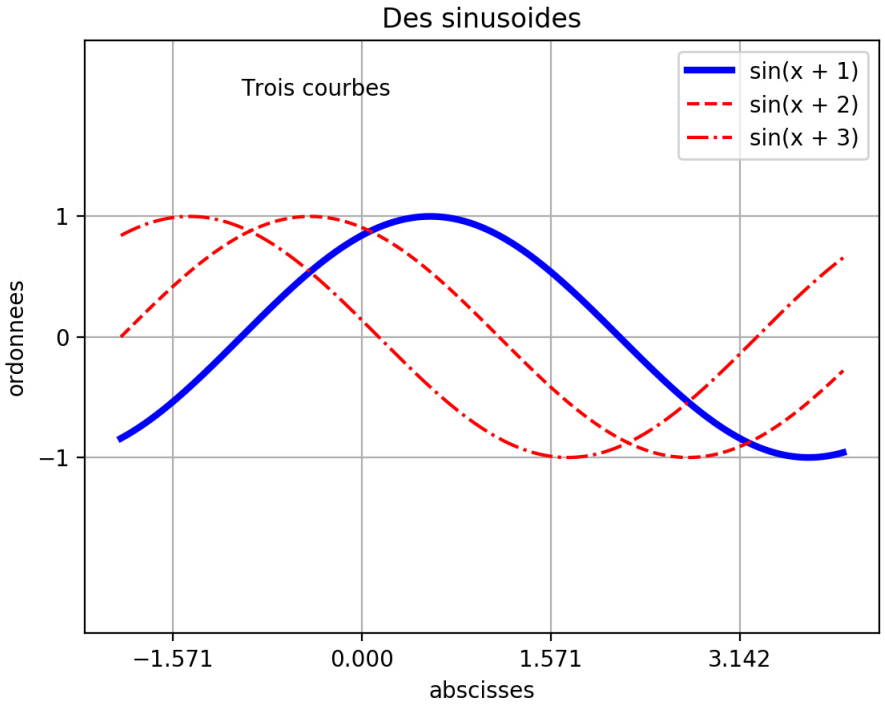
```
import matplotlib.pyplot as plt
import numpy as np

X = np.linspace(-2, 4, 1000) #liste des abscisses entre -2 et 4 avec 1000 points

plt.axis('equal') #repère orthonormé
plt.plot(X, np.sin(X + 1), color='blue', linewidth=3, linestyle='-' , label='sin(x + 1)')
# on gère ainsi la couleur, l'épaisseur du trait, le style du trait et la légende

plt.plot(X, np.sin(X + 2), color='red', linestyle='--', label='sin(x + 2)')
plt.plot(X, np.sin(X + 3), color='red', linestyle='-.', label='sin(x + 3)')

plt.legend(loc='upper \ right') #localisation de la légende donnée par l'instruction label
plt.xlabel('abscisses') #légende pour l'axe des abscisses
plt.ylabel('ordonnees') #légende pour l'axe des ordonnées
plt.title('Des sinusoides') #titre de la figure
plt.text(-1, 2, 'Trois courbes') #texte sur le graphique avec son positionnement
plt.xticks([-np.pi / 2, 0, np.pi / 2, np.pi]) #abscisses qui vont figurer dans le graphique
plt.yticks([-1, 0, 1]) #ordonnées qui vont figurer dans le graphique
plt.grid(True) #ajout d'une grille
plt.show()
```



3.2 Tracer un histogramme

Il est également possible de tracer des histogrammes, voici un exemple :

```
L = [2, 2, 6, 6, 3, 14, 14, 6, 8, 9, 9] # on souhaite tracer l'histogramme qui correspond à la liste L

import matplotlib.pyplot as plt

plt.hist(L, range=(0, 15), bins = 15)
#range=(0, 15) signifie que les abscisses vont de 0 à 15
#bins = 15 signifie que l'on regroupe les valeurs en 15 intervalles (de longueur 1 comme on va de 0 à 15)
plt.show()
```