TP-Cours d'Informatique 00

Découverte de Pyzo – Éléments de langage Python

ITC – PCSI^2 2024-2025 – Lycée Fabert (Metz) – Merci : S.Limal, Y.Loquais, D.Mengel, N.Wipf, A.Lux

I. À la découverte de Pyzo, voyage en terre inconnue

I.1. Organisation de votre travail

Source : N.Wipf A.Lux

Si vous n'utilisez pas votre ordinateur personnel en TP d'informatique, vous **devrez venir à chaque séance avec une clé USB**.

Afin de pouvoir retrouver facilement tous vos fichiers, vous devez créer un dossier **ITC** sur votre clé USB ou sur votre ordinateur personnel.

Vous devez ensuite créer un sous-dossier pour chacun des TP traités. A la fin du premier semestre, l'arborescence de dossiers devrait ainsi ressembler à celle de l'image ci-contre.

Bref, chaque répertoire sera numéroté avec indication du thème abordé dans le TP. Les thèmes abordés au premier semestre sont :

– la **recherche séquentielle dans un tableau** – les **boucles imbriquées** – les **algorithmes dichotomiques** – les **fonctions récursives** – les **algorithmes gloutons** – les **matrices de pixels et images** – les **tris**.

1. Créer le dossier **ITC** et le sous-dossier **TP00-Pyzo+LangagePython** : celui-ci contiendra les fichiers **TP00-[a-z].py** qui seront créés dans ce TP.

I.2. Pyzo : console Python, affectation de variables et typage dynamique

Source : N.Wipf A.Lux

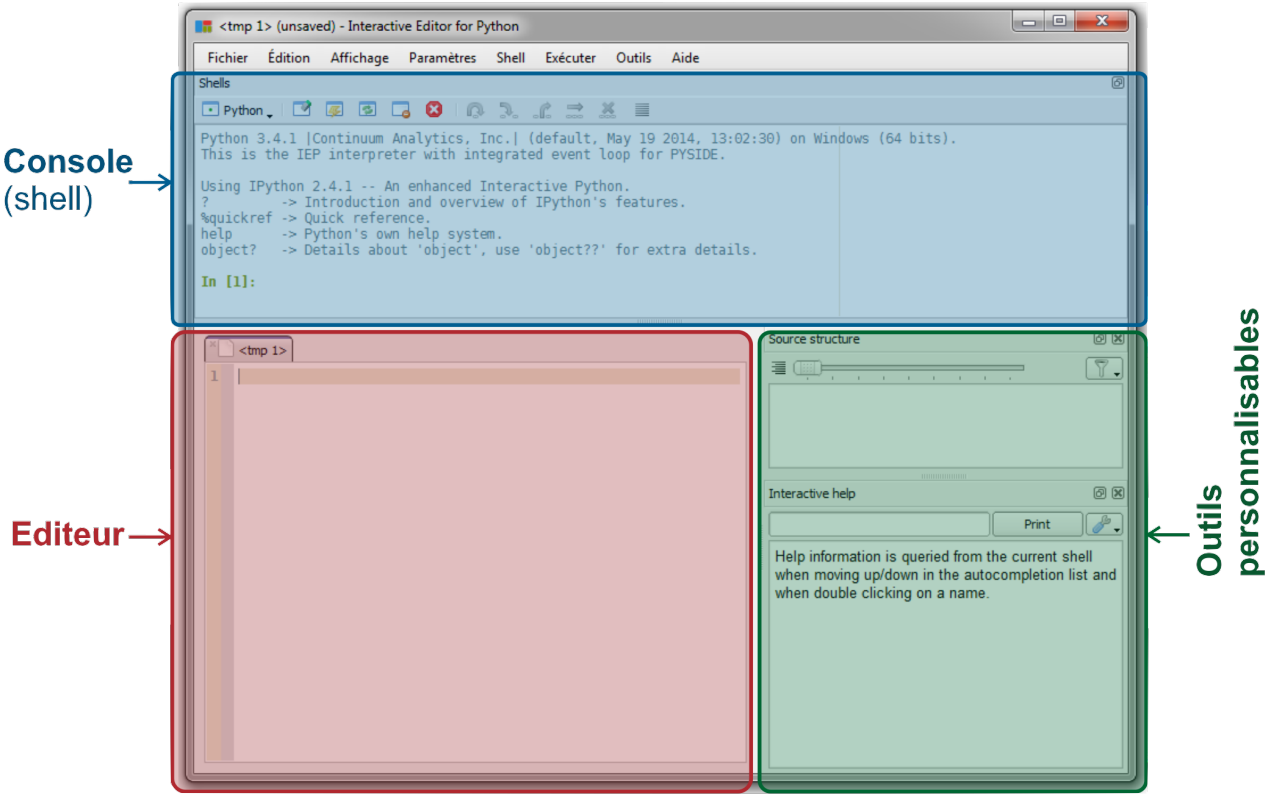
Pour écrire et exécuter des programmes informatiques en langage Python, nous utiliserons cette année **l'environnement de développement intégré (IDE) Pyzo**. Par défaut, la fenêtre de Pyzo est divisée en **trois zones** : **la console**, **l'éditeur** et une dernière **zone d'outils personnalisables**.

— **La console** : elle permet de lancer des commandes qui seront immédiatement interprétées et d'afficher le résultat de l'exécution d'un programme (ou éventuellement le message d'erreur rencontré durant l'interprétation de ce programme).

— **L'éditeur** : seules les instructions écrites dans cette zone peuvent être sauvegardées : **c'est donc cette zone qui nous servira à écrire les programmes**. Dans l'éditeur de code, diverses aides sont fournies comme par exemple la coloration syntaxique automatique, une indentation automatique, la complétion automatique (avec la touche Tab).

— **Les outils personnalisables** : ils peuvent être sélectionnés grâce au menu Outils (Tools) de Pyzo. On utilisera notamment :

- L'outil « Environnement » (Workspace) qui permet de suivre l'état des différentes variables au cours de l'exécution du programme
- L'outil « Aide interactive » (Interactive help) qui permet d'obtenir de l'aide sur la syntaxe des fonctions du langage ou des modules importés.



2. Lancer l'environnement de développement intégré (IDE) Pyzo. Nous travaillons d'abord sur la console python. La console peut être utilisée pour effectuer des calculs :

```
1  5/3; 5//3; 5*3; 5**3; 5%3; 5e3;
```

Donner les résultats obtenus dans le tableau ci-dessous et expliquer chacun des symboles /, //, *, **, % et e en langage Python :

Opération Python	5/3	5//3	5*3	5**3	5%3	5e3
Résultats						

Une **variable** permet de stocker des données pour les réutiliser ensuite. Une **variable** apparaît dans un langage de programmation sous un *nom de variable* à peu près quelconque (il y a cependant ceraines règles : retenez pour l'instant qu'un nom de variable doit commencer par une lettre), mais pour l'ordinateur il s'agit d'une référence désignant une adresse mémoire, c'est-à-dire un emplacement précis dans la mémoire vive de l'ordinateur où se trouve stockée la valeur de cette variable.

L'ensemble des variables définies à un instant donné de l'exécution d'un programme est appelé « **état de l'exécution du programme** ».

3. La console permet également **la déclaration** de variable ou **l'affectation**. On notera d'ailleurs qu'en Python, la déclaration se confond avec la première affectation d'une variable.

```
x = -2; x**2+1; (x,type(x))
y = 2.9; (y,type(y))
z = [x,y,2+1j]; (z,type(z))

a = (x<2); (a,type(a))
b = (x>1); (b,type(b))

t1= "C'est un serpent "
t2 = "Python"
```

```
t1 + t2; (t1,type(t1))

c = (5,2.7); (c,type(c))
d = (5,2,7); (d,type(d))

roues = {"voiture": 4, "vélo":
    2, "tricycle": 3}
roues["vélo"]
(roues,type(roues))
```

Précisez les types de variables obtenus lors de l'exécution du code précédent en complétant le tableau ci-dessous : ce sont les types de variables principaux étudiés cette année.

Variable	x	y	z	a	b	t1	c	d	roues
type									

Contrairement à d'autres langages informatiques (C++, Pascal ...), le langage Python n'a pas besoin que l'on déclare à l'avance les variables ainsi que leur type. L'interpréteur attribue le type d'une variable « à la volée », lors de l'affectation. Cela signifie aussi qu'une variable peut voir son type évoluer au cours de l'exécution d'un programme. On parle de **typage dynamique** de Python.

I.3. Importation de modules dans le langage Python

On appelle **module** tout fichier constitué de code Python (c.-à-d. tout fichier avec l'extension .py) importé dans le langage Python pour offrir de nouvelles fonctionnalités.

Les modules permettent la séparation et donc une meilleure organisation du code. En effet, il et courant dans un projet de découper son code en différents fichiers qui vont contenir des parties cohérentes du programme final pour faciliter la compréhension générale du code, la maintenance et le travail d'équipe si on travaille à plusieurs sur un projet.

4. Rechercher dans l'aide des informations sur les fonctions valeur absolue et racine carrée (utiliser help ou l'aide interactive). Effectuer quelques essais avec ces fonctions :

```
abs(1); abs(-2); x=-4; abs(x**2)
sqrt(2); sqrt(3); sqrt(10**2); sqrt(0.1**2)
```

Quel est le problème ? Et la solution ?

On est souvent amené à importer tout ou partie de modules dans le langage Python, mais **attention** ! Deux modules différents peuvent contenir des fonctions de même nom mais possédant des spécifications (le type et l'ordre de ses paramètres, le type de son résultat ...) différentes.

```
1j**2; (1+1j)**2; sqrt(3+4j)              # Quel est le problème ?
import pylab as pl;  pl.sqrt(3+4j)
```

Expliquer la technique permettant d'importer le module pylab et de distinguer deux fonctions sqrt provenant de deux modules différents :

I.4. Éditeur et gestion des entrées/sorties

Nous allons utiliser maintenant l'éditeur intégré de Pyzo pour écrire du code Python que nous interpréterons/exécuterons ensuite.

5. Copier les lignes de code suivantes dans l'éditeur, enregistrer le fichier **TP00.py** dans votre répertoire de travail puis l'exécuter.

```
1 x=-4                  # Étape 1
2 y = x**2              # Étape 2
3 y                     # Étape 3
```

```
1 a = sqrt(5)           # Étape 1
2 b = sqrt(a**2+3)      # Étape 2
3 b                     # Étape 3
```

Pour ce faire, deux possibilités :

- cliquer sur **Exécuter → Exécuter le contenu de l'onglet courant** (Ctrl+E sous Windows et Cmd+E sous MacOs)
- cliquer sur **Exécuter → démarrer le script** (Ctrl+Shift+E sous Windows et Cmd+Shift+E sous MacOs).

La première option (Ctrl+E) interprète le code de l'éditeur dans la console Python courante (qui contient peut-être des variables définies par l'utilisateur) alors que **la seconde option** (Ctrl+Shift+E) interprète le code dans une nouvelle console dédiée et vierge de toute donnée de l'utilisateur.

Expliquer les différentes lignes du code précédent :

#Étape 1	#Étape 2	#Étape 3

Lorsqu'une expression est évaluée dans l'éditeur, le résultat de cette évaluation n'est pas affiché automatiquement. L'interpréteur n'affiche ce résultat que si cela lui est demandé explicitement : à cet effet, on utilise la fonction **print**().

```
1 print(y)              # Étape 3
```

```
1 print(b)              # Étape 3
```

6. On appelle entrée/sortie toute instruction qui permet d'interrompre le flux d'exécution du programme pour communiquer avec l'utilisateur, donnant ainsi un aspect interactif au programme.

- La fonction `print` permet d'afficher n'importe quel nombre de valeurs fournies en arguments (c'est-à-dire entre les parenthèses). Par défaut, ces valeurs seront séparées les unes des autres par un espace, et le tout se terminera par un saut à la ligne. Vous pouvez remplacer le séparateur par défaut (l'espace) par un autre caractère quelconque (ou même par aucun caractère), grâce à l'argument `sep`.

```
1 print("Bonjour", "à", "tous", sep ="*")
2 print("Bonjour", "à", "tous", sep ="")
3 print("Bonjour", end=""); print("1", "2", "3", sep="§");
```

Listing 1 – Exemples d'utilisation de la fonction `print`

- La fonction `input` provoque une interruption dans le programme courant. L'utilisateur est invité à entrer des caractères au clavier et à terminer avec <Enter>. Lorsque cette touche est enfoncée, l'exécution du programme se poursuit, et la fonction fournit en retour une chaîne de caractères correspondant à ce que l'utilisateur a saisi. Cette chaîne peut alors être assignée à une variable quelconque.

```
1 prenom = input("Votre prénom : "); print("Bonjour,", prenom)
2 print("Entrer un nombre positif quelconque : ", end=" ");
3 ch = input(); nn = int(ch)  # conversion de la chaîne en entier
4 type(ch); type(nn); print("Le carré de", nn, "vaut", nn**2)
```

Listing 2 – Exemples d'utilisation de la fonction `input`

II. Premiers pas en Python

II.1. Séquences d'instructions

Les instructions d'un programme s'exécutent les unes après les autres, dans l'ordre où elles ont été écrites à l'intérieur du script : c'est le flux d'exécution « naturel » d'une séquence d'instructions. Une séquence de plusieurs instructions est généralement appelé *bloc d'instructions*.

Il faut être **extrêmement attentif à l'ordre dans lequel vous placez vos instructions les unes derrière les autres**, car il s'agit d'une source d'erreurs très fréquente.

Si on note e l'état courant et $i(e)$ l'état obtenu en exécutant l'instruction i, alors, après l'exécution des instructions i_1 et i_2, l'état obtenu est $i_2(i_1(e))$.

Enchaîner deux instructions en séquence revient à composer (au sens mathématique) leurs deux actions sur l'état : en général, il n'y a pas commutativité des instructions dans une séquence.

Exemple :

```
1 x = 1
2 x = x + 1
3 x = x * 2
4 y = x + 5
5 x = 2
6 print(y)
```

Listing 3 – Exemple de séquence d'instruct.

```
1 x = 1
2 x = x * 2
3 x = x + 1
4 y = x + 5
5 x = 2
6 print(y)
```

Listing 4 – Exemple de séquence d'instruct.

Dans une séquence, Python exécute les instructions de la première à la dernière, sauf lorsqu'il rencontre une instruction de contrôle de flux, comme une instruction conditionnelle (**if**), une boucle conditionnelle (**while**} ou inconditionnelle (**for**) ou encore les fonctions (**def**).

7. Donner la valeur de y affichée lors de l'exécution de la ligne 5 pour chacun des listings précédents.

Observons à présent la déclaration/l'affection de la ligne 4. Cette instruction s'effectue en deux temps :

- L'expression située à droite du signe = est évaluée en fonction de l'état courant. Ici, l'interpréteur évalue la somme de la valeur de x et de 5 : dans un cas (listing 3) le résultat est 9 et dans l'autre (listing 4) le résultat est 8.
- Dans un deuxième temps, l'interpréteur affecte à la variable dont le nom est y la valeur précédemment calculée.

Conséquence de ce comportement : l'identifiant de y (obtenu par l'instruction `id(y)`) n'a aucun rapport avec celui de x et lorsqu'on change la valeur de x, la variable y n'est pas affectée comme on peut le constater après exécution de la ligne 5 et affichage de la valeur de y sur la ligne 6.

8. On suppose que les variables a et b sont déclarées et contiennent des valeurs entières. La séquence d'instructions suivante se déroule entièrement :

```
1 a = 2*a - b
2 b = a + b
3 a = b - a
4 b = b // 2
```

Listing 5 – Exercice sur les séquences d'affectations

Décrire l'évolution de l'état des variables au cours de l'exécution de cette séquence sur divers exemples puis déterminer les valeurs finales des variables a et b en fonction de leurs valeurs initiales.

II.2. Structures de contrôle

II.2.a Structures de contrôle : la sélection

Une instruction conditionnelle permet d'aiguiller le déroulement du programme dans différentes directions, en fonction des circonstances rencontrées. La commande `if` permet d'exécuter une séquence d'instructions si et seulement si une *condition* est vérifiée par l'état courant. En voici la syntaxe :

`if` *condition* :
    première instruction
    ⋮
    dernière instruction

Exemple 1

```
1 a = int(input("Entrez un nombre entier relatif noté a : "))
2 if (a>100):
3     print("a dépasse la centaine")
4     print("donc 2*a =",2*a,"dépasse 200")
```

Listing 6 – Exemple d'utilisation de la commande `if`

La commande `if` permet de tester la validité de la condition $a > 100$ placée entre parenthèses. Si la condition est vraie, alors le bloc d'instructions que nous avons indenté après le : est exécuté. Si la condition est fausse, rien ne se passe. Notez que les parenthèses utilisées ici avec la commande `if` sont optionnelles : nous les avons utilisées pour améliorer la lisibilité.

La condition évaluée après l'instruction `if` peut contenir la valeur True ou False : on dit qu'il s'agit d'une **variable booléenne**. En voici quelques exemples contenant les opérateurs de comparaison

$==$	$!=$	$<$	$>$	$>=$	$<=$
(égal à)	(différent de)	(strict. inférieur à)	(strict. supérieur à)	(supérieur ou égal à)	(inférieur ou égal à)

$$(a == b) \qquad (a != n) \qquad (a < b) \qquad (a > b) \qquad (a >= b) \qquad (a <= b)$$

En langage Python, la syntaxe de test avec alternative est la suivante :

`if` *condition* :
    bloc d'instructions à exécuter si *condition* a la valeur True
else :
    bloc d'instructions à exécuter si *condition* a la valeur False

Exemple 2

```
1 import random
2 x = random.randint(1,100);
3 y = random.randint(1,100);
4 if x<y:
5     print("x =",x,"est strictement plus petit que y =",y)
6 else:
7     print("x =",x,"est plus grand que y =",y)
```

Listing 7 – Comparaison de deux entiers tirés au hasard

Enfin, dans le cas où l'on souhaite exprimer **plus de deux alternatives** dans une instruction conditionnelle, on peut utiliser la commande `elif` (contraction de `else` et de `if`) qui effectue également une sélection. La syntaxe de cette instruction conditionnelle étendue est la suivante :

`if` *condition1* :
    Bloc d'instructions 1
`elif` *condition2* :
    Bloc d'instructions 2
    ⋮
`elif` *conditionN* :
    Bloc d'instructions N
`else` :
    Autre bloc d'instructions

Chacune des conditions *condition1*,..., *conditionN* est une expression booléenne qui doit pouvoir être évaluée au début de l'exécution de la commande `if`.

Dès que l'une des conditions est réalisée, le bloc d'instructions correspondant est exécuté et le programme passe à l'instruction qui suit toute la structure conditionnelle. Dans le cas où plusieurs conditions sont vérifiées, seule le bloc d'instructions correspondant à la première condition réalisée est exécuté.

Enfin, quand aucune des conditions *condition1*,..., *conditionN* n'est vérifiée, alors, s'il est présent, c'est Autre bloc d'instructions qui est exécuté.

Exemple 3

```
x = int(input("Entrez un entier svp : "))
if x<0:
    print("Cette soupe est trop froide")
elif x>0:
    print("Cette soupe est trop chaude")
else:
    print("Elle est juste comme il faut !")
```

Exercice 1 (Coaching) : Créez un programme de coaching qui :

- demande à l'utilisateur sa taille en mètres et sa masse en kilogrammes,
- calcule son IMC (indice de masse corporelle) en $kg.m^{-2}$,
- calcule les masses masseMin et masseMax qui correspondent respectivement aux masses pour lesquelles l'IMC de l'utilisateur vaudrait 18,5$kg.m^{-2}$ et 25$kg.m^{-2}$

Ce programme devra ensuite afficher :

- la valeur de l'IMC de l'utilisateur,
- l'intervalle de masse « idéal » qui lui permettrait d'avoir un IMC compris entre 18,5$kg.m^{-2}$ et 25$kg.m^{-2}$,
- un conseil à l'utilisateur : perdre du poids si son IMC est supérieur à 25$kg.m^{-2}$, gagner du poids si son IMC est inférieur à 18,5$kg.m^{-2}$ ou conserver son poids lorsque son IMC est dans l'intervalle « idéal ».

II.2.b. Structures de contrôle : les fonctions Python

Lorsqu'une **séquence d'instructions** doit être réalisée plusieurs fois par un programme avec des paramètres parfois différents, on peut l'isoler au sein d'une **fonction**. Une **fonction** est donc un sous-programme qui dépend de zéro, un ou plusieurs paramètres et retourne un résultat (qui peut être un tuple) ou None. Le **corps de la fonction** désigne toutes les instructions la composant et qui peuvent être exécutées si la fonction est appelée.

Une déclaration d'une fonction Python suit la syntaxe classique donnée ci-après :

```
def maFonction(param_1, ..., param_n):
    instruction_1
    ...
    ...
    instruction_p
    return result_1, ..., result_q
```

Toutefois, nous verrons plus tard que l'instruction **return** ne se trouve pas toujours sur la dernière ligne de la déclaration de fonction. On notera que

- la première ligne de la déclaration d'une fonction commence toujours par **def**, se termine par deux-points : . Ici, param_1 à param_n sont les noms des paramètres et result_1 à result_q sont les noms des variables ou les valeurs constituant le résultat retourné par la fonction.
- Le corps de la fonction est **un bloc de code indenté** par rapport à la ligne de définition. **L'indentation est un élément de syntaxe important en langage Python** puisqu'il fait office de délimiteur de blocs d'instructions. Nous y reviendrons.

Voici un exemple de définition d'une fonction numérique simple en langage Python :

```
1 def f(x):
2     P = x**2-5*x+6                # Le retour à la ligne après la ligne 3
3     return P                      # indique la fin de la définition de f
4
5 f(3); P
```

9. Quel est l'affichage produit par ce code Python ? Comment y remédier ?

Lors de l'exécution du code précédent, l'interpréteur Python n'a pas accès à la variable P créée dans un espace-mémoire réservé le temps de l'appel de f par l'instruction f(3) : il s'agit d'une variable « locale », visible depuis le corps de la fonction mais qui est détruite dès la sortie de cette fonction .

Voici un exemple de fonction à plusieurs paramètres et un tuple-résultat :

```
1 def symetrique(a,b,c):
2     sigma1 = a + b + c
3     sigma2 = a*b + a*c + b*c
4     sigma3 = a*b*c
5     return sigma1, sigma2, sigma3
```

10. Définir dans l'éditeur la fonction symetrique et créer la variable F = symetrique(1,2,3), puis afficher sa valeur dans la console.

Si l'on souhaite rendre indépendants les différents éléments d'un tuple, on peut le « dépaqueter » en écrivant une instruction du type : s1,s2,s3=F

11. Afficher successivement les valeurs des variables s1, s2 et s3 ainsi définies.

Exercice 2 (Conversions) : On rappelle que si a et b sont des entiers, alors

★ a//b renvoie le quotient de la division euclidienne de a par b et

★ a%b donne le reste de la div. euclid. de a par b (pratique pour tester la divisibilité de a par b).

1. Écrire la fonction conversionS(h:**int**, m:**int**, s:**int**) → **int**
   - qui prend en argument un triplet de trois entiers h, m et s (correspondant à une durée exprimée en heures / minutes / secondes)
   - et qui renvoie cette durée exprimée en secondes.

2. Écrire la fonction conversionHMS(d:**int**) → (**int**,**int**,**int**)
   - qui prend en argument un entier d correspondant à une durée en secondes
   - et qui renvoie le triplet correspondant heures, minutes, secondes (on pourra commencer par remarquer que d//3600 donne un résultat utile).

   On vérifiera que conversionHMS(4810) renvoie bien (1, 20, 10). Tester également votre programme avec d'autres valeurs.

3. Un tremblement de terre a eu lieu à Sacramento la semaine dernière !
   - la première secousse a été ressentie à 13 h 46 min 12 s,
   - la seconde secousse est intervenue 21 min 54 s après la première
   - puis la troisième secousse a eu lieu 14 min 25 s après la seconde.

   En utilisant les deux fonctions précédentes, identifier l'horaire de la troisième secousse sismique en l'affichant sous la forme « h : min : sec ».

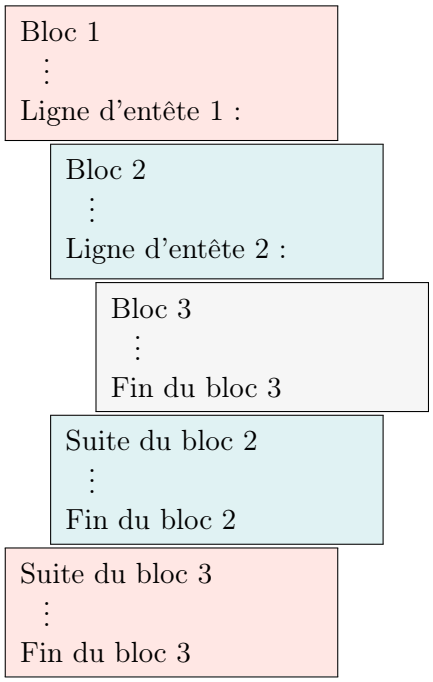
Le constructions utilisées avec les commandes `if` ou `def` sont deux exemples d'instruction composée. Sous Python, les instructions composées ont toujours la même structure : une ligne d'en-tête terminée par un double point, suivie d'une ou de plusieurs instructions indentées sous cette ligne d'en-tête :

```
Ligne d'entête :
    première instruction
      ⋮
    dernière instruction
```

Dans la plupart des autres langages, un bloc d'instructions doit être délimité par des symboles spécifiques (parfois même par des commandes, telles que begin et end). En C++ et en Java, par exemple, un bloc d'instructions doit être délimité par des accolades. Cela permet d'écrire les blocs d'instructions les uns à la suite des autres, sans se préoccuper ni d'indentation ni de sauts à la ligne, mais cela peut conduire à l'écriture de programmes confus, difficiles à relire.

On conseille donc à tous les programmeurs qui utilisent ces langages **de se servir aussi des sauts à la ligne et de l'indentation pour bien délimiter visuellement les blocs**.

Bien entendu, les insctructions composées peuvent être imbriquées, chaque bloc étant indenté par rapport à celui qui le contient suivant le schéma suivant :



Avec Python, **il faut utiliser les sauts à la ligne et l'indentation**, mais en contrepartie vous n'avez pas à vous préoccuper d'autres symboles délimiteurs de blocs. En définitive, Python force donc à écrire du code lisible, et à prendre de bonnes habitudes que l'on conserve lorsqu'on utilise d'autres langages.

II.2.c. Structures de contrôle : itération conditionnelle ou boucle *while*

En programmation, on appelle boucle ou (itération) un système d'instructions qui permet de répéter un certain nombre de fois (voire indéfiniment) toute une série d'opérations. Python dispose de deux types de boucles : la boucle *while* que l'on décrit dans cette partie et la boucle *for* décrite dans la dernière partie de ce TP-cours.

La boucle *while* permet de réaliser une itération contrôlée par une condition d'arrêt. Bien sur, l'exécution de la boucle doit conduire vers la réalisation de cette condition en un temps fini ! On veillera tout particulièrement à ce problème.

Exemple 4

```
1 a = 10240
2 n = 0
3 while (a%2==0):
4     a = a//2
5     n = n +1
6 print(10240," = ",a,"*",2,"**",n,sep="")
```

Listing 8 – Exemple d'itération conditionnelle : la boucle *while*

Le mot while signifie *tant que* en anglais. Cette instruction utilisée à la ligne 3 indique à Python qu'il lui faut répéter continuellement le bloc d'instructions qui suit, tant que le reste de la division euclidienne de a par 2 est nul.

Comme la commande `if` abordée dans la partie précédente, la commande `while` amorce une instruction composée. Le double point à la fin de la ligne introduit le bloc d'instructions à répéter, lequel doit obligatoirement se trouver indenté (c.-à-d. en retrait).

12. À quel moment la boucle *while* s'interrompt-elle ? Qu'est-ce que cela signifie pour a ?

13. Quel est le rôle de n dans cette boucle ? Que représente-t-il en sortie de boucle ?

Exercice 3 : La suite harmonique définie par

$$\forall n \in \mathbb{N}^*,\ H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$$

est croissante et diverge vers $+\infty$. Déterminer le plus petit entier $n \in \mathbb{N}^*$ tel que $H_n > 12$.

```python
import time

quitter = "n"
while quitter != "o":
    print("Heure courante :",time.strftime("%H:%M:%S"))
    quitter = input("Voulez-vous quitter le programme  ? (o/n)")

print("A bientôt")
```

Listing 9 – Une fortune en horloge parlante ...

Méthode (Écrire une itération conditionnelle) :

1. On identifie la condition de la boucle ; il est souvent plus commode de chercher une condition de sortie de la boucle, puis de calculer sa négation ou tout simplement d'utiliser l'opérateur *not*.
2. On écrit le corps de la boucle, en s'assurant que celui-ci modifiera la valeur de la condition au cours des itérations en se rapprochant de la condition d'arrêt.
3. On prévoit une initialisation des variables en amont de la boucle.
4. Il est parfois nécessaire de faire un dernier traitement à la suite de la boucle ; dans tous les cas, on n'oubliera pas de revenir au niveau d'indentation du *while*.

La question de la **preuve de l'arrêt** d'une itération conditionnelle sera posée dans un prochain cours. Pour l'instant, il est important de prendre conscience du problème et de veiller à construire des boucles qui ont une fin !

Dans de nombreux cas, la mise en place d'un compteur permet de contrôler l'arrêt de la boucle *while* et de déterminer le nombre d'itérations de la boucle, comme dans l'exemple ci-dessous.

Exemple 5

```python
somme = 0
n = 0

while 11*n<=1000:
    somme = somme + 11*n
    n = n + 1

print(somme); print(n-1)
```

Listing 10 – Exemple d'utilisation d'un compteur dans une boucle *while*

14. Que fait ce programme ? Quelle représente la valeur de la viariable somme en sortie de boucle ?

15. Quel est le rôle de n dans cette boucle ? Que représente-t-il en sortie de boucle ?

II.2.d. Structures de contrôle : itération inconditionnelle ou boucle *for*

Syntaxe de base et premiers exemples :

L'expression `range(n)` est appelée *itérateur*, que l'on peut voir comme un distributeur d'entiers consécutifs :

- ◇ son premier élément est 0
- ◇ son plus grand élément est $n-1$
- ◇ l'élément venant après l'élément i est l'élément $i+1$.

Cette expression admet d'ailleurs deux formes plus générales, `range(m,n)` dont le premier élément est m (et le dernier $n-1$), et `range(m,n,p)` dont l'élément venant après i est $i+p$ c'est-à-dire l'itérateur précédent parcouru avec un *pas* de p entiers.

```
range(10); list(range(10))  # Conversion d'un intérateur range en liste
list(range(3,11)); list(range(3,11,2))
```

Listing 11 – L'itérable `range`

Il est également possible de préciser un pas p négatif dans l'instruction `range(m,n,p)` avec $m > n$ (sinon l'itérateur est vide). Dans ce cas, la dernière valeur de l'itérateur est $n+1$.

La boucle *for* permet de programmer une action répétitive contrôlée par un compteur. Ce compteur peut être une valeur numérique (dans ce paragraphe) ou l'opérande générique d'une valeur itérable (dans le paragraphe suivant). La syntaxe d'une boucle *for* utilisant l'itérateur `range` est la suivante :

`for` i `in range(...)` :
    bloc d'instructions à exécuter
    et qui dépendent éventuellement de i

Le compteur de boucle noté i est entièrement géré par la boucle `for`. Il n'est pas besoin de l'incrémenter, ni de tester qu'il ne dépasse pas une certaine limite.

Exemple 6

```
n = int(input("Entrez un entier strictement positif : "))
somme = 0
for i in range(n+1):
    somme = somme + i**2
print(somme)
```

Listing 12 – Somme des carrés d'entiers entre 0 et n

Exemple 7

```
from sympy import expand, symbols, cos

x = symbols('x');
for i in range(1,7):
    print(expand(cos(i*x),trig=True).subs(cos(x),x))
```

Listing 13 – Polynômes de Tchebychev

Exercice 4 : Écrire la fonction factorielle (n:**int**)→**int** qui renvoie la valeur de $n!$ en fonction du paramètre $n \in \mathbb{N}$. On utilisera une boucle `for`.

Exercice 5 : Écrire un programme utilisant une boucle **for** qui calcule et affiche la somme de tous les multiples de 11 inférieurs ou égaux à 1000.

Valeurs itérables

On appelle valeur itérable toute variable d'un type composé (d'éléments plus simples) pour lequel :

1. on sait identifier le premier élément que l'on va traiter
2. on sait déterminer l'élément qui vient juste après celui que l'on vient de traiter

Les n-uplets, les chaînes de caractères ou les listes sont des valeurs itérables et peuvent donc servir à définir une boucle `for` :

Exemple 8

```
L = [1, 3, 14, 5, 7, 22, 5, 6]
somme = 0
for i in L:
    somme = somme + i
print(somme)
```

Listing 14 – Somme des éléments d'une liste

Exemple 9

```
s = "Le corbeau et le renard"
S = ""
for c in s:
    print(c)
    if c not in "aeiouy":
        S = S + c.upper()
    else:
        S = S + c
print(S)
```

Listing 15 – Parcours des caractères d'une chaîne – Mise en majuscule des consonnes

Méthode (Choix d'une boucle conditionnelle ou inconditionnelle) :
Si on connaît à l'avance le nombre d'itérations à effectuer, ou plus généralement si on veut parcourir une valeur itérable, on choisit une boucle *for*.

Si la décision d'arrêter la boucle ne peut s'exprimer que par un test, c'est la boucle *while* qu'il faut choisir.

Exercice 6 : Écrire une fonction qui prend en entrée une chaîne de caractères et renvoie sa chaîne-miroir c'est-à-dire la chaîne écrite dans le sens inverse.

Exercice 7 (Somme et nombre de diviseurs) :

1. Écrire la fonction Python sommeDivMax qui à un entier $n \in \mathbb{N}^*$ associe le plus petit entier $k \in [\![1; n]\!]$ pour lequel la somme de ses diviseurs est maximale.
2. Écrire la fonction Python nbDivMax qui à un entier $n \in \mathbb{N}^*$ associe le plus petit entier $k \in [\![1; n]\!]$ qui a le plus grand nombre de diviseurs.

Exercice 8 (Entiers parfaits) : Un entier naturel est dit *parfait* s'il est égal à la somme de ses diviseurs stricts. Écrire une fonction qui retourne la liste de tous les nombres parfaits compris entre les entiers $a \in \mathbb{N}^*$ et $b \in \mathbb{N}^*$.

Exercice 9 (Entiers jumeaux) : Deux nombres premiers sont dits *jumeaux* si leur différence a une valeur absolue égale à 2.

Écrire une fonction qui renvoie la liste de tous les couples de nombres premiers jumeaux compris entre les entiers $a \in \mathbb{N}^*$ et $b \in \mathbb{N}^*$ (avec $a + 1 < b$).

Indication : on pourra utiliser la fonction booléenne isprime du module sympy.

Exercice 10 (Référentiel rebondissant) : Une balle chute de n cm avec $n \in \mathbb{N}^*$. À chaque rebond, la hauteur maximale atteinte par la balle diminue de 10%. Réalisez une fonction qui affiche la hauteur des rebonds tant que celle-ci est strictement supérieure à 5cm et retourne le nombre de rebonds de hauteur strictement supérieure à 5cm.

Exercice 11 (Test de primalité) : Pour déterminer si un entier n supérieur ou égal à 2 est premier, on teste successivement les entiers compris entre 2 et $n-1$ pour savoir si l'un d'entre eux divise n. Dès qu'on a trouvé un tel entier divisant n, on peut affirmer que n n'est pas premier. Si au contraire aucun de ces entiers ne divise n, cela signifie que n est premier.

Écrire la fonction estPremier(n:**int**)→**bool** qui teste la primalité de n.

Exercice 12 (Division euclidienne) : Pour déterminer le quotient et le reste de la division euclidienne de $a \in \mathbb{N}$ par $b \in \mathbb{N}^*$, on se pose la question : « En a, combien de fois b ? ». Un algorithme simple consiste donc à partir de $q = 0$, puis incrémenter q d'une unité tant que $b \times q$ ne dépasse pas a. Dès que $b \times q > a$, on peut en déduire quotient et reste de la division euclidienne de a par b !

Écrire la fonction division(a:**int**,b:**int**)→(**int**,**int**) qui renvoie le quotient et le reste de la division euclidienne du paramètre a par le paramètre b.

Exercices supplémentaires

Exercice 13 (Partie entière) : Pour tout $x \in \mathbb{R}$, il existe un unique $n \in \mathbb{Z}$ tel que $n \leqslant x < n+1$ appelé partie entière de x et noté $n = \lfloor x \rfloor$.

Lorsque $x \geqslant 0$, on pose dans un premier temps $n = 0$ puis on incrémente la valeur de n d'une unité tant que n ne dépasse pas x. Dès que n dépasse x, on peut déterminer la valeur de $\lfloor x \rfloor$. Un raisonnement similaire s'applique lorsque $x < 0$.

Écrire la fonction partieEntiere(x:**float**)→**int** qui renvoie la partie entière de son paramètre x.

Exercice 14 (Critère de divisibilité par 7) : Soit $m \in \mathbb{N}$. À l'écriture de m en base 10 on retire le chiffre des unités puis on retranche deux fois ce chiffre au nombre ainsi obtenu. Le résultat de ces opérations étant noté m', on montre que

$$m \text{ est divisible par 7 si et seulement si } m' \text{ est divisible par 7}$$

Exemple : $\underline{31976} \rightsquigarrow 3197 - 2 \times 6 = \underline{3185} \rightsquigarrow 318 - 2 \times 5 = \underline{308} \rightsquigarrow 30 - 2 \times 8 = \underline{14}$ donc tous les nombres soulignés sont divisibles par 7.

1. Créer une fonction qui au nombre m associe le nombre m'.
2. Écrire un programme qui demande à l'utilisateur d'entrer un entier naturel m et effectue les opérations précédentes tant que le nombre obtenu est supérieur ou égal à 0. Votre programme affichera la suite des nombres ainsi calculés.
3. À partir d'une liste de multiples de 7 à déterminer et du résultat de la question précédente, générer un affichage qui précise si m est divisible par 7 ou non.