Notes de cours
Semaine 1

Cours Turing

1 Programmation Python

Cette première partie des notes de cours est consacrée à la programmation en Python. Elle n'a pas pour vocation d'être un tutoriel de Python et présuppose une première expérience avec la programmation. Cette section permet d'établir un vocabulaire commun et de mettre en lumière des mécanismes parfois subtiles dans Python.

1.1 Concepts généraux

En programmation Python et dans la plupart des langages des *impératifs* (comme C, C++, Java, etc.), il y a deux notions centrales et intimement liées : La notion d'*instruction* et la notion d'*expression*.

1.1.1 Instruction

Un programme Python est une suite d'instructions. Lorsqu'on exécute un programme, ses instructions sont effectuées à la suite, l'une après l'autre, jusqu'à la terminaison du programme. On parle d'exécution *séquentielle*. Ci-dessous est un exemple de programme qui présente plusieurs instructions.

```
import random

print("Est-ce que vous êtes chanceux?")
n = random.randint(1, 6)

if n == 6:
    print("Oui, très !")
elif n == 1:
    print("Pas du tout...")
else:
    print("Moyennement, comme tout le monde au final.")
```

Chaque instruction indique à Python ce qu'il doit effectuer à ce moment-là de l'exécution du programme. Par exemple, une instruction peut demander l'import d'un module, le calcul d'une valeur, l'assignation une variable, ou encore l'interruption d'une boucle. Des instructions plus complexes peuvent servir à définir des fonctions ou contrôler le déroulement de l'exécution du

programme. Ce genre d'instruction complexe peut contenir d'autres instructions à l'intérieur. On abordera des différents types d'instructions par la suite dans le cours.

Erreurs L'exécution d'une instruction peut donner lieu à une *erreur*. Il se peut que l'erreur soit due à un fichier manquant, une division par zéro ou encore, par exemple, l'utilisation d'une variable non déclarée. Lorsqu'une erreur se produit, le programme arrête son exécution et un message d'erreur est affiché, généralement dans la console. Le message d'erreur indique le type d'erreur et le numéro de ligne de l'instruction qui a engendré l'erreur. Ces messages d'erreurs sont très utiles pour diagnostiquer et résoudre des problèmes dans votre code.

Parfois, plusieurs lignes sont indiquées, permettant de retracer en partie l'exécution du programme jusqu'au point de l'erreur. On parle de *stack trace*. Nous aurons l'occasion d'étudier ceci plus en détails dans la suite du cours.

À noter que des mécanismes de capture et de gestion des erreurs existent en Python (`try`/`except`), mais nous ne les aborderons pas pour l'instant.

1.1.2 Expression

Les instructions en Python peuvent, selon le type d'instruction, contenir une voire plusieurs expressions. Une expression représente un calcul à effectuer. Par exemple `3 + 4` est une expression Python qui représente l'addition du nombre 3 au nombre 4. Les expressions peuvent être plus complexes et impliquer plusieurs opérations, comme dans l'expression `3 * 5 + 2` ou dans l'expression `fact(n + 1)`.

Littéraux Une valeur simple comme `4` ou encore `"Bouh !"` est considérée comme une expression. On parle d'expression *littérale* ou de *littéral*. Nous verrons par la suite la syntaxe pour les expressions littérales pour différents types de valeurs supportés par Python.

Appels de fonctions Certaines expressions font appels à des fonctions, comme dans `fact(6)` ou encore `print("Bonjour")`. Pour appeler une fonction, on lui applique une séquence d'arguments.

De manière syntaxique, la séquence d'arguments est délimitée par des parenthèses et apparaît directement après l'expression de la fonction (souvent simplement le nom de la fonction). Les différents arguments sont séparés par des virgules. Parfois, il est possible d'appeler une fonction sans arguments. Dans ce cas, les parenthèses sont tout de même nécessaires, comme dans l'exemple `print()`, sans quoi la fonction n'est pas appelée.

Applications d'opérateurs Certaines expressions font usage d'opérateurs tels que `+`, `*` ou encore `and`. Ces opérateurs ont parfois deux opérandes (on parle alors d'opérateurs binaires) ou un seul opérande (on parle alors d'opérateurs unaires). Les opérateurs unaires, comme `not`, sont notés avant l'expression sur laquelle ils portent. Quant aux opérateurs binaires, ils se notent entre les deux opérandes. On parle de notation *infixée*.

Structure d'une expression Les expressions ont une structure en arborescence qui n'est pas forcément évidente à priori, notamment en présence d'opérateurs.

Des parenthèses peuvent être utilisées pour clarifier ou spécifier la structure d'une expression. Hors présence de parenthèses, la structure de l'expression tient compte de la priorité des opérateurs. La priorité des opérateurs en Python suit les conventions généralement utilisées en mathématiques. Par exemple, la multiplication a priorité sur l'addition.

Pour les plus curieux, afin de mettre en évidence la structure d'une expression voire d'un programme, il est possible d'utiliser le module `ast` de Python, comme dans l'exemple suivant :

```
import ast
expr = "3 * 7 + fact(n) - 1"
tree = ast.parse(expr)
print(ast.dump(tree, indent=4))
```

Le résultat obtenu à l'exécution du programme ci-dessus est le suivant :

```
Module(
    body=[
        Expr(
            value=BinOp(
                left=BinOp(
                    left=BinOp(
                        left=Constant(value=3),
                        op=Mult(),
                        right=Constant(value=7)),
                    op=Add(),
                    right=Call(
                        func=Name(id='fact', ctx=Load()),
                        args=[
                            Name(id='n', ctx=Load())],
                        keywords=[])),
                op=Sub(),
                right=Constant(value=1)))],
    type_ignores=[])
```

L'indentation dans la réponse ci-dessus permet de se rendre compte de la structure arborescente de l'expression. Il y a beaucoup à déchiffrer, il s'agit là de la représentation interne utilisée par Python pour représenter des programmes. Nous aurons normalement la possibilité d'approfondir le sujet en fin d'année !

Valeur Une expression peut être calculée pour obtenir une *valeur*. Par exemple, une fois calculée, l'expression `3 + 4` donne comme valeur `7`. Formellement, on parle de l'*évaluation* d'une expression pour référer au processus de calcul de la valeur d'une expression.

Type En Python, chaque valeur a un *type*, qui indique de quel genre de valeur il s'agit. Par exemple, `7` est de type `int`, pour *integer*, ce qui indique qu'il s'agit d'un nombre entier. Le type d'une valeur peut être obtenu en appelant la fonction `type` sur cette valeur, comme dans l'expression `type(1 + 1)`. Nous aborderons plus tard différents types d'expressions et les opérations qui y sont associées.

Évaluation Lorsque Python procède à l'évaluation d'une expression, il commence par évaluer toutes ses sous-expressions, de gauche à droite. Si les sous-expressions sont elles-mêmes formées d'autres sous-expressions, ses sous-expressions sont aussi évaluées, et ainsi de suite. Une fois que chaque sous-expression est évaluée et réduite à une valeur, Python procède à l'exécution de l'opération à la racine de l'expression. [1] On parle d'*évaluation par valeur* ou encore d'*appel par valeur.* [2]

Effets de bord En plus de résulter en une valeur, l'évaluation d'une expression peut donner lieu à des effets que l'on nomme *effets de bord*. Par exemple, l'évaluation d'une expression peut donner lieu à l'affichage d'un texte à la console, à la création d'un fichier ou encore à l'accélération d'un véhicule motorisé. L'ordre d'évaluation d'une expression prend toute son importance en présence d'effets de bord. En effet, l'ordre dans lequel les sous-expressions sont évaluées influence l'ordre dans lequel les effets de bords sont effectués.

1.2 Types et opérations

1.2.1 `int` - Nombres entiers

Une valeur de type `int` est un nombre entier. On dit aussi que le type `int` *représente* les nombres entiers.

Notation pour littéraux Les nombres entiers peuvent être noté directement en base 10, comme par exemple `7`, `897512` ou encore `-79`. Il est aussi possible d'entrer des nombres en base 2 en préfixant la suite de bits par `0b`, comme dans `0b1001` (ce qui équivaut à écrire `9`), ou en base 16 en préfixant la suite de chiffres hexadécimaux par `0x`, comme dans `0x1CAFE` (ce qui équivaut à écrire `117502`).

Opérations De nombreuses opérations sont supportées de base en Python sur les nombres entiers. Les principales sont répertoriées dans le tableau ci-dessous.

`+`	Addition	
`-`	Soustraction	
`*`	Multiplication	
`/`	Division	La division résulte en une valeur de type `float`.
`//`	Division entière	Partie entière du résultat de la division.
`%`	Modulo	Reste de la division entière.
`**`	Exponentiation	

D'autres opérations seront abordées dans la suite de cours, notamment les opérations *bit à bit*.

1. Les opérateurs logique `and` et `or` font exception à cette règle. Si la valeur obtenue pour la sous-expression de gauche permet de déterminer la valeur de l'expression complète, alors la sous-expression de droite n'est pas évaluée. On parle d'opérateur *court-circuité*.

2. D'autres langages peuvent utiliser d'autres stratégie d'évaluation. Par exemple, le langage Haskell utilise une stratégie d'évaluation appelée *évaluation paresseuse* ou encore *appel par nécessité*.

Opérations de comparaison Les valeurs de type `int` peuvent être comparées en elles. On peut, par exemple, calculer si une valeur est égale à une autre valeur ou encore si une valeur est plus grande qu'une autre. Le résultat de ces opérations de comparaison est une valeur de type `bool` qui représente soit *vrai* soit *faux*. Plus de détails sur le type `bool` sont donnés plus loin.

`==`	Égalité
`!=`	Inégalité
`<`	Plus petit
`<=`	Plus petit ou égal
`>`	Plus grand
`>=`	Plus grand ou égal

Notez qu'on utilise deux signes `=` pour former l'opérateur d'égalité en Python. En effet, comme on le verra par après, le symbole `=` est réservé par Python pour dénoter les affectations de variables.

En Python, il est possible d'enchaîner les opérateurs de comparaison de la même manière qu'il est parfois admis en mathématiques, comme par exemple dans `0 < x < 10` ou encore `12 >= x > y == 3`. Dans ce cas, il faut que toutes les comparaisons soient respectées pour que la condition soit remplie.

Domaine de représentation Le type `int` permet de représenter des nombres positifs, négatifs ou nuls. Contrairement à de nombreux langages, la représentation des nombres entiers par défaut en Python, `int`, permet de représenter des nombres arbitrairement grands ou arbitrairement petits.

1.2.2 `float` - Nombres à virgule flottante

Le type `float` représente les nombres à virgule flottante. Les opérations sur le type `float` sont essentiellement les mêmes que sur le type `int`. Les opérations bit à bit ne sont cependant pas supportées.

Notation pour littéraux Pour noter un expression littérale de type `float`, on écrit le nombre en base 10 et on utilise le symbole `.` comme délimitation entre la partie entière et la partie décimale. Par exemple, on peut écrire `3.5` ou `-6.0`. Il est aussi possible d'utiliser la notation scientifique, comme par exemple dans les expressions `1e10` (pour $1 \cdot 10^{10}$) et `5.4e-6` (pour $5,4 \cdot 10^{-6}$).

Domaine de représentation Le type `float` permet de représenter des nombres à virgules positifs, négatifs ou nuls. Contrairement au type `int`, le domaine de représentation de *float* est fini. Cela signifie qu'il n'est pas possible de représenter précisément tous les nombres à virgule en utilisant le type *float*. Les opérations sur les *float* sont donc des approximations des opérations mathématiques et n'en conservent pas toutes les propriétés.

On utilisera donc généralement pas le type *float* lorsque la précision des calculs est requise. Par exemple, dans le secteur financier, on évitera d'utiliser `float` pour représenter des montants en CHF. À la place, on préférera utiliser des `int` et compter des centimes de CHF.

De même, à cause de imprécisions liées aux opérations sur les `float`, il sera assez rare d'utiliser l'égalité entre deux valeurs de type `float`.

1.2.3 `bool` - Booléens

Le type `bool` représente les valeurs *booléennes vrai* et *faux*. Le nom `bool`, tout comme l'adjectif « booléen », font référence au logicien George Boole.

Notation pour littéraux Il n'y a que deux valeurs booléennes : *vrai* et *faux*. On note `True` la valeur *vrai* et `False` la valeur *faux*.

Opérations Les opérateurs ci-dessous s'appliquent sur des valeurs booléennes et donnent comme résultat une valeur elle aussi booléenne.

`and`	Conjonction (et)	La partie droite n'est pas évaluée si la partie de gauche est fausse.
`or`	Disjonction (ou)	La partie droite n'est pas évaluée si la partie de gauche est vraie.
`not`	Négation (non)	

À noter que pour Python les valeurs booléennes sont aussi des `int`. Formellement, on dit que `bool` est une *sous-classe* de `int`. La valeur `True` correspond à `1` et la valeur `False` à `0`. Cela signifie que l'on peut utiliser des booléens dans des opérations arithmétiques ou dans n'importe quel contexte où l'on s'attend à un `int`.

1.2.4 `str` - Chaînes de caractères

Le type `str` représente des chaînes de caractères en Python, c'est-à-dire plus simplement du texte.

Notation pour littéraux Pour noter un texte en Python, on l'entoure soit de double guillemets (`"`) soit de simples guillemets (`'`). On utilise le caractère d'échappement `\` pour entrer des guillemets ou des caractères spéciaux dans une chaîne, comme par exemple `"Cette \"chaine\" d'exemple avec un \\."` qui représente le texte `Cette "chaine" d'exemple avec un \`. Il est possible de rentrer un saut de ligne via le caractère spécial de saut de ligne noté `\n`.

Opérations Il est possible de calculer le nombre de caractères dans un texte en utilisant la fonction `len`, comme dans `len("Bonjour !")`.

De plus, il est possible d'accéder à un caractère de la chaîne via l'opérateur d'indexation, noté avec des crochets [et], comme dans l'expression `chaine[i]`. Entre les crochets est une expression qui indique la position à accéder dans la chaîne. Il doit s'agir d'une valeur entre 0 (pour le premier élément) et la longueur de la liste *moins un* pour le dernier élément. En Python, chaque élément de la chaîne de caractère est lui-même une chaîne (de longueur 1).

L'opérateur `+` sur les chaînes permet de mettre bout à bout deux chaînes de caractères. On parle de *concaténation* de chaînes de caractères.

En plus de ces différentes opérations, il est possible de comparer entre elles des chaînes à l'aide des différents opérateurs de comparaison comme `==`, `<=` ou encore `>`. L'ordre lexicographique (l'ordre utilisé par les dictionnaires) est utilisé pour comparer des chaînes de caractères.

1.2.5 `None`

La valeur spéciale `None` est utilisée en Python pour dénoter l'absence d'une valeur intéressante. C'est la valeur qui est notamment obtenue lorsque l'on évalue une fonction qui ni retourne pas explicitement de résultat, comme par exemple `print`. La valeur `None` est de type `NoneType`.

Pour tester si une valeur est `None`, on utilise généralement l'opérateur `is` plutôt que l'égalité (`==`), comme dans `expr is None` ou `expr is not None`. L'opérateur `is` est l'égalité de *référence* et indique si les deux expressions font référence au même emplacement dans la mémoire, ce qui sera toujours le cas pour deux valeurs `None`[3].

1.2.6 Conversions de types

En Python, il existe des fonctions pour passer d'un type à l'autre par le biais d'une *conversion*. Chaque type a une fonction associée du même nom qui permet de convertir une valeur d'un type différent vers ce premier type. Par exemple, la fonction `int` permet de convertir des `str` ou des `float` en `int`. Lorsqu'une conversion est impossible, Python émettra une erreur lors de l'évaluation de l'appel à la fonction de conversion. Par exemple, l'expression `int("bonjour")` donnera lieu à une erreur lors de son évaluation car le texte `bonjour` n'est pas lisible comme un nombre en base 10.

1.3 Entrées et sorties

Pour communiquer avec un programme, on utilise ses entrées et ses sorties. Il existe de nombreux canaux de communication possibles avec un programme. Dans un premier temps, nous nous contenterons des plus simples avec les fonctions `print` et `input`.

1.3.1 `print`

La fonction `print` permet d'afficher un message dans la console de l'utilisateur. La fonction ne retourne pas de valeur intéressante mais a un effet de bord lors de son évaluation : afficher un message[4].

La fonction `print` prend un nombre arbitraire d'arguments de types quelconques et les affiche sur une ligne à la sortie, en utilisant un unique espace pour séparer les différentes valeurs. Par exemple, l'évaluation de l'expression suivante aura pour effet de bord l'affichage du message `3 petits chats` à l'utilisateur.

```
print(3, "petits", "chats")
```

3. `None` est un *singleton* en Python, ce qui signifie qu'à l'exécution il n'y a qu'une seule valeur de type `NoneType` en mémoire. Si plusieurs usages de `None` sont faits, ils font toujours tous référence à la même valeur `None` en mémoire.

4. On parle de *procédure* pour parler d'une fonction avec effets de bord et généralement sans valeur de retour intéressante, comme c'est le cas pour `print`

1.3.2 `input`

La fonction `input` permet de demander une chaîne de caractères à l'utilisateur par le biais de la console. La fonction a pour effet de bord d'afficher un message, puis d'interrompre l'exécution du programme (au plein milieu de l'évaluation d'une expression) jusqu'à l'entrée par l'utilisateur d'un texte. La fonction a un argument optionnel qui permet de spécifier le message à afficher à l'utilisateur, comme dans l'exemple suivant. Sans cet argument, aucun message n'est affiché.

```
input("Entrez votre prénom : ")
```

Une fois le message rentré par l'utilisateur dans la console, l'exécution du programme reprend avec la suite de l'évaluation de l'expression courante. Dans le contexte de cette évaluation, la chaîne de caractères entrée par l'utilisateur est utilisée comme valeur pour l'appel à la fonction `input`.

Notez qu'il n'est pas rare d'appeler une fonction de conversion (comme par exemple `int`) directement sur le résultat de `input` afin d'obtenir, par exemple, un nombre de la part de l'utilisateur.

1.4 Variables

Les variables sont un moyen en Python de stocker des valeurs dans la mémoire vive assignée au programme. Une variable est identifiée par son nom. Chaque variable a une *portée*, qui indique dans quelles parties du programme la variable est visible. Nous parlerons plus en détails du concept de portée quand nous aborderons les fonctions.

1.4.1 Assigner une variable

L'assignation de variable est une instruction en Python. On écrit le nom de la variable, suivi du sigle `=`, suivi d'une expression. Lorsque Python exécute cette instruction, il commence par évaluer la partie de droite du sigle égal. Une fois la valeur calculée, elle est stockée en mémoire et sera accessible via le nom spécifié à gauche du sigle égal. À noter qu'il s'agit bien de la valeur qui est stockée, et non de l'expression elle-même. Ainsi, l'expression est évaluée lors de l'assignation, et non lors de l'utilisation de la variable.

1.4.2 Utiliser une variable

Il est possible d'utiliser une variable au sein d'une expression simplement en notant son nom. Par exemple, l'expression `x + 2` fait usage de la variable `x`. La valeur d'une variable est la dernière valeur assignée à cette variable. Si, au moment de l'évaluation, la variable n'existe pas ou n'a pas été encore assignée, une erreur sera levée par Python.

1.4.3 Réassignation de variable

Les variables peuvent voir leur valeur changer au fil du temps. En effet, il est possible d'exécuter une instruction d'assignation de variable même si la variable a déjà une valeur. Dans ce cas, la valeur précédemment stockée en mémoire n'est plus accessible au travers de la variable. Seule la nouvelle valeur est conservée.

Dans certains cas, on utilisera la valeur courante d'une variable dans le cadre d'une instruction d'assignation de la même variable, comme dans l'instruction `x = 3 * x + 1`. Cette utilisation est tout à fait valide en Python, et même relativement courante.

On parle d'*incrémentation* d'une variable lorsque l'on ajoute une valeur à une variable, comme dans l'instruction `x = x + 1`. Ce genre d'opérations étant relativement fréquente, Python permet d'utiliser une syntaxe plus concise : `x += 1`. De même, on peut utiliser l'instruction `x -= 1` à la place de l'instruction `x = x - 1`. On parle dans ce cas de *décrémentation*.

De nombreux autres opérateurs Python, comme `*`, `/` et autres peuvent être utilisé de la même manière. Dans le jargon de Python, on parle d'*opérateurs d'assignation augmentés*.

1.4.4 Échange de variables

Parfois, on cherchera à échanger les valeurs contenues dans deux ou plusieurs variables. Pour ce faire, dans de nombreux langages, il faut faire usage de plusieurs instructions d'assignation et de variables intermédiaires afin de stocker une partie des valeurs autrement perdues.

En Python, on utilisera plutôt la syntaxe `x, y = y, x` qui fait usage de ce qu'on appelle les *tuples*, un sujet que l'on abordera dans la suite du cours.

1.5 Instructions conditionnelles

Les instructions conditionnelles permettent d'exécuter conditionnellement certaines instructions. En Python, on utilise le mot clé `if` pour écrire une instruction conditionnelle, comme dans l'exemple ci-dessous.

```
if x > 3:
    print("x est plus grand que 3")
    print("Chouette, non ?")
```

Juste après le mot clé `if` est une expression. On appelle cette expression la *condition*. La condition dans l'exemple est `x > 3`.

En plus d'une condition, une instruction conditionnelle a aussi un *corps*. Le corps d'une instruction conditionnelle est un bloc de code, une suite d'instructions. Dans l'exemple ci-dessus, les deux instructions situées en dessous de cette première ligne forment le *corps* de l'instruction conditionnelle.

Les instructions du corps d'une instruction conditionnelle sont décalées sur la droite de 4 espaces, ce que l'on appelle une *indentation*. On dit que le bloc de code est *indenté*. Une éventuelle ligne non indentée qui suivrait ne serait pas considérée comme faisant partie du corps de l'instruction conditionnelle, mais comme une instruction à part, à la suite de l'instruction conditionnelle. En Python, l'indentation est utilisée pour délimiter les blocs de code, alors que généralement les autres langages de programmation utilisent d'autres éléments de syntaxe, comme par exemple des paires d'accolades, pour délimiter les blocs.

Lors de l'exécution, lorsqu'une instruction conditionnelle est rencontrée, Python commence par évaluer la condition. Si la condition est *vraie* (`True` ou d'autre valeurs considérées comme *vraies*), alors le corps de l'instruction conditionnelle est exécuté. Sinon, l'exécution de l'instruction conditionnelle termine et l'exécution continue normalement avec l'éventuelle instruction suivante.

1.5.1 `elif`

Une instruction conditionnelle peut être augmentée d'une ou plusieurs autres conditions, chacune accompagnée d'un bloc de code, via le mot clé `elif`. Considérez par exemple l'instruction conditionnelle suivante.

```
if x > 3:
    print("x est plus grand que 3")
    print("Chouette, non ?")
elif x > 0:
    print("x est positif")
    print("C'est déjà ça !")
elif y > 0:
    print("x est négatif ou nul, bouh !")
    print("y par contre est positif")
```

Dans ce cas, lors de l'exécution, les conditions sont évaluées dans l'ordre, une à une et de haut en bas, jusqu'à tomber sur une condition qui est *vraie*. Dans ce cas, le bloc de code correspondant à la condition *vraie*, et uniquement ce bloc, est exécuté. Après l'exécution de ce bloc, l'exécution de l'instruction est considérée comme terminée et l'exécution du programme reprend à l'instruction suivant l'instruction conditionnelle, si une telle instruction existe.

À noter que les conditions qui apparaissent après une condition satisfaite ne sont pas évaluées. Dans le cas où aucune condition n'est satisfaite, alors l'exécution de l'instruction conditionnelle termine et le programme passe simplement à l'instruction suivante.

1.5.2 `else`

Les instructions conditionnelles peuvent être augmentée d'un unique bloc `else` à leur toute fin, comme dans l'exemple ci-dessous.

```
if x > 3:
    print("x est plus grand que 3")
    print("Chouette, non ?")
elif x > 0:
    print("x est positif")
    print("C'est déjà ça !")
elif y > 0:
    print("x est négatif ou nul, bouh !")
    print("y par contre est positif")
else:
    print("J'abandonne, y a vraiment rien qui va !")
```

Comme on peut l'observer dans l'exemple ci-dessus, le mot de clé `else` peut être utilisé en conjonction avec `elif`. Dans ce cas, le bloc du `else` est simplement mis en dernier.

Lors de l'exécution d'une instruction conditionnelle avec un bloc `else`, si la condition est fausse et que toutes les éventuelles conditions supplémentaires apportées via `elif` le sont aussi, alors le bloc de code précédé de `else` est exécuté. Si une des condition se trouvait être vraie, alors ce bloc de code ne serait pas exécuté.

1.6 Boucles

1.7 `while`

La boucle `while` permet d'exécuter un bloc de code tant qu'une condition est remplie. Syntaxiquement, une boucle `while` est composée d'une condition (une expression) et d'un corps (un bloc de code), comme le montre l'exemple ci-dessous.

```
while input("Laissez vide svp: ") != "":
    print("Merci de laisser vide.")
    print("Non mais...")
```

Lors de l'exécution, lorsque Python rencontre une instruction `while`, Python commence par évaluer la condition. Si la condition est fausse, alors l'exécution de l'instruction se termine et Python passe à l'éventuelle instruction suivante. Dans le cas contraire, si la condition est vraie, alors le bloc de code est exécuté. À la fin de l'exécution du bloc de code, la condition est à nouveau évaluée, ce qui peut donner lieu à une nouvelle valeur. À nouveau, si la condition est fausse, alors l'instruction se termine. Au contraire, si l'instruction est vraie, alors le même processus recommence, alternant entre exécution du corps de la boucle et évaluation de la condition, et ce jusqu'à ce que la condition soit fausse (ou que la boucle soit interrompue, comme on le verra dans quelques instants). À noter qu'il se peut qu'une boucle soit exécutée sans fin. On parle dans ce cas de *boucle infinie*. La plupart du temps, la condition d'une boucle dépendra d'un état qui change au fur et à mesure de l'exécution de la boucle et finira par être fausse.

On appelle *itération* les différentes exécutions successives du corps d'une boucle. Ainsi, on parle de la première itération pour faire référence à la première fois que le corps de la boucle est exécuté. De même, la dernière itération d'une boucle fait référence à la dernière exécution du corps de la boucle.

`break` L'instruction `break` permet de sortir d'une boucle. Lors de l'exécution d'une boucle, lorsque Python rencontre l'instruction `break`, il arrête immédiatement l'exécution de la boucle et passe à la suite du programme. Dans le cas de boucles *imbriquées* (une boucle dans le corps d'une autre boucle), seule la boucle la plus interne est interrompue.

`continue` L'instruction `continue` permet de terminer l'exécution du corps de la boucle et de passer directement à l'évaluation de la condition de la boucle puis éventuellement à l'itération suivante si la condition est à nouveau vérifiée. Comme pour `break`, dans le cas de boucles imbriquées, seule la boucle la plus interne est affectée par `continue`.

1.8 `for`

Le mot clé `for` permet aussi de définir des boucles. Les boucles `for` servent à exécuter un bloc de code pour toutes les valeurs d'une *séquence* de valeurs. Syntaxiquement, les boucles `for` s'écrivent comme dans l'exemple suivant :

```
for c in "YMCA":
    print(c)
```

Le mot clé `for` est suivit d'un nom de variable, puis du mot clé `in`, puis d'une expression pour calculer la séquence de valeurs. Cette ligne, terminée par le symbole `:`, est ensuite suivie d'un bloc de code indenté – le corps de la boucle.

Lors de l'exécution, lorsqu'une boucle `for` est rencontrée, Python commence par évaluer l'expression qui calcule la séquence de valeurs. Contrairement aux boucles `while`, cette expression sera évaluée une unique fois. Ensuite, Python procède à l'exécution du corps de la boucle, exactement une fois pour chaque élément de la séquence de valeur précédemment obtenue. Avant chaque itération, Python procède automatiquement à l'assignation de la valeur suivante de la séquence à la variable de la boucle (`c` dans l'exemple). Ainsi, la variable prendra successivement les différentes valeurs contenue dans la séquence.

1.8.1 `break` et `continue`

Les instructions `break` et `continue` peuvent elles aussi être utilisées au sein du corps d'une boucle `for`.

1.9 Séquences en Python

Les boucles `for` permettent de parcourir les éléments d'une séquence. En Python, de nombreux types, comme les chaînes de caractères (`str`), sont des séquences. Nous aurons l'occasion d'aborder d'autres types de séquences dans la suite de ce cours. Pour aujourd'hui, nous aborderons encore simplement la fonction `range` qui permet de retourner des *intervalles* de valeurs.

1.9.1 range

La fonction `range` permet d'obtenir une séquence de valeurs comprises dans un intervalle. Cette fonction est souvent utilisée de pair avec `for`, comme dans l'exemple ci-dessous.

```
for i in range(1, 100):
    print(i * i)
```

La fonction `range` peut prendre un, deux ou trois arguments.

1. Lorsqu'elle reçoit un unique argument, la fonction retourne la séquence des valeurs entières comprises entre 0 (inclus) et l'argument (non inclus).
2. Lorsqu'elle reçoit deux arguments, comme dans l'exemple, la fonction retourne la séquence des valeurs entières comprises entre le premier argument (inclus) et le second argument (non inclus).
3. Lorsqu'elle reçoit trois arguments, la fonction retourne la séquence des valeurs entières comprises entre le premier argument (inclus) et le deuxième argument (non inclus). Le troisième argument sert à spécifier l'écart entre les éléments, par défaut `1`.

2 Complexité temporelle

Souvent, il sera intéressant d'étudier le temps que prend un programme, une fonction, pour s'exécuter et retourner son résultat. Le but sera généralement de concevoir des programmes qui

apportent des réponses rapidement, que ce soit pour des questions de confort d'utilisation ou pour des raisons d'utilisabilité. Par exemple, avoir un programme qui permet de décoder les communications adverses est beaucoup plus utile s'il donne ses réponses rapidement plutôt que des années après la fin d'une guerre.

Dans cette section, nous allons voir quelques outils pour nous permettre d'étudier le temps d'exécution de programmes de manière purement syntaxique, c'est-à-dire simplement en étudiant le code du programme et, de manière importante, sans devoir l'exécuter.

Calculer exactement le temps d'exécution d'un programme est une tâche très difficile. Hors les barrières théoriques, il faudrait tenir compte de tous les détails du matériel sur lequel le programme est exécuté. C'est pourquoi on abordera ce problème de manière plus abstraite et, dans un sens, moins précis.

Le temps d'exécution d'un programme ou simplement d'une fonction dépendra généralement des valeurs qui lui sont données. Il sera intéressant de voir comment le temps d'exécution d'un programme évolue lorsque l'on fournit des données plus grandes (ou plus importantes selon une autre mesure). Pour commencer par un exemple simple, considérez la fonction ci-dessous.

```
def repeter(n):
    for i in range(1, n):
        print("!!!")
```

La fonction définie ci-dessus permet d'afficher `n` lignes contenant le texte `!!!` à la console. Le temps d'exécution de cette fonction grandit avec `n`. Plus `n` est grand, plus le temps d'exécution sera grand. Plus précisément, on s'attend à ce que le temps d'exécution dépende *linéairement* de `n`. Lorsque `n` double, on s'attend à ce que le temps d'exécution de `repeter(n)` double approximativement aussi.

Cette correspondance linéaire n'est pas observée dans le cas de tous les programmes ou de toutes les fonctions. Par exemple, on observe une autre relation avec la fonction suivante.

```
def repeter2(n):
    for i in range(1, n):
        for j in range(1, n):
            print("!!!")
```

Dans ce cas, on s'attend à ce qu'en utiliseant un nombre `n` deux fois plus grand le temps d'exécution soit multiplié par quatre ! Formellement, on dit que la complexité temporelle de `repeter2` est *quadratique* en `n`. En effet, la fonction effectue $\texttt{n}^2$ appels à `print`.

Pour la fonction suivante, la complexité temporelle est encore différente.

```
def repeter3(n):
    for i in range(1, n):
        for j in range(1, n):
            for k in range(1, n):
                print("!!!")
```

Dans ce cas, on parle de complexité *cubique*. Lorsqu'on utilise un nombre deux fois plus grand en entrée, le temps d'exécution est multiplié par 8 (c'est-à-dire 2^3). Le temps d'exécution de cette fonction *ressemble* à $\texttt{n}^3$.

Nous aurons l'occasion d'aborder cette notion de complexité temporelle dans plus de détails dans la suite du cours.