TP Python 3

MESURES PHYSIQUES

Informatique Scientifique – version 2

2010-2011

Robert Cordeau & Laurent Pointal

Informatique :

Rencontre de la logique formelle et du fer à souder.

Maurice NIVAT

Remerciements :

La version 3.1 de ce document est issue d'une réorganisation suite à des critiques de Sabine MARDUEL et Ludovic CONDETTE.

Les versions précédentes avaient bénéficié des corrections attentives de Laurent POINTAL (LIMSI) et de Georges VINCENTS (IUT d'Orsay).

Merci également à Louis CAPRON (S_2, 2009) pour avoir corrigé une coquille.

>>> import this[1]

Préfère :

la beauté à la laideur,

l'explicite à l'implicite,

le simple au complexe

et le complexe au compliqué,

le déroulé à l'imbriqué,

l'aéré au compact.

Prends en compte la lisibilité.

Les cas particuliers ne le sont jamais assez pour violer les règles.

Mais, à la pureté, privilégie l'aspect pratique.

Ne passe pas les erreurs sous silence,

… ou bâillonne-les explicitement.

Face à l'ambiguïté, à deviner ne te laisse pas aller.

Sache qu'il ne devrait avoir qu'une et une seule façon de procéder,

même si, de prime abord, elle n'est pas évidente, à moins d'être Néerlandais.

Mieux vaut maintenant que jamais.

Cependant jamais est souvent mieux qu'immédiatement.

Si l'implémentation s'explique difficilement, c'est une mauvaise idée.

Si l'implémentation s'explique aisément, c'est peut-être une bonne idée.

Les espaces de nommage ! Sacrée bonne idée ! Faisons plus de trucs comme ça.

1 *The Zen of Python,* by Tim Peters (PEP 20), traduction Cécile Trevian et Bob Cordeau

Index des illustrations

Table des matières

Avant-Propos

Il ne faut pas se fier aux apparences sinon on n'aurait jamais mangé d'oursin…

Anonyme

0.1 -Structure des TP

Les TP qui accompagnent le cours d'« informatique scientifique en Python » se composent de huit séances de 4h (la 9e est consacrée au partiel). Ils sont tous organisés de la même façon :

- **manip** : expérimentation du thème de la séance ;
- **programmation** : écriture de scripts.

0.2 -En-tête du TP : préparation et objectifs

Un encadré au début du TP indique ce qu'il vous faut *préparer* afin de pouvoir faire le TP ; il liste entre autres les **notions de cours que vous devez réviser** avant de commencer le TP. Il définit aussi les *objectifs*, c'est-à-dire des points particuliers à étudier dans la séance. Les notions travaillées lors d'une séance sont indispensables pour aborder les séances suivantes. Si certaines notions ne sont pas comprises, n'hésitez pas à questionner l'enseignant chargé du TP ou du cours et à les retravailler en faisant les exercices ad-hoc dans le recueil d'exercices proposés par B.Cordeau.

L'en-tête définit également les fichiers à produire. Leurs noms sont normalisés afin d'être aisément reconnus : ils sont ainsi plus faciles à stocker, à consulter et… à corriger !

0.3 -Texte du TP

Il est **à lire attentivement** ! Le texte précise les *notions* à comprendre et expérimenter, les *algorithmes* à mettre en jeu et les sources à développer. On y trouve aussi des copies d'écran illustrant les sorties des programmes, des notes de bas de pages, des encarts, des rappels mathémathiques…

0.4 -Comment rendre ses exercices pour le prochain TP ?

À chaque séance sauf la dernière, vous devez rendre un exercice individuel.

On vous demande :

- de faire l'exercice sur machine et de le faire exécuter ;
- de ***rendre l'énoncé avec*** votre solution ;
- d'imprimer votre solution (avec l'extension `.py`) en utilisant *Wing* ou un autre environnement, mais ne jamais l'imprimer depuis un fichier texte (car on ne voit pas les indentations) ;
- d'indiquer vos résultats en fin de vos sources (voir Illustration 1: Modèle d'un script en page 11).

0.5 -Comment faire son compte-rendu de TP ?

Chaque binôme doit rendre, en fin de séance, un compte-rendu de ses deux parties de TP, « manip » et « scripts ».

La partie « manip » est rédigée en utilisant les modèles de compte-rendu disponibles pour chaque TP dans les répertoires `TP1`, `TP2`… `TP8` qui sont sur le bureau de l'ordinateur. Vous devez copier dans le modèle les sessions interactives Python (Python Shell, ou « mode calculette »), et y ajouter vos commentaires. Le compte rendu doit être imprimé et rendu en fin de séance.

La partie « scripts » est constituée par les programmes que vous aurez écrit. N'oubliez pas de copier/coller les résultats d'exécution à la fin (entre deux triple guillemets) s'il y a lieu. Les scripts doivent être imprimés et rendus en fin de séance.

0.6 -Où mettre vos fichiers ?

Sur le bureau de travail de votre compte **`infoX`**, sur les ordinateurs en salle de travaux pratiques, se trouvent 8 répertoires : **`TP1`**, **`TP2`**... **`TP8`**. Chacun de ces répertoires contient entre autres un fichier modèle **`.ott`** pour le compte-rendu du TP concerné.

À chaque séance vous devez enregistrer tous les fichiers que vous créez dans le répertoire TP correspondant :

- votre compte-rendu « manip»,
- vos scripts Python **`TPx`** (voir détails ci-dessous),
- les scripts Python dont vous aurez eu besoin,
- autres fichiers (texte…).

En fin de TP, n'oubliez pas d'imprimer votre compte-rendu (manip et scripts) puis de sauvegardez le répertoire **`TPx`** de votre travail sur une clé USB, car les comptes sont « nettoyés » à la fin de chaque séance et tout votre travail est supprimé pour le passage du groupe TP suivant.

0.7 -Le compte-rendu de la partie « Manip »

Ouvrez le fichier modèle de texte *OpenOffice Writer* **`.ott`** correspondant au TP et enregistrez le nouveau document ainsi créé sous le nom **`compte_rendu.odt`** dans le répertoire de travail portant le n° du TP. Remplissez, au début de ce compte-rendu, la partie d'identification (nom, prénom et groupe du binôme et date).

Ensuite, en plus du document compte-rendu TP, ouvrez simultanément l'application EDI *Wing IDE*. **Copier-coller périodiquement** votre travail effectué dans *Wing IDE* vers le compte-rendu dans *OpenOffice Writer* et mettez en forme le texte (utilisez les styles déjà définis dans le modèle). Ajoutez dans le compte-rendu des notes et commentaires pour expliquer ce que vous comprenez, n'oubliez pas d'effacer vos erreurs ou vos tâtonnements, etc. Enfin, **sauvegardez fréquemment** - certains étudiants ont déjà perdu leur travail faute d'avoir sauvegardé.

⚠ **Attention**

Ne pas oublier le plus important : (bis) **ajoutez des commentaires**, vous serez notés sur leur pertinence…

0.8 -Le compte-rendu de la partie « Programme »

Il est constitué par les impressions de vos scripts.

Tous vos scripts doivent être suffixés **`.py`** et être imprimés depuis *Wing* (surtout pas depuis *gedit*).

Les en-têtes de scripts doivent spécifier le nom du fichier correspondant ainsi que les auteurs.

De plus vous devez indiquer en fin de vos scripts les **résultats** que vous obtenez. Pour cela, faites une première exécution et copiez le résultat dans une chaîne multi-lignes comme indiqué sur l'Illustration 1: Modèle d'un script.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""Modèle de script.
Documentation du module.
"""
# fichier: modele_script.py
# auteur: Bob Cordeau

# imports
from math import pi

# ---------- Programme Principal ------------------------------------
r = float(input("Rayon de la sphère en mètre : "))

print("Volume de la sphère de rayon {:.1f} m : {:.2f} m3".format(
      r, 4*pi*r**3/3))

"""
Évaluation de model_script.py

Rayon de la sphère en mètre : 22.4
Volume de la sphère de rayon 22.4 m : 47079.59 m3
"""
```

Illustration 1: Modèle d'un script

L'utilité des différents éléments présents dans l'en-tête des fichiers scripts Python est décrite dans Illustration 5: En-tête d'un script en page 22.

0.9 -La documentation

Les documentations suivantes sont disponibles :

- **La documentation Python** : page d'accueil de firefox
- **Le cours, poly de TP, abrégé Python, reccueil d'exercices, etc** :
  répertoire **`/opt/docs/`**
- **Les utilitaires** : répertoire **`/opt/scriptsTP/`**

Vous avez par ailleurs normalement les copies papier des diapos des cours, avec vos notes personnelles, ainsi que l'impression couleur de l'Abrégé Dense Python 3.1 donnée lors du premier cours.

0.10 -Version de Python

Les TP sont basés sur l'utilisation de Python en version 3.1 ou ultérieure, disponible au téléchargement[2] sur le site www.python.org.

Pensez, dans l'environnement intégré Wing IDE, à spécifier la bonne version de Python (menu ***Editer***, ***Configurer Python…***, personnaliser l'***Exécutable Python*** en indiquant python3).

Si vous en avez besoin pour l'installation sur votre machine personnelle, la description de la configuration de Wing IDE est décrite sur la page http://perso.limsi.fr/pointal/python:wingide.

0.11 -Astuces pratiques

L'environnement de travail pour les TP est Linux Ubuntu. Il permet de disposer de plusieurs **bureaux virtuels**[3] entre lesquels on navigue très rapidement en utilisant les combinaisons de touche Ctrl-Alt-← et Ctrl-Alt-→. Ceci permet par exemple d'avoir, en plein écran, les fichiers sur un bureau, l'environnement de développement sur un autre, la documentation ou le sujet de TP en PDF sur un troisième, et le compte-rendu sur le quatrième.

Pour faire la **copie de textes**, outre les habituels raccourcis Ctr-X (couper), Ctrl-C (copier) et Ctrl-V (coller), il est possible d'utiliser la **copie rapide de textes sous Unix**, à savoir la sélection du texte à copier avec la souris, puis le clic avec le bouton/molette centrale à l'endroit où l'on veut coller le texte sélectionné - éventuellement dans une autre application ou une console.

Pour les codes que l'on vous demande de tester, il est possible de les copier/coller entre la version PDF de l'énoncé du TP et l'environnement de développement Wing IDE.

☞ **Remarque**

Récupération de votre travail : Pour sauvegarder votre travail tout au long des TP, il est **fortement conseillé** de posséder une clé USB, sur laquelle vous enregistrerez tous vos sources, c'est-à-dire tous les fichiers avec l'extension « `.py` », et votre « manip » avec l'extension « `.odt` ».

Mais vous ne devez récupérer votre travail qu'en **fin de séance**.

---

2 Et généralement disponible sous forme de paquet "python3" sous Linux.

3 Pour avoir la même fonctionnalité sous Windows, vous pouvez installer le logiciel virtuawin.

1 - Types, variables, tests

Le plus long des voyages commence par le premier pas.

Lao-Tseu

Préparation :

- les types de base : entiers, flottants, booléens, chaîne de caractères ;
- les variables ;
- les saisies et affichage ;
- les séquences d'instruction et l'alternative if.

Objectifs :

- utilisation d'un « Environnement de Développement Intégré » ;
- utilisation des types de base, des variables et de l'affectation ;
- création des premiers scripts de séquences d'instructions.

Fichiers à produire : `TP1_1.py` `TP1_2.py` `TP1_3.py` `TP1_4.py` `TP1_5.py`

N'oubliez pas de suivre les directives indiquées dans le chapitre Avant-Propos.

1.1 -Manip

1.1.1 - Interpréteur " Calculette "

Python est un *interpréteur* et on peut l'utiliser facilement pour expérimenter le langage en utilisant ce que l'on appelle le **shell Python**. Dans ce mode, Python fonctionne comme une calculatrice avec un ruban déroulant les opérations et les résultats ; vous serez amenés à l'utiliser régulièrement lors des travaux pratiques afin de tester rapidement certaines opérations.

Dans ce mode, le comportement de Python est le suivant :

- Il affiche un message puis une invite (en anglais un « prompt ») : `>>>`.
- Il attend qu'on lui soumette des commandes (par exemple : `2+3`).
- Lorsqu'une commande est validée par l'appui sur Entrée alors elle est évaluée et le résultat de cette évaluation est affiché (sauf s'il n'y a pas de résultat - valeur `None`).
- Si la commande est constituée d'une instruction sur plusieurs lignes, l'invite se transforme en `...` sur les lignes suivantes, jusqu'à ce que l'on saisisse une ligne vide pour indiquer que la commande est terminée et que l'on veut que l'exécution se fasse.
- Une fois une commande évaluée et son résultat affiché, l'interpréteur ré-affiche l'invite et attend la saisie d'une nouvelle commande.

Si vous n'avez aucune interface graphique installée, vous pouvez, dans un terminal ligne de commande, taper : `python3` puis appuyer sur la touche Entrée. Ceci démarre l'interpréteur Python en mode calculette, comme dans Illustration 2 ci-après. Il est alors possible de saisir des expressions Python pour voir quel est le résultat de leur évaluation. Pour sortir de l'interpréteur, il suffit de taper Ctrl+D. (Ctrl+Z sous Windows).

```
[laurent@litchi ~]$ python3
Python 3.1.2 (r312:79147, Apr 22 2010, 16:11:29)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+3
5
>>> 4/3*3.141459*2**3 # Volume d'une sphère de rayon 2.
33.50889599999999
>>>
```

Illustration 2: Interpréteur en ligne de commande

☞ **Remarque**

Dans les lignes de code Python, les **commentaires** sont introduits par le caractère **#** et s'étendent jusqu'à la fin de la ligne. Ils sont destinés à donner des indications aux personnes qui lisent le code, et sont ignorés par Python.

Lorsque l'on fait une erreur dans la ligne de code Python, l'exécution de celle-ci provoque ce que l'on appelle une « levée d'exception », qui par défaut fait afficher diverses informations décrivant l'erreur (cause, ligne où elle a été détectée…). Voici quelques exemples (il y a d'autres catégories d'erreur) :

```
>>> 3*4*(7-2))          # insertion d'une parenthèse fermante de trop
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
Syntax Error: 3*4*(7-2)): <string>, line 110
>>> 3.2/0               # division par zéro
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
builtins.ZeroDivisionError: float division
>>> "256"+4             # opération + entre des types incompatibles
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
builtins.TypeError: Can't convert 'int' object to str implicitly
>>> 4/3*pi*r**2         # utilisation de variables non définies
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
builtins.NameError: name 'pi' is not defined
```

Illustration 3: Levées d'exceptions suite à des erreurs

Python dispose d'Environnements de Développements Intégrés (« EDI » ou « IDE » en anglais) qui lui sont adaptés, et qui facilitent autant l'édition du code Python que les tests et l'exécution. Il est d'ailleurs généralement installé avec un EDI simple et écrit lui-même en Python : IDLE.

Dans ces TP, nous utiliserons un EDI libre pour le monde de l'éducation : Wing IDE (voir illustration 2), aussi bien pour pour la partie« Manip » des tests en mode calculette que pour écrire des scripts et les exécuter.

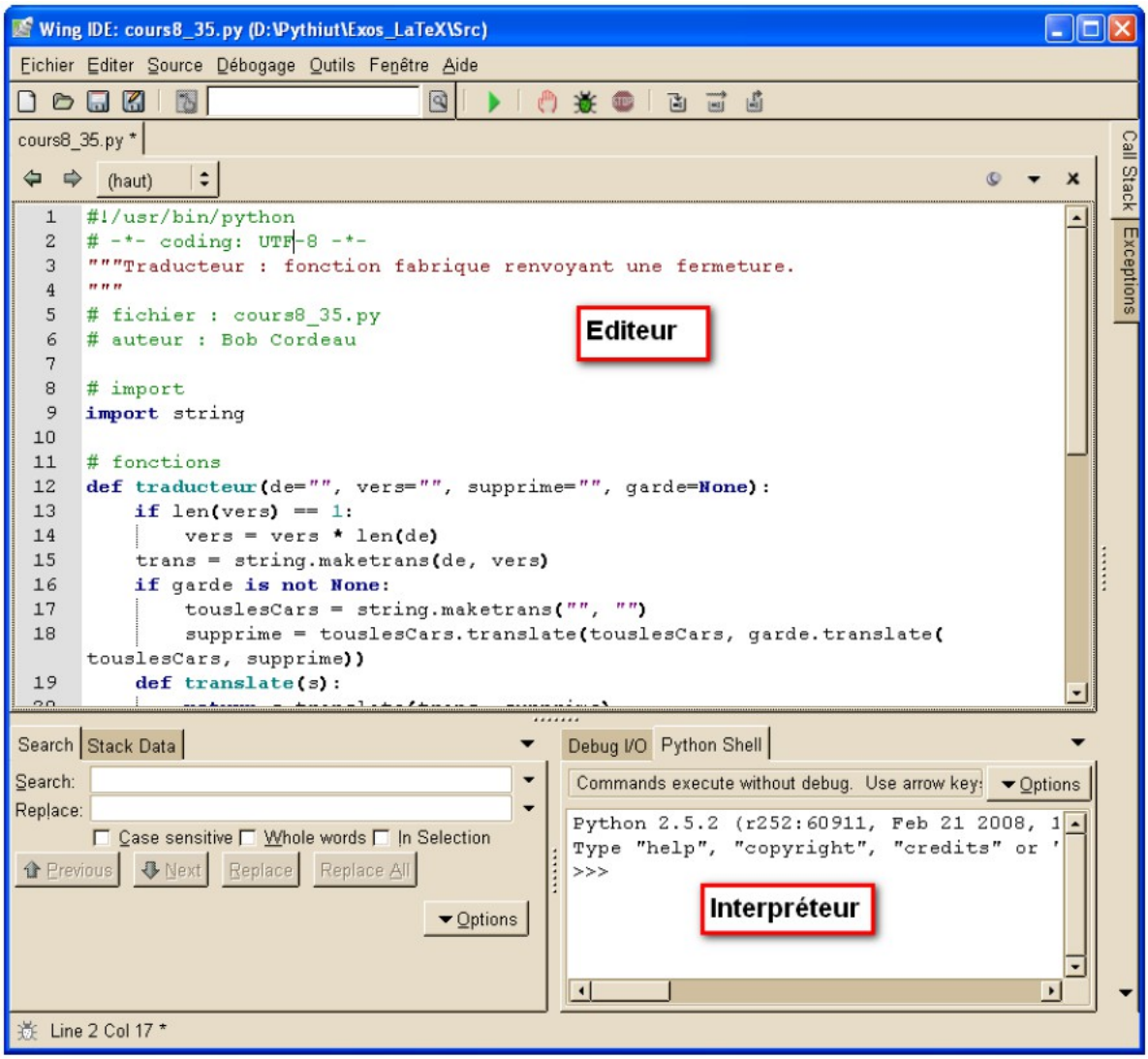

Illustration 4: Wing IDE

Au cas où certaines zones de l'environnement de développement viendraient à être masquées, les raccourcis clavier F1 et F2 et Maj-F2 permettent de les faire réapparaître.

1.1.2 - Types numériques

Le **type entier** `int` et **type flottant** `float`, essayez dans le Python Shell :

```
>>> type(4)
>>> type(4.56)
```

Calculs

Utilisez les quatre **opérations de base** ('**+**', '**-**', '*****', '**/**') sur des entiers relatifs et des flottants.

Utilisez la **variable prédéfinie '_'** pour rappeler le dernier résultat.

Essayez et commentez :

```
>>> 20 / 3
```

```
>>> 20 // 3
>>> 20 % 3
>>> 15 / 0
```

Hiérarchie des opérateurs

Essayez et commentez :

```
>>> 7 + 3 * 4
>>> (7 + 3) * 4
```

Cas particulier de la **division entre entiers**, essayez et commentez :

```
>>> 8 / 4
```

Autres opérateurs

Expérimentez : que font ces opérateurs, commentez ?

```
>>> 20.0 // 3
>>> 20.0 % 3
>>> 2.0 ** 10
>>> 5 ** 2 % 3 # Ici, quel est l'opérateur prioritaire ?
>>> 5 % 2 ** 3 # Et là ?
>>> 18 @ 3
```

☞ **Remarque**

Python dispose d'autres types numériques, entre autre un type standard `complex`, qui peut s'exprimer directement en spécifiant une valeur de partie imaginaire suffixée par un j : `4+3.1j`

Certains types numériques sont disponibles dans des modules spécifiques (fractions, nombres décimaux, etc).

1.1.3 - Type chaîne de caractères

Pour les textes, on parle de *chaînes de caractères*[4] ou tout simplement de *chaînes*.

Le **type chaîne** `str`, essayez :

```
>>> type("toto")
```

Les opérateurs de base des chaînes

Les chaînes peuvent être mises bout à bout (opération de **concaténation**), recopiées un certain nombre de fois (opération de **répétition**) et il est possible de connaître leur nombre de caractère (**longueur**).

Essayez et commentez (attention aux espaces) :

```
>>> "Bonjour " + 'Joe' + " " + """Student"""
>>> '~'*60
>>> '*' * 20 + " Bienvenue " + '*' * 20
>>> len('')
>>> len("Bonjour Joe")
```

4 en anglais *strings*.

1.1.4 - Type logique booléen

Deux valeurs possibles : vrai et faux. On obtient des valeurs booléennes soit en saisissant directement les constantes Python `True` et `False`, soit en calculant des expressions logiques (par exemple des comparaisons).

Le type booléen `bool`, essayez :

```
>>> type(True)
```

Expressions logiques

Essayez et commentez :

```
>>> 4 < 9
>>> (3*8)!=(2*12)
>>> 3.14159 == 3.141592653589793
>>> 4 < 8 < 7
>>> "Bab" > "Bob"
>>> 3 < "trois"
```

Opérateurs logiques

Essayez et commentez :

```
>>> 3 < 4 and 9 == 3*3
>>> 3-1>4 or 3-1>2
>>> not len("Jean")<=3
>>> (3-1>4 or len("Jean")>3) and (9!=3*3)
```

Mémo : tables de vérité

a	b	not a	a or b	a and b
False	False	True	False	False
False	True	True	True	False
True	False	False	True	False
True	True	False	True	True

1.1.5 - Transtypage

Chaînes et nombres flottants ou entiers ne sont pas équivalents, même s'ils peuvent apparaître identiques lors de l'affichag. Le sens des opérateurs n'est pas toujours le même et des types différents ne peuvent pas toujours être combinés. Essayez et commentez :

```
>>> "3" * 10
>>> 3 * 10
>>> "3" * 10.0
>>> "3"+"3"
>>> 3+3
>>> "3"+3
```

Il est possible d'effectuer une conversion d'une valeur d'un type dans un autre, à condition qu'elle soit compatible. Cette opération s'appelle le transtypage[5]. Essayez et commentez :

```
>>> int("12")
>>> int("12.4")
```

5 *cast* en anglais.

```
>>> float("12.4")
>>> str(12)
>>> str(12.4)
>>> bool(12)
>>> bool(0)
```

1.1.6 - Notions de donnée, de variable et d'affectation

L'ordinateur, on vient de le voir, manipule des *données* de différents types. On peut donner des noms à ces données en utilisant des **variables**. Pour l'ordinateur, une variable est une *référence* désignant une adresse mémoire (un emplacement précis dans la mémoire vive). À cette adresse est stockée une valeur bien déterminée et bien typée. C'est la mise en place de cette association nom / valeur qui est appelée l'**affectation**.

Choix des noms de variable

Outre le respect des règles sur la syntaxe des noms (`a`…`z_` suivi de `a`…`z0`…`9_`, en excluant les mots clés du langage)[6], le choix des noms de variables doit se faire de façon à ce que leur lecture fournisse directement une information explicite sur leur utilité. Par exemple : `age`, `maxi`, `moyenne`, `somme_x`…

L'affectation

On *affecte* (ou *assigne*) une valeur à une variable (un nom) en utilisant le signe égal **=**.

⚠ **Attention**

Cela n'a *rien à voir* avec l'opérateur mathématique exprimant l'égalité.

Pour afficher la valeur associée à une variable, on entre simplement son nom comme expression à calculer, on peut même mettre plusieurs variables en les séparant par des virgules.

Essayez :

```
>>> nom = "Student"
>>> prenom = "Joe"
>>> age = 20
>>> etudiant = True
>>> moyenne_notes = 11.5
>>> nom
>>> nom,prenom,age
```

Note : on aurait pû utiliser une *affectation multiple* pour éviter trop de lignes :

```
>>> nom,prenom,age,etudiant="Student","Joe",20,True
```

Les *mots clés réservé du langage* ne peuvent pas être utilisés comme nom de variable, ce sont :

```
False,      class      finally    is         return
None        continue   for        lambda     try
True        def        from       nonlocal   while
and         del        global     not        with
as          elif       if         or         yield
assert      else       import     pass
```

6 Et, bien que possible avec Python 3, l'utilisation de caractères accentués dans les noms des variables est fortement déconseillée. Et interdite pour les TP.

```
break       except      in          raise
```

On vérifie que c'est le cas en essayant :

```
>>> in = 34
```

Lorsque l'on relie un *contenu* (la valeur `20`) à un *contenant* (la variable `age`), on crée une association au moment de l'affectation. Association qui durera jusqu'à ce que l'on ré-affecte la variable avec une autre valeur. Essayez :

```
>>> age = 72
>>> age
```

Lors d'un anniversaire de Joe, on peut utiliser une **incrémentation** pour ajuster son age. Essayez :

```
>>> age = age + 1
>>> age
```

1.1.7 - Fonctions d'entrées/sorties

Ces fonctions sont très rarement utilisées dans le mode calculette. Par contre, elles seront incontournables dans les scripts de la partie programmation.

input

La fonction d'entrée, `input`, permet d'afficher une directive à l'utilisateur et d'attendre qu'il saisisse des informations au clavier en les validant par un appui sur Entrée. La fonction retourne le texte tapé par l'utilisateur sous la forme d'une valeur de type chaîne, que l'on stocke généralement dans une variable afin de pouvoir l'utiliser ultérieurement.

Essayez :

```
>>> formation = input("Quelle est votre formation ? ")
>>> moybac = float(input("Moyenne au BAC ? "))
>>> formation,moybac
```

print

La fonction de sortie, `print`, permet d'afficher des valeurs et des variables. Les caractères spéciaux contenus dans les chaînes sont correctement interprétés à l'affichage. Il est possible d'afficher plusieurs données en une seule fois, en les séparant par des virgules. Essayez et commentez :

```
>>> s = "Chaîne\ns'étale \"ici\" sur\n\tplusieurs\n\tlignes."
>>> s
>>> print(s)
```

En réutilisant les variables définies précédemment, essayez et commentez :

```
>>> print("Bonjour,",prenom,nom,"tu dis avoir",age,"ans.")
>>> print("bonjour","à","tous",sep="---")
```

Amélioration de l'affichage : str.format

L'affichage brut, tel qu'il est réalisé par la fonction `print` en demandant simplement la conversion en texte des données, est parfois difficile à lire. Une *méthode* `format` des chaînes de caractères permet de réaliser une mise en forme plus fine à l'aide d'une syntaxe simple.

Ouvrez le script nommé **TP1_format.py** qui se trouve dans le répertoire TP1, et qui contient les instructions :

```
identite = "{} {}, {} ans".format(prenom,nom,age)
print(identite)
print("{2} ans pour {1} {0}".format(prenom,nom,age))
print("{0:e} {0:f} {0:g}".format(19237.52383210))
print("Angle: {:.2f}°".format(45.8713214332))
print("HxL: {h:.1f}x{l:.1f}m2,{poids:.3f} Kg".format(
                poids=12.6752,l=23.76,h=11.212))
```

Faire un essai en exécutant (en appuyant sur la touche F5).

Quelques commentaires sur la syntaxe de formatage de la méthode **format** :

- Les *{…}* indiquent les endroits où des valeurs doivent être insérées dans le texte, lesquelles et comment les présenter.
- La première information dans les *{}* indique quelle valeur il faut prendre parmi les arguments que l'on donne à la méthode **format**.
  - Par *position* avec une indication numérique, le premier argument a le n°0, le second le n°1, le troisième le n°3, etc.
  - Par *nom*, il suffit de spécifier **nom=*valeur*** pour chaque argument.
  - S'il n'y a pas d'indication, à chaque *{}* correspond un argument, dans l'ordre où ils se présentent.
- La seconde information dans les *{}*, après le *:*, précise comment la valeur doit être présentée.
  - Pour les *nombres flottant*, les lettres de formatage suivantes sont disponibles : ***f*** pour l'affichage scientifique, ***e*** pour l'affichage exponentiel, et ***g*** pour l'affichage optimal (choisit automatiquement entre ***e*** et ***f*** suivant l'ordre de grandeur de la valeur, et limite les décimales). Comme vous le voyez dans les exemples, il est possible de spécifier le nombre de décimales à afficher.
  - Pour les nombres entier, l'affichage se fait par défaut en décimal (base 10). Les lettres de formatage suivantes sont disponibles : ***o*** pour l'affichage en octal (base 8), ***x*** pour l'affichage en hexadécimal (base 16), et ***b*** pour l'affichage en binaire (base 2).

Il existe bien d'autres options pour **format** (alignement de texte, représentation de signe, etc), pour plus de détails il faut consulter la documentation Python 3, section *7.1.3 Format String Syntax*.

1.1.8 - Instruction if

Cette instruction est une *structure de contrôle* de l'exécution du programme. Elle permet de n'exécuter une séquence d'instructions que si une expression logique est vraie. La séquence d'instructions liée au if est un bloc défini au moyen de l'indentation.

Importance de l'indentation

Toutes les structures de contrôles sont des *instructions composées*. Elles ont toutes la même structure : une ligne d'en-tête terminée par un double point, suivie d'une ou de plusieurs instructions indentées (retrait en début de ligne) sous cette ligne d'en-tête.

La règle est très simple :

Toutes les instructions au même niveau d'indentation appartiennent au même bloc.

⚠ **Attention**

Quel que soit l'éditeur utilisé, *ne pas* mélanger les espaces et les tabulations. Il est fortement conseillé de paramétrer son éditeur pour remplacer systématiquement toutes les tabulations par 4 espaces, selon la convention la plus utilisée dans le monde Python.

Exemple avec if

Essayez :

```
>>> if age>18 :
...     print(prenom,"est majeur.")
```

Ouvrez le script nommé **`TP1_if.py`** qui se trouve dans le répertoire TP1, et qui contient les instructions :

```
a = int(input("Valeur de a : "))
if a > 100 :
    print("<a> dépasse la centaine")
    print("C'est excessif, on le réduit de 10%.")
    a = int(a * 0.9)
elif a == 100 :
    print ("<a> vaut 100, rien à redire.")
else :
    print("<a> ne vaut pas cher !")
    print("On l'augmente de 5 unités.")
    a = a + 5
print("Au final, <a> vaut",a)
```

Faire un essai en exécutant le script plusieurs fois (en appuyant sur la touche F5 à chaque fois) et en saisissant différentes valeurs pour **`a`** : **`12`**, **`100`**, **`150`**. Commentez.

1.2 -Programmation

⚠ **Attention**

Conventions de nommage : Pour stocker vos programmes, il vous est demandé d'utiliser des noms normalisés : « **`TPn_s.py`** », où n, entre 1 et 8, est le numéro de la séance, suivi d'un souligné puis de s, numéro du script du TP en cours. Le premier script de ce TP doit donc s'appeler : **`TP1_1.py`**, le deuxième **`TP1_2.py`**, etc.

Dans cette seconde période, on va enregistrer les commandes Python dans un fichier (que l'on appelle souvent un « script »). Pour cela on utilise la fenêtre supérieure de *Wing IDE*.

- ouvrir Wing IDE ;
- ouvrir le fichier **modele_script.py** qui est sur le “Bureau” ;
- cliquez sur ***Fichier/Sauvegarder Sous...***[7] (ou faites simplement Ctrl-Shift-S) et indiquez **TP1_1.py** (rappelez-vous les conventions de nommage et le répertoire de stockage) ;
- ajustez l’en-tête du modèle, puis tapez votre script.

Votre fichier doit commencer par les lignes d’en-tête comme indiqué dans Illustration 5: En-tête d’un script (le fichier **modele_script.py** sur le “Bureau” vous évite à avoir à les re-saisir chaque fois) :

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
"""Modèle de script.
Documentation du module.
"""
# fichier: modele_script.py
# auteur: Bob Cordeau

# imports

# ---------- Programme Principal ----------
```

Illustration 5: En-tête d’un script

Explication du bloc d’en-tête

- `#!/usr/bin/python3`

  ce commentaire spécial[8] assure que vos scripts fonctionneront aussi bien sous Windows que sous Linux, MacOS X ou autre Unix, il doit absolument être sur la première ligne du fichier ;

- `# -*- coding: utf-8 -*-`

  ce commentaire spécial définit l’*encodage*, il permet d’utiliser tous les accents habituels du français, il doit absolument être sur une des deux premières lignes du fichier ;

- ***"""Modèle de script..."""***

  docstring : documentation générale du script, sert pour l'aide en ligne, accessible via **nom_module.__doc__** ;

- `# fichier: modele_script.py`

  nom de votre script (indispensable pour l’impression) ;

- `# auteur: Bob Cordeau`

  nom du binôme (indispensable aussi !) ;

---

7 En version anglaise ***File/Save As...***

8 en anglais le *shebang*.

Ensuite, des commentaires permettent de délimiter les différentes parties du script.

Exécution

Il est fortement conseillé d'enregistrer les scripts, dans le répertoire du TP, avant de les exécuter.

Dans Wing IDE, utilisez le menu ***Débogage*** / ***Démarrer/Continuer*** ou le raccourci clavier F5 afin que l'exécution du script actuellement affiché soit lancée. L'onglet ***E/S du débogueur*** dans la zone d'outils horizontale de *Wing IDE* sert pour les saisies et les affichages.

1.2.1 - Calculs de l'aire d'un rectangle

Programme TP1_1.py

- Saisir deux valeurs flottantes dans deux variables **largeur** et **longueur**.
- Calculer l'aire de la surface correspondante et stockez là dans une variable **aire**.
- Réaliser un affichage propre pour donner le résultat du calcul.

1.2.2 - Somme de deux résistances

Programme TP1_2.py

- Saisir deux valeurs flottantes de résistances dans des variables **r1** et **r2** en indiquant l'unité dans le message (on peut utiliser **"\u2126"** pour afficher un joli Ω) ;
- effectuer l'affichage formaté (deux chiffres significatifs) de la résistance équivalente en série et en parallèle. Ne pas oublier l'unité.

1.2.3 - Equation du second degré

Programme TP1_3.py

Calcul des racines réelles du trinôme : $ax^2 + bx + c = 0$

Juste après l'en-tête du fichier, insérez l'instruction[9] :

```
from math import sqrt
```

Ensuite :

- Saisir trois nombres flottants correspondants aux coefficients et les stocker dans des variables **a**, **b** et **c**.
- Calculer la valeur du discriminant, **delta**.
- En testant la valeur de **delta** (négatif, nul ou positif), afficher qu'il n'existe pas de racine réelle, ou la valeur de la racine réelle double, ou enfin les valeurs des deux racines réelles. N'oubliez pas de formater proprement vos affichages.

1.2.4 - Calcul de la division euclidienne

Programme TP1_4.py

- Saisir deux entiers positifs **a** et **b** ;
- calculer le quotient **q** et le reste **r** ;
- afficher la division euclidienne. Avec par exemple 45 et 11, on doit obtenir :

---

9 Ce calcul nécessite l'utilisation de la racine carrée. La fonction **sqrt** appartient évidemment au module ***math***. Regardez sa syntaxe exacte dans la documentation (ou tapez help(sqrt) dans un interpréteur)

```
Division euclidienne: 45 = 11*4 + 1
```

1.2.5 - Règles logiques

Programme TP1_5.py

Soit la règle administrative suivante, concernant le paiement de l'impôt sur le revenu en un seul règlement (source : http://vosdroits.service-public.fr/F3118.xhtml) :

> **Principe**
>
> Le contribuable peut, s'il le souhaite, opter pour le prélèvement de chacun de ses impôts, à la date limite de paiement qui figure sur l'avis d'imposition.
>
> Pour l'impôt sur le revenu, le contribuable paiera son impôt de 2010, sur ses revenus perçus en 2009, en 1 seule fois :
>
> • si le montant de son impôt de 2009, sur ses revenus perçus en 2008, était inférieur à 337 € ,
>
> • ou s'il est imposé pour la première fois en 2010, sur ses revenus perçus en 2009,
>
> • ou si son impôt de 2009, sur ses revenus perçus en 2008, a été mis en recouvrement après le 15 avril 2010 (la date de mise en recouvrement doit figurer sur l'avis d'imposition),
>
> • ou s'il n'a pas été imposable en 2009, sur ses revenus perçus en 2008, même s'il a déjà été imposé les années précédentes.

À partir de ce texte, identifiez les différentes informations nécessaires pour savoir si le contribuable paiera son impôt obligatoirement en une seule fois. Associez une variable à chaque information, associée à une valeur possible. Écrivez la ou les expressions logiques, utilisant les variables que vous avez définies, et permettant de connaître la décision sur le paiement obligatoire en une seule fois.

2 - Séquences et Boucles

Ce que l'on conçoit bien s'énonce clairement,
Et les mots pour le dire arrivent aisément.

Nicolas Boileau (l'Art poétique)

Objectifs :

- séquences str, tuple, list ;
- structures de contrôle : boucles for et while ;
- rupture de boucles break et continue ;

Fichiers à produire : `TP2_1.py` `TP2_2.py` `TP2_3.py` `TP2_4.py` `TP2_5.py` `TP2_6.py` `TP2_7.py`

N'oubliez pas de suivre les directives indiquées dans le chapitre Avant-Propos.

2.1 -Manip

2.1.1 - Listes et tuples

Ce sont deux conteneurs, qui permettent de stocker des *séquences* de données de tout types[10] (là où les chaînes ne permettent de stocker que des caractères), dans un ordre connu et avec la possibilité d'avoir une répétition de certaines valeurs.

Les **tuples** sont utilisés de façon quasi transparente par Python, ils permettent de créer des séquences de données que l'on ne peut plus modifier ("immutables"). On les écrit simplement entre parenthèses : `( 1, 4, "toto")`

Il est même possible, lorsqu'il n'y a pas d'ambiguïté, d'ôter les parenthèses du tuple. A noter le tuple d'un seul élément : `("un",)` qui doit contenir une virgule pour le distinguer de simples parenthèses de regroupement d'une expression calculée.

Les **listes**, qui seront très utilisées lors des TP, sont des séquences modifiables ("mutables"). On les écrit entre crochets : `[ 7, True, "Hello" ]`

2.1.2 - Indexation de séquences

Comme vu en cours[11], l'indexation dans les séquences (`str`, `tuple`, `list`…), qui permet d'accéder à des éléments ou à des sous-séquences, s'écrit en utilisant l'opérateur `[]`.

Essayez les indexations sur chaînes de caractères et commentez si besoin :

```
>>> s = "Une séquence de caractères"
>>> s[5]
>>> s[-1]
>>> s[4:12]
>>> s[13:]
>>> s[:3]
>>> s[:]
>>> s[::-1]
>>> s[16] = 'C' # On essaie de mettre une majuscule
```

Vérifiez que cela fonctionne aussi sur les listes et les tuples :

```
>>> lst = [ "zero","one","two","three","four", "five","six" ]
```

10 On peut même mélanger différents types de données dans la même séquence.

11 Voir aussi la section « Séquences & Indexation » dans l'Abrégé Dense Python 3.1.

```
>>> lst[2]
>>> lst[:2]
>>> lst[4:]
>>> lst[-1]
...
>>> refs = ( "zéro","un","deux",3,"quatre")
>>> refs[1:4]
...
```

Modification de séquence par l'indexation

Rappel, les chaînes de caractères et les tuples sont des séquences **immutables**, une fois créés il n'est pas possible de les modifier. Essayez :

```
>>> refs[3] = "trois"
>>> s[4:12] = "chaîne"
```

Par contre, les listes sont des séquences **mutables**, il est donc possible de modifier les valeurs en place. Ceci utilise la même notation d'indexation, mais en partie gauche d'une affectation. Essayez et commentez (ré-affichez `lst` après chaque modification pour comprendre ce qui s'est passé) :

```
>>> lst
>>> lst[1] = "un"
>>> lst[3:6] = [ 4,5,6 ]
>>> lst[2:2] = [ "inséré","plusieurs","valeurs" ]
>>> lst[-1] = "données"
>>> lst[-2] = ["un","élément","séquence"]
```

On peut aussi utiliser une notation d'indexation comme argument d'une instruction `del` pour supprimer un ou plusieurs éléments. Essayez et ré-affichez `lst` :

```
>>> del lst[5]
```

Séquences imbriquées

Comme vu en cours, il est possible avec les tuples et les listes d'utiliser comme valeur d'élément tout type de données, entre autres des séquences. Essayez (attention à la ponctuation, éventuellement copiez-collez à partir du PDF du TP) :

```
>>> data=["Pierrot",(4,8,12),[(1.2,5.2),(0.1,9.3)],["un texte" ]]
>>> data[1][0]
>>> data[2][1]
```

Écrivez (exercice à mettre au compte-rendu) les instructions d'indexation imbriquées permettant de récupérer à partir de `data` les valeurs suivantes :

- `1.2`
- `"Pierrot"`
- `"rot"`
- `"texte"`

2.1.3 - Affectation et références

Dans l'exemple suivant :

```
>>> i = 1
>>> msg = "Quoi de neuf ?"
>>> e = 2.718
>>> l = [i, msg, e]
```

On définit dans une zone de la mémoire appelée un **espace de noms**, quatre noms de variables qui sont des **références** vers les valeurs stockées en mémoire.

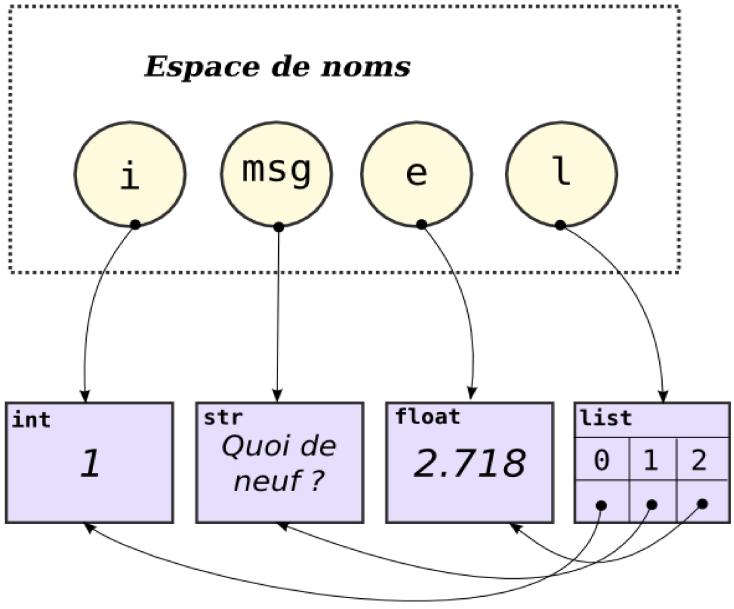

Illustration 6: Références et espaces de noms

On voit dans le diagramme de l'illustration 6 que les éléments contenus "dans" la liste ne sont aussi que des références vers les valeurs en mémoire. Il est d'ailleurs possible de connaître la référence[12] d'une valeur en utilisant la fonction `id`.

```
>>> id(e)
147826572
>>> id(l[2])
147826572
```

Pour toutes les valeurs *immutables* (nombres, booléens, chaînes, tuples…), il n'y a aucun problème car une fois une valeur créée elle ne peut plus être modifiée.

Par contre, pour les valeurs *mutables* (listes…), l'utilisation de références vers les valeurs stockées peut entraîner des modifications indirectes du contenu et des surprises :.

Essayez l'exemple suivant (ouvrez et exécutez le fichier **`TP2_references.py`**) et commentez :

```
liste1 = ['a', 'b', 'c']
liste2 = ['d', 'e', 'f']
```

12 Pour C-Python cette référence est simplement l'adresse en mémoire de la valeur. Elle peut changer d'une exécution sur l'autre suivant les instructions qui ont été effectuées avant la création de la valeur en mémoire.

```
liste3 = [liste1, liste2]
print("Liste1 originale:",liste1)
print("Liste2 originale:",liste2)
print("Liste3 originale:",liste3)
print("liste3[0][2]:",liste3[0][2])
del liste1[1]
print("Modification dans liste1 => liste1:",liste1)
print("liste3 après:",liste3)
```

2.1.4 - Les boucles

Le parcours de séquence (for)

L'instruction **for** est l'instruction idéale pour parcourir les éléments d'une séquence, par exemple d'une liste (ouvrez et exécutez l'exemple suivant **TP2_for.py**) :

```
donnees = [ 1, 3.14, "Bonjour", (7,8,9), [(1,2),(3,4),(5,3)] ]
for v in donnees :
    t = type(v)
    d = v * 2
    print("Valeur: v =",repr(v),"\n\ttype de v:",t,
                                "\n\tdoublé de v:",d)
```

L'exécution vous permet de suivre les différentes valeurs par lesquelles est passé tour à tour **v**, son type ainsi que le résultat d'une multiplication par 2 de la valeur.

La deuxième partie des instructions dans l'exemple (**TP2_for.py**) permet de parcourir les index des données dans la séquence et, dans le cas d'une liste (mutable), de modifier les données si l'on veut :

```
for index in range(len(donnees)) :
    v = donnees[index]
    print("Valeur: v =",repr(v))
    donnees[index] = str(v)+" à l'index "+str(index)
print(donnees)
```

Séries d'entiers (range)

La fonction générateur **range**, présentée en cours, permet de créer des séries d'entiers et de les parcourir dans des boucles **for**.

Essayez :

```
>>> for i in range(10) : print(i)
>>> for i in range(2, 10) : print(i)
>>> for i in range(2, 11, 2) : print(i)
```

Si l'on veut avoir d'un coup la liste complète des valeurs d'un range, il est possible d'écrire :

```
>>> list(range(10))
```

Boucles imbriquées

Il est aussi possible d'imbriquer des boucles, essayez et commentez l'exemple suivant (ouvrez et exécutez le fichier **TP2_forimb.py**) :

```
print("    ",end="")
for col in range(1,4) :
```

```
    print(col,end="  ")
print()
for ligne in range(1,6) :
    print(ligne,end=" : ")
    for col in range(1,4) :
        print(ligne*col,end="  ")
    print()
```

La répétition en boucle (`while`)

L'instruction de boucle **`while`** permet de contrôler l'exécution du bloc d'instructions de la boucle ***tant que*** la condition exprimée est ***vraie***. Voyez l'exemple suivant (ouvrez et exécutez le fichier **`TP2_while.py`**), modifiez le afin de tester d'autres possibilités (changement de condition de boucle, changement de l'expression sur **`i`**, valeur de départ de **`i`** différente…) :

```
i = 7
while i > 0 :                  # ne pas oublier le deux-points !
    print (i,' ',end=' ')      # ne pas oublier l'indentation !
    i = i - 1                  # décrémentation
```

Exemple graphique

Le module ***`turtle`*** permet d'illustrer les boucles de façon ludique. Il offre la possibilité de réaliser des « graphiques tortue[13] », c'est-à-dire des dessins géométriques correspondant à la piste laissée derrière elle par une petite « tortue » virtuelle, dont nous contrôlons les déplacements relatifs sur l'écran de l'ordinateur à l'aide d'instructions simples : **`forward(`*`n`*`)`** pour avancer de ***`n`*** pixels, **`left(`*`a`*`)`** pour tourner à gauche de ***`a`*** degrés, **`right(`*`a`*`)`** pour tourner à droite de ***`a`*** degrés, **`reset()`** pour effacer l'écran et remettre la tortue au centre…[14].

Ouvrez et exécutez le fichier **`TP2_turtle.py`**, qui contient les instructions ci-dessous ; et essayez différentes combinaisons de valeurs pour différents résultats :

```
from turtle import *
for i in range(4) :
    forward(100)
    left(90)
input("Appuyez sur Entrée pour la suite")

reset()
a = 0
while a < 12 :
    forward(150)
    left(150)
    a = a+1
```

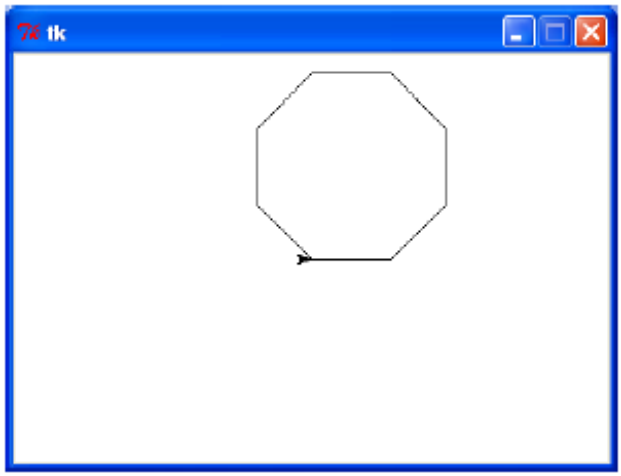

Illustration 7: Un octogone tracé « à la tortue »

13 Inspiré des "turtle graphics" du langage LOGO (fin des années 60, très utilisé dans le domaine de l'éducation).

14 Voir dans la documentation générale de Python la description du module ***`turtle`*** et des nombreuses fonctions qu'il fournit.

```
input("Appuyez sur Entrée pour la suite")
```

2.2 -Programmation

2.2.1 - Multiples de 7

Programme TP2_1.py

- Afficher les 20 premiers termes de la table de multiplication par 7, en signalant au passage (à l'aide d'une astérisque) ceux qui sont des multiples de 3.

Exemple : **`7 14 21 * 28 35 42 * 49…`**

Vous vous rappellerez le paramètre **`end="xxx"`** de la fonction **`print`**, ainsi que l'opérateur modulo calculant le reste de la division entière.

2.2.2 - Octogone

Programme TP2_2.py

- Tracer, à l'aide d'une boucle **`while`** (puis, dans un second temps, à l'aide d'une boucle **`for`**), un *octogone* de 50 pixels de côté (cf. illustration 7).

Vous penserez à calculer le nombre d'itérations nécessaires, et l'angle de rotation entre le traçage de deux côtés consécutifs.

2.2.3 - Combinaisons de dés

Programme TP2_3.py

Vous jouez avec deux dés et vous voulez savoir combien il y a de façons de faire un certain nombre. Bien noter qu'il y a deux façons d'obtenir 3 : 1-2 et 2-1.

- Saisir une valeur entière (indiquer à l'utilisateur que le nombre attendu est entre 2 et 12).
- Calculer, à l'aide de boucles imbriquées, le nombre de possibilités que deux dés à six faces produisent cette valeur.
- Afficher ce résultat.

À l'exécution l'écran devrait ressembler à :

```
Entrez un entier [2..12]. : 9

Il y a 4 façon(s) de faire 9 avec deux dés.
```

2.2.4 - Somme des 10 premiers entiers

Programme TP2_4.py

- Afficher la somme des 10 premiers entiers de trois façons différentes :
  - en utilisant la formule classique : $s=\frac{n(n+1)}{2}$;
  - en utilisant une boucle **`while`** ;
  - en utilisant une boucle **`for`**.

2.2.5 - Epsilon machine

Programme TP2_5.py

Un ordinateur n'est qu'un automate programmé *fini*. Il est sûr que l'on ne peut espérer effectuer des calculs avec une précision infinie, ce qui conduit à des « erreurs d'arrondi ». Quelle précision peut-on espérer ?

On va définir l'*epsilon-machine* comme la plus grande valeur ε telle que[15] :

$$1+\varepsilon=1$$

- Initialiser une variable `dx` à la valeur `1.0`.
- Dans une boucle `while`, diviser `dx` par `2` tant que la condition (`1.0 + dx > 1.0`) est vraie.
- Afficher la valeur dx finale. Comparer cette valeur avec une valeur normalement nulle (exemple la valeur de `sin(π)`).

2.2.6 - Liste de points

Programme TP2_6.py

On dispose d'une liste de coordonnées de points x,y sous la forme d"une liste de tuples :

```
coords = [ (4,5), (9,3), (12,8), (13,7), (18,6), (20,9) ]
```

- Créer une liste vide d'ordonnées `ordos`.
- Remplir la liste d'ordonnées en parcourant la liste des coordonnées.
- Afficher la liste d'ordonnées.

On veut maintenant filtrer le contenu de `coords` afin de borner à 7 toutes les ordonnées qui y sont supérieures.

- Parcourir la liste de `coords`, et y modifier tous les points d'ordonnée `>7` de façon à ce que pour ces points, les ordonnées soient fixées à 7.

Après votre traitement, affichez `coords`, qui devrait contenir :

```
[ (4,5), (9,3), (12,7), (13,7), (18,6), (20,7) ]
```

2.2.7 - Enquête criminelle

Programme TP2_7.py

Vous êtes tranquille dans votre laboratoire lorsque votre responsable arrive avec plusieurs listes de personnes. Trois assassinats similaires ont eu lieu consécutivement dans trois salles de spectacle différentes. Les enquêteurs sur le terrain ont relevé les listes des présents dans chaque salle. Afin de restreindre la liste des suspects, votre première tâche est d'identifier la ou les personnes qui se sont trouvées présentes aux trois spectacles.

Votre programme commencera par une instruction :

```
from enquete_m import *
```

Qui importe dans votre programme les variables globales suivantes :

---

15 Cette définition est conventionnelle car si on définit l'epsilon machine autour d'une valeur différente de 1 on trouve des valeurs légèrement différentes.

`salle1` - nom de la première salle de spectacle.

`spectateurs1` - liste des noms des personnes présentes dans la première salle.

`salle2` - nom de la seconde salle de spectacle.

`spectateurs2` - liste des noms des personnes présentes dans la seconde salle.

`salle3` - nom de la troisième salle de spectacle.

`spectateurs3` - liste des noms des personnes présentes dans la troisième salle.

Vous devez :

- A l'aide de boucles, remplir une nouvelle liste `pres12` qui soit constituée des noms des personnes présentes dans les deux premières salles.
- A l'aide de boucles, prendre ensuite en compte la troisième salle et créer une liste `pres123` qui soit constituée des noms des personnes présentes dans les trois salles.
- Vérifier que le traitement fonctionne, via un contrôle utilisant les opérations sur les ensembles, avec l'expression suivante :

```
set(spectateurs1)&set(spectateurs2)&set(spectateurs3)==set(pres123)
```

Votre résultat a permis d'interpeller un suspect. Les enquêteurs ont analysé ses faits et gestes la veille d"un des crimes, et ils ont un doute sur ce qui concerne ses déplacements. Ils vous demandent de calculer la longueur totale d'un trajet en plusieurs étapes - dont ils ont les coordonnées - et de donner les estimations de temps nécessaires au parcours à différentes vitesses.

L'import issu de `enquete_m` a aussi fourni dans votre programme une autre variable globale :

`trajet` - liste de tuples de coordonnées flottantes x,y en km,km. Par exemple :

```
[(91.68, 53.10), (81.52, 50.14), (43.68, 18.18)]
```

Vous devez :

- A l'aide de boucles, remplir une nouvelle liste `distances`, constituée des distances entre chaque coordonnées consécutives du trajet.
- Afficher cette liste.
- Calculer dans une variable `parcouru` la distance totale parcourue en sommant ces distances, puis afficher cette variable.
- Afficher avec une présentation correcte, pour les vitesses de 25km/h, 40km/h, 50km/h, 60km/h, 75km/h, 90km/h, 120km/h, 130km/h, le temps qui aurais été nécessaire pour parcourir la distance totale. Éviter les répétitions inutile de code

3 - Les fonctions

Le style est une façon simple de dire des choses compliquées.

Jean Cocteau

Objectifs :

- les fonctions et méthodes prédéfinies
  - listes ;
  - chaînes ;
  - dictionnaires ;
- définition des fonctions.

Fichiers à produire : `TP3_1.py` `TP3_2.py` `TP3_3.py` `TP3_4.py` `TP3_5.py` `TP3_6.py` `TP3_7.py`

N'oubliez pas de suivre les directives indiquées dans le chapitre Avant-Propos.

> ⚠ **Attention**
> La notation ***[xxx]*** indique un **paramètre optionnel** que l'on peut donc omettre. Si on fournit le paramètre, on ne met bien sûr pas les crochets, qui sont uniquement là pour la documentation.
>
> Cette notation est utilisée ici ainsi que dans la plupart des documentations informatiques, généralement associée à du texte en italique, comme dans le présent document.

3.1 -Manip

Comme nous l'avons vu en cours, l'utilisation des fonction permet de structurer ses programmes, de réutiliser les parties communes et de mieux les maintenir.

Dans un premier temps nous allons explorer quelques *fonctions prédéfinies* avant d'apprendre à *écrire nos propres fonctions* (il existe bien d'autres fonctions et méthodes, se référer à la documentation Python pour la liste exhaustive).

3.1.1 - Des méthodes et fonctions prédéfinies

Sur les listes

Fonctions. Ces fonctions sont aussi applicables aux autres types de séquences de données (chaînes, tuples…) :

`len(L)` — Renvoie le nombre d'éléments dans `L`.

`sorted(L)` — Crée et renvoie une nouvelle liste, copie de `L`, triée.

`reversed(L)` — Génère les éléments de `L` en ordre inverse.

Méthodes non modificatrices. Ces méthodes renvoient un résultat sans modifier la liste :

`L.count(x)` — Renvoie le nombre d'occurrences de `x` dans `L`.

`L.index(x)` — Renvoie l'indice de la première occurrence de `x` dans `L`.

Exercice : Essayez et commentez :

```
>>> L = [ "jean","marc","yacine","marc","herbert","marc","jean"]
>>> # Affichez la liste et sa longueur.
>>> L.count("marc")
>>> L.index("marc")
```

```
>>> for n in dir(L) : print(n)
```

Quels genres de renseignements nous donne la fonction `dir(L)` ?

Méthodes modificatrices. Ces méthodes, sauf `pop()` renvoient la valeur `None`.

`L.append(x)` — Ajoute l'élément x à la fin de `L`.

`L.extend(L2)` — Ajoute tous les éléments de la liste `L2` à la fin de `L`.

`L.insert(i,x)` — Insère l'élément `x` à l'indice `i` dans `L`.

`L.remove(x)` — Supprime la première occurrence de x dans L.

`L.pop([i])` — Renvoie la valeur de l'élément à l'indice `i` et l'ôte de `L`. Si `i` est omis alors supprime et renvoie le dernier élément.

`L.reverse()` — Renverse sur place les éléments de `L`.

`L.sort()` — Trie sur place les éléments de `L`. Par défaut tri croissant ; un argument optionnel `key` permet de trier suivant le résultat du calcul d'une fonction appliquée à chacune des valeurs ; un argument optionnel booléen `reverse` permet d'inverser le tri.

Sur les chaînes de caractères

Note : ce n'est ici qu'une partie des méthodes disponibles, voir la documentation pour une liste exhaustive.

☞ **Rappel**

Les chaînes de caractères ne sont pas modifiables et donc les méthodes qui *transforment* les chaînes en créent de *nouvelles* et les retournent.

`S.count(sous_chaîne[, début [, fin]])` — Compte le nombre d'occurrences de `sous_chaîne` dans `S` entre `début` et `fin` (0 et len(S) par défaut).

`S.find(sous_chaîne[, début [, fin]])` — Retourne la position de la première occurrence de `sous_chaîne` dans `S` entre `début` et `fin` (0 et len(S) par défaut). Retourne -1 si `sous_chaîne` n'est pas trouvée.

`S.join(séquence)` — Retourne une chaîne résultant de la concaténation des valeurs chaînes issues de `séquence` en insérant `S` comme séparateur entre les valeurs.

`s.split([séparateur[,nb_max]])` — Retourne une liste de chaînes résultant du découpage de la chaîne `S`. Par défaut la séparation se fait sur les blancs (espaces, tabulations, retours à la ligne), mais on peut spécifier un autre `séparateur`. Le paramètre `nb_max` permet de limiter le nombre de découpes.

`S.strip([caractères])` — Supprime les blancs (ou tout caractère dans `caractères`) au début et à la fin de `S`.

`S.replace(ancienne, nouvelle[, nombre])` — Remplace chaque occurrence de la sous-chaîne `ancienne` par `nouvelle` dans `S`. Si `nombre` est donné, seules les `nombre` premières occurrences sont remplacées.

`S.capitalize()` — Transforme le premier caractère de `S` en majuscule et les suivants en minuscules.

`S.lower()` — Convertit toutes les lettres de `S` en minuscules.

`S.upper()` — Convertit toutes les lettres de `S` en majuscules.

`S.center(largeur[,remplissage])` — Centre la chaîne dans une `largeur` donnée, en rajoutant si besoin est des espaces (ou le caractère `remplissage`) de part et d'autre.

`S.zfill(largeur)` — Ajoute si nécessaire des zéros à gauche de `S` pour obtenir la `largeur` demandée.

De plus, on dispose des méthodes booléennes suivantes, dites « de test de contenu» :

`S.isalnum()` — Alphanumérique uniquement (chiffres ou lettres).

`S.isalpha()` — Lettres uniquement.

`S.isdigit()` — Chiffres uniquement.

`S.islower()` — Lettres minuscules uniquement.

`S.isspace()` — Blancs uniquement.

`S.istitle()` — Forme de titre.

`S.isupper()` — Lettres majuscules uniquement.

Exercice : Essayez et commentez :

```
>>> s = "Joe Student"
>>> s.find("Stu")
>>> s.split()
>>> s.upper().center(80,"=")
```

Sur les conteneurs (séquences, ensembles, dictionnaires)

Lors du TP2, nous avons pu voir que le stockage des éléments dans les listes passait simplement par des références. C'est la même chose pour les dictionnaires ou les ensembles. La duplication de ces conteneurs et du contenu nécessite des traitements spéciaux qui sont disponibles en utilisant le module *copy*.

`copy.copy(D)` — Renvoie une copie en *surface*[16] de `D`. Fournit une nouvelle valeur distincte de `D` et égale à `D`, dont le contenu reste partagé avec `D`.

`copy.deepcopy(D)` — Renvoie une copie en *profondeur* de `D`. Fournit une nouvelle valeur distincte de `D` et égale à `D`, dont le contenu est complètement dupliqué et est distinct de celui de `D`.

Sur les dictionnaires

Méthodes non modificatrices. Les méthodes `items()`, `keys()` et `values()` renvoient des vues[17] sur le contenu du dictionnaire, utilisables par exemple dans les boucles `for` ou transformable en listes, avec des valeurs dans un ordre quelconque.

`k in D` — Est vrai (`True`) si `k` est une clé de `D`, faux (`False`) sinon.

---

16 Copie en surface plus simple, pour les séquences : `lst2 = lst1[:]` et pour les dictionnaires : `dico2 = dict(dico1)`.

17 *view* en anglais. Moyen d'accéder aux entrées du dictionnaire en les voyant sous la forme de conteneurs (itérables) de clés ou de valeurs ou de couples (clé, valeur), conteneurs qui restent liés au dictionnaire et en reflètent ses modifications.

`D.items()` — Renvoie une vue des *éléments* (paires clé/valeur) dans `D`.

`D.keys()` — Renvoie une vue des *clés* dans `D`.

`D.values()` — Renvoie une vue des *valeurs* dans `D`.

`D.get(k[, x])` — Renvoie `D[k]` si `k` est une clé de `D` ; sinon `x` (ou `None` si `x` n'est pas précisé).

Méthodes modificatrices :

`D.clear()` — Supprime tous les éléments de `D`.

`D.update(D1)` — Pour chaque clé `k` de `D1`, affecte `D1[k]` à `D[k]`.

`D.setdefault(k[, x])` — Renvoie `D[k]` si `k` est une clé de `D` ; sinon affecte `x` à `D[k]` et renvoie `x`.

`D.popitem()` — Supprime et renvoie un élément (paire clé/valeur) quelconque de `D`.

Exercice : Essayez (respectez bien les accents) et commentez chaque opération :

```
>>> d = { "à":'a',"é":'e',"ù":'u'}    # Affichez d !
>>> d["ê"] = 'e'                      # Re-affichez d !
>>> d.update( { 'à':'A','è':'E'} )    # Re-re-affichez d !
>>> 'e' in d
>>> 'à' in d
```

3.1.2 - Définition des fonctions

Une fonction est une instruction composée définie par :

- une ligne d'en-tête formée :
  - du mot-clé `def`
  - du nom de la fonction
  - de la liste des arguments entre parenthèses
  - du caractère deux-points
- Le bloc d'instructions qui constitue le corps de la fonction est ensuite indenté par rapport à la ligne d'en-tête. Une fonction se termine grâce à une instruction `return`, ou à défaut à la fin de ses instructions ;
- L'instruction `return` permet de retourner une ou plusieurs valeurs à l'expression qui a appelé de la fonction ; un `return` sans valeur spécifiée ou une fonction sans `return` renvoie simplement la valeur prédéfinie `None` ; .
- entre l'en-tête et le corps on insère généralement une *chaîne de documentation*[18], ce qui vous est demandé pour les fonctions que vous définissez en TP.

Exemple élémentaire :

```
>>> def doubler(x):
...     "Retourne le double de <x>"
...     return x * 2
...
```

18 *docstring* en anglais.

```
>>> doubler(7)      # Pourquoi cet appel fonctionne-t-il avec un entier...
>>> doubler(7.2)    # ...un flottant...
>>> doubler("bla") # ...une chaîne de caractères...
>>> doubler({1:'bleu',2:'rouge',3:'vert'}) # Et avec un dictionnaire ?
```

3.2 -Programmation

3.2.1 - Réorganisation de texte

Programme TP3_1.py

À partir d'une variable **chaine** contenant la chaîne suivante :

"Un texte que je veux retravailler"

Remplacer dans **chaine** le *"je veux"* par *"Joe veut"* (interdiction de coder en dur des index).

Découper ensuite **chaine** en une liste de mots, stockée dans une variable **mots**.

A partir de **mots**, créer une nouvelle chaîne qui contienne un séparateur *"+++"* entre chaque mot. Afficher cette nouvelle chaîne.

3.2.2 - Semaine

Programme TP3_2.py

Affecter à une variable **semnum** un dictionnaire qui fasse correspondre les clés numéros des jours de la semaine (entiers 1 à 7) à leurs valeurs noms (chaînes lundi à dimanche).

Créer une seconde variable **semjours**, de type dictionnaire, vide.

En utilisant une boucle parcourant les *éléments* stockées dans **semnum**, remplire **semjours** en faisant correspondre les clés noms des jours à leurs valeurs numéros des jours dans la semaine.

Extraire à partir de **semjours** les noms des jours dans une liste **jours**, que l'on trie ensuite par ordre croissant.

Parcourir cette liste dans une boucle, et pour chaque jour afficher son nom ainsi que son numéro dans la semaine (trouvé via l'un des deux dictionnaires créés précédemment).

3.2.3 - Calcul de π

Programme TP3_3.py

Écrire une fonction **monPi()** avec un argument **p_n**. Cette fonction retourne une valeur approchée de π, sachant que :

$$\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$$

Attention, cette approximation est une quantité *réelle* : utilisez des *flottants*. Pour cela vous avez besoin de la fonction « racine carrée » dont le nom en Python est **sqrt()** et qui est accessible depuis le module ***math***.

Par ailleurs, écrire un programme principal qui saisit un entier **n** positif non nul, qui appelle la fonction **monPi()** et qui affiche vos résultats ainsi que l'erreur relative (en pourcentage) entre votre valeur de π et celle fournie par le module ***math***.

3.2.4 - Suite de Syracuse

On appelle suite de Syracuse toute suite d'entiers naturels vérifiant :

$$u_{n+1}=\begin{cases} \dfrac{u_n}{2} & si \quad u_n\, est\, pair \\ 3u_n+1 & si \quad u_n\, est\, impair \end{cases}$$

Par exemple, la suite de Syracuse partant de u_0=11 est :

11,34,17,52,26,13,40,20,10,5,16,8,4,2,1,4,2,1,4,2,1

En se bornant à 1, on appelle cette suite finie d'entiers le **vol**[19] de 11. On appelle **étape** un nombre quelconque de cette suite finie. Par exemple 17 est une étape du vol de 11. On remarque que la suite atteint une étape maximale, appelée **altitude maximale** du vol. Par exemple 52 est l'altitude maximale du vol de 11.

Programme TP3_4.py

Écrire une fonction `etapeSuivante()` avec un argument `p_u`. Elle reçoit un entier strictement positif, calcule l'entier suivant dans la suite de Syracuse et le retourne. Par exemple `etapeSuivante(5)` doit produire `16` et `etapeSuivante(16)` doit faire `8`, etc.

Écrire une fonction `vol()` avec un argument `p_uinit`. Elle reçoit un entier initial strictement positif, crée une liste vide et, en utilisant la fonction `etapeSuivante()`, calcule et stocke dans cette liste les termes de la suite de Syracuse de l'entier initial jusqu'à 1 inclus. Cette fonction retourne la liste ainsi créée.

Écrire un programme principal qui demande un entier initial à l'utilisateur (pour les tests, utilisez des valeurs entre 2 et 100) puis qui, en utilisant les fonctions déjà définies, affiche proprement les termes de la suite de Syracuse correspondante.

Programme TP3_5.py

Écrire un programme principal qui demande un entier initial à l'utilisateur puis qui calcule et affiche l'altitude maximale de ce vol.

Pour cela, écrire une fonction `altMaxi()` avec un argument `p_uinit`. Elle reçoit un entier et produit l'altitude maximale de la suite de Syracuse commençant par `u0 = p_uinit`. Par exemple, en partant de `u0 = 7`, la suite de Syracuse est : 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1 et par conséquent `altMaxi(7)` vaut `52`.

3.2.5 - Notions de récursivité

La notion de récursivité en programmation peut être définie en une phrase :

Une fonction récursive peut s'appeler elle-même.

Par exemple, trier un tableau de *N* éléments par ordre croissant c'est extraire le plus petit élément puis trier le tableau restant à *N*–1 éléments.

---

19 On a constaté depuis longtemps que pour tous les entiers de départ testés, la suite atteint après un nombre de termes plus ou moins grand, la période 4, 2, 1, 4, 2, 1… Hélas ! pour l'instant tous ceux qui ont essayé de le prouver ont échoué : c'est un « problème ouvert » (une *conjecture*).

Bien que cette méthode soit souvent plus difficile à comprendre et à coder que la méthode classique dite *itérative*, elle est dans certains cas l'application la plus directe et la plus simple de sa définition mathématique.

Voici l'exemple classique de définition par récurrence de la fonction factorielle :

$$n! = \begin{cases} 1 & si \quad n = 0 \\ n(n-1)! & si \quad n \geqslant 1 \end{cases}$$

☞ **Remarque**

Son code est très exactement calqué sur sa définition mathématique.

```
def fact(p_n) :
    if p_n == 0 :
        return 1
    else :
        return p_n * fact(p_n-1)
```

Dans cette définition, la valeur de $n!$ n'est pas connue tant que l'on n'a pas atteint la condition terminale (ici `p_n == 0`). Le programme *empile* les appels récursifs jusqu'à atteindre la condition terminale puis *dépile* les valeurs. Ce mécanisme est illustré par l'illustration 8.

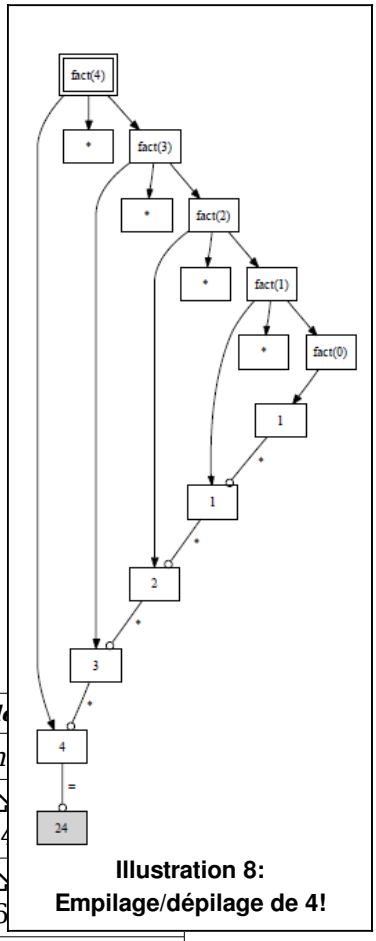

Illustration 8: Empilage/dépilage de 4!

Pour dérouler un programme récursif, il suffit de dérouler tous ses appels dans un tableau comme ci-dessous, jusqu'à atteindre la condition terminale sans appel (par exemple ici quand `n` vaut `0`). Ensuite on fait remonter, à partir de la ligne du bas (sans appel), les valeurs de retour trouvées (3e colonne) jusqu'en haut du tableau.

Calcul	***Appels***	***Valeur d***
fact(n)	*n * fact(n-1)*	*n * fact(n*
fact(4)	4 * fact(3) ⇩	⇧ 24
fact(3)	3 * fact(2) ⇩	⇧ 6
fact(2)	2 * fact(1) ⇩	⇧ 2
fact(1)	1 * fact(0) ⇩	⇧ 1
fact(0)	*sans appel*	⇧ 1

3.2.6 - Suite de Fibonacci

FIBONACCI : une suite doublement récursive

On définit la suite de Fibonacci de la façon suivante :

$$fib_n = \begin{cases} fib_0 & = & 0 \\ fib_1 & = & 1 \\ fib_{n+1} & = & fib_n + fib_{n-1} \end{cases}$$

Programme TP3_6.py

À faire :

- dérouler le programme « à la main », comme expliqué dans le paragraphe précédent, avec **5** comme entrée, remplir et **rendre** le tableau ci-dessous ;
- puis coder la suite de FIBONACCI dans le fichier **TP3_6.py** et vérifier les résultats du tableau avec une boucle d'affichage.

Calcul	***Appels***	***Valeur de retour***
fib(n)	*fib(n-1)+fib(n-2)*	*fib(n-1)+fib(n-2)=*
fib(5)	fib(4)+fib(3) ⇩	⇧
fib(4)	fib(3)+fib(2) ⇩	⇧
fib(3)	fib(2)+fib(1) ⇩	⇧
fib(2)	fib(1)+fib(0) ⇩	⇧
fib(1)	*sans appel*	⇧
fib(0)	*sans appel*	⇧

4 - Les modules

Lorsque vous avez éliminé l'impossible,
ce qui reste, même si c'est improbable,
doit être la vérité.

A. Conan Doyle (Le signe des quatre)

Objectifs :

- coder et utiliser un module ;
- les espaces de nom.

Fichiers à produire : `TP4_1_m.py` `TP4_1.py` `TP4_2_m2.py` `TP4_2_m1.py` `TP4_2.py` `TP4_3_m.py` `TP4_3.py`.

N'oubliez pas de suivre les directives indiquées dans le chapitre Avant-Propos.

4.1 -Manip

Nous allons maintenant expérimenter la technique de codage en utilisant des modules.

4.1.1 - La portée des variables, les espaces de noms

Quand on utilise un *nom* dans un script, l'interpréteur Python le crée, le modifie ou le recherche dans une zone mémoire particulière : « **l'espace de noms** ».

On sait qu'en Python les noms de variables existent dès qu'on leur affecte une valeur. C'est au moment où est réalisé l'affectation que Python sélectionne dans quel espace de noms particulier il place le nom de variable. L'espace de nom utilisé dépend donc de l'endroit où est réalisé l'affectation, ainsi que de la présence de l'instruction `global`[20].

Ce mécanisme d'espaces de noms se retrouve à tous les niveaux, fonctions, objets, classes, modules, packages. La notation qualifiée qui utilise le point "`.`" permet de spécifier un nom dans un espace de noms particulier (par exemple `pi` dans le module `math` : `math.pi`)

☞ **Rappel**

Les **modules**, dès qu'ils sont chargés, créent un *espace de noms global* qui existe pendant toute la durée d'exécution du programme Python. Les **fonctions** et les **méthodes**, *à chaque fois qu'elles sont appelées*, créent un *nouvel espace de noms local* qui disparaît à la fin de l'appel de la méthode ou fonction. Les **objets** qui sont créés (données, valeurs) ont leur propre *espace de noms qualifiés*.

Essayez dans l'interpréteur Python, et commentez :

```
>>> import math
>>> dir(math)
>>> help(math)
>>> math.pi
>>> math.pi.as_integer_ratio()
>>> import datetime
>>> datetime.date.today()
```

Au niveau du module, les affectations créent automatiquement des variables *globales*.

---

20 Il existe aussi l'instruction nonlocal, qui n'est pas abordée ici.

Dans les fonctions et méthodes, par défaut les affectations créent des variables *locales*[21] ; si on a besoin d'affecter une variable de portée *globale* au module qui contient la définition de la fonction ou méthode, on doit utiliser l'instruction **`global`** *nom_variable* pour le spécifier à Python.

⚠ **Attention**

Si vous utilisez un *variable locale* dans une fonction, qui a le même noms qu'une *variable globale* au module, vous avez alors deux variables homonymes mais distinctes car dans deux espaces de noms différents ; les modifications effectuées sur la variable locale ne le sont pas sur la variable globale. Pour n'utiliser que la variable globale au module, il faut spécifier l'instruction **`global`**.

Armé des indications précédentes, testez le petit script **`TP4_global.py`** suivant et commentez :

```
x = "valeur globale"
def f1() :
    print ("--Appel f1()")
    print ("  x dans f1 :", x)
def f2() :
    print ("--Appel f2()")
    x = "valeur locale"
    print ("  x dans f2 :", x)
def f3() :
    print ("--Appel f3()")
    global x
    x = "valeur modifiée"
    y = 12
    print ("  x dans f3 :", x)
    print ("  y dans f3 :", y)
print ("x au niveau global :", x)
f1()
print ("x au niveau global :", x)
f2()
print ("x au niveau global :", x)
f3()
print ("x au niveau global :", x)
print ("y au niveau global :", y)
```

4.1.2 - Les modules

Jusqu'à maintenant, tous les scripts que nous avons écrits n'étaient constitués que d'*un seul* fichier. Or quasiment tous les langages proposent et préconisent le développement multi-fichiers. Quel en est l'intérêt ? En fait ils sont multiples :

- les sources sont plus petits, donc plus faciles à concevoir et à mettre au point ;

---

21 Donc dans l'espace de noms local à la fonction ou méthode, qui disparaît lorsque l'on sort de son exécution.

- on peut ainsi écrire des fonctionnalités qui seront réutilisables dans plusieurs programmes, c'est donc une bonne économie de moyens (on évite de réinventer la roue !) ;
- on peut aller plus loin et regrouper les différentes fonctionnalités dans des éléments souvent appelés bibliothèques.

Chaque langage possède ses propres outils, plus ou moins compliqués. Python propose une technique particulièrement simple : le *module*[22].

Comment faire un module ?

Créons par exemple un module ***aff_m***, contenant une fonction **`afficheur()`** ainsi que son code d'auto-test C'est tout simplement un fichier[23] source Python **`aff_m.py`** contenant :

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
"""Module d'affichage"""
# fichier: aff_m.py
# auteur: Bob Cordeau

# fonction
def afficheur(x) :
    print(x)

if __name__ == '__main__' :
    # Code d'auto-test.
    afficheur(42)
    afficheur("Toto")
```

La structure générale d'un module comporte (autant que besoin) :

- le bloc d'en-têtes avec les directives d'encodage, votre nom, etc,
- les imports des modules utilisés,
- les définitions des fonctions,
- les définitions des classes,
- les définitions des variables globales au module,
- le corps du programme principal du module, généralement précédé du test que la variable **`__name__`** est bien égale à la chaîne **`'__main__'`** (attention, **`__name__`** et **`__main__`** sont encadrés par des *double souglinés* de chaque côté).

Comment utiliser un module ?

Vous savez déjà le faire depuis le deuxième TP !

En effet, le module que vous venez d'écrire s'utilise de la même façon de ceux de la bibliothèque standard Python (vous avez déjà utilisé par exemple le module ***math***). Donc, si dans un script, on a besoin de la fonction **`afficheur()`** du module ***aff_m***, il suffit de l'importer au préalable.

Python propose deux techniques d'importation :

---

22 L'étape suivante que nous n'explorerons pas est le regroupement de modules en *packages* formés d'arborescences de modules.

23 Par convention en TP, on choisit de suffixer les modules Python par ***_m***.

- Première technique, l'import de noms globaux au module :

`from aff_m import afficheur` — Les objets du modules sont directement accessibles (mais peuvent générer des conflits dans l'espace de noms local…) :

```
# import
from aff_m import afficheur
# ---------- Programme Principal ----------------------------
y, z, s = 12, 2.718, "Salut !"
afficheur(y)
afficheur(z)
afficheur(s)
```

La première technique possède une variante commode mais à déconseiller[24] (sauf indication explicite dans la documentation de certains modules) :

```
from aff_m import *
```

- Seconde technique, l'import du module en tant que tel :

`import aff_m` — Les objets du module devront être *pleinement qualifiés*, c'est-à-dire qu'il faudra écrire le nom du module suivi d'un point suivi du nom de l'objet. Par exemple :

```
# import
import aff_m
# ---------- Programme Principal ----------------------------
y, z, s = 12, 2.718, "Salut !"
aff_m.afficheur(y)
aff_m.afficheur(z)
aff_m.afficheur(s)
```

Le module principal

Le module principal est le premier module chargé par Python ; ça peut être celui indiqué à Python sur la ligne de commande : **`python monscript.py`**, ou bien ça peut être le module qui est ouvert dans l'environnement de développement lorsqu'on demande l'exécution du script.

Tous les autres modules, chargés pas des instructions **`import`**, sont des modules secondaires.

Vous remarquerez la dernière partie qui commence par : `if __name__ == '__main__' :`

Ce test permet, si le module est exécuté en tant que module principal[25], d'évaluer les lignes du bloc d'instructions if. Cette construction est très utilisée pour *mettre au point* les modules et fournir ainsi des modules « auto-testables ».

Lorsqu'on importe ce module à partir d'un autre module, cette partie est ignorée car la variable globale `__name__` prend alors comme valeur le nom du fichier d'import.

Module plot_m

Par la suite, nous aurons besoin d'afficher des graphes de courbes. Un module **`GnuPlot`** nous permet de le faire *depuis* un script. Afin d'améliorer la facilité d'utilisation, deux procédures ont été incluses dans le module ***`plot_m`*** (fichier **`plot_m.py`**, source en annexe page Erreur :

24 Car cette syntaxe viole toutes les bonnes recommandations sur les portées des noms !

25 Car la variable globale __name__ prend alors comme valeur '__main__'.

source de la référence non trouvée) qui vous sera utile dans les deux prochains TP et pour certains exercices personnels. Ce module est présent en lecture seule dans le répertoire **/opt/scriptsTP/**. Copiez-le dans votre répertoire de travail pour l'expérimenter et pensez à le copier sur votre clé USB en fin de séance.

Testez ce module et précisez ce qui se passe quand **plot_Fic()** est appelé.

Vous pouvez conserver l'auto-test de ***plot_m*** comme exemple d'écriture de données numériques dans des fichiers textes, et comme exemple d'utilisation de **plot_Fic()**.

Module verif_m

Un second module ***verif_m*** vous sera utile (à enregistrer également pour vos futurs TP, source en annexe page Erreur : source de la référence non trouvée). Il ne comporte deux fonctions **verif()** et **verifSeq()**, que vous pourrez utiliser pour tester les retours numériques de vos fonctions. Ce module est lui-même auto-testé et documenté, ce qui vous en donne la syntaxe.

Copiez le module **/opt/scriptsTP/verif_m.py** dans votre répertoire de travail et expérimentez-le.

4.2 -Programmation

4.2.1 - Réorganisation de code

Programme TP4_1

Le fichier script **TP4_moyennage.py** vous est fournit. Il construit une liste de points de coordonnées (x,y) ayant des valeurs aléatoires entre -20 et +20, puis effectue différents calculs de moyennes et affiche les résultats. Ce code a été écrit vite et mal ; il contient des sections redondantes et utilise des noms de variables incohérents d'une partie à une autre.

A partir du script **TP4_moyennage.py**, créer un nouveau fichier script **TP4_1.py**, qui regroupe dans une fonction **moyenne()** le calcul de moyenne sur une liste donnée en paramètre et qui utilise des appels à cette fonction au lieu des duplications de code. Vérifier que le résultat est identique au script d'origine.

Améliorer ensuite votre script **TP4_1.py** en utilisant des noms de variables homogènes et en utilisant des méthodes identiques pour les traitements similaires. Vérifier que le résultat est identique au script d'origine.

Programme TP4_1_m

La fonction **moyenne()** peut être utile pour d'autres scripts, nous allons faire qu'elle soit réutilisable. Créer un nouveau module **TP4_1_m.py** qui contient cette fonction, mettre en place un auto-test permettant de vérifier son bon fonctionnement.

Remplacer dans **TP4_1.py** la définition de **moyenne()** par son import à partir de **TP4_1_m**.

Vérifier que le résultat est identique au script d'origine.

4.2.2 - Calcul d'intégrale - méthode des rectangles

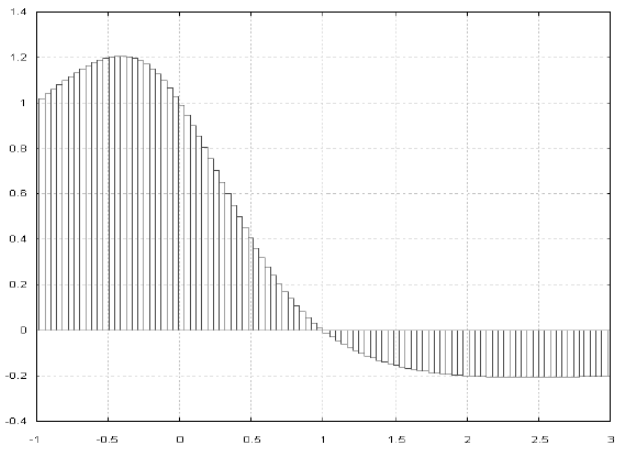

Illustration 9: Méthode des rectangles

Une solution élémentaire de calcul d'une intégrale simple sur un intervalle *[a,b]* est d'utiliser la méthode dite *des rectangles*[26] (voir illustration 9). Elle consiste à calculer l'aire *S(a,b)* délimitée par l'axe 0x, les droites $x=a$ et $x=b$, et la courbe $y=f(x)$.

Quand on divise *[a,b]* en *n* intervalles égaux, et que l'on pose $\Delta x=\frac{b-a}{n}$, on voit que le produit Δx.f(a+iΔx) constitue une valeur approchée de l'aire comprise entre la courbe, l'axe des abscisses et les droites $x = x_i$, $x = x_{i+1}$. La somme algébrique de ces produits constitue une approximation de *S(a,b)* d'autant meilleure que Δx est plus petit.

Le passage à la limite de cette somme constitue l'algorithme de Riemann :

$$\int_a^b f(x)dx=\lim_{n\to\infty}\Delta x\sum_{i=1}^{n} f(a+i\Delta x)$$

4.2.3 - Application au calcul de π

On continue, bien sûr, à utiliser la méthodologie des modules.

Programme TP4_2_m2

Dans le fichier **TP4_2_m2.py** créer une fonction **f()** à intégrer, et qui retourne $\frac{1}{1+x^2}$ (dérivée de la fonction arc-tangente). Auto-tester cette fonction dans le module.

Programme TP4_2_m1

Dans le fichier **TP4_2_m1.py**, créer la fonction **integrale()**. Elle reçoit quatre arguments : les deux bornes de l'intervalle d'intégration **p_a** et **p_b**, le nombre de pas de la somme[27] **p_nbpas** et le nom de la fonction à intégrer **p_fct**.

La fonction effectue une boucle permettant de découper **p_nbpas** morceaux entre les bornes **p_a** et **p_b**, appelle pour chaque valeur de ce découpage la fonction **p_fct** afin de calculer l'ordonnée de la courbe intégrée, et enfin cumule la somme des aires pas × hauteurs.

Le fonction retourne la somme des aires ainsi calculée.

Auto-tester votre fonction dans le module en intégrant la fonction unité entre 1 et 5 en 100 pas.

```
def unite(x) :
    return 1.0
```

26 Ou plus exactement « algorithme de Riemann ».

27 Une optimisation simple et efficace de la formule de Riemann consiste à remarquer que la fonction à intégrer est *linéaire*, et qu'il est donc inutile de faire **n** multiplications de Δx.

Programme TP4_2

En intégrant entre 0 et 1 la fonction **`f()`** précédemment définie, on obtiendra un résultat approché de π/4 car tan(π/4) vaut 1.

Écrire le programme principal (fichier **`TP4_2.py`**) qui saisit le nombre de pas d'intégration, appelle la fonction **`integrale()`** avec les quatre arguments voulus et affiche une approximation de π. Ne pas oublier d'importer la fonction **`f()`** depuis le module ***`TP4_2_m2`*** et la fonction **`integrale()`** depuis le module ***`TP4_2_m1`***.

Ensuite, à l'aide d'une boucle, affichez avec 5 décimales vos approximations de π pour 10, 1 000 et 100 000 pas d'intégration.

4.2.4 - Application au calcul de e

On souhaite calculer une valeur approchée de *e* (base du logarithme népérien) en utilisant l'algorithme de Riemann. On cherche la borne *b* telle que :

$$I = \int_1^b \frac{dx}{x} = 1$$

En effet, si *b*=*e*, alors *I*=1. Comme de plus la fonction logarithme est continue et monotone, on est fortement tenté de faire varier *b* en utilisant une méthode « dichotomique » pour approcher *e* avec la précision souhaitée.

Principe de la dichotomie : diviser pour trouver !

Il s'agit d'un procédé qui détermine, avec une marge d'erreur connue, ε, la solution d'une équation $f(x_0) = k$ (une intégrale dans notre cas) à l'intérieur d'un intervalle donné [*g*,*d*]. Cet algorithme n'est valable que si la fonction est *continue* et *monotone* sur l'intervalle de recherche.

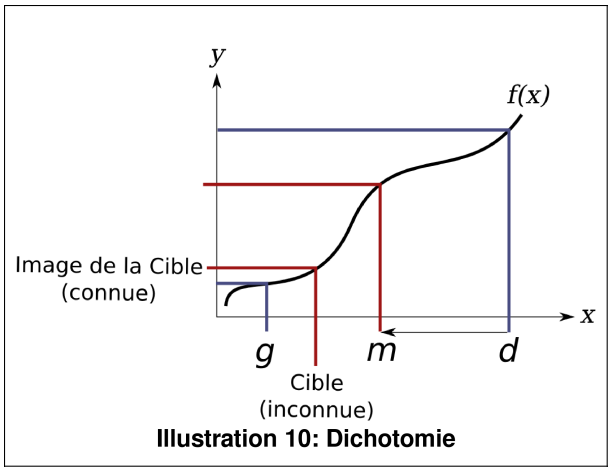

Illustration 10: Dichotomie

Son algorithme est le suivant (pour une fonction croissante[28]) :

Tant que l'intervalle de recherche est plus grand que l'erreur prévue : $|d-g|>\epsilon$

- on calcule le milieu *m* de l'intervalle :

$$m = \frac{g+d}{2}$$

- on teste la valeur de la fonction en ce point :

$$\text{Si} \quad f(m) < k \quad \text{alors } g = m, \text{ sinon } d = m$$

Ainsi, à chaque étape on restreint d'une moitié l'intervalle où se trouve la valeur recherchée.

28 Si la fonction est décroissante, il faut ajuster le test du $f(m) < k$.

Programme TP4_3

Codage de l'algorithme

Nous allons nous resservir des modules que nous avons déjà écrit, ainsi nous n'aurons qu'un minimum de code à développer :

Créer un module ***TP4_3_m*** contenant une fonction **`dicho(p_fonc, p_cible, p_gauche, p_droite, p_eps)`**, où les arguments représentent :

- la fonction dans laquelle on recherche un zéro,
- la valeur de la cible (k dans l'explication principe de la dichotomie),
- les bornes gauche et droite d'un intervalle réel de recherche,
- la précision de recherche.

La fonction **`dicho`** retourne trois valeurs[29] : les bornes gauche et droite recadrées par dichotomie et le nombre d'itérations de la boucle de dichotomie (c'est le compteur de la boucle while).

Écrire un auto-test comportant :

- la définition d'une fonction **`xmoins1(x)`** retournant **`x-1`** ;
- l'appel à la fonction **`dicho()`** pour trouver une solution de l'équation *xmoins1(x)=0* dans l'intervalle **`(0.5, 2.0)`**.

Application au calcul de *e*

Tout d'abord, ajouter dans le module ***TP4_2_m2*** une nouvelle fonction **`g()`** qui retourne **`1/x`**.

Nous avons maintenant tout ce qu'il nous faut pour écrire le programme principal **`TP4_3.py`** en utilisant au mieux les « briques » déjà écrites.

- importer les fonctions **`dicho()`**, **`integrale()`** et **`g()`** de leur module respectif ;
- saisir un encadrement de la borne supérieure de l'intégrale (c'est-à-dire un encadrement de *e* - conseiller *gauche* < 2 et *droite* > 3 à l'utilisateur) et une précision de calcul.

Il reste à appeler la fonction **`dicho()`** avec les bons arguments…

Quelle est la fonction à laquelle il faut appliquer la dichotomie ? Elle n'existe pas encore, nous sommes donc conduit à écrire une fonction spécifique à notre problème[30].

Définir une fonction **`calcul_e(p_x)`**, qui se contente de retourner le résultat d'un appel à la fonction **`integrale()`** avec les paramètres suivants :

- borne inférieure : 1.0 ;
- borne supérieure : x (elle sera trouvée par la fonction dicho()) ;
- pas d'intégration : 10^5 ;
- fonction à intégrer : g.

Enfin appeler la fonction **`dicho()`** et afficher l'approximation trouvée ainsi que le nombre d'itérations nécessaires à la fonction pour converger vers la valeur demandée.

---

29 Sous la forme d'un tuple.

30 Et qui possède, bien sûr, la même *signature* (mêmes paramètres et valeur de retour) que dans l'algorithme général.

5 - Les fonctions (suite) et les fichiers

Tout passe.
L'art robuste seul a l'éternité.
Le buste survit à la cité.
Théophile Gautier

Objectifs :

- documenter les fonctions et les modules ;
- approfondir le passage des arguments ;
- gérer les fichiers textuels ;

Fichiers à produire : `TP5_module_m TP5_1.py TP5_2_m.py TP5_2.py. TP5_3_m.py. TP5_3.py. TP5_3bis.py`.

N'oubliez pas de suivre les directives indiquées dans le chapitre Avant-Propos.

5.1 -Manip

5.1.1 - Documenter les fonctions et les modules

On l'a déjà dit, documenter ses scripts est important. Mais Python offre beaucoup plus[31].

Par exemple :

```
#!/usr/bin/python3
# -*- encoding: utf-8 -*-
# Fichier:TP5_ras.py
"""Ce module permet de tester le système d'auto-documentation de Python.

Il définit simplement une fonction, bien documentée, et son code d'auto-test.
"""
def ras() :
    """Cette fonction est bien documentée mais ne fait presque rien."""
    print ("rien à signaler")

if __name__ == '__main__' :     # Bloc d'auto-test
    print("\n+++ Documentation stockée dans un attribut de la fonction:")
    print(ras.__doc__)
    print("\n+++ Exemple d'appel à la fonction:")
    ras()
```

Lors d'une session en mode calculatrice la documentation issue du code source est directement accessible via l'aide en ligne[32] :

```
>>> from TP5_ras import ras
>>> help(ras)
Help on function ras in module TP5_ras:

ras()
    Cette fonction est bien documentée mais ne fait presque rien.
```

31 *Batteries included !*

32 Certains outils permettent de générer directement des manuels (HTML, PDF) à partir des documentations du code source.

À faire : créer un module `TP5_module_m` sur ce modèle, comprenant :

- deux fonctions `y1(x)` et `y2(x)`, avec leur documentation[33], calculant respectivement les expressions mathématiques $y_1 = 3x + 1$ et $y_2 = 2 - 7x$;
- un seul bloc d'auto-test, à la fin du module[34], avec pour chaque fonction :
  - un affichage de la documentation ;
  - des tests appelant les fonctions avec des valeurs significatives afin de vérifier leur bon fonctionnement (on pourra éventuellement utiliser des appels à la fonction `verif()` du module `verif_m` cité lors du TP4).

⚠ **Attention**

Donc, **dorénavant**, vous êtes priés de suivre les bonnes pratiques et d'ajouter systématiquement :

- une **documentation** intégrée à toutes les fonctions que vous écrirez ;
- un « **auto-test** » à tous vos modules - qui vérifie le bon fonctionnement de vos fonctions.

5.1.2 - Approfondir le passage des arguments

Valeurs par défaut

Jusqu'à maintenant, nos fonctions étaient définies avec un nombre fixe (éventuellement zéro) d'arguments que nous étions obligés de tous utiliser, dans l'ordre, à l'appel. Dans la définition d'une fonction, il est possible (et souvent souhaitable) de préciser une *valeur par défaut* pour certains paramètres. On obtient ainsi une fonction qui peut être appelée avec une partie seulement des arguments attendus. Voir la fonction `init_port()` dans l'exemple qui suit.

Nombre variables d'arguments

Certaines fonctions, comme par exemple `sum()`, acceptent qu'on leur fournisse un nombre variables d'arguments. Ceci est réalisé en déclarant un paramètre spécifique à la fonction, préfixé par une étoile (ex. `*params`). Lors de l'appel à la fonction, les arguments variables sont regroupés dans un tuple qui est passé dans le paramètre défini avec l'étoile. Voir la fonction `ecrire_port()` dans l'exemple qui suit.

Nombre variable d'arguments nommés

Certaines fonctions, comme par exemple la méthode `s.format()` des chaînes, acceptent qu'on leur fournisse un nombre variable d'arguments avec des noms. Ceci est réalisé en déclarant un paramètre spécifique à la fonction, préfixé par deux étoiles (ex. `**options`). Lors de l'appel à la fonction, les arguments variables nommés sont regroupés dans un dictionnaire qui est passé dans le paramètre défini avec la double étoile ; les noms étant utilisés comme clés du dictionnaire. Voir la fonction `parametres_fonctionnement()` dans l'exemple qui suit.

Exemple

Le script ci-dessous est disponible dans le fichier `TP5_portserie.py`. Essayez le et commentez le résultat de son exécution :

---

33 *docstring* en anglais.

34 En utilisant `if __name__ == '__main__':`.

```
def init_port(vitesse=9600, parite='paire', nb_bits=8, nb_stop=1) :
    print ("Initialisation port série à",vitesse,"bits/s")
    print ("    Parité:", parite)
    print ("    Bits de donnéee:", nb_bits)
    print ("    Bits d'arrêt:", nb_stop)

init_port()
init_port(parite="nulle", vitesse=28800)
init_port(2400, "impaire", 7, 2)

def ecrire_port(*chaines) :
    for texte in chaines :
        print("Ecriture de: {!r} + LF".format(texte))

ecrire_port("Hello périphérique")
ecrire_port("Canal 3","Mode Volts","Courant Continu","Gain 5","Mesurer")
ecrire_port("Canal 2","Mode Amperes","Gain Auto","Mesurer")

def parametres_fonctionnement(**options) :
    noms_params = sorted(options.keys())
    for nom in noms_params :
        val = options[nom]
        print("Réglage paramètre {} à {!r}".format(nom,val))

parametres_fonctionnement(nom1="valeur1",nom2="valeur2",trucmuche=True)
parametres_fonctionnement(abscisse=-3.2,ordonnee=4.7,hauteur=8.12,temps=232.2)
parametres_fonctionnement(parite="nulle", vitesse=28800)
```

5.1.3 - Les fichiers textuels

Le but de ce paragraphe est de pouvoir lire ou écrire des fichiers « texte », c'est-à-dire lisibles par tout éditeur de texte.

Répertoire courant

Les exercices suivants vont écrire des fichiers dans le répertoire courant. Auparavant, vous devez vérifier dans quel répertoire vous êtes positionné. Pour cela, dans la fenêtre de l'interpréteur, tapez :

```
>>> from os import getcwd,chdir
>>> getcwd()
```

Si besoin, vous pouvez changer ce répertoire courant en utilisant la fonction **`chdir()`**. Faites-le pour aller dans votre répertoire de travail (**`/homes/infoX/Bureau/TPn`**).

Note : Si vous travaillez sur un script dans Wing IDE, le répertoire courant est automatiquement positionné par l'environnement de développement au répertoire qui contient ce script.

Ouverture d'un fichier

En Python, l'accès aux fichiers est assuré par l'intermédiaire d'un *objet-fichier*, un type de données dédié aux accès aux fichiers, qui est créé par un appel à la fonction interne **`open()`**.

Cette fonction est s'utilise ainsi :

```
f = open(nomfichier[,mode][,encoding='xxx'])
```

- le premier argument est une chaîne contenant le nom du fichier à ouvrir[35] ;
- le second argument est une chaîne précisant le mode d'ouverture :
  - `"w"` *(write)* indique un mode en écriture,
  - `"r"` *(read)* indique un mode en lecture (mode par défaut si argument omis),
  - `"a"` *(append)* indique un mode en ajout.
- le troisième argument est une chaîne indiquant l'encodage des caractères du fichier (par défaut spécifique au système d'exploitation)[36].

Notons qu'en mode `"w"`, le fichier est créé et que s'il préexistait, l'ancien est effacé et remplacé par le nouveau (d'où l'utilité du mode `"a"`). Il existe d'autres caractères de modes d'ouverture[37],.

Lorsque les opérations sur un fichier sont terminées, il faut le fermer par un appel à sa méthode `close()` qui permet de libérer les ressources allouées pour la gestion de l'accès au fichier et de s'assurer que les données stockées par le système d'exploitation dans des zones mémoire tampons sont effectivement écrites sur le disque.

```
f.close()
```

Écriture séquentielle dans un fichier

Essayez et commentez (note : mettez des accents dans les textes écrits) :

```
>>> f = open("monFichier.txt","w",encoding='utf-8')
>>> f.write("Hé bien ça démarre sec.")
>>> f.write("Là, on a 0123456789 valeurs !")
>>> f.close()
```

Et avec votre explorateur de fichiers, ouvrez le fichier `monFichier.txt` dans le répertoire courant. Les deux chaînes écrites se sont retrouvées concaténées, ce qui est normal car elles ont été écrites l'une après l'autre *sans séparateur*.

☞ **Remarques**

- Ne pas confondre le *nom du fichier* (chaîne) sur disque avec le nom de variable de l'*objet-fichier* qui y donne accès (objet retourné par `open()`).
- Le fichier est sur le disque dur alors que l'*objet-fichier* est dans la mémoire vive de l'ordinateur (celle dont le contenu est perdu lorsqu'on éteint la machine).
- La méthode `write()` réalise l'écriture proprement dite. Son argument est une chaîne de caractères, le plus souvent formatée. Elle retourne le nombre de caractères écrits. Elle n'ajoute pas de retour à la ligne (caractère `'\n'`), il faut le spécifier dans la chaîne à écrire.

---

35 Nom éventuellement préfixé par un chemin absolu ou relatif si le fichier ne se trouve pas dans le répertoire courant. Comme séparateur entre les noms, utilisez des `/` et non des `\`.

36 Ce paramètre a beau être optionnel, il vaut mieux expliciter (et donc savoir) dans quel encodage sont les données textuelles. Couramment : `utf-8`, `latin1`, `ascii`.

37 On peut ajouter dans la chaîne de mode :`"b"` (mode binaire), `"t"` (mode texte, par défaut), `"+"` (lecture et écriture).

- La méthode **`close()`** referme le fichier. En raison des caches d'optimisation du système d'exploitation, il n'est pas garanti que les modifications faites soient écrites physiquement sur le disque tant que le fichier n'est pas fermé (ou qu'un appel à la méthode **`flush()`** ne force cette écriture).

Lecture séquentielle dans un fichier

On peut maintenant relire les données que l'on vient de stocker. Essayez et commentez :

```
>>> f= open("monFichier.txt",'r',encoding='utf-8')
>>> t = f.read()
>>> t
>>> f.close()
```

Ré-essayez la séquence d'ouverture/lecture/fermeture, en remplaçant la valeur du paramètre **`encoding`** par *`'latin1'`* et commentez.

☞ **Remarques**

- La méthode **`read()`** lit l'ensemble des données présentes dans le fichier et les affecte à une variable de type chaîne de caractères.
- La méthode **`read()`** peut être utilisée avec un argument entier qui indique alors le nombre de caractères à lire à partir de la position courante dans le fichier. Ceci permet de lire le fichier par morceaux.
- Comme précédemment, la méthode **`close()`** referme le fichier. Ceci permet de s'assurer qu'on libère les ressources allouées par le système d'exploitation pour l'accès au fichier.

Réécriture dans un fichier

Essayez simplement de faire une nouvelle écriture dans le fichier *existant* **`monFichier.txt`** :

```
f = open("monFichier.txt","w",encoding='utf-8')
f.write("Une ligne de texte\n")
f.close()
```

Ouvrez le fichier texte créé avec un éditeur, et commentez.

Maintenant exécutez plusieurs fois les instructions suivantes (vous pouvez les mettre dans un module et le ré-exécuter simplement plusieurs fois avec Wing IDE). Commentez, :

```
import time
f = open("monFichier.txt","a",encoding="utf-8")
f.write("On est à "+str(time.time())+"\n")
f.close()
```

Boucle de lecture dans un fichier

Les objets-fichiers sont des objets itérables (que l'on peut utiliser comme séquence dans une boucle **`for`** par exemple). À chaque itération ils fournissent la ligne de texte suivante dans le fichier. On peut ainsi écrire :

```
# Afficher les lignes d'un fichier qui contiennent une chaîne particulière:
f = open("monFichier.txt")
for line in f :
    if 'chaîne' in line : print(line)
```

```
f.close()
```

Écriture de données

Jusqu'ici, nous avons simplement écrit des chaînes de caractères. Le script **TP5_ecritdata.py**, qui vous est fournit, essaie de créer un fichier qui sauvegarde d'autres données : un nombre flottant et une liste de chaînes.

```
# Fichier : TP5_ecritdata.py
s = "Une phrase"
x = 23.432343
lst = [ "Uno","Dos","Tres" ]
f = open("data.txt",'w',encoding='utf-8')
f.write(s)
f.write("\n")
f.write(x)
f.write("\n")
f.write(lst)
f.write("\n")
f.close()
```

Exécutez ce script et commentez pourquoi cela ne fonctionne pas ?

5.2 -Programmation

5.2.1 - Écriture de données

Écrire un script (fichier **TP5_1.py**) qui soit une adaptation de **TP5_ecritdata.py**, mais qui fonctionne correctement de façon à ce que, après son exécution, le fichier texte **data.txt** contienne :

```
Une phrase
23.432343
Uno Dos Tres
```

5.2.2 - Courbe sinus cardinal

Programme TP5_2

Enregistrement et affichage des points d'une courbe :

- dans le module *TP5_2_m*, définissez la fonction sinus cardinal :

$$sinc(x)=\frac{\sin(\pi.x)}{\pi.x}$$

  en n'oubliant pas de gérer le cas particulier où x vaut 0, et en retournant 1 dans ce cas, car $\lim_{x\to 0}\frac{\sin(x)}{x}=1$.

- Dans le programme principal (fichier **TP5_2.py**) :
- importez la fonction **sinc()** ainsi que la procédure plot_Fic() du module *plot_m* ;
  - définissez **n = 256** ;
  - écrivez dans le fichier **sinc.dat** **n** lignes formées des valeurs de **x** et **y**, séparées par un blanc, telles que :

$$x=\frac{i-(n/2)}{20}, i\in[0,(n-1)]$$
$$y=sinc(x)$$

- enfin affichez la courbe par un appel à **plot_Fic()**.

5.2.3 - Mesures d'absorption

Le fichier **mesures.txt** qui vous est fourni contient le résultat de mesures prises au cours d'une expérience de spectroscopie d'absorption EXAFS[38] sur une feuille d'or à des fins d'étalonnage (mesures réalisées au centre Synchrotron Elettra de Trieste en Italie).

```
11800 -0.4806
11805 -0.481994
11810 -0.48344
…
```

La première colonne de valeurs contient des énergies en électron-volts, et la seconde colonne les coefficient d'absorption mesurés pour chaque énergie. Vous pouvez tracer simplement la courbe correspondant à ces mesures :

```
>>> import plot_m
>>> plot_m.plot_Fic([("mesures.txt","points")])
```

Programme TP5_3_m

On veut, pour commencer, pouvoir lire ces données et les stocker dans une liste de tuples (énergie, coefficient d'absorption) afin de pouvoir réaliser des traitements numériques :

```
[(11800,-0.4806),(11805,-0.481994),(11810,-0.48344),…]
```

Créer un nouveau module **TP5_3_m.py**, qui contienne une fonction **lire_donnees()**. Elle prend un seul argument **p_nomfichier** qui contient le nom du fichier de données à lire.

Cette fonction lit le contenu du fichier[39], et effectue les traitements nécessaires afin de séparer les valeurs, les convertir en flottants et retourner au final une liste de tuples.

Programme TP5_3

Si vous avez tracé la courbe correspondante, avec ***plot_m*** comme indiqué, vous avez pu voir qu'il existe un seuil où le coefficient d'absorption monte brusquement, suivi d'un pic. L'énergie du pic, est un des éléments caractérisant le matériau de l'échantillon, on va donc écrire un programme Python pour identifier cette énergie à partir des données expérimentales.

Dans le module ***TP5_3_m.py*** déjà créé, ajoutez une fonction **energie_pic()** prenant un seul paramètre **p_mesures**, qui contiendra lors de l'appel la liste des valeurs des mesures. Cette fonction retourne l'énergie correspondant au pic d'absorption le plus élevé.

Pour trouver cette énergie, vous pouvez soit faire une boucle de recherche de maximum, soit utiliser la méthode **sort()** des listes en spécifiant le critère de tri comme indiqué dans le cours 5, et prendre l'élément le plus grand.

---

38 EXAFS : Extended X-Ray Absorption Fine Structure - un flux de photons est envoyé sur un échantillon, et on mesure l'énergie absorbée par celui-ci en faisant varier l'énergie des photons. Voir aussi les expériences XANES (X-ray Absorption Near Edge Structure).

39 On considère que le contenu du fichier est constitué de lignes contenant chacune deux valeurs séparées par un ou plusieurs espaces, comme le fichier mesures.txt.

Créez un nouveau module **TP5_3.py** qui utilise les fonctions définies dans ***TP5_3_m*** afin de charger les données et calculer le pic du seuil. Sachant que sur une courbe d'EXAFS de l'or, on trouve un maxima d'absorption caractéristique[40] à 11945 eV (celui qui nous intéresse) et à, indiquez le décalage des instruments en comparant l'énergie mesurée au pic à l'énergie théorique que l'on devrait avoir.

Programme TP5_3bis

La façon de faire est ici simplifiée car nos mesures le permettent. Il faudrais normalement commencer par localiser le seuil où l'absorption augmente brusquement, et ne prendre en compte que le premier pic qui suit immédiatement ce seuil ; il y a en effet des courbes d'absorption où d'autres pics, plus éloignés du seuil, ont des valeurs plus élevées.

Créez un nouveau module **TP5_3bis.py**, dans lequel vous définissez une première fonction **index_seuil()** qui prenne en paramètre une liste **p_mesures** et retourne l'index de la mesure à partir de laquelle l'absorption augmente rapidement - cherchez le maxima sur les valeurs des pentes entre deux énergies consécutives.

Créez une seconde fonction **maxima_seuil()** qui prenne en paramètre une liste **p_mesures** et un entier **p_index**, et qui retourne la valeur de l'énergie pour le premier maxima trouvé sur l'absorption pour les mesures commençant à partir de l'index donné.

En combinant ces deux fonctions, réalisez la même opération de recherche d'énergie du pic d'absorption, qui elle fonctionnera normalement avec d'autres mesures réelles.

40 Il y en a aussi normalement un à 11970 eV, visible sur la courbe.

6 - Les fonctions (fin)

(…) la méthode est bonne.
C'est brutal, c'est maussade, mais c'est simple comme bonjour, inratable.
Georges Duhamel (Chronique des Pasquier)

Objectifs :

- Réaliser un programme plus important.
- Face à un problème réel (physique ou mathématique), apprendre à structurer les données en utilisant les caractéristiques du langage utilisé ;
- Apprendre à concevoir les différentes fonctions pour résoudre le problème posé (analyse descendante) en le découpant ;
- Synthétiser ces éléments dans le programme principal (analyse ascendante) ;
- Si une donnée est mal structurée ou une fonction mal conçue, reprendre l'analyse au début.

Fichiers à produire : `TP6_1_m.py` `TP6_1.py`.

N'oubliez pas de suivre les directives indiquées dans le chapitre Avant-Propos.

6.1 -Programmation

Un hôpital a installé récemment un générateur électrique 240 Volts pour palier à une éventuelle panne d'alimentation secteur. Ce générateur est mis régulièrement en marche afin de s'assurer qu'il fonctionne encore bien. Les responsables techniques se sont aperçus que la tension de sortie du générateur baisse lentement au cours de son fonctionnement. Ils ont donc réalisé une série de mesures pendant 50 minutes lors de la dernière mise en marche, et ont ainsi récupéré des valeurs de tension avec les temps de mesure correspondants (exprimés en minutes). Votre travail consiste à mettre en place un outil permettant d'analyser ces mesures et d'indiquer au bout de combien de temps la tension *moyenne* en sortie du générateur risque de passer en dessous des 240 Volts. Les responsables techniques demandent aussi de connaître le coefficient de diminution de la tension de sortie qui permettrait de tenir trois jours au dessus de 240 Volts.

6.1.1 - Droites de régression

Introduction

On rappelle que si un nuage de points donné a une forme *allongée* il existe différentes méthodes pour créer l'équation d'une droite représentant *au mieux* ce nuage. En fait, suivant la méthode, on obtient des droites différentes, à moins que les points ne soient parfaitement alignés, auquel cas toutes les méthodes se valent et donnent toutes l'équation de la droite sur laquelle sont les points du nuage.

On va envisager ici la méthode des *moindres carrés*.

Cette méthode consiste à calculer la somme des carrés des écarts entre les points du nuage et les points d'une droite *estimée* et à choisir comme meilleure droite celle pour laquelle la somme des carrés des écarts est minimum.

Il reste alors à choisir la façon de calculer les *écarts* et il y a deux choix possibles classiques :

- soit pour chaque point du nuage on utilise la différence entre l'ordonnée de ce point et celle du point de la droite qui aurait la même abscisse ;
- soit pour chaque point du nuage on utilise la différence entre l'abscisse de ce point et celle du point de la droite qui aurait la même ordonnée.

En utilisant la première méthode, on obtient la droite de régression de y en x, et en utilisant la deuxième la droite de régression de x en y.

Droite de régression de y en x

Il est tout d'abord nécessaire de structurer les données manipulées en utilisant au mieux les ressources de notre langage.

Une ***droite*** peut être représentée par deux informations :

- **a** son coefficient directeur ;
- **b** son terme constant

Un tuple Python, contenant les deux flottants dans cet ordre fera l'affaire. Par exemple la droite $y=7x-2$ est représentée par le tuple : `(7.0, -2.0)`.

De même, un ***point*** est un tuple ordonné, composé de ses deux coordonnées, abscisse et ordonnée.

Enfin un ***nuage*** sera représenté par une liste de points, c'est-à-dire une liste de tuples.

Nommage des objets

Nous allons manipuler deux objets avec chacun un fichier de points associé :

nuage — un nuage de points contenant vos données de départ ;

mc — la droite des moindres carrées, que vous allez calculer.

Conception du module de fonctions

Il faut maintenant concevoir un module contenant les différentes fonctions utiles au programme. Vous allez définir les fonctions suivantes[41], en nommant correctement les paramètres (que leur nom ait un sens et que la règle du préfixe `p_` soit respectée) :

`creeNuage()` — fonction qui reçoit un nombre de points et qui renvoie un *nuage*. Son rôle est de fournir de quoi faire des tests en générant des points aléatoires (dans certaines limites…) et en les stockant dans le fichier textuel `nuage.dat` dans votre répertoire de travail. Elle vous est fournie dans le fichier `TP6_1_m-depart.py` ;

`ecartOrdo()` — fonction qui reçoit un *nuage* et une *droite*. Cette fonction renvoie un `float` : la somme des carrés des écarts entre les ordonnées des points du nuage et des points de la droite spécifiée ;

`pointMoyen()` — fonction qui reçoit un *nuage*. Elle renvoie un *point* qui représente l'abscisse moyenne et l'ordonnée moyenne du nuage ;

`variance()` — fonction qui reçoit un *nuage* et le *point moyen* du nuage et qui renvoie le tuple de flottants (var_x, var_y) ;

`coVariance()` — fonction qui reçoit un *nuage* et le *point moyen* du nuage et qui renvoie le flottant cov_{xy} ;

`calculMC()` — fonction qui reçoit un *nuage* et qui renvoie une *droite* correspondant à la droite des moindres carrés.

---

41 Voir la section de rappels de statistiques à la fin de ce TP.

`ecritDroite()` — procédure qui reçoit un *nuage* ainsi qu'une *droite* et un *nom de fichier*. Elle ne renvoie rien mais écrit dans le fichier textuel dont on a donné le nom les points en ordonnée de la droite pour les abscisses du nuage.

Méthode de développement du module TP6_1_m

Écrire votre module en respectant la méthodologie suivante :

1. reprenez **`creeNuage`** et **`creeNuageTest`** ;
2. dans l'auto-test, pour pouvoir tester les fonctions sur des calculs réalisables à la main, choisissez la droite de test y=2x+1 et utilisez la fonction **`creeNuageTest`** qui crée un « petit nuage » de test à 3 points correspondant à cette droite : (1.0, 3.0), (2.0, 5.0) et (3.0, 7.0). L'appel à **`creeNuageTest`** remplace donc un appel à la fonction **`creeNuage`** pour les tests de validité des calculs ;
3. pour chaque fonction :
   - codez la fonction
   - la tester numériquement à l'aide de **`verif`** ou **`verifSeq`** en utilisant le nuage et la droite de test (vérifiez les fonctions au fur et à mesure)
4. toujours dans l'auto-test du module :
   - appelez la fonction **`creeNuage`** avec un nombre de points (ex. 200).
   - affichez le nuage à l'aide de la fonction **`plot_Fic`**
   - calculez la droite des moindres carrés, sauvegardez ses points dans un fichier **`mc.dat`** et, avec **`plot_Fic`**, affichez sur la même courbe cette droite ainsi que le nuage de points, afin de vérifier visuellement la corrélation. Affichez l'écart en ordonnée de la droite par rapport au nuage.

Le programme principal TP6_1

Les mesures réalisées sur le générateur électrique vous sont fournies dans le fichier **`mesures_generateur.dat`**. Celui ci contient des lignes composées de deux nombres flottants, sur deux colonnes séparées par un espace. La première colonne est le temps en minutes et la seconde colonne la mesure de tension correspondante.

Voici quelques indications pour écrire le programme principal (fichier **`TP6_1.py`**) :

- importez tout le module ***`TP6_1_m`*** ;
- depuis le module ***`plot_m`***, importez **`plot_Fic`** ;
- écrivez le code de lecture du fichier de données (rappel, cours 4 sur les fichiers) et stockez les valeurs lues dans une liste (comme le nuage de points) ;
- calculez la droite des moindres carrés pour les mesures (affichez ses paramètres) ;
- calculez et affichez la durée estimée pour arriver en moyenne en dessous de 240 Volts ;
- calculez et affichez les paramètres d'une droite qui, en partant de la même tension initiale à la mise en route du générateur, permettrait à celui-ci de tenir trois jours au dessus de 240 Volts.

Pour visualisez vos résultats, sauvegardez les points de la droite des moindres carrés dans un fichier **`mc.dat`**, les points de la droite à 3 jours dans un fichier **`d3jours.dat`** et faites un appel :

```
plot_Fic([('mesures_generateur.dat','points'),('mc.dat','lines'),
('d3jours.dat','lines')])
```

Rappels de statistique

- Pour un ensemble de points $M_i(x_i,y_i)$ où i varie de 0 à $(N-1)$, on appelle ***point moyen*** le point de coordonnées :

$$\bar{x}=\sum_{i=0}^{N-1}\frac{x_i}{N} \quad \text{et} \quad \bar{y}=\sum_{i=0}^{N-1}\frac{y_i}{N}$$

- On appelle ***variance*** d'une suite de réels x_i (respectivement y_i), où i varie de 0 à $N-1$, les réels :

$$var_x=\sum_{i=0}^{N-1}\frac{(x_i-\bar{x})^2}{N} \quad \text{et} \quad var_y=\sum_{i=0}^{N-1}\frac{(y_i-\bar{y})^2}{N}$$

- On appelle ***covariance*** de la suite (x_i,y_i), où i varie de 0 à $N-1$, le réel :

$$cov_{xy}=\sum_{i=0}^{N-1}\frac{(x_i-\bar{x})(y_i-\bar{y})}{N}$$

- Équation de la ***droite des moindres carrées*** :

$$\hat{y}=\frac{cov_{xy}}{var_x}x+\left(\bar{y}-\bar{x}\frac{cov_{xy}}{var_x}\right)$$

- Le ***coefficient de corrélation*** est enfin le réel :

$$cor_{xy}=\sqrt{\frac{cov_{xy}^2}{var_x\,var_y}}$$

7 - Introduction à la POO

On place son rêve si loin, tellement loin, tellement hors des possibilités de la vie,
qu'on ne pourrait rien trouver dans la réalité qui le satisfasse ;
alors, on se fabrique, de toutes pièces, un objet imaginaire !

Roger Martin Du Gard (Jean Barois, p. 117)

Objectifs :

- Comprendre la notion de classe et d'instance de classe (objet) ;
- maîtriser les constructeurs (avec généralement des valeurs par défaut) ;
- maîtriser la surcharge des opérateurs ;
- maîtriser la « représentation » des objets.

Fichiers à produire : **`TP7_1_m.py`** **`TP7_2_m.py`**.

N'oubliez pas de suivre les directives indiquées dans le chapitre Avant-Propos.

7.1 -Manip

Comme les objets de la vie courante, les objets informatiques peuvent être très simples ou très compliqués. Ils peuvent être composés de différentes parties, qui sont elles-mêmes des objets, ceux-ci étant constitués à leur tour d'objets plus simples, etc.

7.1.1 - Expérimenter les classes

Classe Point

En vous servant de votre cours, essayez et commentez cet exemple de la classe **`Point`** :

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-
"""Module de la classe Point."""
# fichier : point_m.py
# auteur : Bob Cordeau
# import
from math import hypot

# classe
class Point(object):
    """Definit les points de l'espace 2D."""
    def __init__(self, x=3.0, y=4.0):
        "Constructeur avec valeurs par defaut."
        self.x, self.y = x, y

    def distance(self):
        "Retourne la distance euclidienne du point à l'origine."
        return hypot(self.x, self.y)

    def __str__(self):
        "Modele d'affichage des attributs d'un Point."
        return "[x = {:g}, y = {:g} ; d = {:g}]" .format (self.x,
                 self.y, self.distance())

# Auto-test ------------------------------------------------------
```

```
if __name__ == '__main__':
    help(Point)
    p = Point()
    print (p)
    p2 = Point(3.7, 8.1)
    print (p2)
```

Bien remarquer l'utilisation de la méthode spéciale **`__init__()`** qui permet de créer automatiquement des objets **`Point`** dans un état initial connu.

Dans l'auto-test, l'opération fondamentale est l'***instanciation***, c'est-à-dire la création d'un objet de la classe **`Point`**. À la ligne suivante, on affiche l'objet **`p`** grâce à la méthode spéciale **`__str__()`** qui fournit un modèle à la fonction **`print`**.

Vous noterez enfin l'utilisation des « docstrings » dans les classes (l'appel à la documentation par **`help(Point)`** affiche bien ces informations).

Héritage

Comme nous l'avons vu en cours, l'un des principaux atouts de la *POO* (Programmation Orientée Objet) est l'*héritage* (ou la *dérivation*) : c'est la possibilité de se servir d'une classe préexistante pour en créer une nouvelle qui possédera quelques fonctionnalités différentes ou supplémentaires. Ce procédé permet de créer toute une hiérarchie de classes allant du général au particulier.

À partir de l'exemple suivant, instanciez trois objets un **`Mammifere`**, un **`Carnivore`** et un **`Chien`**, et imprimez leurs attributs. Commentez.

```
class Mammifere(object):
    def __init__(self) :
        self.caractMam = "il allaite ses petits ;"

class Carnivore(Mammifere):
    def __init__(self) :
        Mammifere.__init__(self)
        self.caractCar = "il se nourrit de la chair de ses proies ;"

class Chien(Carnivore):
    def __init__(self) :
        Carnivore.__init__(self)
        self.caractCh = "son cri se nomme aboiement ;"
```

Bien noter l'appel, dans une sous-classe, du **`__init__()`** de la classe parente, afin que celle-ci procède aussi à ses initialisations.

7.1.2 - Un module de classes

Tout comme dans le cas déjà vu des fonctions, on peut faire un module de classes. Un exemple est donné ci-dessus avec la classe **`Point`**.

7.2 -Programmation

7.2.1 - Classes parallèlépipède et cube

Module de classe TP7_1_m

Sur le modèle de la classe **`Point`**, écrivez un module « auto-testable » ***`TP7_1_m`***.

Ce module doit implémenter la classe **`Parallelepipede`** :

- prévoir un constructeur avec des valeurs par défaut pour les longueur, largeur et hauteur ;
- affectez le nom de cette figure (**`'parallèlépipède'`**) comme attribut d'instance ;
- définissez la méthode **`volume()`** qui retourne le volume d'un parallélépipède rectangle ;
- munir la classe d'une méthode de représentation permettant que l'affichage d'une instance de **`Parallelepipede`** produise :

```
Le parallèlépipède de côtés 12, 8 et 10 a un volume de 960.
```

Testez votre module, puis ajoutez une classe **`Cube`** qui hérite de la classe **`Parallelepipede`**. Son constructeur aura une valeur d'arête par défaut et, dans sa définition :

- appellera le constructeur de la classe **`Parallelepipede`** ;
- surchargera son propre nom (attribut d'instance) : **`'cube'`**.

L'affichage d'un objet **`Cube`** de côté 10 doit produire le message :

```
Le cube de côtés 10, 10 et 10 a un volume de 1000.
```

7.2.2 - Classe Fraction

Module de classe TP7_2_m

Implémentez la classe **`Fraction`** représentant les fractions rationnelles réduites.

Cette classe possède :

Un **constructeur** qui possède les propriétés suivantes :

- gestion de valeurs par défaut : **`numerateur`** et **`denominateur`** initialisés à 1,
- interdiction d'instancier une fraction ayant un dénominateur nul (dans ce cas on lèvera une exception **`ValueError`** contenant un message indiquant l'erreur),
- définition de trois attributs d'instance :
  - **num** la valeur absolue du numérateur ;
  - **den** la valeur absolue du dénominateur ;
  - **signe** le signe de la fraction[42], qui vaudra **`1`** si numérateur * dénominateur ≥0, **`-1`** sinon.
- enfin rendre une fraction simplifiée, si c'est possible, grâce à l'appel à la méthode **`simpliFrac()`**.

Une **méthode spéciale** **`__str__()`**, permettant d'afficher par exemple :

```
>>> print (Fraction(6, -9))
```

42 Mettre le signe à part permet de gérer élégamment la représentation des fractions du type $\frac{a}{-b}$.

```
(-2,3)
```

Les méthodes de **surcharge d'opérateurs** :

- `__neg__(self)` retourne la Fraction opposée (`- f`)
- `__add__(self,other)` retourne la Fraction somme (`f + g`)
- `__sub__(self,other)` retourne la Fraction différence (`f - g`)
- `__mul__(self,other)` retourne la Fraction produit (`f * g`)
- `__truediv__(self,other)` retourne la Fraction rapport (`f / g`)
- `__floordiv__(self,other)` retourne la Fraction rapport (`f // g`)

Une **méthode `simpliFrac()`** faisant appel à une méthode **`PGCD()`** détaillée plus bas.

> Le rôle de cette procédure est de changer les attributs d'instance de la fraction sur laquelle elle travaille afin que celle-ci soit réduite.

Voici les tests que vous devez réaliser et commenter :

```
# Auto-test ------------------------------------------------------
if __name__ == '__main__':
    f = Fraction()
    print ("Constructeur par défaut : f =", f)
    #~ print "\nFraction interdite :" # à tester une fois !
    #~ frac = Fraction(2, 0)
    g = Fraction(10, 30)
    print ("\ng =", g)
    print ("\topposée :", -g)
    print ("\tg.num =", g.num)
    print ("\tg.den =", g.den)
    h = Fraction(7, -14)
    print ("\nh =", h)
    print ("\ng + h =", g+h)
    print ("g - h =", g-h)
    print ("g * h =", g*h)
    print ("g / h =", g/h)
```

Algorithme d'Euclide - calcul du PGCD (Plus Grand Commun Diviseur)

Pour vous aider à simplifier vos fractions, voici un codage de l'algorithme d'EUCLIDE, suivi d'explications.

```
class Fraction(object) :
    …
    def PGCD(self) :
        "Algorithme d'Euclide"
        x,y = self.num,self.den
        while y :
            x,y = y, x%y
        return x
```

Si *a* et *b* sont deux nombres entiers naturels (*non nuls*) qui ne sont pas premiers entre eux, l'ensemble de leurs diviseurs communs comporte d'autres éléments que le nombre 1. Cet ensemble admet un plus grand élément appelé le PGCD.

Il y a deux façons de procéder pour rechercher le PGCD de deux nombres.

Par exemple pour 360 et 108 :

1. par l'utilisation de la décomposition en produits de facteurs premiers :

   $$360 = 2^3.3^2.5$$
   $$108 = 2^2.3^3$$

   Le PGCD est obtenu en faisant le produit de tous les facteurs qui figurent à la fois dans les deux décompositions avec leur exposant le plus bas :

   $$pgcd = 2^2.3^2 = 36$$

2. Par l'algorithme d'EUCLIDE. On prend le reste précédent le reste nul :

   360/108=3, **reste 36**

   108/36=3, reste 0

On rappelle la propriété suivante :

$$PGCD(x,y).PPCM(x,y)=x.y$$

8 - Partiel blanc de révision

Je m'étais rendu compte que seule la perception grossière et erronée
place tout dans l'objet, quand tout est dans l'esprit...

Proust (le Temps retrouvé)

Objectifs :

Se préparer en travaillant sur deux sujets déjà proposés lors de partiels.

Fichiers à produire : **`TP8_1_m.py`** **`TP8_1.py`** **`TP8_2_m1.py`** **`TP8_2_m2.py`** **`TP8_2.py`**.

N'oubliez pas de suivre les directives indiquées dans le chapitre Avant-Propos.

8.1 -Programmation

8.1.1 - Le jeu de Marienbad

Ce jeu – appelé également « jeu des allumettes » – nécessite deux joueurs et 21 allumettes.

Les 21 allumettes sont réparties en 6 tas, avec **`i`** allumettes dans le **`i`**^{e} tas : une allumette dans le premier tas, deux dans le deuxième, etc.

Chacun à son tour, les joueurs piochent dans un seul tas le nombre d'allumettes souhaité. *Le joueur qui prend la dernière allumette perd la partie.*

Votre programme devra faire jouer deux joueurs humains. Au cours du jeu, l'affichage se présentera simplement sous la forme suivante (si Bob est l'un des deux joueurs)

```
tas        : (1, 2, 3, 4, 5, 6)
allumettes : [0, 2, 3, 2, 4, 4]
-------- Prochain joueur : Bob --------
```

Le module de classe TP8_1_m

Ce module contient la classe **`Marienbad`** qui comprend les méthodes suivants :

- un constructeur **`__init__()`** permettant de spécifier les noms des joueurs **`p_joueur1`** et **`p_joueur2`**, avec des valeurs par défaut adaptées (**`"joueur1"`** et **`"joueur2"`**), et qui met en place les attributs d'instance suivants :
  - **`tas`** de type tuple d'entier avec des valeurs de 1 à 6,
  - **`allumettes`** de type liste d'entier avec les valeurs initiales des tas d'allumettes,
  - **`joueurs`** de type tuple de deux chaîne de caractères initialisé à partir des paramètres de construction,
  - **`tour`** un entier qui permet d'alterner les joueurs à chaque tour (indication : il sert d'indice à l'attribut **`joueurs`**).
- **`__str__()`** qui affiche l'état du jeu en cours.
- **`verifie()`** qui renvoie un booléen : vrai s'il est possible d'enlever **`p_n`** allumettes dans le tas **`p_t`**, faux sinon, la fonction devra vérifier non seulement si le nombre d'allumettes restantes est suffisant, mais aussi que le numéro du tas est valide (si ce n'est pas le cas, elle devra retourner faux).
- **`enlever()`** qui réalise un tour de jeu :mise à jour des tas en enlevant **`p_n`** allumettes dans le tas **`p_t`** et évolution de l'indicateur du numéro de tour en cours. Cette méthode considérera que **`verifie()`** a été précédemment appelée et ne fera donc aucun contrôle.
- **`termine()`** qui renvoie un booléen : vrai si le jeu est terminé (il n'y a plus d'allumette), faux sinon.

Le module principal TP8_1

Dans le programme principal :

- Saisissez le nom des deux joueurs puis créez un objet représentant une nouvelle partie du jeu ;
- lancez le jeu (c'est une boucle tant que le jeu n'est pas terminé).

  À chaque tour vous devez :

  - afficher l'état du jeu,
  - demander au joueur en cours le numéro du tas **`t`** et le nombre d'allumettes **`n`** qu'il désire ôter,
  - vérifier si son choix est valide, sinon expliquer l'erreur et recommencer un tour de jeu avec le même joueur,
  - supprimer **`n`** allumettes du tas **`t`**,
  - vérifiez si le jeu est terminé,
- annoncez le gagnant (ou le perdant).

8.1.2 - Les polynômes

Introduction

Un polynôme de degré n est de la forme :

$$a_0 + a_1x + a_2 x^2 + a_3 x^3 \ldots + a_n x^n$$

On va stocker les polynômes sous la forme de tuples : $(a_0, a_1, a_2, a_3, \ldots, a_n)$.

Par exemple le polynôme :

$$7 + 2.3\ x - 9.12\ x^2 + 7.8\ x^4$$

Est stocké sous la forme :

```
(7, 2.3, -9.12, 0, 7.9)
```

Un tuple représentant un polynôme doit contenir **au moins** une valeur, éventuellement nulle. Attention, un tuple à une seule valeur s'exprime avec une virgule (ex: **`(3,)`**).

Le module de fonctions TP8_2_m1

Ce module contiendra les fonctions suivantes :

`poly_calcul(p, x)` — Prend en paramètre un tuple polynôme **`p`** et une valeur numérique **`x`** et retourne l'évaluation numérique du polynôme avec cette valeur :

```
>>> poly_calcul((7,2,3),2)
23
```

`poly_coef(p)` — Prend en paramètre un tuple polynôme **`p`** et retourne le degré le plus élevé du polynôme ayant un coefficient non nul, ou 0 si tous les coefficients sont nuls ;

`poly_add(p1, p2)` — Prend en paramètres deux tuples polynômes **`p1`** et **`p2`**, et retourne le tuple polynôme correspondant à la somme de **`p1`** et **`p2`** (polynôme ayant comme coefficient pour chaque degré la somme des coefficients de **`p1`** et **`p2`** pour ce degré).

`poly_chaine(p)` — Prend en paramètre un tuple polynôme **`p`** et retourne une chaîne correspondant à une forme lisible de ce polynôme :

```
>>> poly_chaine((23.4, 1.1, 11.3, 0, 0, 4, 12.85))
"23.4+1.1x+11.3x^2+4x^5+12.85x^6"
```

`poly_simplifie(p)` — qui prend en paramètre un tuple polynôme **`p`** et retourne un tuple polynôme équivalent en ayant supprimé les coefficients nuls des degrés supérieurs au degré du polynôme :

```
>>> poly_simplifie((4, 0, 12, -2, 0, 4, 9.3, 0, 0, 0))
(4, 0, 12, -2, 0, 4, 9.3)
```

Le module de classe TP8_2_m2

Ce module définira la classe **`Polynome`**, et utilisera les fonctions du module de fonctions afin de définir les méthodes suivantes :

- méthode d'initialisation, permettant de créer une instance de la classe **`Polynome`** avec comme argument un tuple contenant les coefficients du polynôme tel que défini précédemment ;
- méthode de représentation textuelle, retournant la représentation d'un polynôme sous forme de chaîne lisible ;
- méthode d'addition, permettant d'ajouter une instance de la classe **`Polynome`** à une autre instance de cette classe, et retournant une nouvelle instance de **`Polynome`** correspondant à la somme des deux autres ;
- méthode **`evaluer(self, x)`** permettant de réaliser l'évaluation du polynôme avec la valeur **`x`** donnée et retournant le résultat de cette évaluation ;
- méthode **`degre(self)`** retournant le degré du polynôme.

Le module principal TP8_2

Ce module importe la classe **`Polynome`** et réalise les deux traitements suivant :

- Saisie des degrés d'un polynôme (on pourra utiliser la fonction standard **`eval("chaîne")`** afin de permettre à l'utilisateur de saisir l'ensemble des coefficients séparés par des virgules.
- Création d'un fichier texte **`polyres.txt`** contenant en première ligne la représentation textuelle lisible du polynôme, et dans les lignes suivantes, les valeurs entières de -10 à 10 suivies d'un caractère tabulation (**`"\t"`**) de l'évaluation de ce polynôme pour cet entier.

9 - Annexes

9.1 -Bibliographie

[1] Swinnen, Gérard,
Apprendre à programmer avec Python 3,
Eyrolles, 2010.

[2] Summerfield, Mark,
Programming in Python 3,
Addison-Wesley, 2e édition, 2009.

[3] Martelli, Alex,
Python en concentré,
O'Reilly, 2004.

[4] Lutz, Mark et Bailly, Yves,
Python précis et concis,
O'Reilly, 2e édition, 2005.

[5] Ziadé, Tarek,
Programmation Python. Conception et optimisation,
Eyrolles, 2e édition, 2009.

[6] Ziadé, Tarek,
Python : Petit guide à l'usage du développeur agile,
Dunod, 2007.

[7] Ziadé, Tarek,
Expert Python Programming,
Packt Publishing, 2008.

[8] Younker, Jeff,
Foundations of Agile Python Development,
Apress, 2008.