1
Bonnes pratiques pour le TIPE

1. Un TIPE est l'occasion de traiter des données numériques, en volume plus ou moins important.
Avant de se lancer, il faut être conscient d'un certain nombre de *bonnes pratiques* qui faciliteront ce travail et amélioreront sa qualité.
Ceux qui se lanceront sans avoir pris conscience de ces bonnes pratiques les découvriront à l'usage (je suis optimiste), à leurs dépens, et l'énergie qu'ils auront perdue à ces occasions développera leur mémoire et l'importance qu'ils attacheront à ces bonnes pratiques (ça sent l'expérience personnelle, hein ?).

2. Commentaires du code
Tout code informatique doit être commenté, cette injonction est assez souvent répétée pour être bien connue.
Mais quels sont les vrais enjeux ?
2.1 Il importe de comprendre que *le premier destinataire de ce code est... soi-même.*
Une particularité du TIPE est la durée de sa réalisation : ce travail s'étend sur plusieurs mois. Si on doit modifier du code qu'on a écrit deux ou quatre mois auparavant, il est probable qu'on n'aura pas conservé tous les détails en mémoire. Les commentaires doivent servir à comprendre rapidement et précisément un code qu'on aura écrit, qu'on aura oublié et qu'on doit relire.
2.2 Il y a deux niveaux de lecture du code :
- un niveau superficiel, où on se contente de connaître les arguments et les valeurs renvoyées par chaque fonction (et peu importe comment ça marche) ;
- un niveau approfondi, où on doit comprendre le fonctionnement interne des fonctions.

La lecture superficielle suffit pour créer un nouveau code utilisant les fonctions existantes (dans le cas où les fonctions existantes suffisent à nos nouveaux besoins).
La lecture approfondie permet de décider si on peut modifier une fonction existante pour qu'elle réponde à de nouveaux besoins ou s'il faut écrire une nouvelle fonction.
2.3 Avec un peu de pratique, un choix judicieux des noms de variables et des noms de fonctions permet de comprendre le code dès la première lecture. Dans ces conditions, les commentaires sont superflus !

Pour écrire un bon code, il faut chercher à écrire un code qui rende les commentaires inutiles.

À ce sujet, une lecture attentive de *Coder proprement* de Robert C. Martin (Pearson, 2009) sera source de profit.

2.4 Les deux codes suivants produisent le même résultat : une valeur approchée de l'intégrale de f sur un segment $[a,b]$ calculée en suivant la méthode des rectangles. Le second code est plus efficace que le premier (moitié moins de calculs) mais le premier code est bien plus parlant.

```
def integrales(f, intervalle, N):
    a, b = intervalle
    dx = (b-a)/N
    aire_totale = 0.
    for i in range(N):
        x, y = a+i*dx, a+(i+1)*dx
        aire_trapeze = dx*(f(x) + f(y))/2
        aire_totale += aire_trapeze
    return aire_totale
```

```
def integrales(f, a, b, N):
    d = (b-a)/N
    s = (f(a)+f(b))/2
    for i in range(1, N):
        s += f(a+i*d)
    return s*d
```

2.5 Le code suivant est encore plus efficace, puisqu'il tire parti de la puissance du module numpy. Les noms de variables ont été soigneusement choisis pour rester au plus près des idées mathématiques.
Mais qui, à part un pythonien fervent, peut trouver ce code limpide ?

```
import numpy as np
def integrales(f, intervalle, N):
    a, b = intervalle
    X = np.linspace(a, b, N+1)    # (N+1) points pour N sous-segments
    Y = f(X)                      # si la fonction f le permet !
    base = (b-a)/N
    aires_trapezes = base*(Y[1:]+Y[:-1])/2
    return np.sum(aires_trapezes)
```

3. Mise à l'épreuve du code

Il est facile de comprendre qu'il est plus facile de corriger un code de quelques lignes qu'un code de plusieurs pages — c'est la vieille histoire de l'aiguille dans la meule de foin.
Il faut en tirer les conséquences : *chaque fonction écrite doit être testée sans attendre*, c'est-à-dire avant d'écrire une autre fonction.
Quelle doit être l'attitude d'un professeur ? Si votre code ne fonctionne pas, il efface toutes les fonctions qui n'ont pas été testées sauf une. Vous n'avez testé aucune des fonctions que vous avez écrite ? Il vous reste la première fonction que vous avez écrite... À vous de la tester !

4. Représentation des données

Les données numériques (résultats des mesures ou bilans des opérations de traitement des mesures) doivent être présentées de façon claire et honnête.
La *clarté* est obtenue par une **représentation graphique** (courbe, histogramme, nuage de points...) dotée d'un titre et d'une légende.
L'*honnêteté* consiste à ne pas camoufler un fait important et à ne pas mettre en valeur un fait secondaire. Il faut savoir résister à la tentation...

5. Persistance des données

On distingue deux types de données : les données produites par un algorithme *déterministe* et les données *aléatoires*.

5.1 Un **algorithme déterministe** produira exactement les mêmes résultats à chaque exécution. Mais si l'exécution d'un tel algorithme est longue (plusieurs minutes ou plusieurs heures), il n'est pas raisonnable de devoir exécuter plusieurs fois cet algorithme : les valeurs produites par cet algorithme doivent être enregistrées pour pouvoir être traitées plusieurs fois sans avoir à les calculer à nouveau.
Qu'on pense à un cas extrême : il est important de représenter graphiquement les résultats de l'algorithme, il n'est pas raisonnable de recommencer tous les calculs pour changer le titre de la figure ou la couleur de la couleur.
5.2 Un **algorithme aléatoire** (utilisant le module `random`) produira des résultats différents à chaque fois, de même que des **mesures expérimentales**.
Il est très intéressant dans ce cas d'enregistrer les données recueillies afin de pouvoir comparer les résultats obtenus lors de plusieurs expériences. C'est ainsi qu'on pourra se faire une idée juste de ce qui est permanent (= ce qu'on retiendra de l'expérience) et de ce qui est variable (= ce qui relève des incertitudes) dans les résultats.
5.3 Dans tous les cas, *les résultats obtenus doivent être enregistrés en vue d'être traités.*

6. Sécurité des données

Les données que vous avez obtenues dans votre travail doivent être conservées **en lieu sûr**.
Mais dans le monde de l'informatique, aucun lieu n'est sûr. La sécurité de vos données sera assurée par la multiplication des supports de sauvegarde :

— une copie sur votre disque dur,

— une copie sur un support amovible (clé USB, disque dur externe...),

— une copie en ligne

et vous pourrez commencer à être tranquille pour vos résultats.
Rappelez-vous le proverbe : *En informatique, la paranoïa est le début de la sécurité.*

2
Chaînes de caractères

1. Nous allons passer en revue les opérations les plus utiles sur les chaînes de caractères, toujours dans l'optique d'écrire et de lire des fichiers contenant des données numériques.

2. Mon objectif est de vous indiquer ce qu'il est possible de faire avec Python (à peu près tout est possible et déjà programmé...). Il vous appartiendra de tirer le meilleur parti de ces ressources pour obtenir un résultat de qualité avec un minimum de peine.
N'hésitez pas à me faire part de vos demandes si vous êtes bloqués sur un point : il y a sans doute une solution simple.

I
Caractères d'échappement

3. Les **caractères d'échappement** sont des caractères spéciaux : ils sont traités comme des caractères bien qu'on les écrive sous la forme de chaînes de deux caractères.
Nous présentons ici les cinq caractères les plus fréquemment utilisés.

4. Fin de ligne
En insérant le caractère `\n` dans une chaîne, on écrit cette chaîne sur deux lignes.

```
print("Bonjour le monde")   # une seule ligne
print("Bonjour\nle monde")  # deux lignes
```

5. Tabulation
Pour reproduire l'effet de la touche de tabulation dans un texte, il faut insérer le caractère `\t` dans ce texte.

```
print("Bonjour\tles\tpetits\tenfants !")
print("À\tdemain\tles\tparents !")
```

6. Guillemets
Une chaîne de caractères peut être délimitée au choix par des guillemets simples `'` ou des guillemets doubles `"`.
Pour insérer des guillemets dans une chaîne de caractères, il faut recourir aux caractères d'échappement `\'` et `\"`.

```
print("Il prétend \"Elle m\'a dit \'Va voir !\' et je n\'ai rien vu\".")
```

7. Contre-oblique
Tous les caractères d'échappement commencent par une contre-oblique `\`. Il faut donc recourir au caractère d'échappement `\\` pour insérer une contre-oblique dans une chaîne de caractères.

```
print("On va à la ligne en insérant le caractère spécial \\n.")
```

II

Fichiers csv

8. Structure des fichiers csv

Les fichiers csv (= *comma separated value*) sont les fichiers de tableur les plus simples :

- ils ne contiennent qu'une seule feuille de calcul ;
- ce sont des fichiers textes, lisibles par un œil humain, par opposition aux fichiers binaires, que seule la machine est capable d'interpréter ;
- aucune mise en page n'est possible, les fonctions du tableur ne sont pas enregistrées (seul le résultat numérique produit par une fonction est enregistré) ;
- chaque ligne de la feuille de calcul est représentée par une ligne dans le fichier : les données présentes sur cette ligne sont séparées par un délimiteur (en général point-virgule, virgule ou tabulation, mais aussi deux-points ou espace).

9. Si la structure des fichiers csv vous est inconnue, il est intéressant d'ouvrir un fichier de tableur (xls, ods, xlsx...) et le fichier csv correspondant avec un éditeur de texte.
On notera aussi que la taille des fichiers csv est sensiblement inférieure à celles des autres fichiers.

10. Élimination des espaces avec strip

La fonction `strip` permet de supprimer tout ce qui est inutile, c'est-à-dire les espaces au début et à la fin d'une chaîne de caractères. C'est en particulier utile pour supprimer le caractère `\n` qui figure à la fin des chaînes de caractères obtenues lors de la lecture ligne par ligne d'un fichier.

```
s = "  Un petit texte sans importance. \n"
print("|"+s+"|")           # texte écrit sur deux lignes
print("|"+s.strip()+"|")   # texte écrit sur une seule ligne
```

11. Consulter la documentation en ligne pour l'usage d'arguments avec `strip` (ce qui permet de supprimer autre chose que des espaces) et pour découvrir les fonctions analogues `lstrip` et `rstrip` (pour limiter la portée de `strip` à la partie gauche ou à la partie droite de la chaîne de caractères).

12. Écriture d'une ligne

Dans un fichier, on ne peut écrire que des chaînes de caractères, pas des nombres !

12.1 Pour écrire une ligne dans un fichier csv, il faut

- d'abord convertir les valeurs à enregistrer en chaînes de caractères ;
- regrouper ensuite ces chaînes de caractères dans un itérable (liste, tableau, tuple...) ;
- fusionner ces chaînes de caractères en une seule chaîne de caractères, en prenant soin d'insérer un séparateur entre les différentes valeurs ;
- ajouter un saut de ligne `\n` à la fin de cette chaîne avant de l'écrire dans le fichier.

12.2 Commande join

La commande `join` est très pratique ! Elle s'applique à la chaîne de caractères qui sert de séparateur (le point-virgule dans l'exemple suivant) et prend l'itérable en argument.

```
L_S = [ str(x) for x in L ]     # conversion en chaînes de caractères
ligne = ";".join(L_S)           # fusion des chaînes séparées par des points-virgules
fichier.write(ligne+"\n")       # écriture dans un fichier avec saut de ligne
```

Sans la commande `join`, le code serait bien plus long — et surtout moins clair !

```
ligne = L_S[0]
for i in range(1, len(L_S)):
    ligne += ";" + L_S[i]
```

13. Lecture d'une ligne

En lisant un fichier, on ne récupère que des chaînes de caractères — pas des nombres !

13.1 Au moyen d'une boucle (`for` ou `while`), on récupère chaque ligne du fichier sous la forme d'une chaîne de caractères. Pour retrouver les données numériques correspondantes, il faut

- commencer par éliminer le saut de ligne à la fin de la chaîne (avec `strip`);
- puis tronçonner la chaîne de caractères en fonction du délimiteur à l'aide de la fonction `split`;
- on dispose alors d'une liste de chaînes de caractères, qu'il faut convertir en liste d'entiers ou en liste de flottants (selon les données qu'on a à traiter).

```
ligne    = ligne.strip()                  # élimination des espaces
data_str = ligne.split(";")               # séparation des données
data_num = [ float(s) for s in data_str ] # conversion des données en nombres
```

13.2 Commande split

La fonction `split` sert à tronçonner une chaîne de caractères en fonction d'un délimiteur. Le délimiteur disparaît du résultat et le résultat obtenu dépend du délimiteur.

```
s = "0.123,4.567,8.901"
data_virgule = s.split(",")     # liste de 3 chaînes
data_point   = s.split(".")     # liste de 4 chaînes
data_etoile  = s.split("*")     # liste d'une seule chaîne
```

III

Présentation des nombres

14. Qu'il s'agisse d'afficher des résultats à l'écran ou de les enregistrer dans un fichier, il est utile de bien présenter ces résultats pour en faciliter la lecture.

15. Principe général

Nous allons maintenant voir comment modifier des chaînes de caractères en fonction de divers paramètres, en ayant pour objectif de mettre en forme des nombres.

Une chaîne de caractères peut être modifiée à l'aide de la commande `format` selon le principe général suivant.

```
s = "bla bla {champ1:specification1} bla bla {champ2:specification2}"
print(s.format(champ1, champ2))
```

Les parties modifiables sont indiquées par un couple d'accolades. Chaque accolade fait référence à un **champ de remplacement** (= l'un des arguments de la fonction `format`) et à une **spécification** qui indique comment mettre en forme ce champ.

III.1 Syntaxe des champs de remplacement

16. Un champ de remplacement peut être une chaîne de caractères ou, plus généralement, tout objet que Python sait transformer en chaîne de caractères (par exemple un nombre).
Il n'est donc pas nécessaire de convertir explicitement le champ de remplacement en chaîne de caractères.

```
x = 3.141592                # flottant
print(x)                    # conversion automatique
print("pi = "+x)            # erreur !
print("pi = "+str(x))       # erreur corrigée
print("pi = {}".format(x)) # conversion automatique
```

17. Les champs peuvent rester vides : dans ce cas, il faut que le nombre de `{}` soit égal au nombre d'arguments de `format` : ces arguments sont alors distribués dans l'ordre.

```
s = "Bonjour {}, bonjour {} et bonjour {}."
print(s.format("mesdames", "mesdemoiselles", "messieurs"))
```

18. Si un argument de `format` doit être utilisé plusieurs fois, il est pratique d'indiquer entre les accolades l'**indice** de chaque argument.

```
s = "Bonjour {0}, bonjour {1} ! \nAu revoir {0}, au revoir {1}"
print(s.format("mesdames", "messieurs"))
```

Il peut être intéressant de remplacer les indices par des **clés** : le code est alors plus lisible et les valeurs associées aux différentes clés peuvent être données dans un ordre quelconque.

```
s = "Bonjour {mesdames}, bonjour {messieurs}"
S = s.format(messieurs="Castor et Pollux", mesdames="Hélène et Clytemnestre")
print(S)
```

19. La documentation en ligne décrit d'autres possibilités pour définir les champs de remplacement (notamment inspirées de la programmation orientée objet).

III.2 Syntaxe des spécifications

20. Dans les exemples précédents, les champs de remplacement étaient intégrés tels quels dans la chaîne de caractères.
Quand on manipule des nombres, il peut être intéressant (pour des raisons de lisibilité) de spécifier une manière particulière de représenter ces nombres.
Je rappelle que je présente ici une version *simplifiée* du formatage des données.

21. Le format standard des spécifications est le suivant.

```
[[fill]align][sign][0][width][.precision][type]
```

Il y a donc six paramètres possibles et tous ces paramètres sont optionnels (c'est la raison pour laquelle ils se trouvent entre crochets). Si une spécification est choisie (c'est-à-dire dès qu'un des six paramètres est fixé), elle doit toujours être précédée de `:` pour la distinguer d'un champ.

21.1 Le paramètre de remplissage `fill` est une chaîne de caractères quelconque. S'il est présent, ce paramètre doit impérativement être suivi par le paramètre d'alignement `align`, qui est l'un des quatre caractères suivants : `'<'`, `'>'`, `'='` ou `'^'`.

21.2 Le paramètre `sign` est l'un des trois caractères suivants : `'+'`, `'-'` ou `' '`.

21.3 Les paramètres `width` et `precision` sont des entiers *strictement* positifs. D'après le format standard, l'entier `precision`, s'il est présent, doit impérativement être précédé d'un point.
21.4 Le paramètre `type`, s'il est présent, est un caractère choisi dans une liste assez vaste. On y reviendra plus loin.

22. Analyse de la spécification
Il est important de comprendre que les valeurs possibles et les positions respectives des différents paramètres permet de savoir sans ambiguïté quels sont les paramètres présents et quels sont les paramètres omis, ainsi que les valeurs respectives des différents paramètres présents. C'est assez astucieux !
22.1 Pour la spécification `{:0>012.3f}`, l'alignement suit le paramètre `'>'` avec le caractère `'0'` pour le remplissage ; le signe n'est pas précisé (valeur par défaut) ; il y un `'0'` avant la largeur, qui prend la valeur 12 ; la précision est fixée à 3 et le type de nombre est `'f'`.
22.2 Autre exemples.

Spécification	fill	align	sign	0	width	precision	type
`{:>+8}`	défaut	`'>'`	`'+'`	non	8	défaut	
`{:>=+08.4e}`	`'>'`	`'='`	`'+'`	oui	8	4	`'e'`
`{:020.12}`		non	défaut	oui	20	12	défaut
`{:<<16.8g}`	`'<'`	`'<'`	défaut	non	16	8	`'g'`
`{:0>4}`	`'0'`	`'>'`	défaut	non	4	défaut	défaut
`{:.4f}`		non	défaut	non	défaut	4	`'f'`

23. Signe
Par défaut, la valeur du paramètre `sign` est le caractère `'-'`. Dans ce cas, seul apparaît le signe des nombres négatifs.
Lorsque ce paramètre prend la valeur `'+'`, tous les nombres sont affichés avec un signe, qu'ils soient positifs ou négatifs.
Enfin, avec la valeur `' '` (c'est un espace), les nombres négatifs sont affichés avec un signe `'-'` et un espace remplace le signe `'+'` devant les nombres positifs.

```
spec = [ "({:+})", "({:-})", "({: })" ]
valeurs = [ 123, -456, 12.34, -56.789 ]
for s in spec:
    for v in valeurs:
        print(s.format(v))
```

24. Alignement
Pour que l'alignement apparaisse, il faut imposer une largeur, c'est-à-dire attribuer une valeur au paramètre `width`, en s'assurant que les données (nombres ou chaînes de caractères) à représenter n'excèdent pas cette largeur.
24.1 Le remplissage est assuré par défaut par des espaces. On peut choisir un caractère de remplissage arbitraire en donnant une valeur particulière au paramètre `fill`.

24.2 Lorsque le paramètre d'alignement prend la valeur `'^'`, le champ de remplacement est centré dans l'espace disponible.
Avec `'>'`, le champ de remplacement est aligné à droite (valeur par défaut pour les nombres).
Avec `'<'`, il est aligné à gauche (valeur par défaut pour les chaînes de caractères.

```
spec = [ "({:>12})", "({:<12})", "({:*>12})", "({:-<12})", "({:#>12})" ]
x = 12.34
for s in spec:
    print(s.format(x))
```

Le code précédent affiche successivement :

```
(       12.34) (12.34       ) (*******12.34) (12.34-------) (#######12.34)
```

24.3 La valeur d'alignement `'='` ne s'applique qu'aux nombres : dans ce cas, le signe du nombre (s'il y en a un) est aligné à gauche et la valeur absolue du nombre est alignée à droite.
Avec le code

```
spec = [ "({:>12})", "({:=12})", "({:*=12})", "({:#<12})", "({:#>12})" ]
x = -12.34
for s in spec:
    print(s.format(x))
```

on obtient cette fois :

```
(      -12.34) (-      12.34) (-******12.34) (-12.34######) (######-12.34)
```

Cas des entiers

25. Pour les entiers, le paramètre `type` peut être omis.
25.1 Le paramètre `precision` est interdit : faute de virgule, il ne saurait y avoir de chiffres après la virgule. la présence de ce paramètre produirait une erreur.
25.2 Le paramètre `width` donne la longueur minimale de la chaîne de caractères produite.
Si le nombre de chiffres est inférieur à cette largeur, la chaîne est complétée par des espaces (à gauche du nombre).
Si le nombre de chiffres est supérieur ou égal à cette largeur, aucun espace n'est introduit et le nombre entier est affiché entièrement (pas de troncature).
Ainsi, la chaîne définie par `"{:8}".format(12345)` est égale à `'  12345'` alors que la chaîne définie par `"{:3}".format(12345)` est égale à `'12345'`.

26. Exemple
Un programme renvoie une liste d'entiers compris entre 0 et 9 999 et on souhaite afficher lisiblement la liste de ces entiers.
26.1 Comme chaque entier est écrit avec quatre décimales au plus, on va représenter chaque entier par une chaîne de six caractères (dont au moins deux espaces).
26.2 Par défaut, on va afficher les entiers sur dix colonnes. Il faut donc découper la liste des entiers en sous-listes de longueur 10.

26.3

```
def afficher(liste_entiers, n_col=10):
  nb_lig = len(liste_entiers)//n_col + 1
  for i in range(nb_lig):
    rangee = liste_entiers[n_col*i : n_col*(i+1)]  # tranche de 10 éléments
                                                   # conversion en chaînes de longueur 6
    rangee_s = [ "{:>6}".format(j) for j in rangee ]
    print("".join(rangee_s))
```

Cas des flottants

27. Pour les flottants, on a le choix entre :

— la notation scientifique avec `e` ou `E` ;

— la notation en virgule fixe avec `f` ou `F`, pour laquelle le paramètre `precision`, s'il est indiqué, est un entier strictement positif qui fixe le nombre de chiffres après la virgule ;

— la notation générale (mi-flottante, mi-scientifique, selon ce que Python juge préférable) avec `g` ou `G`, pour laquelle le paramètre `precision` fixe le nombre total de chiffres significatifs (et non plus le nombre de chiffres après la virgule) ;

— les pourcentages avec `%`.

IV

Localisation

28. On appelle **localisation** la prise en compte des particularités typographiques de chaque langue : choix du séparateur décimal, séparation des milliers, composition de la date, présentation des unités monétaires...

IV.1 La localisation du pauvre

29. Le principal problème qu'on peut rencontrer dans le cadre d'un TIPE tient au **séparateur décimal** : pour Python, les flottants sont écrits à l'aide d'un *point* (notation anglo-saxonne) tandis que pour des logiciels *francisés* (le tableur de LibreOffice pour ne citer que lui), les flottants sont écrits avec une *virgule* (notation française).

30. Il est impossible de faire travailler Python avec des virgules, il n'est pas recommandé de reconfigurer LibreOffice en anglais. C'est donc au moment où Python et LibreOffice vont échanger des fichiers que nous allons intervenir.

31. Pour écrire un fichier csv compatible avec la version francisée de LibreOffice, il faut donc transformer les points décimaux en virgules.
Si nous disposons d'une chaîne de caractères

```
s = "1.23;2.34;3.45;4.56;5.67;7.89\n"
```

nous pouvons la découper en fonction du séparateur décimal avant de recoller les morceaux à l'aide du bon séparateur décimal.

```
L = s.split(".")
s = ",".join(L)
```

On peut alors écrire cette chaîne de caractères dans le fichier csv, que LibreOffice lira sans encombres.

32. Inversement, si la lecture d'un fichier csv nous donne la chaîne de caractères

```
s = "1,23;2,34;3,45;4,56;5,67;7,89\n"
```

la même méthode nous permet de transformer les virgules en points.

```
L = s.split(",")
s = ".".join(L)
```

On peut alors découper la chaîne de caractères selon les points-virgules et convertir les sous-chaînes de caractères en flottants compréhensibles par Python.

```
L = s.strip().split(";")
data = [ float(x) for x in L ]
```

IV.2 Le module locale

33. Le module `locale` a pour objectif de réaliser les opérations de localisation de manière rigoureuse, sans qu'on ait besoin de s'occuper des détails.
Cependant, ce qui fonctionne efficacement avec Linux ne fonctionne pas, ou mal, avec Windows. Il est probable que vous ne pourrez pas utiliser ce qui suit au lycée...

34. Avant toute chose, il faut savoir quelle localisation est prévue par défaut sur l'ordinateur. L'instruction

```
locale.getlocale()
```

renvoie par exemple `('fr_FR', 'UTF-8')` : le clavier est français (de France...) et les caractères suivent l'encodage Unicode.

35. Activation
Pour activer la localisation par défaut, il faut le demander explicitement.

```
locale.setlocale(locale.LC_NUMERIC, '') # pour les nombres seulement
locale.setlocale(locale.LC_ALL, '')     # pour les nombres, dates, unités monétaires
```

On peut activer n'importe quelle localisation, du moment qu'elle est comprise par la machine — *ηλασ σελακηλοσ*.

36. Utilisation
On peut alors respecter les codes typographiques (pour les chaînes de caractères et les fichiers) ainsi que la représentation des nombres entiers et flottants en Python.

36.1 Pour convertir une chaîne de caractères en nombre, il faut utiliser l'une des commandes suivantes.

```
n = locale.atoi(s)    # au lieu de int(s) pour les entiers
x = locale.atof(s)    # au lieu de float(s) pour les flottants
```

36.2 Pour convertir un nombre en chaîne de caractères, il faut utiliser la commande suivante.

```
s = locale.str(x)
```

On peut aussi laiser la commande `format` travailler à notre place en indiquant qu'il s'agit d'un nombre de type `'n'` (pour les entiers aussi bien que pour les flottants). Pour les flottants, cela revient à choisir le type `'g'` mais en tenant compte de la localisation choisie.

```
s = "La valeur cherchée est {:.4n}.".format(x)
```

Si on préfère utiliser la notation en virgule fixe, il faut recourir à une autre commande et à une autre syntaxe. (Mais pourquoi ?)

```
s = locale.format_string("La valeur cherchée est %4.1f.", x)
```

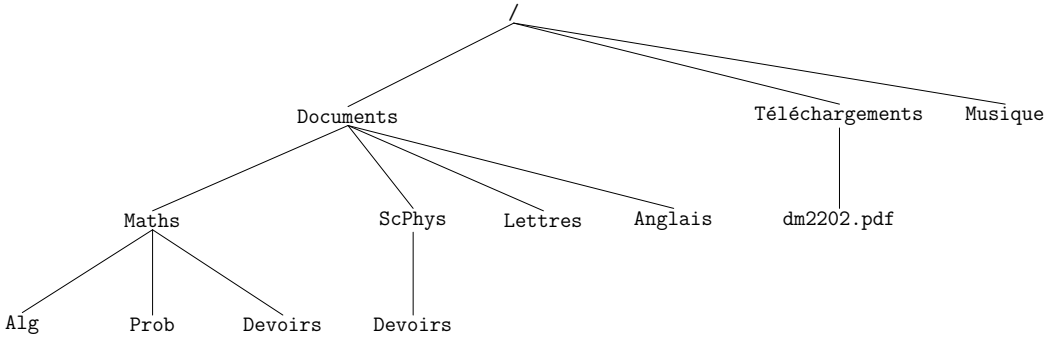
Dans cet exemple, le nombre est représenté par une chaîne de 4 caractères dont 1 caractère pour le séparateur décimal et 1 chiffre après la virgule (même si ce chiffre est nul).

3
Manipulations de fichiers

1. On présente sommairement deux modules : le module `os` (pour *Operating System*) qui va nous permettre de naviguer dans l'arborescence des fichiers et le module `io` (pour *Input/Output*) afin de lire et d'écrire des fichiers.
On se bornera à décrire le fonctionnement élémentaire des fonctions les plus utiles. L'aide en ligne est à votre disposition pour tout complément d'information.

I
Chemins d'un fichier

2. Fichiers et répertoires sont rangés dans un système de fichiers qui a la forme d'un arbre.



2.1 La **racine** de cet arbre est notée `"/"`. Sauf si on travaille sur une machine équipée de Windows...
Au lycée, *normalement*, votre répertoire personnel est identifié par la racine `"K:"` et vos documents devraient se trouver dans le répertoire `"K:Docs"`.
2.2 Le **chemin absolu** d'un répertoire ou d'un fichier `"F"` donne les noms des différents répertoires qu'il faut parcourir depuis la racine pour arriver à `"F"`.
Les chemins absolus commencent donc toujours par le caractère `"/"` : c'est même à ça qu'on les reconnaît.
2.3 Le **chemin relatif** de `"F"` est le chemin à parcourir pour aller du répertoire courant (il faut donc savoir où l'on se trouve) jusqu'à `"F"`.

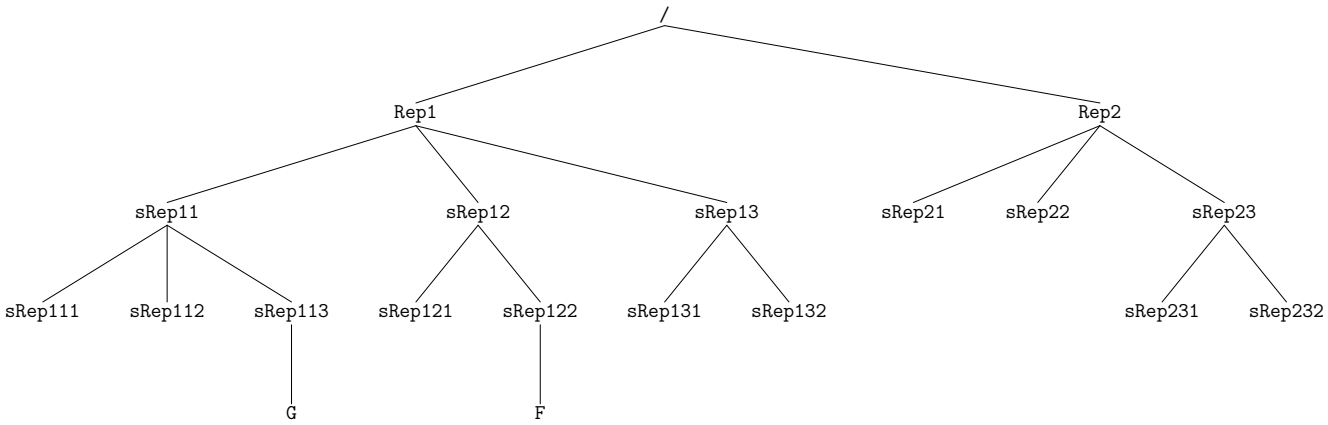
3. Par convention, un répertoire dont on donne le nom est toujours un **répertoire fils** du répertoire courant.
Un répertoire peut avoir plusieurs répertoires fils.
Si ces répertoires fils sont assez nombreux, c'est que le répertoire courant est mal organisé...

4. À l'exception évidente du répertoire racine, chaque répertoire est contenu dans un autre répertoire, dit **répertoire parent**.
Le répertoire courant est noté `"."` et le répertoire parent du répertoire courant est toujours noté `".."`.

5. Il est possible (aussi bizarre que ça paraisse) qu'un répertoire `"MP"` contienne un sous-répertoire `"sujets"`, qui contient lui-même un sous-répertoire `"MP"`.
Si le répertoire courant est `"sujets"`, alors le chemin relatif `"MP/F.txt"` identifie un fichier dans le répertoire fils `"MP"` tandis que le chemin relatif `"../F.txt"` identifie un fichier dans le répertoire parent `"MP"`.
(Je préconise de ne jamais organiser votre système de fichiers de cette manière.)

6. Exemples
Considérons deux fichiers `"F"` et `"G"`, dont les chemins absolus sont les suivants.

```
"/Rep1/sRep12/sRep122/F"
"/Rep1/sRep11/sRep113/G"
```



6.1 Si on se trouve dans le répertoire `"/Rep1/sRep12/sRep122"`, le chemin relatif de `"F"` est simplement `"F"` ou encore `"./F"` (si on ressent le besoin de préciser qu'il s'agit bien d'un fichier situé dans le répertoire).
6.2 Si je me trouve dans le répertoire `"/Rep1/sRep12"`, on peut descendre directement vers `"F"` tandis qu'il faut d'abord remonter dans l'arborescence avant de descendre vers `"G"`.

```
"sRep122/F"
"../sRep11/sRep113/G"   # on remonte vers "Rep1"
```

Le chemin relatif de `"F"` est aussi `"./sRep122/F"`, si on souhaite insister sur le fait que `"sRep122"` est bien un sous-répertoire du répertoire courant.
6.3 Si je me trouve dans le répertoire `"/Rep1/sRep11/sRep112"`, les chemins relatifs vers `"F"` et `"G"` sont respectivement

```
"../../sRep12/sRep122/F" # on remonte jusqu'à "Rep1"
"../sRep113/G"           # on remonte seulement jusqu'à "sRep11"
```

6.4 Si je me trouve dans `"/Rep2/sRep23/sRep232"`, il faut remonter jusqu'à la racine dans les deux cas :

```
"../../../Rep1/sRep12/sRep122/F"
"../../../Rep1/sRep11/sRep113/G"
```

Dans ce cas, autant repérer les deux fichiers par leurs chemins absolus, ce sera plus simple !

II

Module os

7. Fonction getcwd

La fonction `os.getcwd()` (pour *get Current Working Directory*) renvoie une chaîne de caractères contenant le chemin absolu (= depuis la racine de l'arborescence) du répertoire.

8. Fonction chdir

Si le répertoire actuel n'est pas le bon, il faut en changer. L'instruction `os.chdir(path)` (pour *Change Directory*) nous envoie dans le répertoire dont l'adresse est donnée par la chaîne de caractère `path` passée en argument.

8.1 Si le premier caractère de la chaîne `path` est une oblique (`/`), le nouveau répertoire est identifié par son chemin absolu.

8.2 Sinon, le nouveau répertoire est identifié par son chemin relatif au répertoire courant (= celui qui est indiqué par `getcwd`).

8.3 Dans les deux cas, il est essentiel que l'adresse fournie désigne un répertoire existant, dans lequel on dispose des droits en lecture et/ou écriture — sans quoi, une erreur se produit et met fin au programme.

8.4 Aller au répertoire parent

Pour remonter dans l'arborescence (si on n'est pas déjà à la racine...), on peut facilement accéder au répertoire parent du répertoire courant.

```
os.chdir("..")
```

9. Fonction mkdir

L'instruction `os.mkdir(path)` (pour *Make Directory*) sert à créer un sous-répertoire d'un répertoire existant.

9.1 L'argument `path` est le chemin du répertoire créé, qu'il s'agisse du chemin absolu ou du chemin relatif au répertoire courant.

Le nom du nouveau répertoire est donné par la sous-chaîne de l'argument `path` comprise entre la dernière oblique et la fin de la chaîne.

9.2 On ne peut pas créer un répertoire qui existe déjà !

```
os.mkdir("Rep") # Création d'un nouveau répertoire
os.mkdir("Rep") # Erreur : le répertoire existe déjà !
```

9.3 On ne peut créer qu'un seul sous-répertoire à la fois !

Le code suivant indique comment créer un répertoire pour chacun des TP de l'année, chacun de ces répertoires étant un sous-répertoire du nouveau répertoire `Rep2021`.

```
os.mkdir("Rep2021")
for i in range(11):
    os.mkdir("Rep2021/TP{:0>2}".format(i+1))
```

10. Fonction listdir

La fonction `os.listdir(path=None)` renvoie la liste des fichiers et répertoires contenus dans le répertoire dont le chemin est passé en argument. Fichiers et répertoires sont décrits par une chaîne de caractères (leur chemin relatif au répertoire identifié par l'argument `path`).

En l'absence d'argument, la fonction s'applique au répertoire courant. Les instructions `os.listdir()` et `os.listdir(".")` sont donc équivalentes.

11. Fonction rename

Comme on s'en doute, l'instruction `os.rename(source, dest)` renomme `"dest"` le fichier appelé `"source"` (dans le répertoire courant).

12. Cas pratique

Pour les besoins du trombinoscope, je copie dans un répertoire `"Trombi22"` les photos prises à l'aide de mon smartphone. Ce répertoire contient d'autres fichiers que ces photos (en particulier des scripts Python), mais il ne contient pas d'autres photos que ces photos d'identité.
Sur mon smartphone, le nom de chaque fichier est de la forme `"IMG_aaaammjj_hhmmss.jpg"`, de telle sorte que l'ordre alphabétique des noms de fichier soit exactement l'ordre chronologique des prises de vue et donc aussi l'ordre alphabétique des élèves.
Nous allons simplifier la dénomination des fichiers en conservant l'ordre des fichiers.

```
L = os.listdir()                                # liste de tous les fichiers
L_jpg = [ f for f in L if f[-4:]==".jpg" ] # liste des fichiers d'extension ".jpg"
L_jpg.sort()                                    # la même liste, triée
for (i, f) in enumerate(L_jpg):
    id = "{:0>2}.jpg".format(i)
    os.rename(f, id)
```

Les photos sont ainsi renommées `"00.jpg"`, `"01.jpg"`, ..., `"33.jpg"` (pour une classe de 34 élèves) *sans intervention manuelle*.
Un code analogue permet de faire apparaître l'identité de chaque élève dans le nom du fichier, ce qui permettra d'inclure ces photos dans Pronote (et toujours sans intervention manuelle).

III

Module io

13. Fonction open

L'instruction `io.open(f, mode, encoding)` ouvre un le fichier `f` et retourne un flot (*stream*).

13.1 L'argument `f` est le nom du fichier à ouvrir. Si ce fichier ne se trouve pas dans le répertoire courant, l'argument `f` peut être au choix le chemin absolu qui mène à ce fichier ou le chemin relatif au répertoire courant.

13.2 Mode

L'argument `mode` précise dans quel but le fichier doit être ouvert (lecture ou écriture).
Par défaut, la valeur de cet argument est `'r'` (pour *read*) : le fichier est ouvert pour être lu.
Il y a trois manières usuelles d'ouvrir un fichier en écriture :

- si `mode='w'` (pour *write*), le fichier est créé s'il n'existe pas ; mais s'il existe, son contenu est effacé ;
- si `mode='a'` (pour *append*), le fichier est créé s'il n'existe pas ; s'il existe, son contenu est conservé et toutes les opérations d'écriture sont effectuées à la fin du fichier existant.
- si `mode='x'`, le fichier est créé ; si un fichier portant le nom `f` existe déjà, un erreur apparaît.

13.3 Encodage

L'*encodage* est la manière dont les caractères qui composent le fichier sont représentés en binaire. Tous les encodages sont compatibles pour les lettres non accentuées et les chiffres — et c'est à peu près tout...
L'encodage standard est aujourd'hui *unicode*. Il est donc recommandé de préciser systématiquement `encoding="utf-8"` (en lecture ou en écriture) pour ne pas se retrouver coincé à avec un encodage spécifique au système d'exploitation (on risquerait de ne pas pouvoir lire le fichier sur un autre ordinateur).

14. Fonction close

Tout flot ouvert, en écriture ou en lecture, doit être fermé avec la fonction `close`.
Il est particulièrement important de fermer un flot ouvert en écriture, car c'est cette opération qui termine véritablement les opérations d'écriture : un fichier non fermé risquerait d'être incomplet !

15. Lectures ligne à ligne

Lorsque toutes les lignes d'un fichier jouent le même rôle, le moyen le plus simple de lire le fichier ligne à ligne est une boucle `for`.

```
foo = io.open("toto.txt", mode='r', encoding='utf-8')
for L in foo: # les lignes du fichier sont affectées successivement à la chaîne L
    ....      # traitement des données
foo.close()
```

Si le fichier contient une ligne d'en-tête, il vaut mieux passer par une boucle `while` et demander explicitement de lire chaque ligne avec la fonction `readline`. Lorsque la fin du fichier est atteinte, cette fonction renvoie la chaîne vide `""`.

```
foo = io.open("Recap.csv", mode='r', encoding='utf-8')
en_tete = foo.readline() # première ligne
...                      # traitement de la première ligne
ligne = foo.readline()   # deuxième ligne
while (ligne!=""):       # tant que la fin du fichier n'est pas atteinte
    ....                 # traitement des données
    ligne = foo.readline()  # on passe à la ligne suivante
foo.close()
```

16. Écriture d'un fichier

Pour écrire des données dans un fichier, celui-ci doit être ouvert en écriture... On écrit la chaîne de caractères `s` sur le fichier à l'aide de l'instruction `foo.write(s)` où `foo` est le flot associé au fichier concerné.

```
foo = io.open("test.txt", mode='w', encoding='utf-8')
for i in range(100):
    foo.write("l.{:>3} bla bla bla\n".format(i))
foo.close()
```

Si on omet le caractère de saut de ligne `\n`, les données sont écrites sur une seule ligne, à la suite les unes des autres. Si c'est voulu, ce n'est pas un problème !

```
foo = io.open("test2.txt", mode='w', encoding='utf-8')
for i in range(100):
    foo.write("l.{:>3} bla bla bla - ".format(i))
foo.close()
```

17. Quelques exemples

17.1 Affichage à l'écran du contenu d'un fichier

```
foo = io.open("Effectif_MP.csv", mode='r', encoding="utf-8") # ouvert en lecture
for ligne in foo:
    print(ligne)
foo.close()    # fermeture du flot
```

17.2 On considère un tableau rassemblant les notes des différents élèves aux compositions. La première ligne du tableau contient l'identifiant de chaque colonne. Les autres lignes contiennent le nom de chaque élève ainsi que ses différentes notes. On suppose connue une fonction `moyenne(data)` dont l'argument est une ligne du fichier et qui renvoie le nom de l'élève et sa moyenne.

```
foo = io.open("Compos.csv", mode='r', encoding="utf-8")
en_tete = foo.readline()   # lecture de la première ligne (en-tête)
data = foo.readline()      # lecture de la seconde ligne (première ligne de données)
while (data!=""):
    eleve, moyenne = moyenne(data)
    print("Moyenne de {:>#25} :\t{:3.1f}".format(eleve, moyenne))
    data = foo.readline()
foo.close()
```

18. Exemple complet

Dans le sous-répertoire `"Donnees"` du répertoire courant, le fichier `"Exp1.csv"` contient les résultats d'une expérience.

18.1 Ce fichier commence avec une ligne d'en-tête et les flottants sont écrits à la française (avec une virgule).

Pression (en Pa)	Durée théorique (en s)	Durée mesurée (en s)
1013	0	0
675	1,386	2
410	3,09	4,2
220	5,22	6
130	7,02	10,3
72	9,06	13,1
40	11,04	16,4

18.2 On prépare Python à lire des données francisées.

```
import locale
locale.setlocale(locale.LC_NUMERIC, '')
```

18.3 On lit le fichier : d'abord la ligne d'en-tête, puis les lignes de données numériques.

```
import io
fichier = io.open("./Donnees/Exp1.csv", "r", encoding="utf-8")
en_tete = fichier.readline().split(";")
mesures = []
for ligne in fichier:
    data = ligne.strip().split(";")           # liste de chaînes de caractères
    data_f = [ locale.atof(x) for x in data ] # liste de flottants
    mesures.append(data_f)
fichier.close()
```

18.4 Pour faciliter le traitement des données, on convertit la liste de liste en tableau.

```
import numpy as np
mesures = np.array(mesures)
```

18.5 Il reste à traduire graphiquement ces données.

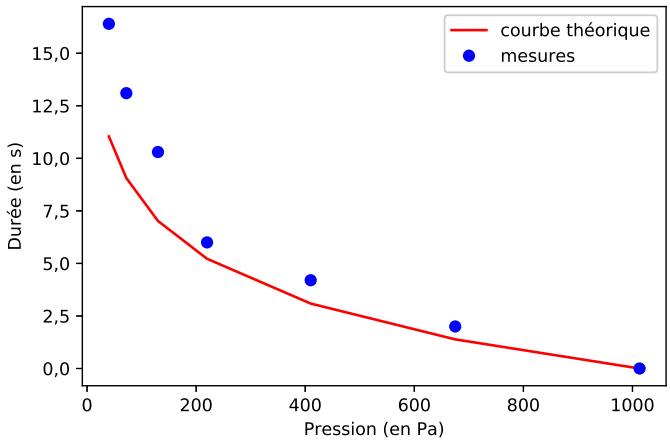
```
import matplotlib.pyplot as plt
X = mesures[:,0] # pressions
Y = mesures[:,1] # durées théoriques
Z = mesures[:,2] # durées mesurées
plt.plot(X, Y, 'r', label="courbe théorique")
plt.plot(X, Z, 'bo', label="mesures")
```

18.6 Un graphe sans légende n'est pas un graphe !

```
plt.legend()
plt.xlabel(en_tete[0])
plt.ylabel("Durée (en s)")
y_tks = np.arange(0, 16, 2.5)                    # positions
y_tks_s = [ "{:.4n}".format(y) for y in y_tks ]  # étiquettes
plt.yticks(y_tks, y_tks_s)
```

On peut souhaiter que les étiquettes soient mieux alignées le long de l'axe des ordonnées. Dans ce cas, il faut les écrire en virgule fixe.

```
y_tks_s = [ locale.format_string("%4.1f", y) for y in y_tks ]
```

18.7 Pour utiliser le graphe ainsi produit dans une présentation, on le sauvegarde dans un répertoire prévu à cet effet.

```
plt.savefig("./Figures/Exp1.pdf")
```