TP 2 : Voyage en Pythonerie

Objectifs :

- Utilisation des modules standards.
- Révisions sur les tracés de courbes.
- Utilisation des modules d'ingénierie scientifique.

Outre l'enseignement de notions importantes d'informatique théorique, l'informatique pour tous permet de présenter un certain nombre d'outils d'ingénierie scientifique. Ces outils sont utiles dans les autres disciplines (mathématiques, sciences physiques, chimie et sciences de l'ingénieur), soit pour faciliter certaines études, soit pour les rendre possible lorsqu'on est confronté à des problèmes n'ayant pas de solution analytique. Ce TP a pour objectif de présenter un choix de modules intéressant pour résoudre rapidement les problématiques scientifiques nécessitant l'usage d'outils informatiques.

I Modules "Batteries included"

Un grand nombre de modules sont inclus dans l'installation de base du langage Python. La liste de ces modules et leurs documentations sont disponibles à l'adresse suivante : `https://docs.python.org/3/library/index.html`

Remarque 1. *Il est préférable d'importer les modules avec la syntaxe suivante :* `import module as mod` *Une fonction du module s'appelera alors dans le programme par* `mod.fonction()`*. Cette syntaxe évitera les confusions entre fonctions de différents modules ayant le même nom. Une fois le module importé, une aide est disponible dans la console en tapant* `help('module')`*, ou* `help('module.fonction')`*.*

1.1 Le module math

Le module `math` permet d'importer les fonctions et constantes mathématiques usuelles.

Commande Python	Constante/Fonction mathématique
pi, e, exp(x), log(x), log(x, a)	$\pi, e = exp(1), exp(x), ln(x), log_a(x)$
pow(x, y), floor(x), abs(x), factorial(n)	$x^y, \lfloor x \rfloor, \lvert x \rvert, n!$
sin(x), cos(x), tan(x), asin(x),...	fonctions trigonométriques
sinh(x), cosh(x), tanh(x), asinh(x),...	fonctions hyperboliques

1.2 Le module random

Ce module propose diverses fonctions permettant de générer des nombres pseudo-aléatoires qui suivent différentes distributions mathématiques. On parle de nombres pseudo-aléatoires car il est assez difficile d'écrire un algorithme qui soit réellement non-déterministe (c'est à dire qui produise un résultat totalement imprévisible), mais ce module permet de simuler assez bien l'effet du hasard. Voici quelques fonctions de ce module.

`random.randrange(p, n ,h)`	choisit un entier aléatoirement dans `range(p, n, h)`
`random.randint(a, b)`	choisit un entier aléatoirement dans l'intervalle $[a, b]$
`random.choice(seq)`	choisit un entier aléatoirement dans la séquence `seq`
`random.random()`	renvoie un décimal aléatoire dans $[0, 1[$
`random.shuffle(seq)`	mélange les éléments dans la séquence `seq`

1.3 Le module time

Ce module présente des fonctions permettant de faire des mesures de durées d'exécution (voir TP1) ou des pauses dans l'exécution du programme.

`time.time()`	Renvoie le temps en secondes depuis epoch (1^{er} janvier 1970) au format flottant
`time.sleep(secs)`	suspend l'exécution du programme pendant une durée de `secs` secondes (format flottant possible)

1.4 Des modules pour faire des calculs exacts

L'usage en informatique des flottants induit des erreurs d'arrondis dès les premiers calculs lorsqu'on utilise des décimaux ou des fractions. Si on souhaite manipuler ces nombres sans erreur, deux modules peuvent s'avérer utiles.

1.4.1 Le module decimal

Du fait du stockage des flottants en base 2 par l'ordinateur, le simple test suivant :
`(0.1 + 0.1 + 0.1) == 0.3` renvoie `False`.
Le module decimal peut éviter ce problème.

```
>>> import decimal as dec
>>> dec.Decimal('0.1') + dec.Decimal('0.1') + dec.Decimal('0.1')
Decimal('0.3')
>>> dec.Decimal('0.1') + dec.Decimal('0.1') + dec.Decimal('0.1') == dec.Decimal('0.3')
True
```

Par défaut, la précision du module decimal est de 28 décimales. On peut afficher la précision en cours et la modifier de la manière suivante.

```
>>> dec.getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999, capitals=1,
clamp=0, flags=[], traps=[InvalidOperation, DivisionByZero, Overflow])
>>> dec.getcontext().prec = 100  # Fixe la nouvelle précision à 100 décimales
```

1.4.2 Le module fractions

Ce module permet de manipuler sans erreur des nombres rationnels.

```
>>> import fractions as f
>>> 1 / 3 + 2 / 5
0.7333333333333334
>>> f.Fraction(1, 3) + f.Fraction(2, 5)
Fraction(11, 15)
>>> _ - f.Fraction(6, 15)
Fraction(1, 3)
>>> float(_)
0.3333333333333333
```

1.5 Le module csv

Le format "Comma-separated values" est un format informatique ouvert représentant des données tabulaires sous forme de "valeurs séparées par des virgules" ; c'est le format le plus couramment utilisé pour importer ou exporter des données d'une feuille de calcul d'un tableur. Un fichier `csv` est un fichier texte, dans lequel chaque ligne correspond à une rangée du tableau ; les cellules d'une même rangée sont séparées par une virgule. Á partir d'un tableur, il suffit de sauvegarder une feuille de calcul en précisant le format `csv` pour fabriquer un tel fichier.

Remarque 2. *Pour éviter les désagréments liés à l'absence de standardisation du séparateur décimal, il est fortement conseillé de paramétrer son tableur pour choisir le point (et non la virgule) comme séparateur décimal. Avec LibreOffice, il faut aller dans le menu Outils > Options > Paramètres linguistiques > Langue > Environnement Linguistique et choisir une combinaison langue/pays utilisant le point comme séparateur décimal, comme "français (Suisse)" par exemple (penser modifier alors Monnaie par défaut).*

Pour ouvrir un fichier avec Python, on importe le module `csv` et on utilise la syntaxe suivante.

```
import csv
with open('Approximations_pi_eleve.csv', newline = '') as f:
        lire = csv.reader(f)
        for ligne in lire :
                print(ligne)
```

Pour écrire dans un fichier, on utilise l'instruction `open` avec l'option `'w'` pour écrire (et dans ce cas la méthode `writer` va écraser le fichier s'il existe déjà) ou `'a'` pour ajouter du contenu aux lignes suivantes.

```
with open('Approximations_pi_votreNom.csv', 'w', newline = '') as f:
        ecrire = csv.writer(f)
        for ligne in tableau_donnees :
                ecrire.writerow(ligne)
```

Projet 1. *Approximation de π*

On considère les quatre suites définies pour $n \in \mathbb{N}$:

$$u_n = \sum_{k=0}^{n} \frac{1}{(k+1)^2}, \qquad v_n = \sum_{k=0}^{n} \frac{1}{(k+1)^6} \qquad w_n = \sum_{k=0}^{n} \frac{(-1)^k}{2k+1}, \qquad r_n = \frac{\sqrt{8}}{9801} \sum_{k=0}^{n} \frac{(4k)!(1103+26390k)}{(k!)^4 396^{4k}}$$

Leonhard Euler (Suisse, 1707-1783) a mis en évidence le fait que $\lim_{n\to+\infty} u_n = \frac{\pi^2}{6}$. On a également $\lim_{n\to+\infty} v_n = \frac{\pi^6}{945}$ et James Gregory (Écosse, 1638-1675) a démontré que $\lim_{n\to+\infty} w_n = \frac{\pi}{4}$. La dernière formule est due à Srinivasa Ramanujan (Inde, 1887-1920) qui l'a fournie sans aucune démonstration, on a $\lim_{n\to+\infty} r_n = \frac{1}{\pi}$.

1. *Programmer une fonction* `u_n` *qui prend en entrée un entier n et qui renvoie la valeur de u_n, puis une fonction* `pi1` *qui renvoie une estimation de la valeur de π à partir de u_n. Créer de même trois fonctions* `pi2`*,* `pi3` *et* `pi4` *afin d'estimer la valeur de π à partir de trois autres fonctions calculant v_n, w_n et r_n.*

   Remarque 3. *Ne pas oublier de renseigner la "docstring" des fonctions.*

   On va comparer les approximations de π ainsi obtenues. Les fichiers d'installation de Python contiennent déjà une approximation de π la plus précise possible compte tenu de la limitation des flottants. La valeur approchée de π est obtenue avec le module `math` *par* `math.pi`*.*
2. *Programmer une fonction qui renvoie le nombre de décimales correctes après la virgule d'une estimation de π. Elle aura pour argument la valeur d'une estimation de π.*
   Un fichier `Approximations_pi_eleve.ods` *est dans le répertoire :*
   `Informatique\Eleves\votreClasse\TP2`*.*
   Il fournit la structure d'un tableau pour afficher les résultats de ces comparaisons.
3. *Enregistrer le fichier* `Approximations_pi.ods` *sous le format* `csv`*. Récupérer les données de ce fichier dans un programme.*
4. *Calculer les termes à afficher dans ce tableau, puis enregistrer vos valeurs dans un fichier* `Approximations_pi_votreNom.csv` *dans un répertoire de vos documents. Ouvrir ce dernier fichier avec le tableur LibreOffice et constater le résultat.*

II Tracés de courbes avec Python

Le package `matplotlib.pyplot` permet de tracer les courbes. Dans toute la suite, on considère qu'on a importé ce module ainsi :

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

On importe également le module `numpy` pour avoir accès aux fonctions mathématiques sur les tableaux :

```
import numpy as np
```

2.1 Tracé d'une courbe simple

Soit par exemple une fonction :

```
def f(x) :
        return (np.sin(x) / (1 + x ** 2))
```

Pour tracer cette courbe en fonction de x, il faut créer une liste de points où l'on désire évaluer la fonction. L'instruction `linspace` permet de répartir la liste de manière linéaire entre les deux bornes avec le nombre de points souhaités. Dans l'exemple ci-dessous le nombre de points est égal à 200.

```
x1 = np.linspace (0, 8 * pi, 200)
```

Il ne reste plus qu'à tracer la courbe.

```
plt.plot(x1, f(x1))
plt.show()
```

2.2 Tracés de verticale et horizontale

Pour tracer une verticale pour un x donné, il faut créer une liste de points en x où x est constant et créer une liste en y avec ses bornes de variations. Il faut que les listes soient de même taille, soit par exemple 200 points. Dans l'exemple ci-dessous, x varie de [5,5] et y varie de [0,10].

```
xv = np.linspace (5, 5, 200)
yv = np.linspace (0, 10, 200)
```

Il ne reste plus qu'à tracer la courbe.

```
plt.plot(xv, yv)
plt.show()
```

Exercice 1.
Faire de même pour tracer une horizontale $y = 5$ pour les mêmes bornes de valeurs qui seront notées `(xh, yh)`.

2.3 Tracé de courbes par superposition

Pour superposer des courbes, il faut mettre `plt.show()` à la fin de l'ensemble des tracés.

```
plt.plot(xv, yv)
plt.plot(xh, yh)
plt.show()
```

2.4 Des options de mise en forme des graphiques

On reprend l'exemple du premier tracé. Mais à présent, on gère la taille de l'image, on ajoute un titre, une légende pour les deux axes, une grille et on définit la limite du rendu en x.

```
# On gère la taille de la figure (x, y) en cm
plt.figure(figsize=(6, 4))
# Ajout d'un titre
plt.title("Courbe de y = f( x )")
# Légende pour l'axe des abscisses
plt.xlabel("x")
# Légende pour l'axe des ordonnées
plt.ylabel("y")
# Ajout d'une grille
plt.grid(True)
# Limite le rendu à la plage [0, 8 pi] en abscisse
plt.xlim([0, 8 * pi])
# On change la couleur, l'épaisseur, le style et l'étiquette
plt.plot(x1, f(x1), color = "red", linewidth = 2.5, linestyle ="?", label = "f(x)")
# On localise l'étiquette
plt.legend(loc = 'upper right')
plt.show()
```

La figure est alors beauoup plus lisible. Le titre, les légendes des axes, le quadrillage augmente fortement la plus value.
Les options pour le dessin des courbes sont les suivantes :

- `color = 'color'`, color = couleur du trait. Les couleurs standards sont :
  white, black, red, green, blue, cyan, magenta, yellow.
- L'épaisseur du trait est défini avec `linewidth = z`, z = épaisseur du trait en pt.
- Le style du trait est défini avec `linestyle = 'ls'`.
  Voici les différents styles :

: → ligne en pointillés	- → ligne pleine (défaut)	* → pas de courbe
– → ligne en tirets	-. → alternance tiret-point	

- Le nom de la figure est défini avec `label = 'nomdelafigure'`.
- La localisation de l'étiquette est définie avec `plt.legend(loc = 'localisation')`.
  Voici les différentes localisations :

`upperleft` ou 2	`uppercenter` ou 9	`upperright` ou 1
`centerleft` ou 6	`center` ou 10	`centerright` ou 5 ou 7
`lowerleft` ou 3	`lowercenter` ou 8	`lowerright` ou 7

On peut aussi marquer certains points de la courbe associée. Il suffit de changer la valeur de `linestyle` et de choisir une liste de points plus petite. On dispose aussi des options `markersize`, `markerfacecolor`, `markeredgecolor` et `markeredgewidth` pour modifier la taille, la couleur, la couleur du bord et la largeur du bord des marqueurs.
Dans l'exemple ci-dessous, la courbe possède initialement 300 points. On choisit de mettre en évidence uniquement 15 points uniformément répartis.

```
# Choix de l'ensemble des points où on évalue la fonction
x2 = np.linspace (0, 8 * pi, 15)
plt.plot (x2, f(x2), linestyle = '*', marker = 'o', markersize = 15,
markerfacecolor = 'cyan', markeredgecolor = 'blue', markeredgewidth = 2)
...
plt.show()
```

Les options pour le dessin des courbes sont les suivantes :

- Le type du marqueur est défini avec `marker = 'ma'`, `ma` étant le type de marqueur.
  Voici les différents types de marqueurs :

o → boulette	+ → plus	. → point	s → carré
x → croix	* → étoile	∧ → triangle	

- La taille x du marqueur avec `markersize = x`.
- La couleur `color` du marqueur avec `markerfacecolor = 'color'`.
- La couleur `color` du bord du marqueur avec `markeredgecolor = 'color'`.
- La largeur y du bord du marqueur avec `markeredgecolor = y`.

On utilise alors le principe de superposition précédemment défini.

2.5 Tracés en échelle logarithmique

Exemple de tracé en échelle logarithmique décimal sur l'axe des abscisses.

```
x3 = np.linspace(1, 100)
def g(x) :
        return(x)
# En échelle log décimal pour les abscisses
plt.semilogx(x3,g(x3))
plt.show()
# En échelle linéaire pour les abscisses
plt.plot(x3, g(x2))
plt.show()
```