Rappels sur le langage Python : typage dynamique et importation

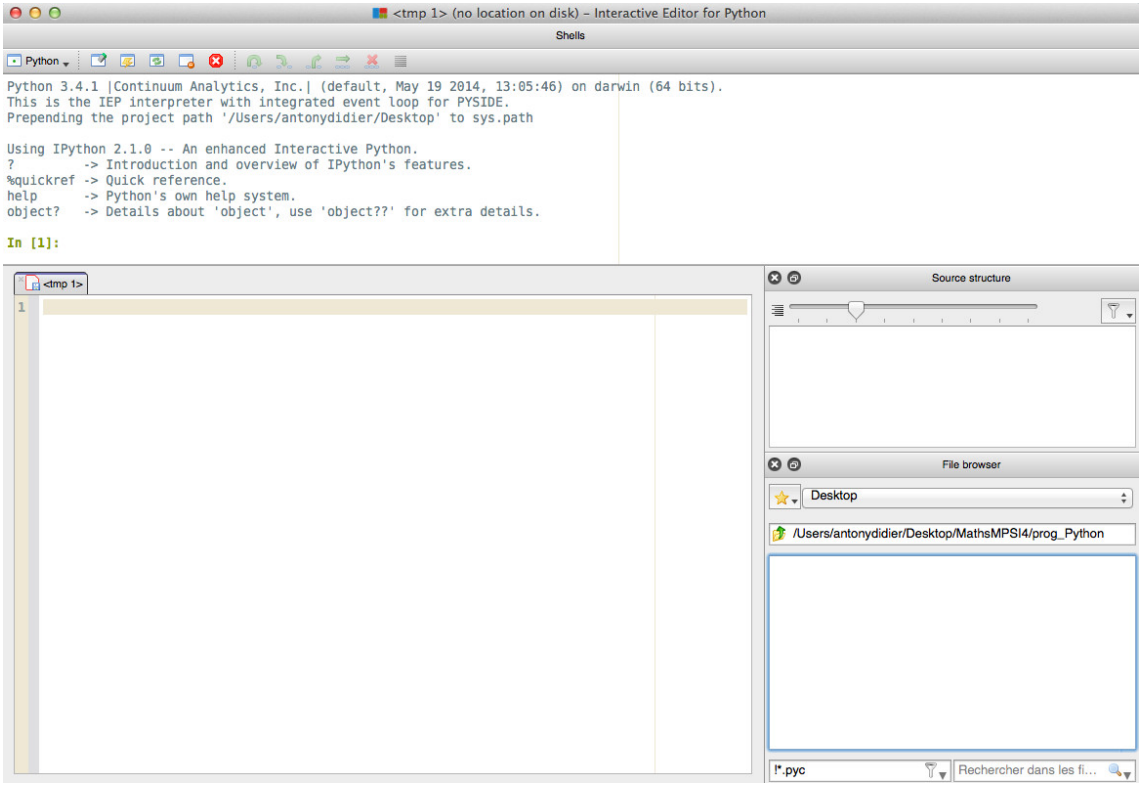
Ces dernières années, vous avez dû apprendre un nouveau langage de programmation avec ses particularités : le langage Python. Si celui-ci est assez accessible de par sa simplicité de programmation, nous reviendrons ici sur quelques notions fondamentales afin de faire tourner des exemples classiques sur machine.

1 Présentation de l'interface et premier exemple

Au cours de cette année, nous n'apprendrons pas à programmer des séquences d'instructions sous un format binaire, mais nous privilégierons le langage **Python**, un **langage de haut niveau** parmi d'autres : ce type de langage représente en fait un ensemble de mots-clefs qui définissent un code compréhensible par chacun d'entre nous et qui sera interprété par la machine avant d'être exécuté. Si la programmation semble aisée, cela nécessite d'abord :

1. d'installer le langage Python sur la machine (par exemple sur www.anaconda.com)
2. puis d'utiliser un logiciel **interpréteur** (par exemple sur www.pyzo.org) qui nous facilitera l'édition et l'interprétation de nos instructions, et ceci afin que les tâches soient effectuées par l'ordinateur.

Concrètement, l'environnement de travail se présentera de la façon suivante :



dans lequel on pourra distinguer :

- un module d'édition qui nous permet de construire nos différents scripts d'instructions dans le langage Python ;
- une console interactive qui permet d'exécuter certaines commandes.

La plupart du temps, on cherchera donc à construire nos programmes dans l'éditeur sous la forme de **fonctions**, c'est à dire qu'à partir de données formelles appelées aussi **arguments**, le programme renverra un ou plusieurs **résultats**, puis ces fonctions seront appelées dans la console.

Remarque Dans la construction de nos scripts, il faudra veiller à la **structure physique** de nos programmes et les sauts de ligne ou indentation seront indispensables. D'ailleurs, on fera souvent appel à des blocs d'instructions classiques :

$$\begin{cases} \text{les boucles conditionnelles } \texttt{if, elif, elif..., else} \\ \text{les boucles itératives } \texttt{for...} \quad \texttt{ou while...} \end{cases}$$

Exercice 1 (résolution d'une équation du second degré à coefficients réels). [ ]
On considère l'équation $ax^2 + bx + c = 0$, où $(a, b, c) \in \mathbb{R}^3$ et $a \neq 0$.

Dans le langage Python, construire un programme $racines(a : float, b : float, c : float) \to tuple$ calcule le discriminant associé, puis renvoie les racines éventuellement complexes de l'équation.

Remarque Il y aura toujours plusieurs façons de construire un programme, mais on préfèrera des instructions compréhensibles à l'économie des lignes. Cela facilite le debugage et c'est cette attitude qu'il faudra privilégier tout au long de l'année... on pourra même songer à renseigner nos fonctions en précisant par exemple :

- le type des variables utilisées dans la définition de la fonction,
- une description sommaire de la fonction à placer en préambule et entre guillemets """ ..."",
- des commentaires bien placés à l'aide du symbole #,
- tester quelques conditions en amont ou en aval à l'aide de la commande `assert`.

On pourra ensuite lire le fichier d'aide associé à l'aide de la commande `help`.

Exercice 1 (résolution d'une équation du second degré à coefficients réels). [ ]
On considère l'équation $ax^2 + bx + c = 0$, où $(a, b, c) \in \mathbb{R}^3$ et $a \neq 0$.

Dans le langage Python, construire un programme $racines(a : float, b : float, c : float) \to tuple$ calcule le discriminant associé, puis renvoie les racines éventuellement complexes de l'équation. *On veillera à renseigner la fonction ainsi construite.*

2 Typage dynamique et importation de librairies additionnelles

Le langage Python est un langage léger : il ne nécessite pas de **déclarer** au préalable les variables qui seront utilisées : on parle de **typage dynamique**, c'est à dire que pour chaque variable, l'interpréteur identifie sa **classe** à l'assignation et ceci en fonction des commandes utilisées. On fera donc attention aux différentes classes des objets mais aussi à leurs **attributs** (des propriétés propres à la classe de l'objet) et aux nombreuses **méthodes associées** (des fonctions particulières définies pour les objets d'une même classe).

Par exemple, vous avez déjà travaillé avec :

- les **objets booléens** du type `bool` et qui ne prennent que deux valeurs `True` ou `False`. Ces valeurs nous permettent de vérifier des conditions logiques qu'elles soient simples ou multiples.
- les **nombres entiers** : ce sont les objets du type `int` et parmi les opérations associées les plus courantes, il y a :

commande Python	interprétation
`x//y`	renvoie le quotient de la division euclidienne de x par y
`x%y`	renvoie le reste de la division euclidienne de x par y
`range(x)`	renvoie la liste des entiers $[0, 1, \dots, x-1]$

- les **nombres réels** : ce sont les objets du type `float` et parmi les opérations associées les plus courantes, il y a :

commande Python	interprétation
`round(x)` ou `round(x,n)`	renvoie la valeur arrondie de x à 10^{-n} près
`abs(x)`	renvoie la valeur absolue de x

  ainsi que toutes les opérations usuelles.

Et si on veut aller plus loin, il sera parfois nécessaire d'importer des **librairies additionnelles** : il s'agit généralement de modules externes qui contiennent des fonctions déjà programmées. Ici, nous prendrons souvent l'habitude d'importer la librairie **math** ajoutant les fonctions mathématiques usuelles qui ne sont pas intégrées par défaut :

```
In : import math
```
python

On peut alors obtenir une rapide description des fonctions données :

```
In : help(math)

Help on module math:
(...)
FUNCTIONS
acos(...)
acos(x)
Return the arc cosine (measured in radians) of x.
(...)
```
python

On retiendra en particulier le nom des fonctions usuelles ainsi que la donnée des constantes e et π :

`acos`, `acosh`, `asin`, `asinh`, `atan`, `atanh`, `ceil`, `cos`, `cosh`, `degrees`, `exp`, `fabs`, `factorial`, `floor`, `fsum`, `log`, `log10`, `sin`, `sinh`, `radians`, `sqrt`, `tan`, `tanh` et `e`, `pi`

Par contre, ces fonctions sont associées à la librairie importée et cela exige de préfixer les fonctions par le nom du module ou d'un alias bien choisi... Alors, pour simplifier nos scripts, on préfèrera, quand c'est possible, importer le module d'une autre façon :

```
In : from math import *; sqrt(169)
                                    13.0
```
python

Au fil de l'année, on sera donc amené à travailler avec différents modules en fonction de nos besoins et on pourra d'ores et déjà retenir l'existence des librairies suivantes :

- le module **cmath** pour travailler sur les nombres complexes,
- le module **random** permettant de générer des nombres pseudos-aléatoires,
- le module **time** pour gérer l'horloge interne du système,
- les modules **pylab** ou **matplotlib** pour représenter les fonctions usuelles ou des objets géométriques,
- le module **numpy** pour travailler sur les tableaux,
- le module **scipy** apportant de nombreuses méthodes d'approximation pour la résolution de problèmes mathématiques.

3 Le cas particulier des données structurées

Certaines classes nous permettront de gérer des objets constitués eux-mêmes d'autres objets, on parle plutôt de **données structurées** ou de **conteneurs**.
L'avantage de ces objets, c'est que tous les termes contenus seront numérotés de 0 à la longueur-1, et on distinguera :

- **les n-uplets du type** *tuple*
  Ils représentent la classe la plus courante et leur définition est implicite puisqu'il suffira de juxtaposer les éléments à l'aide d'une virgule, avec des parenthèses ou non.

- **les ensembles du type** *set*
  Ils représentent une classe très pratique car on y retrouve toutes les opérations ensemblistes usuels. On pourra construire de tels ensembles à l'aide d'accolades ou tout simplement par la commande de conversion `set`.

- **les chaînes de caractères du type** *string*
  Elles sont construites soit en utilisant la fonction de conversion `str`, soit à l'aide de guillemets simples ' ' ou doubles " ", selon la présence d'apostrophes dans la chaîne donnée :

```
In : s,t= 'sans apostrophe!',"avec l'apostrophe"; type(s); type(t)
                                        <class 'str'>
                                        <class 'str'>
                                                                    python
```

- **les listes du type** *list*
  Les **listes** permettent de représenter des séquences modifiables. C'est le type que l'on préférera manipuler tant les méthodes sont nombreuses. On pourra définir une telle liste de trois façons :

  1. en convertissant une variable de type *tuple* grâce à la commande `list`,
  2. en complétant la liste au fur et à mesure dans un programme itératif avec une boucle `for` ou `while`,
  3. en décrivant la séquence contenue dans la liste. On parle alors de **liste par compréhension** pour laquelle les éléments sont exprimés en fonction de l'indice associé :

```
In : L = [k**2 for k in range(0,11)]; L
                        [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
                                                                    python
```

  Et en plus, on pourra même y ajouter des **instructions conditionnelles** :

```
In : L = [k**2 for k in range(0,11) if k%2 == 0]; L
                        [0, 4, 16, 36, 64, 100]
                                                                    python
```

  Contrairement aux n-uplets, on pourra cette fois-ci en modifier le contenu et on fera bien entendu attention à l'indexation car les éléments sont toujours numérotés de 0 jusqu'à la longueur de la liste obtenue par la commande `len` -1 :

```
In : L = list(range(11)); L[5] = 0; print(L); len(L)
                        [0, 1, 2, 3, 4, 0, 6, 7, 8, 9, 10]
                                        11
                                                                    python
```

  Si de plus, on cherche à supprimer un élément, on pourra toujours le faire au moyen de la commande `del` :

```
In : del(L[2:5]); L
                        [0, 1, 0, 6, 7, 8, 9, 10]
                                                                    python
```

  En fait, la donnée de deux indices sous la forme `i:j` nous permet d'extraire les éléments d'une liste, mais on veillera à bien comprendre que l'élément d'indice i est toujours inclus, alors que l'élément d'indice j est exclus.

  Pour finir, on présente ici quelques opérations déjà rencontrées dans des programmes précédents :

commande Python	interprétation
`x in L` ou `x not in L`	teste si x appartient ou non à la liste L
`L1 + [x]`	ajoute l'élément x à la liste $L1$
`L1 + L2`	renvoie la concaténation des listes $L1$ et $L2$
`L * n`	renvoie la concaténation de n-fois la liste L
`max(L)` ou `min(L)`	renvoie le maximum ou le minimum de la liste L

Cette classe possède des fonctions plus spécifiques encore, appelées aussi **attributs**, et qui permettent d'obtenir des propriétés liées à l'objet lui-même :

commande Python	interprétation
`L.index(x)`	renvoie le premier indice de l'élément qui vaut x
`L.count(x)`	renvoie le nombre d'occurrences de x dans L

De la même façon, on retrouvera cette notation pointée dans l'utilisation des **méthodes**. Ce sont des fonctions qui opèrent sur la liste donnée :

commande Python	interprétation
`L.append(x)`	ajoute l'élément x à la fin de la liste L
`L1.extend(L2)`	ajoute à la fin de $L1$ les éléments de $L2$
`L.insert(i,x)`	insère au rang i l'élément x
`L.remove(x)`	supprime la première occurrence de x dans L
`L.reverse()`	permet de retourner la liste L en inversant les éléments
`L.sort()`	permet d'ordonner la liste L

Remarque Ces commandes existent, mais encore une fois, on ne vous demande pas de toutes les connaître mais plutôt de les reconstruire.

4 Applications

Ces premiers exemples ont pour seul but de vous aider à mieux appréhender l'environnement et la syntaxe du langage Python.

Exercice 2 (approximation de e et comparaison de la complexité en temps). [ ]
On rappelle que le **développement en série entière de l'exponentielle** nous donne par exemple :

$$e = \sum_{k=0}^{+\infty} \frac{1}{k!}$$

1. (a) Dans le langage Python, construire la fonction itérative $facto(n : int) \to int$ qui renvoie la valeur de $n!$.
   (b) En déduire le programme $somme1(n : int) \to float$ renvoie la valeur de $S_n = \sum_{k=0}^{n} \frac{1}{k!}$ à l'aide de votre fonction $facto$.
   (c) De la même façon, construire le programme $somme2(n : int) \to float$ qui renvoie la valeur de S_n sans votre fonction $facto$, mais en calculant $k!$ à partir de l'itération précédente.
2. Importer le module `time`, puis définir la fonction $comparaison(\epsilon : float) \to list$ qui calcule les valeurs de S_n par les programmes $somme1$ et $somme2$, et renvoie les temps de calcul $[t1, t2]$ nécessaires pour que $|\mathrm{e} - S_n| \leq \epsilon$.

Exercice 3 (suite de Syracuse et temps de vol). [ ]
On définit la **suite de Syracuse** par :

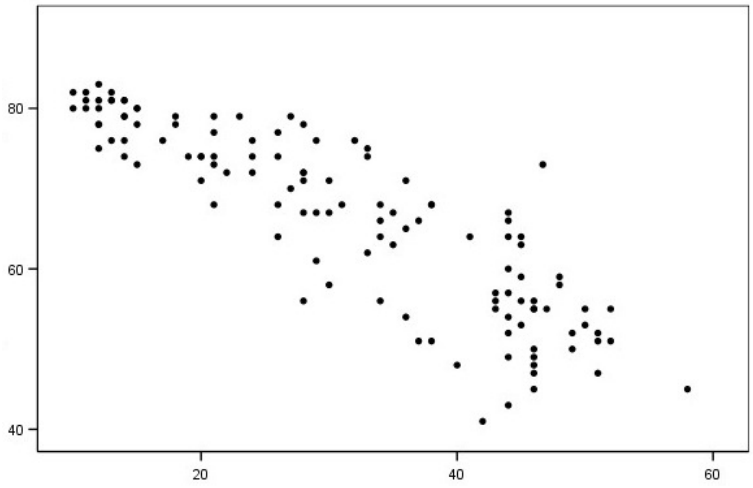
$$u_0 \in \mathbb{R} \text{ et pour tout } n \in \mathbb{N},\ u_{n+1} = \begin{cases} \frac{u_n}{2}, & \text{si } u_n \text{ est pair} \\ 3u_n + 1, & \text{si } u_n \text{ est impair} \end{cases}$$

1. Dans le langage Python, écrire la fonction $syracuse(n : int, u_0 : float) \to float$ qui renvoie le n-ième terme de la suite.
2. Construire le programme $syraliste(u_0 : float) \to list$ qui calcule les termes de la suite tant que $u_n \neq 1$, puis renvoie la liste $[u_0, \ldots, 1]$ ainsi que la longueur de celle-ci, aussi appelé temps de vol.
3. Pour quelle valeur de $u_0 \in [\![2, 1000]\!]$ le temps de vol est-il le plus élevé ?

Exercice 4 (nuage de points et droite de régression linéaire). [ ]

Lorsqu'on étudie certains phénomènes physiques, on peut être amené à relier des paramètres, c'est à dire à chercher une relation de dépendance entre ces paramètres. Dans le cas le plus simple, quand les données relevées sur deux paramètres fournissent un nuage de points alignés, on pourra faire l'hypothèse que le modèle est affine et on cherchera naturellement à identifier la **droite de régression linéaire** qui s'approche au plus près de chacun des points.

Concrètement, si $X = [x_0, \dots, x_{n-1}]$ et $Y = [y_0, \dots, y_{n-1}]$ désignent les mesures effectuées pour deux paramètres physiques, le nuage de points associés sera constitué des points de coordonnées (x_k, y_k):



On se propose de construire une fonction qui nous permettra de déterminer l'équation de la droite de régression linéaire traduisant une relation affine de la forme $Y = aX + b$.
Dans les programmes suivants, on évitera de faire appel à la commande `sum`.

1. Construire une fonction *moyenne* qui pour une liste X donnée, renvoie la moyenne des coefficients définie par :
$$\overline{x} = \frac{1}{n}(x_0 + \ldots + x_{n-1})$$
2. Construire une fonction *covariance* qui pour deux listes X, Y données, renvoie la covariance de X et Y définie par :
$$cov(X,Y) = \frac{1}{n}\sum_{k=0}^{n-1}(x_k - \overline{x})(y_k - \overline{y})$$
On pensera à anticiper ici et imposer une pré-condition pour que les listes X et Y aient la même taille.
3. Construire une fonction *variance* qui pour une liste X donnée, renvoie la variance de X définie par :
$$var(X) = \frac{1}{n}\sum_{k=0}^{n-1}(x_k - \overline{x})^2$$
4. Construire alors la fonction principale *reglin* qui pour deux listes de mesures données X, Y, renvoie les coefficients a et b de la droite de régression linéaire, définis par :
$$a = \frac{cov(X,Y)}{var(X)} \text{ et } b = \overline{y} - a\overline{x}$$
puis affiche la représentation graphique de la droite d'équation $y = ax + b$ sur le segment $[m, M]$ où m, M désignent les valeurs minimales et maximales de la liste X.
*Pour cette représentation, on n'hésitera pas à importer les librairies **numpy** et **matplotlib**, et faire appel aux commandes `linspace` et `plot`, qui sont très efficaces pour construire des graphiques.*

Exercice 5 (recherche d'un plus grand élément et place de celui-ci). [ ]

On considère une liste L constituée de n nombres réels et une matrice $A \in \mathcal{M}_{np}(\mathbb{R})$.

1. Dans le langage Python, construire la fonction *chercherliste*$(L : list) \to tuple$ qui renvoie la valeur du plus grand élément de la liste L, ainsi que un indice correspondant à sa place.
2. De la même façon, construire la fonction *cherchermatrice*$(A : array) \to tuple$ qui renvoie la valeur du plus grand élément de la matrice A, ainsi que un couple d'indices correspondant à sa place.