ITC – cours n°1

Introduction à la programmation et à Python

Ceci est un support de cours, mais sans pratique ce polycopié est insuffisant. vous pouvez retrouver le cours et des fichiers python associés sous divers formats sur cahier de prépa.

I. Introduction générale

1.1. Informatique, algorithmique, programmation

L'informatique est généralement définie comme l'ensemble des sciences et techniques de traitement automatisé de l'information par une machine (en général un ordinateur). Comme pour de nombreuses sciences, on peut séparer artificiellement des aspects plus théoriques (modèles, concepts, calculs...) et des questions appliquées (techniques de mise en œuvre, déploiement, maintenance...), la séparation n'étant pas toujours nette.

L'algorithmique est la science des algorithmes, c'est-à-dire des procédures automatisées qui permettent d'accomplir une certaine tâche. L'enseignement d'informatique de tronc commun (ITC) se concentre en particulier sur cet aspect (plutôt théorique en soi) : nous verrons de nombreux algorithmes pour répondre à un certain nombre de problématiques, ainsi que des réflexions théoriques sur leur efficacité, leur pertinence selon le contexte.

La programmation est le fait de transcrire concrètement un algorithme afin qu'il puisse être exécuté par une machine : il s'agit donc (en partie) de l'aspect appliqué de l'informatique. Nous nous attacherons simplement à programmer les algorithmes que nous aurons vu afin de mieux les comprendre et manipuler, l'aller-retour entre réflexion théorique et mise en œuvre pratique étant indispensable pour progresser. Cependant, nous ne verrons pas de techniques avancées de programmation dans ce cours.

1.2. Langage de programmation

Une machine numérique stocke toutes ses données sous forme binaire (des 0 et des 1), et notamment les instructions qui composent un algorithme qu'elle doit exécuter. Cependant, l'algorithme doit être écrit par un être humain (appelé·e développeur·euse ou programmeur·euse dans le cas qui nous occupe), et il est impensable de rédiger des algorithmes très avancés directement en langage binaire, trop éloigné de notre mode de pensée. La solution trouvée est de rédiger nos algorithmes dans un **langage de programmation** codifié, qui sera ensuite traduit en binaire par un autre programme ; ce programme intermédiaire est appelé **compilateur** (s'il fournit un fichier exécutable qui pourra être lancé autant de fois qu'on le souhaite) ou **interpréteur** (s'il lit le code et exécute les instructions à la volée).

Les langages sont qualifiés de plus ou moins bas niveau selon leur proximité avec les problématiques spécifiques à la machine : s'il faut gérer par des instructions précises la mémoire, des opérations élémentaires etc..., on parle d'un langage bas niveau ; à l'inverse, un langage haut niveau cachera es opérations à l'utilisateur, et on pourra se concentrer sur l'algorithme lui-même ; le code en devient plus concis, par exemple :

- en assembleur x86 (Linux), considéré comme très bas niveau :

```
section .data
    hellomsg :      db 'Hello, World !',10
    hellosize :     equ $-hellomsg
section .text
    global _start
_start :
    mov eax,4           ; appel système "write" (sys_write)
    mov ebx,1           ; file descriptor, 1 pour stdout
    mov ecx, hellomsg   ; adresse de la chaîne a afficher
    mov edx, hellosize  ; taille de la chaîne
    int 80h             ; exécution de l'appel système
    mov eax,1           ; appel système "exit"
    mov ebx,0           ; code de retour
    int 80h
```

- en C, considéré comme bas niveau :

```c
#include <stdio.h>     // bibliothèque entrées/sorties
int main(int argc, char **argv) {  // point d'entrée
  printf("Hello, World\n") ;       // fonction d'affichage
  return 0;                        // code de retour
}
```

- en Python, considéré comme haut niveau :

```python
print("Hello, World")  # fonction d'affichage
```

1.3. L'interpréteur interactif et le fichier de code

Python est un langage interprété, ce qui signifie que le code est lu, traduit en code binaire et exécuté dans la foulée. Cela permet une grande souplesse (on peut décider de faire exécuter une nouvelle instruction rapidement) mais empêche certaines optimisations[1].

Dans l'environnement de travail Spyder que nous utiliserons cette année (voir figure 1), on distingue plusieurs zones :

- la zone d'édition de code, à gauche ;
- l'interpréteur interactif (ou console), en bas à droite ;

[1] Notons que la question des optimisations fines ne sera pas abordée dans ce cours ; seules les optimisations par choix d'un algorithme plutôt qu'un autre nous intéresseront. Dans ce cadre, il est bien plus commode pour l'apprentissage de travailler avec un langage interprété

- la zone d'exploration de variables, en haut à droite.

L'interpréteur interactif permet d'exécuter des commandes Python à la volée, en tapant directement. Dans ce cas, le résultat de l'opération (s'il y en a un) est affiché immédiatement. On peut par exemple directement taper des calculs numériques et voir les résultats ; il s'agit donc de se servir de l'interpréteur comme d'une calculatrice. Nous verrons dans la pratique que l'interpréteur peut servir ponctuellement à tester certaines hypothèses en phase de débuggage. Par ailleurs, c'est là que s'affichent les messages d'erreur en cas de code ne respectant pas le standard.

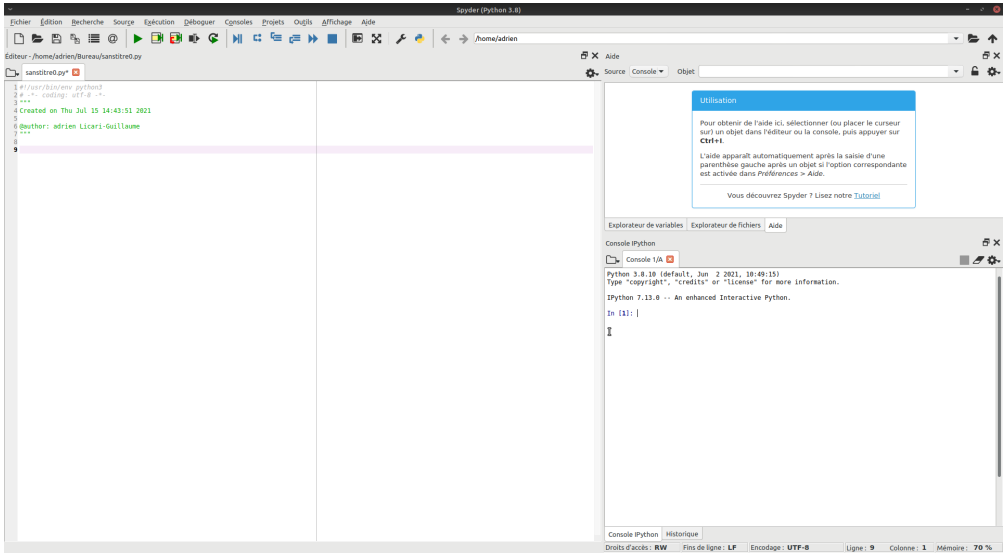


Figure 1 : L'environnement de travail Spyder

Lorsqu'on souhaite enchaîner plusieurs instructions (nous ferons assez souvent des codes de plusieurs dizaines de lignes), il n'est pas commode de tout retaper dans l'interpréteur à chaque fois que l'on souhaite changer un paramètre ou faire un nouveau test. On peut alors écrire plusieurs instructions dans un fichier texte, accessible dans la zone de gauche de Spyder ; on pourra alors exécuter le fichier, c'est-à-dire envoyer l'ensemble de ses instructions dans l'interpréteur : tout se passe comme si on avait tapé chaque ligne du fichier dans l'interpréteur.

1.4. Commentaires

On peut voir dans les exemples précédents des informations pour le lecteur/la lectrice, ici écrites en français (et donc pas dans le langage de programmation) ; elles ne sont pas lues par le compilateur/ l'interpréteur, qui les ignore simplement. Ces informations sont là uniquement pour lecture, soit de la personne qui l'a programmé (ce qui sert à se rappeler la structure, ou des points subtils), soit de quelqu'un d'autre qui va reprendre/ adapter/améliorer le code. Ces éléments sont appelés des **commentaires**. En Python, les commentaires sont annoncés avec le symbole # : tout ce qui se trouve au-delà, sur la même ligne, sera ignoré.

Il est absolument indispensable d'écrire des commentaires en nombre correct : il ne s'agit pas de doubler les choses évidentes, mais simplement de donner des points d'ancrage. C'est par la pratique que l'on apprend à mettre la bonne quantité de commentaires, mais un bon point de départ reste que « s'il n'y a aucun commentaire, c'est qu'il n'y en a pas assez ».

1.5. Mise en forme

Au-delà des commentaires, on peut également mettre en forme d'une certaine manière le code afin de simplifier la lecture. par exemple, si on prend les deux lignes suivantes,

```
a = a + b*c
a=a+b*c
```

elles vont produire le même résultat pour la machine, mais la lecture est différente. Si on s'habitue à une certaine façon de typographier le code, cela simplifie la lecture ; à l'inverse, un code dont la mise en forme change attire l'œil et empêche de se concentrer sur le cœur de l'algorithme. En Python, il existe une convention générale (appliquée par de nombreuses personnes) définie dans un document appelé PEP8. Dans ce cours, nous prendrons l'habitude de suivre ces recommandations.

II. Les bases de Python

2.1. Variables simples

a) Notion de variable

Afin de pouvoir effectuer des algorithmes intéressants, il faut pouvoir retenir certaines valeurs en cours d'exécution, par exemple :

- dans la recherche d'un plus grand élément, quelle est la plus grande valeur déjà trouvée ?
- dans un jeu de gestion : combien d'habitants sont actuellement dans la ville ?
- pour afficher un réseau social : combien de likes a actuellement ce post ?
- dans un navigateur web : quelle était la page précédente ?
- dans un logiciel de comptabilité : quel est le salaire de tel employé ?
- etc...

On comprend donc que toute information doit être stockée dans une variable pour être utilisable ; si on perd la variable, on perd l'information.

b) Nombres

Les nombres entiers correspondent à des variables de type `int` et les nombres réels « à virgule » au type `float` . Nous reviendrons plus tard dans l'année en détail sur leur représentation binaire. On donne une valeur à une variable en utilisant l'opérateur d'affectation :

```
In [1]: a = 2
In [2]: b = 7.2
In [3]: c = -2e-6
```

Au moment de l'exécution, Python choisit le type adapté pour la variable : ici `a` sera un entier, tandis que `b` et `c` seront des réels. Notons immédiatement la possibilité d'utiliser la notation scientifique : `-2e-6` $= -2 \cdot 10^{-6}$. Si on veut que `a` soit un réel (et non un entier) valant 2, il faudra écrire explicitement un réel : `a = 2.0`.

On dispose pour ces nombres des opérateurs usuels : addition `+`, soustraction `-`, multiplication `*`, division `/`, puissance `**`, auxquels s'ajoutent pour les entiers la division euclidienne `//` et le reste par cette division (ou modulo) %. Il faut donc faire attention avec les entiers : le résultat de `3 / 2` ne sera pas un entier, puisqu'il s'agit de 1, 5. Cela est en fait toujours vrai avec l'opérande `/` : puisque Python ne sait pas a priori si le résultat serait un entier ou non, il convertit toujours en réels pour effectuer l'opération. Par exemple, `1/1` renvoie `1.0` !

Les opérateurs mathématiques sont évalués dans l'ordre de priorité usuel ; si on souhaite le modifier, il faut utiliser des parenthèses, comme d'habitude en mathématiques :

```
In [1]: 2 + 3*4
Out[1]: 14
In [2]: (2+3) * 4
Out[2]: 20
```

En plus de nombres « usuels », Python propose une représentation de l'infini : `x = float('inf')` fait de `x` un nombre réel supérieur à n'importe quel autre. Ceci est une spécificité de Python, tous les langages ne le proposent pas.

- Python propose aussi un type natif pour les nombres complexes : le symbole `j` sert à définir la partie imaginaire, par exemple `1 - 2j` représente le complexe $1 - 2i$. Cela n'est pas à retenir, puisque non au programme.
- Le PEP8 donne les recommandations suivantes :
  - mettre des espaces autour de l'opérateur d'affectation : `a = 2` et non `a=2` ;
  - mettre des espaces autour des opérateurs de plus basse priorité : `+` et `-` a priori, sauf si des parenthèses viennent modifier l'ordre ;
  - ne pas mettre d'espace contre les parenthèses/crochets ;
  - lorsqu'on veut mettre un petit commentaire au bout d'une ligne, mettre deux espaces avant le signe `#` ; si on commente ainsi plusieurs lignes successives, aligner les commentaires verticalement.

c) *Booléens*

Les variables booléennes sont des variables qui ne peuvent prendre que deux valeurs : `True` (vrai) et `False` (faux). On peut simplement déclarer une variable de ce type :

```
proposition_1 = True
```

Comme pour les nombres, Python constate que `True` est un booléen, donc il en déduit que la variable `proposition_1` l'est également.

Une façon d'obtenir des booléens est d'utiliser un opérateur de comparaison sur d'autres variables : égalité `==`, inégalité `!=`, ordre strict `<` ou `>`, ordre large `<=` ou `>=`. On peut également transformer `True` en `False` (et vice-versa) avec l'opérateur `not` et composer les booléens avec les opérateurs `and` et `or` :

```
In [1]: a, b, c = 1, 1, 2
In [2]: a == b
Out[1]: True
In [3]: a == c
Out[2]: False
In [4]: not (a == c)
Out[3]: True
In [5]: (a == b) or (a == c)
Out[4]: True
In [6]: (a == b) and (a == c)
Out[5]: False
```

L'opérateur de comparaison `==` n'est pas l'opérateur d'affectation `=` ! Si ces deux opérateurs sont confondus en mathématiques, ils sont très différents en informatique :

- l'opérateur d'affectation évalue d'abord ce qui se trouve à sa droite, puis fait en sorte que le nom à sa gauche représente la valeur obtenue ; par exemple, si `a` vaut 2, alors `b = a**2 - 1` va d'abord évaluer ce qui se trouve à droite comme un entier valant 3, puis affecter à `b` cette valeur (suite à cette instruction, `b` vaut 3) ;
- l'opérateur de comparaison prend deux objets et vérifie s'il s'agit du même ; en fonction, il renvoie un booléen égal à `True` ou `False`.

On dit des opérateurs `or` et `and` qu'il sont paresseux : dès que la réponse peut être déterminée, elle est renvoyée et les calculs stoppés. Par exemple, dans l'expression `(a == b) or (a == c)`, si `a == b` est évaluée à `True`, alors `a == c` ne sera même pas évaluée, puisque, quel que soit le résultat de cette évaluation, l'ensemble sera nécessairement `True`. À l'inverse, une condition en deux parties avec and ne sera pas entièrement évaluée si une de ces composantes est évaluée à `False`, puisqu'automatiquement l'ensemble serait `False`.

L'égalité entre deux réels est parfois difficile à tester : en effet, pour des raisons de représentation binaire, certaines opérations qui devraient donner le même résultat ne le donnent pas, par exemple :

```
In [1]: a = 2.2*3.0  # a représente en fait 6.6000000000000005
In [2]: b = 2.0*3.3  # b représente 6.6
In [3]: a == b
Out[1]: False
```

Pour tester l'inégalité de deux réels, il vaut donc mieux tester « s'ils sont suffisamment proches pour que leur différence semble être un problème de représentation ». Une fonction du module[2] `math` propose une fonction permettant une telle comparaison :

```
In [1]: from math import isclose  # simplifie la comparaison de réels
In [2]: a, b = 2.2*3.0, 2.0*3.3
In [3]: a == b
Out[1]: False
In [4]: isclose(a, b)
Out[2]: True
```

d) Chaînes de caractères

Pour contenir du texte, on utilise une variable de type `str` (pour « character string », ou « chaîne de caractères »). Une valeur de type `str` est délimitée par des guillemets simples ou triples :

```
message = "Bonjour à toi aventurier·ère !"
```

Nous reviendrons en détails sur ce type de données dans un chapitre dédié.

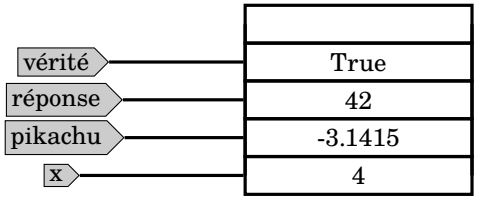
2.2. Fonctionnement détaillé de l'affectation

Pour éviter un certain nombre d'erreurs, il est nécessaire d'avoir une idée correcte de la façon dont Python retient les données, et de la méthode d'affectation. En Python, un nom est une **référence (ou étiquette) attachée à un emplacement en mémoire**[3].

Par exemple, si on exécute le code suivant :

```
x = 4
pikachu = -3.1415
réponse = 42
vérité = (réponse == réponse)
```

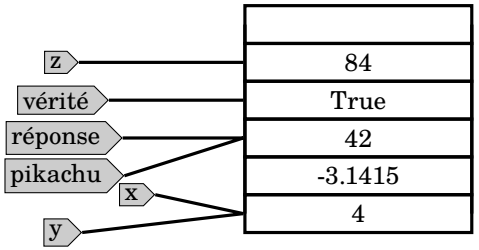
alors on peut se représenter l'état de la mémoire comme sur le schéma ci-contre. Si ensuite on exécute les instructions suivantes




```
y = x
pikachu = réponse
z = x*pikachu/2
```

alors Python

- ligne 1 : **évalue** l'expression de droite, en lisant la valeur associée à la référence `x` : il en déduit que l'expression de droite vaut 4, valeur qui est déjà référencée en mémoire : il **attache** la référence `y` (créée à cette occasion) à cette valeur ;
- ligne 2 : **évalue** l'expression de droite, en lisant la valeur associée à la référence `réponse` : il en déduit que l'expression de droite vaut 42, valeur qui est déjà référencée en mémoire : il **attache** la référence `pikachu` (simplement déplacée puisqu'elle existait déjà) à cette valeur ;
- ligne 3 : **évalue** l'expression de droite, en lisant les valeurs associées aux références `réponse` et `x` ,: il en déduit que l'expression de droite vaut 84, valeur qui n'est pas référencée en mémoire : il **inscrit en mémoire** cette valeur, puis il **attache** la référence `z` (créée à cette occasion) à cette valeur.



On obtient finalement un état de la mémoire représenté sur le schéma ci-contre.

Fonctionnement de l'affectation

Cette séquence est toujours la même pour une affectation :

- évaluer l'expression à droite de l'opérateur ;
- inscrire la valeur en mémoire si elle ne l'est pas déjà ;
- attacher la référence de gauche à la valeur.

Quelques conséquences importantes sont les suivantes :

- l'opérateur d'affectation n'est absolument pas symétrique (contrairement au signe = en mathématiques) : la partie de gauche et celle de droite ne sont pas du tout traitées de la même façon ;
- en particulier, la partie de gauche doit être une **référence**, donc un **nom valide** ; la partie de droite peut être une expression, mais par exemple `a + b = 3` n'a aucun sens, puisque `a + b` n'est pas un nom mais une expression ;
- on peut écrire des affectations qui « semblent fausses » sur le plan mathématique ; un exemple très courant est l'incrémentation (ou la décrémentation) :

```
nb_vies = nb_vies + 1
```

Si le signe `=` était une égalité, cela serait faux ; mais il s'agit d'une affectation : on lit le nombre de vies (par exemple 4), on ajoute 1 (dans l'exemple, cela donne 5), puis on attache l'étiquette `nb_vies` à cette nouvelle valeur. Ainsi, la variable `nb_vies` vaut 1 de plus après cette instruction qu'avant.

Cette opération de modification d'une variable est si courante qu'elle a droit à ses propres opérateurs pour alléger l'écriture :

[2] Nous parlerons des modules dans un prochain chapitre
[3] Pour simplifier, disons en mémoire RAM

```
a += b  # identique à a = a + b
a -= b  # identique à a = a - b
a *= b  # identique à a = a * b
a /= b  # identique à a = a / b
```

On peut enfin faire des **affectations multiples** : dans ce cas, les expressions de droite sont **toutes évaluées** avant que les étiquettes ne soient affectées. Par exemple,

```
a, b, c = 2, 3, 4
a, b = a**2, a*b  # après cela, a vaut 2**2 -> 4 et b vaut 2*3 -> 6
```

Pour finir, notons qu'évaluer une variable qui n'a jamais été définie conduit à une erreur assez compréhensible si on la lit : par exemple, `b = a + pouet` après les instructions précédentes conduit à

```
NameError: name 'pouet' is not defined
```

Enfin, donnons les règles de nommage : les variables ont des noms qui peuvent contenir des caractères alphanumériques (y compris les accents en Python 3), mais ne peuvent pas commencer par un chiffre. On peut également utiliser l'underscore `_` mais pas `-` (pour ne pas le confondre avec une soustraction). Le PEP8 recommande de commencer par une minuscule et de séparer les mots d'un même nom avec l'underscore, par exemple `mon_pokemon`.

> Dans les exemples simples du cours, les variables s'appellent souvent `a`, `x` ...qui sont des noms peut explicites. En code « réel » (même dans un petit exercice), il est recommandé d'utiliser au plus des noms explicites, même si cela est plus long à taper : cela améliore la lisibilité du code et évite de mettre trop de commentaires. On préférera donc `nb_étudiants` à `n`.

III. Fonctions : « ranger » des algorithmes

3.1. Principe

Un algorithme peut être défini comme une suite d'instructions qui prend des entrées et renvoie un résultat. Dans un premier temps[4], nous allons confondre cette notion d'algorithme avec celle de **fonction**.

Une fonction en informatique est un moyen de grouper en un « bloc » une séquence d'instructions, cette séquence pouvant être ajustée à travers un ou plusieurs paramètres qu'on lui fournit (c'est d'ailleurs ce qui fait l'intérêt des fonctions). Puisque cette séquence d'instructions sera appelée à plusieurs reprises, on lui donne un nom (une étiquette) et elle sera stockée quelque part en mémoire. La syntaxe de définition usuelle est la suivante :

```
def nom_de_la_fonction(arguments):  # les arguments sont les entrées
    instruction_1
    ...
    instruction_n
    return résultat  # sorties
```

On voit apparaître un **bloc** d'instructions, ouvert par les deux-points et l'indentation, et dont la fin est annoncée par la fin de l'indentation.

Pour appeler la fonction, on utilise son nom suivi de l'opération d'appel de fonction[5] `()` (les parenthèses), et on donne les paramètres entre les parenthèses.

Prenons l'exemple suivant :

```
def addition(a, b):
    c = a + b
    return c
```

On peut alors l'utiliser dans l'interpréteur :

```
In [1]: addition(3, 7)
Out[1]: 10
In [2]: addition(-4.5, 7)
Out[2]: 2.5
In [3]: addition(0, 7)
Out[3]: 7
```

On constate donc que la fonction (l'agorithme) reste la même, mais si on change les valeurs d'entrée, les instructions s'appliquent sur des valeurs différentes et conduisent à un résultat différent.

> Le PEP8 recommande la mise en forme suivante :
>
> - pas d'espace contre les parenthèses ;
> - une espace après les virgules dans les paramètres ;
> - **deux** lignes blanches avant et après la définition d'une fonction ;
> - on peut sauter une ligne de temps en temps dans le corps d'une fonction, pour séparer des blocs cohérents entre eux.
>
> Par ailleurs, il est recommandé de donner des noms explicites aux fonctions, aux paramètres et aux variables internes ; on comprend mieux ce qui ce passe lorsqu'on lit la ligne `loyer = trouver_loyer(numero_appartement)` que `pikachu = tatayoyo(moutarde)`

3.2. Fonctions mathématiques

On peut avoir accès à de nombreuses fonctions mathématiques usuelles en utilisant le module `math` :

- les fonctions trigonométriques sin, cos, tan etc... ;
- exp, log, ln (attention : `log` représente le logarithme népérien ; le logarithme décimal

[4] Nous approfondirons les choses plus tard

[5] Le nom « opérateur d'appel de fonction » n'est pas officiel, mais il est pratique pour le penser.

est `log10`) ;
- partie entière `floor` ;
- la constante π y est aussi définie.

Il faut pour les utiliser importer le module, par exemple

```
import math as m
x = 1.0
y = m.sin(x)
```

On peut obtenir de l'aide sur une fonction à l'aide de la commande `help` dans l'interpréteur :

```
In [1]: help(m.sin)
| Help on built-in function sin in module math :
|
| sin(x, /)
|     Return the sine of x (measured in radians).
```

IV. Structures de contrôle

4.1. Branchements conditionnels

Un programme doit pouvoir prendre une décision par lui-même dans un contexte donné ; pour cela, on utilise une structure conditionnelle :

```
if expression_booléenne:
    instruction_1
    instruction_2
    ...
    instruction_n
suite_du_programme
```

Dans ce cas, l'expression booléenne est évaluée (ce peut être une variable booléenne, ou encore une expression évaluée à la volée comme `a == b`) ; si elle vaut `True` , les instructions dans le bloc indenté sont exécutées, sinon elles ne le sont pas.

Il faut ici absolument respecter la syntaxe :
- les deux-points après la condition, pour amorcer le bloc d'instructions conditionnelles (le PEP8 recommande de ne pas mettre d'espace avant les deux-points) ;
- l'indentation doit être respectée dès la ligne suivante : c'est elle qui démarre le bloc d'instructions à exécuter sous condition ;
- on revient à l'indentation précédente pour délimiter la fin du bloc d'instructions conditionnelles.

On peut également choisir d'exécuter d'autres instructions si la condition n'est pas remplie, à l'aide du mot-clé `else` :

```
def entrée_en_boîte(âge):
    if âge < 18:
        réponse = "Vous ne pouvez pas entrer en boîte."
    else:
        réponse = "Allez-y bonne soirée !"
    return réponse
```

ou encore enchaîner les conditions avec `elif` :

```
def taille(nombre):
    if nombre < 10:
        réaction = "Mais c'est tout petit !"
    elif a < 100:
        réaction = "C'est pas mal ça..."
    else:
        réaction = "Oulah, que c'est grand !"
    return réaction
```

Quoiqu'il arrive, au plus un bloc d'instructions sera exécuté : si un bloc voit sa condition évaluée à True , les instructions de ce bloc sont exécutées, puis le code saute à la fin de la structure de branchement conditionnel.

4.2. Boucles

a) Boucle while

Pour répéter une action jusqu'à ce qu'une condition soit remplie, on utilisera `while` . Attention, si la condition est vérifiée avant d'entrer dans la boucle, celle-ci ne sera jamais parcourue ! Il est donc important d'initialiser correctement la condition de boucle.

```
while condition_est_True:
    instruction_1
    instruction_2
    ...
    instruction_n
suite_du_programme
```

Par exemple, l'algorithme code suivant boucle jusqu'à ce que l'utilisateur trouve la valeur attendue :

```
def trouver_nombre():
    a = 0
    réponse = 42
    while a != réponse:
        a = récupérer_saisie()   # ne pas se préoccuper des détails ici
        a = int(a)
    return "Bravo !"
```

Les boucles `while` amènent souvent à une erreur classique : la boucle infinie. La condition doit pouvoir devenir fausse, ce qui signifie que les variables associées à la condition

peuvent changer dans le bloc de la boucle. Par exemple, ceci

```
a = 0
while a == 0 :
    b += 1
```

ne s'arrête jamais, puisque `a` ne change pas de valeur dans la boucle, et ne peut donc jamais devenir non nulle. Si jamais votre code ne termine pas, les boucles `while` sont donc de bons endroits à regarder...

On peut enfin stopper une boucle `while` avant que la condition ne soit remplie, avec l'instruction `break` : par exemple, on peut ainsi amorcer la boucle précédente sans initialiser `a` :

```
def trouver_nombre():
    réponse = 42
    while True:
        a = récupérer_saisie()
        a = int(a)
        if a == réponse:
            break
    return "Bravo !"
```

Dès que l'instruction `break` est rencontrée, l'exécution du code reprend après le bloc `while` .

Dans l'exemple précédent, l'usage de `break` est parfaitement redondant avec celui de la boucle simple : la condition de sortie est déplacée vers l'intérieur du corps de boucle. En revanche, `break` prend son sens quand il existe une condition de sortie normale, et une condition exceptionnelle.

b) *Boucle for*

On veut souvent répéter une opération sur un ensemble d'entités ; citons quelques exemples :

- on peut chercher les entiers entre 1 et n qui vérifient une certaine propriété, on écrira alors des instructions que l'on voudra voir exécuter pour **chaque cas** $i = 1, i = 2 \ldots$ jusqu'à $i = n$;
- on peut mettre en paiement le salaire de tous les employés d'une entreprise : on dira alors pour que **chaque employé**, on exécutera la même série d'instructions (récupérer le salaire, le RIB, mettre en paiement) pour chaque employé ;
- on peut passer chaque pixel d'une image en noir et blanc ; dans ce cas, pour **chaque pixel**, on exécutera la même série d'instructions (récupérer la couleur, calculer l'équivalent en niveaux de gris, changer la couleur) ;
- dans un jeu de tir, on peut mettre à jour les positions des balles à chaque frame : dans ce cas, pour **chaque balle**, on appliquera les mêmes instructions (récupérer la position et la vitesse, calculer la nouvelle position, mettre à jour).

Dans tous ces cas, on remarque qu'on applique un même traitement à un ensemble d'occurrences. Pour l'instant, nous nous concentrons sur le cas d'une collection de nombres entiers ; nous verrons les autres cas au fur et à mesure. Pour écrire une telle itération, la syntaxe est

```
for elt in collection:  # pour l'ensemble des instructions du bloc,
    instruction_1       #      la variable elt prendra chaque
    instruction_2       #      valeur de la collection
    ...
    instruction_n
suite_du_programme
```

On peut générer une collection d'entiers compris entre 0 et $n-1$ avec l'instruction `range(n)` ,
par exemple

```
for i in range(100):
    x = 2*i
```

affecte successivement à la variable `x` les nombres pairs entre 0 et 198.

> **!** Attention aux bornes ! Par défaut en Python, on inclut le premier et on exclut le dernier. Si vous avez un doute, vous pouvez toujours taper `help(range)` dans la console...

On peut également générer des collections d'entiers plus subtiles à l'aide de range , car elle accepte plus d'un paramètre ; par exemple

- `range(3, 12)` créé une collection d'entiers de 3 (inclus) à 12 (exclus) ;
- `range(3, 12, 2)` créé une collection d'entiers de 3 (inclus) à 12 (exclus) espacés de 2 ;
- `range(12, 3, -1)` créé une collection d'entiers de 12 (inclus) à 3 (exclus) espacés de -1.

De même que pour la boucle `while` , on peut utiliser l'instruction `break` pour stopper une boucle avant la fin.

V. Listes

5.1. Définition, usage basique

Une liste Python est un **ensemble mutable de variables**, c'est-à-dire que les éléments qui le composent peuvent être changés. On les définit à l'aide des crochets :

```
In [1]: notes = [12, 4, 20, 9, 8, 8, 11, 14]
```

On définit également **l'opérateur concaténation** `+` et **l'opérateur répétition** `*` , permettant de créer de nouvelles listes à partir des précédentes :

```
In [2]: x, y = [12, -5, True], [34, 2]
In [3]: z = x + y  # z référence la liste [12, -5, True, 34, 2]
In [4]: t = y*3    # t référence la liste [34, 2, 34, 2, 34, 2]
```

On peut enfin accéder à la taille et aux éléments de la liste avec les syntaxes suivantes :

```
In [5]: len(z)        # renvoie la taille de z, ici 5
In [6]: a = z[0] + 2  # z[0] vaut 12 donc a vaut 14
In [7]: z[2]
Out[1]: True
In [8]: u = z[2:4]    # u est la liste [True, 34]
```

Comme toujours, les indices démarrent à 0 et finissent à $n-1$, n étant le nombre d'éléments dans le tuple.

Soyons attentifs à la syntaxe `[:]` : on dit que l'on fait **une tranche** de la liste. La syntaxe est similaire à celle de `range`, avec quelques ajouts :

- `z[:j]` prend tous les éléments du 0 (inclus) au j^{e} (exclus) ;
- `z[i:]` prend tous les éléments du i^{e} (inclus) au dernier ;
- `z[i:j]` prend tous les éléments du i^{e} (inclus) au j^{e} (exclus) ;
- `z[i:j:2]` prend tous les éléments du i^{e} (inclus) au j^{e} (exclus) de deux en deux ;
- `z[i:j:-1]` prend tous les éléments du i^{e} (inclus) au j^{e} (exclus) dans l'ordre inverse (cette expression suppose donc que $i > j$) ;
- l'indice -1 est valide et représente le dernier élément ; de même -2 est l'indice de l'avant-dernier et ainsi de suite.

Les listes sont **mutables** : on peut changer la valeur d'un élément de la liste :

```
In [5]: notes[3] = 11
In [6]: notes
Out[4]: [12, 4, 20, 11, 8, 8, 11, 14]
```

Une liste peut contenir des éléments de types différents ; par exemple `[2, 2.72, "bonjour", True]` est une liste valide. Cela est néanmoins considéré comme une mauvaise pratique : comme on peut changer les éléments, il vaut mieux que tous les éléments aient toujours le même comportement, donc le même type.

5.2. Construction d'une liste

Une méthode classique pour remplir une liste est de partir d'une liste vide et de rajouter des éléments à la fin avec l'instruction `append` qui **rajoute un élément à la fin de liste** :

```
def liste_carrés(n):
    carrés = []              # carrés est une liste vide
    for i in range(n):       # pour chaque valeur de i entre 0 et n-1,
        carrés.append(i**2)  # on ajoute la valeur i**2 à la liste
    return carrés
```

À la fin de cette série d'instructions, `carrés` n'est donc plus une liste vide : l'instruction `append` **modifie la taille de la liste**.

On peut également utiliser des formules avec une boucle `for` directement dans les crochets : on appelle cela des **listes en compréhension** : le code suivant est donc équivalent au précédent[6] :

```
carrés = [i**2 for i in range(n)]
```

On peut enfin créer des listes par **concaténation** ou par **répétition**, à l'aide des opérateurs vus précédemment :

```
x = [4, 5, 7]
y = [9]*4  # répétition : contient [9, 9, 9, 9]
z = x + y  # concaténation : z contient [4, 5, 7, 9, 9, 9, 9]
```

L'opposé de `append` est `pop` : elle enlève le dernier élément et le renvoie[7] :

```
In [1]: n = 50
In [2]: carrés = [i**2 for i in range(n)]
In [3]: len(carrés), carrés[-1]
Out[1]: 50, 2401
In [4]: x = carrés.pop()
In [5]: len(carrés), x
Out[2]: 49, 2401
```

5.3. Identité de deux listes

Le caractère mutable des listes donne de la souplesse pour leur utilisation, mais provoque des difficultés de compréhension. Par exemple, prenons le code suivant :

```
In [1]: a = [3, 2, 1]
In [2]: b = a
```

Dans ce cas, les étiquettes `a` et `b` représentent la **même liste** en mémoire ; en conséquence, l'instruction `a[0] = -1` changera non seulement l'élément 0 de la liste `a`, mais aussi celui de la liste `b` (puisque ce sont les mêmes !) ; on peut s'en convaincre en affichant les deux :

```
In [3]: a[0] = -1
In [4]: a
Out[1]: [-1, 2, 1]
In [5]: b
Out[2]: [-1, 2, 1]
```

En revanche, on aurait pu définir une liste différente de `a`, contenant les mêmes éléments ; on les distingue grâce à l'opérateur `is` : dans le cas précédent

```
In [6]: a is b
Out[3]: True
```

[6] Ou presque : le résultat est exactement le même, mais pour des raisons subtiles que nous aborderons plus tard, leur rapidité n'est pas la même

[7] En réalité, on peut insérer/retirer un élément avec `pop` et `insert` (similaire à `append`) ailleurs qu'en dernière position ; cependant ces usages ont été exclus du programme car ils peuvent, selon les circonstances, être très coûteux en temps de calcul, ce qui complexifie des raisonnements que nous verrons dans des chapitres ultérieurs.

tandis que

```
In [1]: a = [3, 2, 1]
In [2]: b = [3, 2, 1]
In [3]: a == b  # a et b ont-elles les mêmes éléments ?
Out[1]: True
In [4]: a is b  # a et b sont-elles la même liste en mémoire ?
Out[2]: False
In [5]: a[0] = -1
In [6]: a, b
Out[3]: [-1, 2, 1], [3, 2, 1]
In [7]: a == b  # a et b ont-elles les mêmes éléments ?
Out[4]: False
```

5.4. Parcourir des listes

Il est très courant de parcourir des listes ; par exemple, considérons une liste de nombres donnée `nombres` et essayons simplement d'afficher tous ses éléments : on peut utiliser un compteur i qui prenne chaque valeur des indices corrects de la liste $i \in 0, 1, \dots, n-1$, soit :

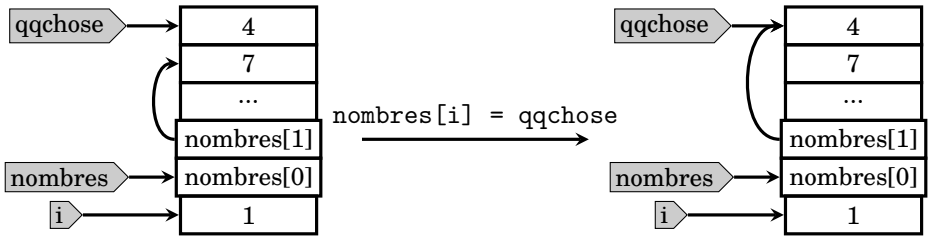
```
n = len(nombres)
for i in range(n):  # i va parcourir l'ensemble {0, 1, ..., n-1}
    x = nombres[i]  # pour une valeur de i donnée, x vaut nombres[i]
    # autres instructions
```

On peut également faire la boucle for non pas sur la collection $\{0, 1, \dots, n-1\}$ des indices, mais directement sur la collection $\{\ell_0, \ell_1, \dots, \ell_{n-1}\}$ des valeurs de la liste :

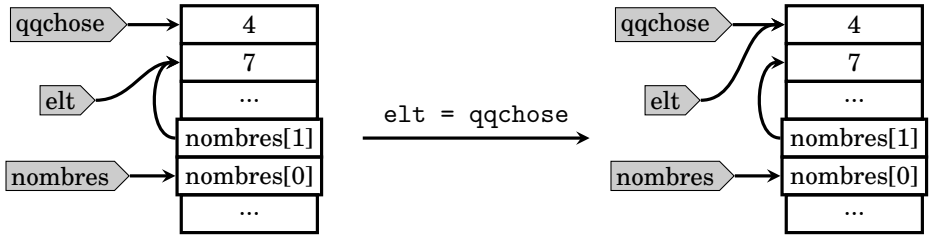
```
for elt in nombres:  # elt va parcourir l'ensemble des valeurs de nombres
    x = elt          # pour une valeur de elt donnée, x vaut elt
    # autres instructions
```

Cette deuxième écriture présente une différence notable avec la première, même si, ici, elle mène au même résultat :

- avec la première formulation, on peut modifier la liste, puisqu'on peut écrire `nombres[i] = qqchose` ; en effet, les `nombres[i]` sont eux-mêmes des références vers les éléments de la liste ;



- on ne peut pas le faire avec la deuxième, puisque `elt` est ici une copie d'une valeur contenue dans la liste ; ainsi `elt = qqchose` modifiera la valeur attachée à la référence `elt`, mais pas celle attachée à la référence `nombres[i]`, qui reste inchangée.



On peut de cette manière faire une copie d'une liste : en effet, on peut créer une nouvelle liste, et recopier les éléments de la première un à un :

```
def copie(liste):
    n = len(nombres)
    copie = [0]*n        # on prépare une liste de la bonne taille
    for i in range(n):   # on doit pouvoir accéder à copie[i]
        copie[i] = nombres[i]
    return copie
```

Il existe une fonction toute faite qui fait exactement la même chose : la bien nommée `copy` Le code précédent (avant les tests) est donc équivalent à `copie = nombres.copy()`. On peut également utiliser une tranche contenant tous les éléments pour générer une copie : `copie = nombres[:]`.

> On dit de cette copie qu'elle est superficielle : en effet, on copie directement les étiquettes présentes en `nombres[i]` ; mais si cette étiquette pointe vers une liste, alors `nombres[i]` et `copie[i]` représentent deux étiquettes différentes sur la même liste. Ce point très subtil sera revu et travaillé en TD.

ITC – cours n°1

Exercices

Les bases de Python

Variables simples : calculs, fonctions mathématiques

1 – Effectuer les calculs suivants à l'aide de l'interface Python. On se posera systématiquement la question de l'utilité des parenthèses dans chaque cas :

$2+(5\times 7)=$..... $(2+5)\times 7=$..... $2-(5-7)=$..... $(2-5)-7=$.....

$(2\times 5)^3=$..... $2\times(5^3)=$..... $3^{(2^3)}=$..... $(3^2)^3=$.....

$\frac{24}{2\times 3}=$..... $\frac{24}{2}\times 3=$..... $\frac{24}{\frac{2}{3}}=$..... $\frac{\frac{24}{2}}{3}=$.....

2 – Déterminer (sans utiliser l'interpréteur Python) les résultats que donneront les expressions Python suivantes, puis vérifier que Python fournit bien le résultat attendu.

```
 2 + 5 * 7 = ..............        2 * * 2 * * 3 = ..............
24 / 2 * 3 = ..............          24 / 2 / 3 = ..............
```

3 – Effectuer, à l'aide de Python, les calculs suivants ; on pourra importer le module `math`

$\sqrt{2+\sqrt{2+\sqrt{2}}}=$ $\sqrt[3]{7}=$ $\arccos(0.5)=$ $\tan(\pi/6)=$

$e^{e^2}=$ $\ln(2^{30})=$ $\log_{10}(1023)=$ $\log_2(1023)=$

Fonctionnement de l'affectation, calculs avec variables

1 – On donne la séquence d'instructions suivante :

```
a, b, c = 2, 3, 5
a, b, c = b, c, a
a, b, c = 7, a+2, a*a
a, a, a = a, b, c
a, b, c = a*c, b//a, c/a
a, b, c = b+c, a, a%c
a, b, c = (a+b)//(a-c), a**2+1, b//a-4*c
```

Déterminer, sans Python, les valeurs désignées par les noms `a`, `b` et `c` après chaque instruction ; une fois cela fait, vérifier avec Python en recopiant ce code dans l'éditeur (on pourra ajouter `print(a, b, c)` entre chaque ligne).

Structures de contrôle

Branchements conditionnels

1 – Écrire une fonction `calcul` avec en entrée deux variables `nombre1` et `nombre2`, qui renvoie une variable `résultat` égale à leur produit si celui-ci est inférieur à 1000, et leur somme sinon. On pourra tester dans l'interpréteur

```
In [1]: calcul(20, 30)
Out[1]: 600
In [2]: calcul(40, 30)
Out[2]: 70
```

2 – Écrire une fonction `plat` avec en entrée deux variables booléennes `végétarien` et `chaud` ; elle renvoie une variable `choix` de type `str`, qui contient le choix de menu selon ces deux variables :

- pour un repas végétarien chaud, une lasagne d'aubergines ;
- pour un repas végétarien froid, un assortiment de mezze libanais ;
- pour un repas carniste chaud, des spaghettis bolognaise ;
- pour un repas carniste froid, une salade César.

On pourra tester, par exemple

```
In [1]: plat(True, False)
Out[1]: Mezze libanais
```

3 –Si vous avez traité la question précédente avec des conditions imbriquées, la reprendre avec une seule indentation, en utilisant les opérateurs `and` et/ou `or` ; et vice-versa.

4 – On rappelle que la fonction arcsin est définie pour tout réel compris dans l'intervalle $[-1;1]$. Écrire une fonction `arcsin_protégé` qui prend en paramètre un réel `x` et renvoie une variable `résultat` valant $\arcsin x$ si celui-ci est défini, et 100 sinon[8].

5 – Écrire une fonction `FizzBuzz` qui prend en paramètre un entier `nombre` et qui renvoie une variable `résultat` telle que

- si le nombre est divisible par 3, elle vaut « Fizz » ;
- si le nombre est divisible par 5, elle vaut « Buzz » ;
- si le nombre est divisible par 3 **et** par 5, elle vaut « FizzBuzz » ;
- sinon, elle vaut le nombre lui-même.

6 – On visite un musée ; l'entrée est gratuite si l'on a moins de 26 ans **ou** si l'on est professeur (peu importe la discipline) ; elle est de 5€ sinon. Il suffit de remplir l'une des deux conditions pour être admis gratuitement à l'intérieur (bien sûr, si vous remplissez les deux, l'offre reste valable). Écrire une fonction `prix_musée` qui prend en paramètre une variable booléenne `est_enseignant` et une variable entière `âge` ; elle renvoie une variable `prix` représentant le prix de l'entrée. On pourra tester quelques cas, par exemple

[8] Cette façon de faire, qui consiste à choisir une valeur de retour incohérente par convention pour signifier un problème, n'est absolument pas la meilleure ; mais elle suffira dans le cadre de ce petit exercice

```
In [1]: prix_musée(True, 55)
Out[1]: 0
In [2]: prix_musée(False, 21)
Out[2]: 0
In [3]: prix_musée(False, 27)
Out[3]: 5
```

Boucles

Boucle while

1 – On considère une laverie automatique ; dans celle-ci, deux types de machines sont employées : des machines économes capables de laver 5 kg de linge au maximum et d'autres, plus gourmandes, capables de laver 10 kg :

- si le linge fait moins de 5 kg, il faut le mettre dans une machine de 5 kg ;
- s'il fait entre 5 kg et 10 kg, il faut le mettre dans une machine de 10 kg ;
- enfin, s'il fait plus, il faut répartir le linge entre machines de 10 kg et, dès qu'il reste 5 kg ou moins, mettre le linge restant dans une machine 5 kg.

Écrire une fonction `machines` qui prend en paramètre une variable réelle `masse` donnant la masse de linge en kg ; elle renvoie combien de machines de 10 kg et de 5 kg seront nécessaires. Il vous est interdit d'utiliser une division (ce serait plus simple, mais on doit travailler les boucles while là...). On pourra par exemple tester :

```
In [1]: machines(23)
Out[1]: 2, 1
```

2 – La suite de Syracuse est une suite à valeurs dans $\mathbb{N}$ définie par la relation de récurrence suivante :

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

Il existe une conjecture disant que celle-ci atteint toujours 1 à partir d'un certain rang.

Écrire une fonction `syracuse` qui prend en paramètre un entier `u0` et renvoie le rang auquel la suite vaut 1 pour la première fois (on appelle ce rang le temps de vol de la suite). On pourra tester

```
In [1]: syracuse(23)
Out[1]: 15
```

3 – Rappel : le PGCD de deux nombres est un entier qui est le plus grand diviseur que ces deux nombres ont en commun. Ainsi, le PGCD de 427 et 84 est 7, car $427 = 7 \times 61$ et $84 = 7 \times 12$. Pour calculer le PGCD de deux nombres a et b, voici l'algorithme d'Euclide :

- on calcule le reste r de la division euclidienne $a//b$;
- si r est nul, alors b est le PGCD de a et b ;
- sinon, on remplace a par b et b par r, puis on recommence jusqu'à arriver à l condition précédente.

Écrire une fonction `pgcd` qui prend en paramètres deux variables entières `a` et `b` et renvoie leur PGCD selon l'algorithme d'Euclide.

4 – On donne l'algorithme suivant :

```
def algo_mystère(x):
    y = x
    while y % 3 == 0:
        y = y // 3
    return y
```

On exécute `algo_mystère(45)` : expliquer à chaque boucle la valeur de `y`, le nombre de boucles effectuées, et la valeur renvoyée par l'algorithme.

Boucle for

1 – Expliquer ce que fait le programme suivant on précisera à chaque boucle le type et la valeur de chaque variable.

```
def programme_mystère(n):
    somme = 0
    for i in range(10):
        somme += i
    return somme
```

Taper le code dans Spyder pour vérifier la réponse.

2 – Écrire une fonction `affiche_cubes` prenant un paramètre entier n et affichant les cubes des n premiers entiers non nuls. Pour afficher, on utilisera la fonction `print`.

3 – Écrire une fonction `calcule_somme_cubes` prenant un paramètre entier n et calculant et affichant la somme de ces n cubes.

4 – Écrire une fonction `factorielle` prenant un paramètre entier n et calculant $n!$ (on rappelle que $k!$ correspond au produit des entiers strictement positifs inférieurs ou égaux à k).

5 – On définit la suite $(u_n)_{n\in\mathbb{N}}$ par $u_0 = 0$ et $u_{n+1} = \sqrt{2+u_n}$. Écrire une fonction `suite` prenant en paramètres deux entiers i et j et qui affiche les termes de u_i à u_j (inclus).

Listes

Écrire des fonctions qui renvoient une liste contenant :

1 – les noms de 5 personnes ;

2 – les entiers de 1 à 100 ;

3 – les carrés des entiers de 1 à 100 ;

4 – les 100 premiers termes (à partir de $n = 0$) de la suite définie par $u_n = 3n - 1$;

5 – les termes $n = 800$ à $n = 952$ de la suite précédente.

6 – Écrire une fonction `syracuse_2` qui prend en paramètres deux entiers n et u_0 et renvoie une liste contenant les n premiers termes (à partir de $n = 0$) de la suite de Syracuse, avec l'initialisation à u_0.