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 dits *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'affectation d'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 les 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 `*`. 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 `-` dans l'expression `-x`, 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.

Contenu avancé

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.

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 modification de la mémoire du programme, 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.

Exemple Comme exemple pour illustrer les deux derniers points, considérez le programme suivant, qui définit une fonction `double` un peu bavarde. Un appel a la fonction double avec un nombre en argument aura pour résultat le double de ce nombre. Ainsi, l'expression `double(4)` aura pour valeur 8. En plus de cela, l'appel aura pour effet de bord l'affichage d'un texte en sortie dans le console. Dans le cas de l'expression `double(4)`, le texte `Je dois doubler 4` sera affiché.

```
def double(n):
    print("Je dois doubler", n)
    return 2 * n

print("Le premier résultat est", double(3))
print("Puis viennent", double(double(5)), "et", double(7))
```

Le résultat de l'évaluation de ce programme est donc, en suivant l'ordre d'évaluation indiqué plus haut :

```
Je dois doubler 3
Le premier résultat est 6
Je dois doubler 5
```

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*.

```
Je dois doubler 10
Je dois doubler 7
Puis viennent 20 et 14
```

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 comparaisons et les opérations *bit à bit*.

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 entiers 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.

1.2.3 `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.

Il est aussi possible de multiplier une chaîne de caractère par un nombre entier, ce qui permet de faire répéter une chaîne un nombre donné de fois.

```
print("Hello !\n" * 100)
```

1.2.4 `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`.

1.2.5 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 Sortie d'un programme

Très souvent, un programme devra communiquer des informations à l'utilisateur. Il existe de nombreux canaux de communication possibles avec un programme, et nous en aborderons quelques-un dans ce module. Dans un premier temps, nous nous contenterons d'une fonction assez rudimentaire : la fonction `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 [3].

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")
```

1.4 Variables

Les variables sont un moyen en Python de stocker des valeurs dans la mémoire vive affectée 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 Affecter une variable

L'affectation de variable est une instruction en Python. On écrit le nom de la variable, suivi du sigle `=`, suivi d'une expression, comme dans :

```
n = 6 * 7
```

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. L'expression est évaluée lors de l'affectation, et non lors de l'utilisation de la variable. En plus de la question de la performance, cette distinction prend toute son importance lorsque le calcul de l'expression a des effets de bord. Le calcul n'est effectué qu'une unique fois, au moment de l'affectation.

3. 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`

Ainsi, dans l'exemple suivant, un seul affichage du texte `Je double 11` sera effectué, et ce malgré que la variable est utilisée à plusieurs reprises :

```
def double(n):
    print("Je double", n)
    return 2 * n

x = double(11)
print(x + x + x)
```

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 affectée à cette variable. Si, au moment de l'évaluation, la variable n'existe pas ou n'a pas été encore affectée, une erreur sera levée par Python.

1.4.3 Réaffectation de variable

Les variables peuvent voir leur valeur changer au fil du temps. En effet, il est possible d'exécuter une instruction d'affectation 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'affectation 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'affectation 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'affectation 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 Les fonctions

Les fonctions sont un outil essentiel en programmation. Cet outil permet de structurer les programmes et permet de réutiliser du code. Dans cette première partie, nous allons nous intéresser aux fonctions en Python : comment les appeler et comment les définir.

1.5.1 Appel de fonction

Pour faire un appel de fonction, on fait suivre le nom de la fonction à appeler par une paire de parenthèses. Entre les parenthèses sont spécifiés, dans l'ordre, les différents *arguments* de la fonction. Chaque argument est une expression.

```
print(3 * 4, 6 * 7)
```

Lors de l'évaluation d'un appel de fonction, la valeur de l'expression qui indique quelle fonction appeler est calculée en premier. La plupart du temps, cette expression est simplement le nom de la fonction (comme `print` dans le code ci-dessus), mais il se peut aussi que la fonction à appeler soit le résultat d'une expression plus complexe, comme nous pourrons parfois le voir à l'avenir.

Une fois que l'expression qui indique quelle fonction appeler a été évaluée, les arguments sont calculés, un à un et de gauche à droite. Une fois toutes ces expressions évaluées, l'appel à la fonction est finalement effectué. Lors de cet appel, les instructions spécifiées par la fonction sont exécutées. Une fois ces instructions effectuées, l'appel de fonction est à son tour réduit à une valeur, qui dépend de la valeur éventuellement retournée lors de l'exécution des instructions de la fonction. Nous verrons sous peu comment spécifier quelles instructions doivent être effectuées lors d'un appel de fonction et comment donner une valeur de retour à la fonction.

1.5.2 Définition de fonction

Dans la section précédente, nous avons vu comment appeler des fonctions et les règles qui régissent l'ordre d'évaluation lors d'un appel de fonction. Dans cette section, nous allons examiner comment définir des fonctions.

Syntaxe Pour définir une fonction en Python, on utilise le mot-clé `def`. Sur la même ligne que ce mot-clé, on inscrit le nom de la fonction suivit de ses *paramètres* entre parenthèses. Finalement, on termine la ligne par un symbole `:` (deux points). En-dessous de cette ligne et indentées d'un cran sur la droite se trouvent les instructions de la fonction. On appelle ces instructions le *corps* de la fonction.

```
def dire_bonjour(nom, age):
    if age > 20:
        print("Bonjour", nom, ".")
        print("J'espère que vous passez une belle journée.")
    else:
        print("Salut", nom, "!")
        print("Ça va bien ou quoi ?")
```

Remarque Formellement, une définition de fonction est une instruction. Une définition de fonction peut se situer à n'importe quel endroit où une instruction est attendue. Il est donc possible de définir des fonctions au sein de boucles ou d'autres fonctions, par exemple, bien que par défaut on préfèrera définir les fonctions au niveau principal du programme (le niveau sans indentation) si possible.

Exécution de la définition Lorsqu'une instruction de définition de fonction est exécutée, la fonction est enregistrée en mémoire sous le nom spécifié. Il s'agit exactement du même procédé que lors d'une *affectation de variable*. Le nom de la fonction joue le rôle de nom de variable. La valeur stockée dans la variable est la *fonction elle-même*. En Python, les fonctions sont des valeurs.

Étant donnée la définition de la fonction `dire_bonjour` plus haut, l'instruction suivante permet d'afficher le type de la valeur stockée dans la variable `dire_bonjour`. En l'occurence, il s'agira du type *function*.

```
print(type(dire_bonjour))  # Affiche <class 'function'>
```

Comme les fonctions sont des valeurs en Python, est possible de les manipuler autrement qu'en les appelant directement. Il est par exemple possible de passer des fonctions comme arguments à d'autres fonctions, ou même de retourner des fonctions depuis d'autres fonctions.

1.5.3 Arguments, paramètres et exécution d'un appel de fonction

Lorsqu'un appel de fonction est évalué, le corps de la fonction est exécuté. Avant cela, une valeur est affectée à chaque paramètre de la fonction. La valeur affectée à un paramètre est la valeur de l'argument correspondant. Lors d'appels différents, les paramètres peuvent se voir affecter des valeurs différentes.

```
dire_bonjour("Albert", 15)
dire_bonjour("Beatrice", 52)
```

Dans l'exemple ci-dessus, deux appels sont faits à la fonction `dire_bonjour`. Dans le cas du premier appel, le paramètre `nom` se verra attribué la valeur `"Albert"` et `age` la valeur `15` avant l'exécution du corps de la fonction. Dans le cas du second appel, le paramètre `nom` aura comme valeur `"Beatrice"` et `age` la valeur `52` avant la nouvelle exécution du corps de la fonction.

1.5.4 Valeur de retour

Dans le corps d'une fonction, il est possible d'utiliser l'instruction `return` pour indiquer la valeur de retour d'une fonction. Après le mot clé `return` est indiquée une expression. Le résultat de l'évaluation de cette expression est utilisé comme valeur de retour de la fonction. La valeur de retour est la valeur qui est utilisée comme résultat de l'évaluation de l'appel de la fonction.

```
def additionner(a, b):
    return a + b
```

L'instruction `return`, lorsqu'elle est exécutée, met fin immédiatement à l'exécution du corps de la fonction. Toutes les instructions suivantes du corps de la fonction sont ignorées. Par exemple, dans le code ci-dessous, l'instruction d'appel à la fonction `print`, de même que l'instruction `return n`, ne seront jamais exécutées.

```
def double(n):
    return 2 * n
    print("Hahahaha !")
    return n
```

```
# A pour valeur 10, n'affiche rien.
double(5)
```

Notez au passage qu'il est possible d'utiliser plusieurs instructions `return` au sein d'une même fonction. La première à être exécutée donne la valeur de retour de la fonction et met fin immédiatement à l'exécution du corps de la fonction.

1.5.5 Portée des variables

Les paramètres d'une fonction, tout comme les variables affectées dans le corps de la fonction, sont visibles uniquement à l'intérieur de la fonction. On dit qu'elles ont une *portée locale*. Ces variables ne sont visibles qu'au sein de la fonction et non à l'extérieur du corps de la fonction.

```
def verifier_mot_de_passe(mot):
    if mot != "1234":    # Vérifie si le mot de passe est
        return False     # différent de 1234
    print("Mot de passe correct !")
    return True

verifier_mot_de_passe("hello")

# La variable mot n'est pas visible ici.
# L'instruction suivante donnera lieu à une erreur.
print(mot)
```

Toute variable du même nom que la variable locale se trouvant dans le contexte de la définition de la fonction sera cachée par cette variable locale. Les deux variables, bien qu'ayant le même nom, ne sont pas autrement liées. La modification d'une variable n'affectera pas l'autre variable. Voyez par exemple le code suivant qui met en avant ce phénomène.

```
x = 1
y = 1

def foo(x):
    # On ne peut pas lire le x et le y du contexte global ici.
    # Les deux variables sont cachées.
    y = 2

foo(2)

print(x)  # Affiche 1, le x du contexte global n'est pas modifié.
print(y)  # Affiche 1, le y du contexte global n'est pas modifié.
```

Si une variable est uniquement lue et jamais affectée dans le corps d'une fonction, alors la variable ne sera pas considérée comme locale mais sera à la place prise hors du contexte de la fonction, comme dans l'exemple suivant.

```
x = 1

def foo():
    print(x)

foo()  # Affiche 1

x = 2
foo()  # Affiche 2
```

1.5.6 `global` et `nonlocal`

Les mots-clés `global` et `nonlocal` permettent, au sein d'une fonction, de contourner les règles exprimées plus haut et de quand même modifier une variable définie hors du corps de la fonction. Leur usage est assez rare. Dans un soucis d'exhaustivité, ces mots-clés sont évoqués ici.

Le mot-clé `global` s'utilise pour accéder à une variable définie dans le contexte global, comme dans l'exemple ci-dessous.

```
x = 1

def foo():
    global x
    x += 1

print(x)  # Affiche 1
foo()
print(x)  # Affiche 2
foo()
print(x)  # Affiche 3
```

Contenu avancé

Le mot-clé `nonlocal` est plus technique. Il permet de modifier, au sein d'une fonction, une variable définie au sein d'une autre fonction dont le corps contiendrait la définition de la première fonction. Voici un exemple.

```
def foo():
    x = 1
    def bar():
        nonlocal x
        print(x)
        x += 1
    return bar

f1 = foo()
f1()  # Affiche 1
f1()  # Affiche 2

f2 = foo()
f2()  # Affiche 1
f2()  # Affiche 2
f2()  # Affiche 3

f1()  # Affiche 3
```

Notes de cours
Semaine 2

Cours Turing

1 Programmation Python

Pour cette semaine, nous allons continuer notre exploration du langage Python et aborder les concepts de *valeurs booléennes*, d'*expressions conditionnelles* et de *boucles*.

1.1 Le type `bool`

Jusqu'à présent nous avons vu que les expressions Python, lorsque calculées, donne lieu à un résultat. Ces résultats, appelés *valeurs* dans le jargon de Python, peuvent être classifiés par *type*. Nous avons abordé déjà différents types : `int` pour les nombres entiers, `float` pour les nombres à virgules, `str` pour les chaînes de caractères ou encore `function` pour les fonctions. Le type `bool` est au autre type de valeurs, qui représente les valeurs *vrai* et *faux*. Ces valeurs sont appelées des valeurs *booléennes* ou parfois simplement *booléens*. 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)	Les deux côtés doivent être vrais pour que le résultat soit vrai.
`or`	Disjonction (ou)	Un côté à vrai suffit pour que le résultat soit vrai.
`not`	Négation (non)	Le résultat est vrai si et seulement si l'opérande est fausse.

Le comportement des différents opérateurs booléens est récapitulé dans le programme d'exemple ci-dessous.

```
print(False and False)  # Affiche False
print(False and True)  # Affiche False
print(True and False)  # Affiche False
print(True and True)  # Affiche True
```

```
print(False or False)  # Affiche False
print(False or True)  # Affiche True
print(True or False)  # Affiche True
print(True or True)  # Affiche True

print(not False)  # Affiche True
print(not True)  # Affiche False
```

Remarque À noter que la partie de droite d'un `and` ou d'un `or` n'est pas évaluée (calculée) lorsque la partie de gauche suffit pour déterminer le résultat. Ainsi, dans l'exemple suivant, l'expression ne donnera pas d'erreur de division par 0 même si la variable `n` a pour valeur 0.

```
print(n > 0 and 1 / n < 0.05)
```

Contenu avancé

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 Opérations de comparaison

Les valeurs de type `int`, `float` et `str` abordées au dernier cours peuvent être comparées entre 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

```
print(4 == 5)  # Affiche False
print(4 == 4)  # Affiche True

print("Hello" == "Hell")  # Affiche False
print("Hello" == "Hello")  # Affiche True

print(4 != 5)  # Affiche True
print("Hello" != "Hello")  # Affiche False
```

```
print(6 > 4)   # Affiche True
print(6 > 8)   # Affiche False
print(6 > 6)   # Affiche False
print(6 >= 6)  # Affiche True
```

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

Contenu avancé

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.

1.3 Instructions conditionnelles

La semaine dernière, nous avions abordé un certain nombre de différentes *instructions* qu'il est possible de donner à Python :

Expression Effectuer le calcul d'une expression.

Affectation de variable Effectuer le calcul d'une expression et stocker le résultat (la valeur) dans une *variable*.

Définition de fonction Définir une fonction et la stocker dans une variable.

Ces quelques instructions nous ont déjà permis de réaliser des programmes intéressants. Cette semaine, nous allons aborder deux nouvelles familles d'instructions, à savoir :

1. Les instructions conditionnelles.
2. Les boucles.

Commençons par les instructions conditionnelles. Les instructions conditionnelles permettent d'exécuter conditionnellement certaines instructions en fonction de la valeur de certaines expressions (appelées *conditions*). 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` vient 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*, comme pour le corps d'une définition de fonction.

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. [1]

Lors de l'exécution, lorsqu'une instruction conditionnelle est rencontrée, Python commence par évaluer (calculer la valeur de) 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.

1.3.1 `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.

1.3.2 `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")
```

1. 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.

```
    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 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.

1.4 Boucles `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. [2]

```
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. Par exemple, il est fréquent d'avoir une condition qui dépend d'une variable qui est modifiée dans le corps de la boucle, comme dans le programme suivant qui affiche les chiffres de 0 à 9 :

```
x = 0
while x < 10:
    print(x)
    x = x + 1
```

Nous verrons la semaine prochaine que pour ce genre d'utilisations une autre famille de boucles est privilégiée : les boucles `for`.

2. La fonction `input` permet de demander à l'utilisateur d'entrer une chaîne de caractères. La chaîne entrée est ensuite utilisée comme valeur de retour de la fonction.

Remarque 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.

Contenu avancé

Il existe deux instructions qui permettent de gérer plus finement le contrôle de l'exécution d'une boucle, à savoir les instructions `break` et `continue`.

`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`.

Lorsque raisonnable, il est conseillé de ne pas recourir à ces instructions car elle peuvent complexifier la compréhension du comportement d'une boucle.

Notes de cours
Semaine 3

Cours Turing

1 Les séquences en Python

Ces premières semaines de cours, nous avons vu comment concevoir des programmes plus ou moins simples en Python. Les programmes que vous avez conçus jusqu'à présent travaillaient sur des valeurs relativement simples, comme par exemple des nombres entiers ou encore des formes (`Graphic` dans PyTamaro). C'est en manipulant ces différentes valeurs que vos programmes réalisent les différentes tâches qu'on veut leur faire accomplir. Cette semaine, nous allons enrichir nos connaissances en Python en introduisant d'autres types de valeurs très intéressants, les *séquences*.

En termes très généraux, une séquence est simplement une valeur qui contient d'autres valeurs, appelées les *éléments* de la séquence.

```
ma_sequence = (1, 2, 3, 4, 5)  # Un tuple de cinq éléments
print(ma_sequence)  # Affiche (1, 2, 3, 4, 5)
print(len(ma_sequence))  # Affiche 5
```

Dans une séquence, les éléments sont ordonnés, c'est-à-dire qu'ils sont rangés dans un certain ordre. Une séquence peut contenir zéro, un ou plusieurs éléments. Comme on le verra, il sera possible d'effectuer différentes opérations sur les séquences, comme par exemple compter le nombre d'éléments qu'elles contiennent ou encore accéder à un élément à une certaine position dans la séquence.

Prenons un exemple concret avec un type de séquence particulièrement utile en Python, et déjà abordé dans les semaines précédentes : les *chaînes de caractères*. Une chaîne de caractères représente une séquence ordonnée de valeurs : des caractères.

```
ma_chaine = "Bonjour"
print(len(ma_chaine))  # Affiche 7
print(ma_chaine[0])  # Affiche 'B'
```

Il existe plusieurs types de séquences en Python, chacun ayant ses propres caractéristiques et les opérations qui peuvent être effectuées sur ces séquences. Dans ces notes de cours, nous allons aborder trois types de séquences très utilisés en Python :

1. les *chaînes de caractères*,
2. les *tuples*,
3. les *listes*.

1.1 Chaînes de caractères

Une chaîne de caractères est une séquence ordonnée de caractères. Nous avons déjà vu comment former une chaîne de caractères en Python, en entourant une suite de caractères par des guillemets simples ou doubles.

```
"Bonjour"
'Bonjour'
```

Longueur On appelle la longueur d'une chaîne de caractères le nombre de caractères qu'elle contient. La longueur d'une chaîne de caractères est accessible via la fonction `len`.

```
len("Bonjour")  # A pour valeur 7
```

Accès aux caractères Pour accéder directement à un caractère d'une chaîne de caractères, on indique la position du caractère entre deux crochets ([et]).

```
"Bonjour"[0]  # A pour valeur 'B'
```

La position du premier caractère est 0, celle du deuxième caractère 1, et ainsi de suite. Comme l'on commence par la position 0, le dernier caractère d'une chaîne de longueur n a pour position $n - 1$.

Il est aussi possible d'indiquer une position négative. Dans ce cas, les positions sont comptées depuis la fin, -1 étant le dernier caractère, -2 l'avant-dernier, etc.

```
lettres = "ABC"
lettres[-1]  # A pour valeur 'C'
lettres[-2]  # A pour valeur 'B'
```

Notez aussi que la position à accéder peut aussi être spécifiée par une expression plus complexe qu'un nombre entier littéral.

```
k = 2
"ABC"[1 + k - 3]  # A pour valeur 'A'
```

Lorsque l'on tente d'accéder à une position qui n'existe pas dans la chaîne, Python lance une erreur de type `IndexError`.

Notez que Python ne possède pas de type caractère distinct du type chaîne de caractères. En Python, un caractère est simplement une chaîne de caractères de longueur 1.

```
"Bonjour"[6]  # A pour valeur 'r'
len("Bonjour"[6])  # A pour valeur 1
"Bonjour"[6][0]  # A aussi pour valeur 'r'
```

Sous-parties La même syntaxe que pour l'accès à un caractère permet de référer à une sous-chaîne de la chaîne. Dans ce cas, on utilisera non pas une seule position, mais deux positions séparées par un symbole deux-points (:).

```
"ABCDE"[2:4]  # A pour valeur "CD"
```

Le premier nombre indique la position de début de la sous-partie. La valeur à cette position est incluse dans le résultat. Le deuxième nombre indique la position de fin de la sous-partie. Contrairement à la position de début, la valeur à la position de fin n'est pas incluse. Les positions de début et de fin peuvent être omises. Lorsque la valeur de début est omise, la sous-partie commence au début de la chaîne. Lorsque la valeur de fin est omise, la sous-partie termine à la fin de la chaîne.

Parcours des caractères Grâce à la fonction `len` et à la possibilité d'accéder à des caractères via leur position, il est possible, à l'aide d'une boucle `while`, de passer en revue tous les caractères d'une chaîne et ainsi d'exécuter des instructions pour chaque caractère.

```
mot = "Python"
i = 0
while i < len(mot):
    print(mot[i])
    i += 1
```

Nous verrons aussi plus loin dans ces notes de cours qu'il est possible d'utiliser une boucle `for` pour parcourir les caractères d'une chaîne de caractères.

Concaténation L'opérateur `+` permet de *concaténer* deux chaînes de caractères pour former une unique chaîne. La chaîne résultante de la concaténation contient les caractères de la première chaîne suivies des caractères de la deuxième chaîne.

```
"Bonjour, " + "le monde!"  # A pour valeur "Bonjour, le monde!"
```

Appartenance L'opérateur `in` permet de tester l'appartenance d'un caractère à une chaîne de caractères, comme dans l'exemple ci-dessous.

```
print("o" in "Bonjour")  # Affiche True
print("x" in "Bonjour")  # Affiche False
print("o" not in "Bonjour")  # Affiche False
print("x" not in "Bonjour")  # Affiche True
```

1.2 Tuple

Le deuxième type de séquences que nous allons aborder est celui des *tuples*. En Python, un tuple représente une séquence ordonnée de valeurs, potentiellement de différents types. Contrairement aux chaînes de caractères, les tuples peuvent contenir des valeurs arbitraires et pas seulement des caractères.

Construction Pour former un tuple, on note simplement, entre parenthèses, les différentes expressions séparées par des virgules. Suivant le contexte, les parenthèses sont optionnelles.

```
("Boxe", "Natation", "Tennis")
```

Les tuples peuvent contenir aussi des valeurs de type différents, et même d'autres collections. Notez aussi que l'on peut donner des expressions plus complexes que de simples valeurs littérales.

```
("Jessica", 3 * 7, 170, ("Photographie", "Esca" + "lade"))
```

Déconstruction Il n'est pas rare en Python, étant donné un tuple, de vouloir affecter chaque élément à une variable. Pour ce faire, on peut lister sur la gauche du signe = lors d'une assignation non pas une seule variable, mais une séquence de variables séparées par des virgules.

```
personne = ("Albert", 19, 180, ("Foot", "Échecs"))  # Construction
prenom, age, taille, hobbies = personne  # Déconstruction
print(age)  # Affiche 19
```

Accès aux éléments Pour accéder directement à un élément d'un tuple, on indique la position de l'élément entre deux crochets ([et]), comme pour les chaînes de caractères.

```
("A", "B", "C")[0]  # A pour valeur "A"
```

La position du premier élément est 0, celle du deuxième élément 1, et ainsi de suite, tout comme pour les chaînes de caractères.

Sous-parties De même que pour les chaînes de caractères, la syntaxe pour accéder à un élément d'un tuple permet aussi de référer à une sous-partie du tuple lorsqu'on utilise deux positions séparées par un symbole deux-points (:).

```
("A", "B", "C", "D", "E")[2:4]  # A pour valeur ("C", "D")
```

Longueur On appelle le nombre d'éléments d'un tuple sa longueur. La longueur est accessible via la fonction `len`.

Parcours des éléments Grâce à la fonction `len` et à la possibilité d'accéder à des éléments via leur position, il est possible, à l'aide d'une boucle `while`, de passer en revue tous les éléments d'un tuple et ainsi d'exécuter des instructions pour chaque élément.

```
fruits = ("pommes", "bananes", "poires")
i = 0
while i < len(fruits):
    print("J'aime les", fruits[i])
    i += 1
```

Concaténation L'opérateur + permet de *concaténer* deux tuples pour former un unique tuple. Le tuple résultant de la concaténation contient les valeurs du premier tuple suivies des valeurs du deuxième tuple.

```
t1 = (1, 2)
t2 = (3, 4)
t1 + t2  # A pour valeur (1, 2, 3, 4)
```

Appartenance L'opérateur `in` permet de tester l'appartenance d'une valeur à un tuple, comme dans l'exemple ci-dessous.

```
ps = (2, 3, 5, 7)
print(3 in ps)  # Affiche True
print(4 in ps)  # Affiche False

print(3 not in ps)  # Affiche False
print(4 not in ps)  # Affiche True
```

1.3 Listes

Toutes les opérations que l'on a vues sur les tuples sont applicables de la même manière sur les listes. À priori, la seule différence est qu'à la place d'utiliser les parenthèses pour former une liste, on utilise les crochets.

```
["Elsa", "Anna", "Olaf", "Kristoff", "Sven"]
```

Pourquoi donc faire une distinction entre tuples et listes en Python ? La réponse vient de ce qu'on appelle la *mutabilité*.

Mutabilité À la différence des tuples, les listes sont *modifiables*. On dit qu'elles sont *mutables*. Prenons un exemple.

```
noms = ["Elsa", "Anna", "Olaf", "Kristoff", "Sven"]
noms.pop()  # Supprime le dernier élément
print(len(noms))  # Affiche 4
noms.pop()  # Supprimer le dernier élément
print(len(noms))  # Affiche 3
print(noms)  # Affiche ["Elsa", "Anna", "Olaf"]
```

Dans l'exemple ci-dessus, la liste stockée dans la variable `noms` voit son contenu modifié par l'exécution de la méthode `pop`, et ce alors même que la variable `noms` n'est pas modifiée. Notez qu'il n'y a aucune réaffectation de la variable, et pourtant la liste a bel et bien changé.

Par le passé, lorsque nous voulions modifier une valeur, nous passions par une réaffectation de la variable qui contenait la valeur. La valeur en soi n'était pas modifiée, seulement la variable. La valeur stockée était simplement remplacée par une autre valeur.

```
n = 3
print(n)  # Affiche 3
n = 4
print(n)  # Affiche 4
print(3)  # Affiche 3
```

Dans l'exemple ci-dessus, les valeurs en soi ne changent pas, ce sont les variables qui changent. Au début, la variable `n` prend pour valeur 3, puis ensuite elle prend pour valeur 4. La valeur 3 quant à elle ne change pas. 3 reste 3 et ne devient pas 4.

Dans le cas des listes, et des autres valeurs mutables que l'on pourra aborder, la situation sera différente : la valeur *elle-même* pourra être modifiée.

Aliasing Ce phénomène de modification de valeurs sans modification de variables peut parfois amener à des situations difficiles à comprendre, surtout lorsque la même valeur est accessible sous différents noms (ce que l'on nomme *aliasing*).

```
xs = [1, 2, 3]
ys = xs

print(ys)  # Affiche [1, 2, 3]
xs.pop()  # Enlève le dernier élément de la valeur stockée dans xs
print(ys)  # Affiche [1, 2]
```

Dans l'exemple ci-dessous, la valeur stockée dans la variable `ys` a changé en cours d'exécution alors qu'un élément a été retiré de la valeur stockée dans `xs`. La situation s'explique facilement car il s'agit *de la même valeur* qui est stockée à la fois dans `xs` et dans `ys`. Pour éviter cette situation, il est possible de faire une *copie* de la liste.

Copie d'une liste Parfois, il sera utile, comme dans le cas discuté juste à l'instant, avec la problématique d'aliasing, de faire une copie d'une liste. Une copie est une liste en tout point identique mais qui a sa propre identité. Les opérations faites sur une liste n'affectent pas les éventuelles copies de la liste. Pour copier une liste en Python, on appelle la méthode `copy`.

```
from copy import deepcopy

xs = [1, 2, 3]
ys = deepcopy(xs)  # Stocke une copie de xs dans ys

print(ys)  # Affiche [1, 2, 3]
xs.pop()  # Enlève le dernier élément de la valeur stockée dans xs
print(xs)  # Affiche [1, 2]
print(ys)  # Affiche [1, 2, 3]
ys.pop()
print(xs)  # Affiche [1, 2]
```

Dans l'exemple ci-dessus, on observe que la liste originale et sa copie ne sont pas affectées par les modifications effectuées sur l'autre liste.

Opérations sur les listes

Maintenant que nous avons mis en lumière ce phénomène de mutabilité, passons en revue les différentes opérations qui modifient le contenu d'une liste.

Modification des éléments Grâce à une assignation, il est possible de remplacer la valeur stockée à une certaine position dans la liste. Pour cela, sur la partie de gauche du `=`, on utilisera la même syntaxe que pour accéder à l'élément.

```
animaux = ["chats", "chiens", "poissons"]
animaux[2] = "oiseaux"
print(animaux)  # Affiche ["chats", "chiens", "oiseaux"]
```

Ajout d'éléments à la fin La méthode `append` permet d'ajouter un élément à la fin d'une liste.

```
xs = [1, 2, 3]
xs.append(4)
print(xs)  # Affiche [1, 2, 3, 4]
```

La méthode `append` permet d'ajouter un unique élément à la fin. Pour ajouter plus d'un élément à la fois, on utilisera la méthode `extend`. La méthode `extend` permet d'ajouter une séquence d'éléments à la fin d'une liste.

```
xs = [1, 2, 3]
ys = [4, 5, 6]
xs.extend(ys)
print(xs)  # Affiche [1, 2, 3, 4, 5, 6]
print(ys)  # Affiche [4, 5, 6]
```

Notez au passage que la liste de valeurs passée en argument à `extend` n'est elle pas modifiée.

Ajout d'éléments à des positions arbitraires La méthode `insert` permet d'insérer un élément à une position donnée dans la liste.

```
xs = ["A", "B", "D", "E"]
xs.insert(2, "C")
print(xs)  # Affiche ["A", "B", "C", "D", "E"]
```

Il est cependant important de noter que d'insérer un élément au milieu d'une liste est une opération beaucoup plus complexe pour Python que d'ajouter un élément à la fin. Alors que l'ajout d'un élément à la fin d'une liste prend un temps constant [1] (qui ne dépend pas de la taille de la liste), ajouter un élément au milieu de la liste demande à Python de déplacer tous les éléments qui suivent la position d'insertion. Cette opération prend, dans le pire des cas, un temps linéaire dans le nombre d'éléments de la liste.

Suppression d'éléments à la fin La méthode `pop` permet de retirer le dernier élément d'une liste. La méthode retourne l'élément ainsi retiré.

```
xs = [1, 2, 3]
print(xs.pop())  # Affiche 3
print(xs.pop())  # Affiche 2
print(xs.pop())  # Affiche 1
print(xs)  # Affiche []
```

Suppression d'éléments à des positions arbitraires La méthode `pop` peut aussi s'utiliser pour retirer un élément à une position quelconque de la liste. Pour cela, on ajoute comme argument à l'appel à `pop` la position à laquelle retirer une valeur.

1. En moyenne. Certains ajouts peuvent prendre plus de temps, mais la moyenne est constante. On parle de complexité temporelle amortie.

```
xs = [1, 2, 3]
print(xs.pop(1))  # Affiche 2
print(xs.pop(0))  # Affiche 1
print(xs.pop(0))  # Affiche 3
print(xs)  # Affiche []
```

Tout comme pour l'ajout, la suppression d'éléments à la fin d'une liste est plus rapide qu'à des positions arbitraires de la liste.

Trier une liste On appelle trier une liste le fait de mettre les éléments de la liste dans l'ordre, généralement du plus petit au plus grand. Pour trier une liste en Python, on utilise la méthode `sort`.

```
xs = [4, 3, 5, 1, 2]
xs.sort()
print(xs)  # Affiche [1, 2, 3, 4, 5]
```

2 La boucle `for`

Une boucle `for` permet de parcourir les éléments d'une séquence, un par un, en exécutant à chaque fois le corps de la boucle avec l'élément courant.

```
for element in ("Python", "C++", "Java"):
    print("J'aime le langage", element)
```

La variable `element` prend successivement pour valeur chaque élément de la séquence. Le nom de la variable peut être choisi librement. Avant chaque itération, la prochaine valeur de la séquence est affectée à la variable.

L'avantage d'une boucle `for` par rapport à une boucle `while`, en plus d'être plus concise, est qu'il n'est pas nécessaire de gérer une variable d'index (ce qui est souvent source d'erreurs).

```
i = 0
xs = ("Python", "C++", "Java")
while i < len(xs):
    element = xs[i]
    print("J'aime le langage", element)
    i += 1 # Oublier cette ligne crée une boucle infinie !
```

La fonction `range` La fonction `range` est souvent utilisée en conjonction avec la boucle `for`. La fonction `range` permet de générer une séquence de nombres entiers.

```
for i in range(5):
    print(i)  # Affichera 0, puis 1, puis 2, puis 3, puis 4
```

Lorsque la fonction `range` est appelée avec un seul argument, elle génère les entiers de 0 jusqu'à l'entier juste avant l'argument, donnant ainsi une séquence de longueur égale à l'argument.

La fonction `range` peut aussi prendre deux arguments, un début et une fin.

```
for i in range(3, 6):
    print(i)  # Affichera 3, puis 4, puis 5
```

Dans ce cas, la séquence générée commence à l'entier donné en premier argument et se termine juste avant l'entier donné en deuxième argument.

La fonction `range` peut aussi prendre un troisième argument, un pas.

```
for i in range(3, 8, 2):
    print(i)  # Affichera 3, puis 5, puis 7
```

Parcours avancés Parfois, on souhaitera tout de même avoir accès à la position de l'élément courant dans une boucle `for`. Pour ce faire, on utilisera la fonction `enumerate`, comme ceci :

```
fruits = ("pommes", "bananes", "poires")
for i, fruit in enumerate(fruits):
    print(i, fruit)  # Affichera 0 pommes, 1 bananes, puis 2 poires
```

Parfois, on voudra parcourir les éléments dans l'ordre inverse. Pour cela, on pourra utiliser la fonction `reversed`.

```
fruits = ("pommes", "bananes", "poires")
for fruit in reversed(fruits):
    print(fruit)  # Affichera poires puis bananes puis pommes
```

D'autres fois, on voudra parcourir les éléments du plus petit au plus grand. Pour cela, on utilisera la fonction `sorted`.

```
fruits = ("pommes", "bananes", "poires")
for fruit in sorted(fruits):
    print(fruit)  # Affichera bananes puis poires puis pommes
```

Parfois, on voudra parcourir plusieurs séquences de valeurs en même temps. Si le but est de passer en revue toutes les combinaisons possibles de valeurs, on utilisera des boucles imbriquées.

```
fruits = ("pommes", "bananes", "poires")
couleurs = ("rouges", "jaunes", "vertes")
for fruit in fruits:
    for couleur in couleurs:
        print(fruit, couleur)  # Affichera toutes les combinaisons
```

Si, au contraire, on cherche à parcourir en parallèle deux ou plusieurs séquences, on utilisera la fonction `zip`, comme dans l'exemple suivant.

```
fruits = ("pommes", "bananes", "poires")
couleurs = ("rouges", "jaunes", "vertes")
for fruit, couleur in zip(fruits, couleurs):
    print(fruit, couleur)  # Affichera pommes rouges
                           # puis bananes jaunes
                           # puis poires vertes
```

Au contraire, on cherchera parfois à obtenir toutes les combinaisons possibles de valeurs de plusieurs séquences. Pour cela, on utilisera soit des boucles imbriquées, soit la fonction `product` du module `itertools`.

```
# Avec des boucles imbriquées
fruits = ("pommes", "bananes", "poires")
couleurs = ("rouges", "jaunes", "vertes")
for fruit in fruits:
    for couleur in couleurs:
        print(fruit, couleur)  # Affichera toutes les combinaisons

# Avec itertools.product
from itertools import product

fruits = ("pommes", "bananes", "poires")
couleurs = ("rouges", "jaunes", "vertes")
for fruit, couleur in product(fruits, couleurs):
    print(fruit, couleur)  # Affichera toutes les combinaisons
```

3 Réductions

Une réduction est une opération qui prend une séquence de valeurs et qui calcule une unique valeur à partir de cette séquence.

Le calcul de la somme des éléments d'une liste est un exemple classique de réduction.

```
n = 0
xs = [1, 2, 3, 4, 5]
for x in xs:
    n += x
print(somme)  # Affiche 15
```

Une autre réduction classique est le calcul du produit des éléments d'une liste.

```
p = 1
xs = [1, 2, 3, 4, 5]
for x in xs:
    p *= x
print(p)  # Affiche 120
```

Il existe de nombreuses autres réductions, comme par exemple le calcul du minimum ou du maximum d'une liste, ou encore le comptage du nombre d'éléments d'une liste qui satisfont une certaine propriété.

Les réductions sur les séquences sont très courantes en programmation.

La plupart suivent le schéma suivant :

```
accumulateur = valeur_initiale
for element in sequence:
    accumulateur = operation(accumulateur, element)
```

Fonctions de réduction Python propose plusieurs fonctions de réduction intégrées, comme par exemple `sum`, `min`, `max`, `all` et `any`.

Notes de cours
Semaine 4

Cours Turing

Introduction

Lors de cette quatrième semaine de cours, nous allons aborder des aspects importants de la programmation en Python qui vont au-delà du langage de programmation. Jusqu'à présent, nous avons appris les bases de la syntaxe Python, les types de données fondamentaux, les structures de contrôle et les fonctions. Nous allons nous intéresser aujourd'hui non pas à de nouvelles constructions du langage, mais plutôt à des pratiques et des conventions qui permettent d'écrire du code de meilleure qualité.

Nous allons aborder les sujets suivants :

— Guide de style Python
— Nomenclature
— Documentation en Python (commentaires, docstrings, annotations de type)
— Tests unitaires

Ces différents sujets devraient vous permettre de découvrir un aspect plus pratique de la programmation en Python. Ils vous aideront à écrire du code plus lisible, plus maintenable et plus fiable. Ils sont particulièrement importants lorsque vous travaillez en équipe ou sur des projets de grande envergure ou complexité. Je vous encourage vivement à appliquer ces bonnes pratiques dans vos projets de programmation futurs.

1 Guide de style Python

Un guide de style est un ensemble de conventions et de recommandations visant à améliorer la lisibilité, la cohérence et la maintenabilité du code. En Python, le guide de style le plus largement adopté est le PEP 8 : `https://peps.python.org/pep-0008/`. Le PEP 8 couvre divers aspects du style de code, notamment la mise en forme du code, la nomenclature, les conventions de codage et les bonnes pratiques. Je vous invite à aller le consulter pour plus de détails.

Voici quelques recommandations clés du PEP 8 :

- **Indentation :** Utiliser 4 espaces par niveau d'indentation. Éviter les tabulations.
- **Longueur des lignes :** Limiter la longueur des lignes à 79 caractères pour le code et 72 caractères pour les commentaires et les docstrings.
- **Espaces dans le code :** Utiliser des espaces autour des opérateurs (par exemple, `x = 1 + 2`) et après les virgules. Éviter les espaces inutiles à l'intérieur des parenthèses, crochets ou accolades.
- **Nomenclature :** Utiliser des noms explicites et descriptifs pour les variables.
- **Commentaires :** Utiliser des commentaires pour expliquer le code, mais éviter les commentaires évidents. Les commentaires doivent être clairs et concis.
- **Docstrings :** Utiliser des docstrings pour documenter les fonctions. Les docstrings doivent décrire brièvement ce que fait la fonction documentée, ses paramètres et sa valeur de retour.
- **Imports :** Grouper les imports en trois sections : imports standard, imports de bibliothèques tierces, et imports locaux. Chaque section doit être séparée par une ligne vide.

Le but d'un guide de style est de garantir que le code est facile à lire et à comprendre, même pour les développeurs qui ne l'ont pas écrit. En suivant un guide de style, on facilite la collaboration entre plusieurs personnes travaillant sur le même projet. Certains des points mentionnés ci-dessus seront abordés plus en détail dans les sections suivantes.

2 Nomenclature

La nomenclature est le terme utilisé pour décrire la façon de nommer les variables, fonctions et autres éléments dans le code. Une bonne nomenclature est essentielle pour rendre le code lisible et compréhensible.

2.1 Variables

Les noms de variables doivent être descriptifs et refléter clairement leur contenu ou leur rôle dans le programme.

2.1.1 Syntaxe

En Python, les noms de variables doivent respecter les règles suivantes :
— Ils doivent commencer par une lettre (a-z, A-Z) ou un underscore (_).
— Ils peuvent contenir des lettres, des chiffres (0-9) et des underscores.
— Ils sont sensibles à la casse (par exemple, `variable` et `Variable` sont deux variables différentes).
— Ils ne doivent pas être des mots réservés de Python (comme `if`, `for`, `while`, etc.).

2.1.2 Conventions

En Python, la convention de nommage recommandée pour les variables est le *snake_ case*, où les mots sont en minuscules et séparés par des underscores. Par exemple : `total_price`, `user_name`, `item_count`.

La plupart du temps, on préférera des noms de variables formés de mots complets plutôt que des abréviations. Par exemple, on utilisera `total_price` au lieu de `tp`.

Cependant, pour des variables temporaires ou dans des contextes très locaux, des noms courts comme `i`, `j` ou `temp` peuvent être acceptables. Ainsi, dans une boucle `for`, il est courant d'utiliser `i` comme variable d'itération :

```python
for i in range(10):
    print(i)
```

De même, pour des variables représentant des valeurs abstraites ou mathématiques, des noms comme `x`, `y` ou `z` sont couramment utilisés.

```python
def calculate_distance(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5

point_a = (1, 2)
point_b = (4, 6)
distance = calculate_distance(point_a, point_b)
print("Distance:", distance)
```

Dans le cas de variables contenant des collections (listes, dictionnaires, ensembles), il est souvent utile d'utiliser des noms au pluriel pour indiquer qu'elles contiennent plusieurs éléments. Par exemple : `users`, `items`, `prices`. Cela permet de clarifier le type de données que la variable contient. Cela permet aussi d'utiliser des noms au singulier pour les éléments individuels lors du parcours de la collection à l'aide d'une boucle.

```
users = ["Alice", "Bob", "Charlie"]
for user in users:
    print("Hello,", user)
```

De même, pour des collections abstraites, on peut utiliser des noms comme `values` ou même `xs`, `ys` et `zs`.

```
def map_list(f, xs):
    result = []
    for x in xs:
        result.append(f(x))
    return result
```

2.2 Fonctions

Les fonctions doivent également avoir des noms descriptifs qui indiquent clairement leur objectif ou leur action.

2.2.1 Syntaxe

Les noms de fonctions suivent les mêmes règles que les noms de variables.

2.2.2 Conventions

En Python, la convention de nommage recommandée pour les fonctions est également le *snake_ case*. Par exemple : `calculate_total`, `get_user_info`, `process_data`.

Les noms de fonctions doivent généralement être des verbes ou des phrases verbales qui décrivent l'action effectuée par la fonction. Par exemple : `send_email`, `fetch_data`, `save_record`. Cela est très courant pour les fonctions avec des effets de bord (modification d'une variable globale, écriture dans un fichier, affichage à l'écran, etc.).

Pour les fonctions qui retournent une valeur sans effet de bord, on utilisera souvent un nom qui décrit la valeur retournée. Par exemple : `rectangle`, `sum`, `max_value`.

Dans le cas de fonctions qui retournent un booléen, il est courant d'utiliser des préfixes comme `is_`, `has_`, `can_` ou `should_`. Par exemple : `is_valid`, `has_permission`, `can_execute`.

Concernant les paramètres de fonctions, les mêmes conventions de nommage que pour les variables s'appliquent.

3 Documentation en Python

Documenter son code est une pratique essentielle en programmation. Une bonne documentation facilite la compréhension, la maintenance et la collaboration sur le code. En Python, il existe plusieurs façons de documenter son code. Nous allons aborder deux méthodes principales : les commentaires et les *docstrings*.

3.1 Commentaires

Les commentaires sont des lignes de texte dans le code qui ne sont pas exécutées par l'interpréteur Python. Ils sont utilisés pour expliquer le fonctionnement du code, fournir des informations supplémentaires ou marquer des sections importantes.

3.1.1 Syntaxe

En Python, les commentaires commencent par le symbole # et s'étendent jusqu'à la fin de la ligne. Les commentaires peuvent être placés sur une ligne à part ou à la fin d'une ligne de code. Lorsqu'un commentaire est placé à la fin d'une ligne de code, il est généralement précédé d'au moins deux espaces pour améliorer la lisibilité.

```
# Ceci est un commentaire sur une seule ligne

x = 10  # Ceci est un commentaire à la fin d'une ligne de code
```

Pour faire un commentaire sur plusieurs lignes, on place le symbole # au début de chaque ligne.

```
# Ceci est un commentaire
# sur plusieurs lignes
# Chaque ligne commence par un #
```

Les éditeurs de code, tels que VSCode, offrent souvent des raccourcis pour commenter ou décommenter rapidement des blocs de code.

3.1.2 Bonnes pratiques

Les commentaires doivent être utilisés judicieusement pour améliorer la compréhension du code. Les commentaires ne doivent pas être utilisés pour expliquer des choses évidentes ou redondantes. Par exemple, le commentaire suivant est inutile :

```
x = 10  # On stocke la valeur 10 dans la variable x
```

Il est préférable d'utiliser des commentaires pour expliquer le pourquoi du code, les décisions de conception ou les parties complexes. Par exemple :

```
frames = []

# Phase de chute
```

```python
for i in range(0, 101):
    p = (i / 100) ** 2  # Progression quadratique (accélération)
    frames.append(frame(p, red, black))

# Phase de rebond
for i in range(99, 0, -1):  # 100 et 0 sont exclus (déjà présents)
    p = (i / 100) ** 2
    frames.append(frame(p, red, black))
```

Écrire des commentaires utiles n'est pas toujours facile et demande de la pratique. Je vous encourage vivement à écrire des commentaires dans votre code. Pour chaque commentaire, questionnez-vous sur son utilité. Évitez les paraphrases et privilégiez les explications qui éclairent les lecteurs sur les aspects importants du code. C'est en pratiquant cet exercice que vous deviendrez plus à l'aise avec l'écriture de commentaires pertinents.

3.2 Docstrings

En Python, les docstrings sont des chaînes de caractères utilisées pour documenter notamment les fonctions. Elles sont placées immédiatement après la définition de l'élément qu'elles documentent et sont délimitées par des triples guillemets (généralement doubles).

```python
def ma_fonction(param1, param2):
    """Description de ce que fait la fonction.

    Plus de détails sur le fonctionnement de la fonction.

    Paramètres:
        param1: Description du paramètre 1.
        param2: Description du paramètre 2.

    Retourne:
        Description de la valeur de retour.
    """
    # Corps de la fonction
    pass
```

Les docstrings sont accessibles via l'attribut `.__doc__` de l'objet documenté. Par exemple :

```python
print(ma_fonction.__doc__)
```

Les docstrings sont particulièrement utiles pour fournir des informations sur l'utilisation des fonctions, leurs paramètres, leurs valeurs de retour et tout autre détail pertinent. Elles sont également utilisées par des outils de documentation automatique pour générer des documents à partir du code source.

Il n'existe pas de format strictement imposé pour les docstrings, mais il est recommandé de suivre des conventions cohérentes. Les conventions changent suivant les différentes communautés, équipes, projets ou encore outils de documentation.

Par exemple, voici la docstring de la fonction `rectangle` de PyTamaro, telle qu'elle apparait dans le code source[1] :

```
def rectangle(width: float, height: float, color: Color) -> Graphic:
    """
    Creates a rectangle of the given size, filled with a color.

    :param width: width of the rectangle
    :param height: height of the rectangle
    :param color: the color to be used to fill the rectangle
    :returns: the specified rectangle as a graphic
    """
    # Reste du code...
```

Notez au passage l'utilisation d'*annotations de type* dans la définition de la fonction. Nous abordons ce sujet dans la section suivante.

3.3 Annotations de type

Les annotations de type en Python sont une fonctionnalité qui permet de spécifier les types de données attendus pour les paramètres d'une fonction et pour sa valeur de retour. Elles sont introduites en Python 3.5 et sont principalement utilisées pour améliorer la lisibilité du code et faciliter la détection d'erreurs potentielles lors du développement.

3.3.1 Syntaxe

Les annotations de type sont ajoutées à la définition d'une fonction en utilisant le symbole `:` après le nom du paramètre, suivi du type attendu.

```
def ma_fonction(param1: int, param2: str):
    # Corps de la fonction
    pass
```

Dans cet exemple, `param1` est annoté comme un entier (`int`) et `param2` comme une chaîne de caractères (`str`).

Pour annoter le type de retour d'une fonction, on utilise le symbole `->` suivi du type de retour attendu, placé avant les deux-points de la définition de la fonction.

```
def addition(a: int, b: int) -> int:
    return a + b
```

Ici, la fonction `addition` prend deux entiers en paramètres et retourne également un entier.

1. Accessible ici : https://github.com/LuCEresearchlab/pytamaro/blob/main/pytamaro/primitives.py

3.3.2 Utilisation

Les principales raisons d'utiliser des annotations de type sont les suivantes :

— **Documentation :** Elles servent de documentation intégrée au code, aidant les développeurs à comprendre rapidement les types de données attendus par une fonction.

— **Éditeurs de code :** De nombreux éditeurs de code (comme VSCode) utilisent les annotations de type pour fournir des fonctionnalités avancées, telles que l'autocomplétion, la détection d'erreurs en temps réel et la navigation dans le code.

— **Outils de vérification de type :** Des outils externes, comme *mypy*, peuvent être utilisés pour analyser le code et vérifier que les types utilisés sont cohérents avec les annotations. Ils peuvent être intégrés dans l'éditeur de code ou exécutés séparément.

Il est important de noter que les annotations de type en Python sont facultatives et n'affectent pas l'exécution du code.

3.3.3 Types courants

Voici quelques types couramment utilisés dans les annotations de type en Python :

— `int` : Entier
— `float` : Nombre à virgule flottante
— `str` : Chaîne de caractères
— `bool` : Booléen (True ou False)
— `list` : Liste
— `dict` : Dictionnaire
— `set` : Ensemble
— `tuple` : Tuple (une liste immuable)
— `None` : Absence de valeur utile

Les noms de *classes* peuvent également être utilisés comme types. Par exemple, dans PyTamaro, on peut annoter une fonction pour indiquer qu'elle retourne un objet de type `Graphic` ou qu'elle prend un paramètre de type `Color`.

```
from pytamaro import Graphic, Color

def square(size: float, color: Color) -> Graphic:
    return rectangle(size, size, color)
```

3.3.4 Types de collections

Pour les collections, comme les listes, il est possible de spécifier le type des éléments qu'elles contiennent en utilisant la syntaxe `list[Type]`. Par exemple, une liste d'entiers peut être annotée comme `list[int]`.

```
def my_sum(numbers: list[int]) -> int:
    return sum(numbers)
```

Pour les tuples, on peut spécifier les types de chaque élément en utilisant la syntaxe `tuple[Type1, Type2, ...]`. Par exemple, un tuple contenant un entier et une chaîne de caractères peut être annoté comme `tuple[int, str]`.

3.3.5 Unions de types

Il est possible d'indiquer qu'un paramètre ou une valeur de retour peut être de plusieurs types en utilisant la syntaxe `Type1 | Type2`. Par exemple, une fonction qui peut retourner soit un entier, soit une chaîne de caractères peut être annotée comme `int | str`.

```
def process(value: int | str) -> None:
    if isinstance(value, int):
        print("C'est un entier :", value)
    else:
        print("C'est une chaîne de caractères :", value)
```

3.3.6 Types optionnels

Parfois, on souhaite indiquer qu'un paramètre est optionnel, ou alors qu'une valeur de retour peut ne pas être présente. Dans les deux cas, on utilise généralement la valeur spéciale `None` pour représenter l'absence de valeur. Pour annoter un paramètre ou une valeur de retour qui peut être d'un certain type ou `None`, on utilise la syntaxe `Type | None`.

4 Tests

Les tests sont une partie essentielle du développement logiciel. Ils permettent de vérifier que le code fonctionne comme prévu et de détecter les erreurs. Il existe plusieurs types de tests, mais nous allons nous concentrer sur les tests *unitaires*, qui consistent à tester des unités individuelles de code. Dans notre cas, les unités de code que nous allons tester sont les *fonctions*.

4.1 Décomposition en fonctions

La décomposition en fonctions est une pratique de programmation qui consiste à diviser un programme en fonctions indépendantes et réutilisables. Chaque fonction doit avoir une responsabilité claire et accomplir une tâche spécifique.

En plus de faciliter la lecture et la maintenance du code, la décomposition en fonctions facilite également le processus de test du code. En effet, en isolant des parties spécifiques du code dans des fonctions, on peut tester chaque fonction individuellement, ce qui permet de détecter plus facilement les erreurs et de s'assurer que chaque partie du code fonctionne correctement.

4.2 Le module `unittest`

Python fournit un module intégré appelé `unittest` pour écrire et exécuter des tests unitaires. Voici un exemple simple de test unitaire utilisant le module `unittest` :

```
import unittest

from my_module import addition  # Importer la fonction à tester

# La classe de test
class TestAddition(unittest.TestCase):
    def test_add_positive_numbers(self):
        self.assertEqual(addition(2, 3), 5)

    def test_add_negative_numbers(self):
        self.assertEqual(addition(-2, -3), -5)

if __name__ == '__main__':
    unittest.main()
```

Dans cet exemple, nous avons une fonction `addition` que nous voulons tester et qui est définie dans un module appelé `my_module`. La classe `TestAddition` hérite de `unittest.TestCase` et contient deux méthodes de test : `test_add_positive_numbers` et `test_add_negative_numbers`.[2]

2. Le module `unittest` utilise des concepts de programmation orientée objet comme les classes et l'héritage. Nous n'avons pas encore abordé ces concepts dans le cours. Ne vous inquiétez pas si vous ne comprenez pas tout de suite comment cela fonctionne. L'important est de comprendre l'idée générale des tests unitaires et la façon dont ils sont écrits.

Lorsque vous exécutez ce script, le module `unittest` exécute automatiquement toutes les méthodes de test définies dans la classe `TestAddition`. Chaque méthode de test utilise des *assertions* pour vérifier que le résultat de la fonction `addition` est correct. Si une assertion échoue, le test est considéré comme ayant échoué.

Écrire des tests unitaires est une bonne pratique qui permet de détecter des erreurs rapidement. Cependant, il n'est pas toujours évident de trouver de bons tests à écrire. Quels cas de figure faut-il tester ? Il n'existe malheureusement pas de recette miracle. Voici quelques conseils pour vous aider à écrire de bons tests :

— Testez les cas normaux : Commencez par tester les cas d'utilisation de votre fonction anticipés comme les plus courants.
— Testez les cas limites : Pensez aux cas extrêmes ou aux valeurs limites qui pourraient poser problème. Par exemple, si votre fonction prend un nombre en entrée, testez les valeurs négatives, zéro et les très grands nombres. Si votre fonction prend une liste en entrée, testez les listes vides ou les listes avec un seul élément.
— Testez les erreurs : Si votre fonction peut lever des exceptions, écrivez des tests pour vérifier que les exceptions sont bien levées dans les situations appropriées.

Aussi, certaines fonctions sont plus faciles à tester que d'autres. Essayez dans votre pratique de **découper votre code en fonctions testables** et d'écrire des tests pour ces fonctions.

Conclusion

Dans cette quatrième semaine de cours, nous avons exploré des aspects importants de la programmation en Python qui vont au-delà des constructions de base du langage.

Les différents éléments que nous avons abordés aujourd'hui sont essentiels pour écrire du code de qualité. Ils contribuent à rendre le code plus compréhensible et facilitent la collaboration entre plusieurs développeurs. Je vous encourage vivement à appliquer ces bonnes pratiques dans vos projets de programmation. Comme pour toute compétence en programmation, la maîtrise de ces aspects vient avec la pratique. N'hésitez pas à expérimenter, à écrire du code, à le documenter et à le tester régulièrement.

Notes de cours
Semaine 5

Cours Turing

Introduction

Cette semaine, nous allons nous intéresser à des sujets assez différents :
— La génération de nombres pseudo-aléatoires à l'aide du module `random` de Python.
— L'utilisation de l'aléatoire (ou pseudo-aléatoire) pour résoudre des problèmes.
— Les *dictionnaires*, une structure de données en Python.
— La définition de listes (et dictionnaires) par compréhension.

1 Génération de nombres pseudo-aléatoires avec `random`

Le module `random` de Python rassemble plusieurs fonctions en rapport avec l'aléatoire. Le module est inclus dans la bibliothèque standard de Python, il n'est donc pas nécessaire de l'installer séparément.

Dans cette section, nous allons voir comment utiliser ce module pour générer des nombres pseudo-aléatoires, choisir des éléments pseudo-aléatoires dans une liste, ou encore mélanger une liste. La documentation complète du module `random` est disponible à l'adresse suivante :

`https://docs.python.org/3/library/random.html`

1.1 Fonctions courantes

Le module `random` fournit plusieurs fonctions pour générer des nombres pseudo-aléatoires. Il contient aussi des fonctions pour mélanger des listes ou choisir des éléments aléatoires dans une séquence.

Voici quelques fonctions couramment utilisées :

— `random.random()` : génère un nombre flottant aléatoire dans l'intervalle $[0.0, 1.0)$.
— `random.randint(a, b)` : génère un entier aléatoire entre `a` et `b` (inclus).
— `random.choice(xs)` : choisit un élément aléatoire dans la séquence `xs`.
— `random.shuffle(xs)` : mélange la liste `xs`.

Exemple

Voici un exemple d'utilisation du module `random` :

```python
import random

# Affiche un nombre à virgule flottante aléatoire entre 0.0 et 1.0
print(random.random())

# Affiche un entier aléatoire entre 1 et 10 (compris)
print(random.randint(1, 10))

# Affiche un élément aléatoire dans une liste
fruits = ['pomme', 'banane', 'cerise']
print(random.choice(fruits))

# Mélange la liste
random.shuffle(fruits)

# Affiche la liste mélangée
print(fruits)
```

Le résultat de ce programme sera différent à chaque exécution. Essayez de l'exécuter plusieurs fois pour voir les différences !

1.2 Le *seed*

Les générateurs de nombres pseudo-aléatoires utilisent un algorithme déterministe pour produire une séquence de nombres qui semblent aléatoires. En partant d'une même valeur initiale, appelée *seed*, le générateur produira toujours la même séquence de nombres.

On peut définir la *seed* en utilisant la fonction `random.seed(value)`. Cela est utile pour reproduire des résultats lors de tests ou de débogage.

Exemple

Voici un exemple montrant l'effet du *seed* :

```
import random

# Définir le seed
random.seed(42)

# Générer une séquence de nombres aléatoires
for _ in range(5):
    print(random.randint(1, 100))

print("---- Réinitialisation du seed ----")

# Réinitialiser le seed
random.seed(42)
# Générer à nouveau la même séquence de nombres aléatoires
for _ in range(5):
    print(random.randint(1, 100))
```

Les deux boucles afficheront la même séquence de nombres :

```
82
15
4
95
36
---- Réinitialisation du seed ----
82
15
4
95
36
```

1.3 Autres distributions

Le module `random` fournit aussi des fonctions pour générer des nombres selon différentes distributions statistiques. Vous pouvez par exemple générer des nombres suivant une distribution uniforme, normale (gaussienne), exponentielle, etc. Pour plus de détails, consultez la documentation du module `random`.

2 Utilisation de l'aléatoire en informatique

L'aléatoire est un outil puissant en informatique qui trouve des applications dans des domaines très variés. Quelques exemples de domaines d'application de l'aléatoire sont discutés ci-dessous.

2.1 Simulation

L'aléatoire est souvent utilisé pour simuler des phénomènes complexes dans divers domaines, tels que la physique, la biologie, l'économie, etc. Par exemple, on peut utiliser des nombres pseudo-aléatoires pour modéliser le comportement de particules dans un gaz, la propagation d'une épidémie, ou encore les fluctuations du marché boursier. Des modèles probabilistes sont souvent employés pour représenter ces phénomènes, et la génération de nombres aléatoires permet de simuler des scénarios variés.

2.2 Algorithmique

Certains algorithmes utilisent la génération de nombres pseudo-aléatoires pour obtenir des résultats approximatifs de manière plus rapide que des algorithmes déterministes. Par exemple, comme on le verra en exercices, il est possible d'estimer la valeur de π en utilisant une méthode probabiliste appelée la méthode de Monte Carlo.

Contenu avancé

Le hasard peut aussi être utilisé dans la résolution de jeux, comme les échecs ou le go, où des algorithmes comme Monte Carlo Tree Search (MCTS) explorent les coups possibles en simulant des parties aléatoires pour évaluer les positions. Dans ce contexte, l'aléatoire permet de sonder efficacement un grand espace de possibilités de parties.

Contenu avancé

D'autres algorithmes utilisent l'aléatoire pour éviter des situations pathologiques dans lesquelles un algorithme déterministe pourrait être inefficace. C'est le cas par exemple de l'algorithme de tri rapide (*quicksort*) qui choisit un pivot aléatoire pour diviser la liste à trier. Dans ce genre d'algorithme, le résultat n'est pas affecté par l'aléatoire, mais la performance de l'algorithme peut être améliorée en utilisant un choix aléatoire.

2.3 Optimisation

L'aléatoire est également utilisé dans des algorithmes d'optimisation, tels que les algorithmes génétiques ou le recuit simulé (*simulated annealing*). Ces algorithmes s'inspirent de processus naturels pour explorer l'espace des solutions possibles et trouver de bonnes solutions à des problèmes complexes. L'utilisation de l'aléatoire permet de diversifier la recherche et d'éviter de rester bloqué dans des optima locaux.

2.4 Cryptographie

Finalement, un autre domaine où l'aléatoire est crucial est la cryptographie. Les systèmes de chiffrement modernes reposent sur des clés secrètes qui doivent être générées de manière (pseudo-)aléatoire pour garantir la sécurité des communications. Vous aurez l'occasion d'en apprendre davantage sur ce sujet dans le cadre du prochain module de ce cours. Vous y verrez notamment des méthodes de génération de nombres pseudo-aléatoires.

Contenu avancé

Il est important de noter que pour des applications en cryptographie, les générateurs de nombres pseudo-aléatoires standards tels que ceux fournis par le module `random` de Python ne sont pas adaptés, car ils peuvent être prédits par un attaquant.
Des générateurs de nombres aléatoires cryptographiquement sécurisés, comme ceux disponibles dans le module `secrets` de Python, doivent être utilisés à la place.

3 Dictionnaires en Python

Un dictionnaire en Python est une structure de données qui permet de stocker des paires clé-valeur. Chaque clé est unique et est associée à une valeur.

3.1 Création de dictionnaires et accès aux valeurs

Voici comment créer un dictionnaire en Python :

```
# Création d'un dictionnaire vide
mon_dictionnaire = {}

# Création d'un dictionnaire avec des paires clé-valeur
mon_dictionnaire = {'clé1': 'valeur1', 'clé2': 'valeur2'}

# Accès à une valeur via sa clé
valeur = mon_dictionnaire['clé1']  # valeur sera 'valeur1'
```

La méthode `get` permet aussi d'accéder à une valeur en fournissant une clé. Contrairement à l'accès direct via les crochets, `get` ne génère pas d'erreur si la clé n'existe pas dans le dictionnaire. Il est possible de spécifier qui sera retournée si la clé n'existe pas. Si aucune valeur par défaut n'est fournie, la valeur `None` sera retournée en cas de clé inexistante.

```
mon_dictionnaire = {'clé1': 'valeur1', 'clé2': 'valeur2'}

# Accès à une valeur via sa clé avec get
valeur = mon_dictionnaire.get('clé1', 'valeur_par_défaut')
print(valeur)  # Affiche valeur1

valeur_inexistante = mon_dictionnaire.get('clé_inexistante', 'valeur_par_défaut')
print(valeur_inexistante)  # Affiche valeur_par_défaut
```

3.2 Modification de dictionnaires

Les dictionnaires sont des structures mutables, ce qui signifie que vous pouvez ajouter, modifier ou supprimer des paires clé-valeur après leur création.

```
# Création d'un dictionnaire
mon_dictionnaire = {'clé1': 'valeur1', 'clé2': 'valeur2'}

# Ajout d'une nouvelle paire clé-valeur
mon_dictionnaire['clé3'] = 'valeur3'

# Modification d'une valeur existante
mon_dictionnaire['clé1'] = 'nouvelle_valeur1'
```

```python
# Suppression d'une paire clé-valeur
del mon_dictionnaire['clé2']

# Affichage du dictionnaire
print(mon_dictionnaire)  # Affiche {'clé1': 'nouvelle_valeur1', 'clé3': 'valeur3'}
```

3.3 Itération sur les dictionnaires

Vous pouvez itérer sur les clés, les valeurs ou les paires clé-valeur d'un dictionnaire en utilisant des boucles `for`. Par défaut, itérer sur un dictionnaire parcourt ses clés. Il est aussi possible d'utiliser les méthodes `keys()`, `values()` et `items()` pour obtenir respectivement les clés, les valeurs et les paires clé-valeur.

```python
mon_dictionnaire = {'clé1': 'valeur1', 'clé2': 'valeur2'}

# Itération sur les clés
for cle in mon_dictionnaire:
    print(cle)

# Autre façon d'itérer sur les clés
for cle in mon_dictionnaire.keys():
    print(cle)

# Itération sur les valeurs
for valeur in mon_dictionnaire.values():
    print(valeur)

# Itération sur les paires clé-valeur
for cle, valeur in mon_dictionnaire.items():
    print(cle, valeur)
```

3.4 Applications des dictionnaires

Les dictionnaires sont très utiles pour représenter des données structurées, comme par exemple des informations sur des personnes, des produits, etc.

```python
# Représentation d'une personne
personne1 = {
    'nom': 'Martin',
    'prénom': 'Claire',
    'âge': 25,
}

personne2 = {
    'nom': 'Dupont',
    'prénom': 'Jean',
```

```
    'âge': 30,
}
```

Dans ce genre d'utilisation, chaque entrée du dictionnaire représente un attribut de l'object (la personne, le produit, etc.), avec la clé correspondant au nom de l'attribut et la valeur correspondant à la valeur de l'attribut.

Les dictionnaires sont aussi très utiles pour associer des données à des clés qui proviennent d'une collection de choses, comme par exemple par associer un numéro de téléphone à des noms, ou une population à des villes.

```
# Dictionnaire associant des noms à des numéros de téléphone
annuaire = {
    'Alice': '079 123 45 67',
    'Bob': '078 234 56 78',
}

# Dictionnaire associant des villes à leur population
populations = {
    'Berne': 146348,
    'Zurich': 448664,
    'Genève': 203856,
}
```

Les dictionnaires sont aussi très utilisés aussi lors de calculs, par exemple pour compter des occurrences d'éléments dans une liste.

```
# Compter les occurrences de chaque fruit dans une liste
fruits = ['pomme', 'banane', 'orange', 'pomme', 'orange', 'banane', 'pomme']
compteur = {}
for fruit in fruits:
    if fruit in compteur:
        compteur[fruit] += 1
    else:
        compteur[fruit] = 1
print(compteur)  # Affiche {'pomme': 3, 'banane': 2, 'orange': 2}
```

Contenu avancé

Les dictionnaires sont aussi très utilisés comme *caches* pour mémoriser des résultats de calculs coûteux. Voici un exemple simple :

```
# Fonction pour calculer le n-ième nombre de Fibonacci
cache = {}
def fibonacci(n):
    if n in cache:
        return cache[n]
    if n <= 1:
        return n
    result = fibonacci(n - 1) + fibonacci(n - 2)
    cache[n] = result
    return result

print(fibonacci(500))
```

Dans cet exemple, le dictionnaire `cache` est utilisé pour mémoriser les résultats des appels précédents à la fonction `fibonacci`. Cela permet d'éviter de recalculer plusieurs fois les mêmes valeurs, ce qui améliore considérablement les performances de la fonction.

4 Compréhensions de listes et de dictionnaires

Les compréhensions de listes et de dictionnaires sont des syntaxes concises pour créer des listes ou des dictionnaires en Python.

4.1 Compréhensions de listes

Une compréhension de liste permet de créer une nouvelle liste en appliquant une expression à chaque élément d'une séquence existante. Mais le mieux est peut-être de voir un exemple !

```
# Création d'une liste des carrés des nombres de 0 à 9
carrés = [x**2 for x in range(10)]
print(carrés)  # Affiche [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Il est aussi possible d'ajouter une condition pour filtrer les éléments de la séquence.

```
# Création d'une liste des carrés des nombres pairs de 0 à 9
carrés_pairs = [x**2 for x in range(10) if x % 2 == 0]
print(carrés_pairs)  # Affiche [0, 4, 16, 36, 64]
```

Les compréhensions de listes sont souvent plus concises et lisibles que les boucles traditionnelles pour créer des listes. Comparez par exemple avec la version utilisant une boucle `for` :

```
# Version avec compréhension de liste
puissances_de_2 = [2**x for x in range(10)]
print(puissances_de_2)  # Affiche [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]

# Version avec boucle for
puissances_de_2 = []
for x in range(10):
    puissances_de_2.append(2**x)
print(puissances_de_2)  # Affiche [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

Bien entendu, l'utilisation de `range` dans les exemples ci-dessus n'est pas obligatoire. On peut utiliser n'importe quelle séquence, comme une liste ou une chaîne de caractères.

```
# Création d'une liste des lettres en majuscules dans une chaîne de caractères
chaine = "Quelle Belle Journée"

print([c.lower() for c in chaine if c.isupper()])  # Affiche ['q', 'b', 'j']
```

Les compréhensions de listes sont souvent utilisées en conjonction avec les différentes fonctions de la bibliothèque standard de Python pour manipuler (`enumerate`, `zip`, etc.) ou réduire des séquences (`sum`, `all`, `any`, etc.). Le code résultant est souvent très concis et lisible.

```
prix = [20, 10, 5, 15, 30]
quantites = [2, 0, 1, 4, 3]

total = sum([p * q for p, q in zip(prix, quantites)])
print(total)  # Affiche 195
```

Contenu avancé

Les compréhensions de listes peuvent également être imbriquées pour créer des listes à partir de plusieurs séquences. Voici un exemple :

```
# Création d'une liste de tous
# la chaine de caractères i + j
# avec i dans ["0", "1", "2"]
# et j dans ['A', 'B']
produits = [
    str(i) + j
    for i in range(3)
    for j in "AB"
]
print(produits)  # Affiche ['0A', '0B', '1A', '1B', '2A', '2B']
```

Il est aussi possible d'ajouter des conditions dans les compréhensions imbriquées.

```
# Création d'une liste de tous
# les couples (i, j)
# avec i dans [0, 1, 2]
# et j dans [0, 1, 2]
# où i + j est pair
somme_paires = [
    (i, j)
    for i in range(3)
    for j in range(3)
    if (i + j) % 2 == 0
]
print(somme_paires)  # Affiche [(0, 0), (0, 2), (1, 1), (2, 0), (2, 2)]

# Création d'une liste de tous
# les couples (i, j)
# avec i dans [0, 1, 2]
# où i est pair,
# et j dans [0, 1, 2]
# où j est pair
deux_pairs = [
    (i, j)
    for i in range(3)
    if i % 2 == 0
    for j in range(3)
    if j % 2 == 0
]
print(deux_pairs)  # Affiche [(0, 0), (0, 2), (2, 0), (2, 2)]
```

4.2 Compréhensions de dictionnaires

Les compréhensions de dictionnaires permettent de créer des dictionnaires de manière concise, similaire aux compréhensions de listes.

Voici un exemple :

```
# Création d'un dictionnaire des carrés des nombres de 0 à 3
carrés_dict = {x: x**2 for x in range(4)}
print(carrés_dict)  # Affiche {0: 0, 1: 1, 2: 4, 3: 9}
```

Tout comme pour les compréhensions de listes, il est possible d'ajouter une condition pour filtrer les éléments.

```
# Création d'un dictionnaire des carrés des nombres pairs de 0 à 3
carrés_pairs_dict = {x: x**2 for x in range(4) if x % 2 == 0}
print(carrés_pairs_dict)  # Affiche {0: 0, 2: 4}
```

La compréhensions de dictionnaires permettent de facilement convertir des listes de paires clé-valeur en dictionnaires.

```
cles = ['a', 'b', 'c']
valeurs = [1, 2, 3]
cles_valeurs = zip(cles, valeurs)
dictionnaire = {k: v for k, v in cles_valeurs}
print(dictionnaire)  # Affiche {'a': 1, 'b': 2, 'c': 3}
```

Notes de cours
Semaine 6

Cours Turing

Introduction

Pour cette sixième semaine, nous allons aborder le concept de *récursivité* en programmation. La récursivité est un phénomène où une entité se définit en termes d'elle-même. Le concept peut sembler abstrait au premier abord, mais nous allons l'aborder de manière progressive avec des exemples concrets.

En programmation, la récursivité se manifeste principalement à travers les fonctions récursives et les structures de données récursives. Nous verrons également comment la récursivité est utilisée dans des techniques algorithmiques telles que *diviser pour régner*.

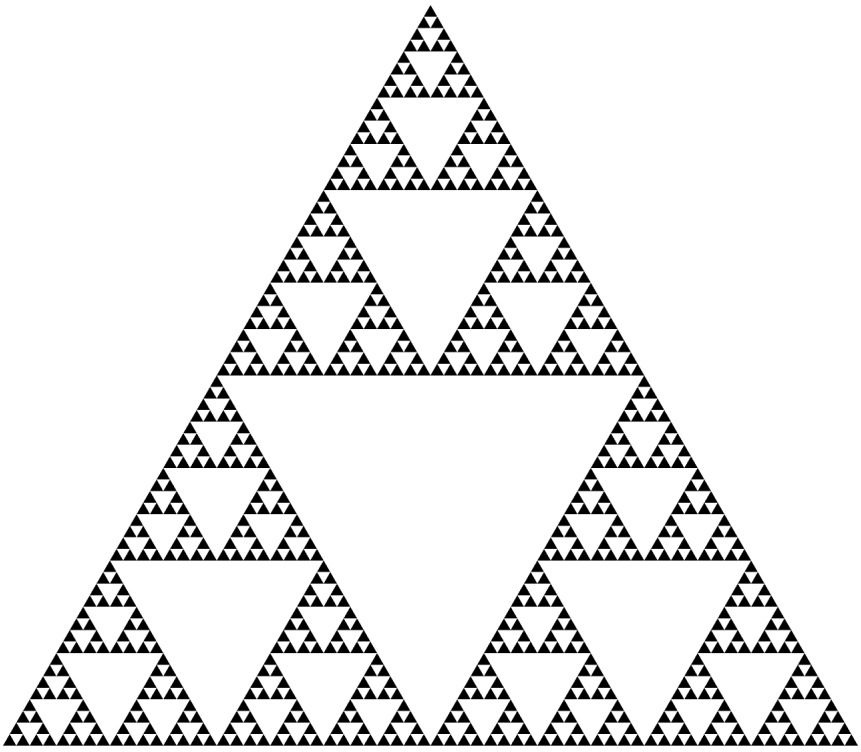


FIGURE 1 – Exemple de structure récursive : le triangle de Sierpiński

1 Fonctions récursives

On appelle une fonction *récursive* une fonction qui contient des appels à elle-même dans son corps. La récursivité est une technique de programmation puissante qui permet parfois d'implémenter des algorithmes de manière plus simple et plus élégante que les approches itératives (utilisant des boucles).

1.1 Exemple : compte à rebours

Par exemple, ci-dessous un exemple simple de fonction récursive en Python qui affiche un compte à rebours :

```python
def compte_a_rebours(n: int):
    if n == 0:
        print("Décollage !")
    else:
        print(n)
        compte_a_rebours(n - 1)
```

Dans cet exemple, la fonction `compte_a_rebours` s'appelle elle-même avec un argument décrémenté jusqu'à atteindre le cas où n est égal à 0.

On appelle **cas de base** les cas où la fonction ne s'appelle pas elle-même, ce qui permet d'éviter de faire des appels récursifs infinis. Les cas où la fonction s'appelle elle-même sont appelés **cas récursifs**.

Notez que nous aurions aussi pu, dans cet exemple, utiliser une boucle pour réaliser le compte à rebours.

```python
def compte_a_rebours_boucle(n: int):
    while n != 0:
        print(n)
        n -= 1
    print("Décollage !")
```

1.2 Exemple : factorielle

Un autre exemple classique de fonction récursive est la définition de la fonction factorielle. Pour un entier naturel n, la factorielle de n, notée $n!$, est définie comme le produit de tous les entiers positifs inférieurs ou égaux à n.

L'opération peut être définie de manière *récursive* avec les deux cas suivants :

— **Cas de base :** La factorielle de 0 est 1, c'est-à-dire :

$$0! = 1$$

.

— **Cas récursif :** Pour tout entier naturel $n > 0$, la factorielle de n est donnée par :

$$n! = n \cdot (n-1)!$$

En Python, la fonction factorielle peut être définie de manière récursive comme suit :

```
def factorielle(n: int) -> int:
    if n == 0:
        return 1
    else:
        return n * factorielle(n - 1)
```

Dans l'exemple ci-dessus, la fonction `factorielle` s'appelle elle-même pour calculer la factorielle de `n - 1`. Contrairement à l'exemple du compte à rebours, le retour de la fonction est utilisé dans le calcul du résultat final.

Tout comme l'exemple précédent, il est possible de définir la fonction factorielle sans utiliser la récursivité, en utilisant une boucle à la place :

```
def factorielle_boucle(n: int) -> int:
    resultat = 1
    for i in range(1, n + 1):
        resultat *= i
    return resultat
```

1.3 Exemple : suite de Fibonacci

La suite de Fibonacci est une séquence de nombres où chaque nombre est la somme des deux précédents. La suite commence généralement par les nombres 0 et 1. Ainsi, les premiers termes de la suite de Fibonacci sont :

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

La suite de Fibonacci peut être définie de manière récursive avec les deux cas suivants :

— **Cas de base :**

$$F(0) = 0$$

$$F(1) = 1$$

— **Cas récursif :** Pour tout entier naturel $n \geq 2$,

$$F(n) = F(n-1) + F(n-2)$$

En Python, la fonction pour calculer le n-ième terme de la suite de Fibonacci de manière récursive peut être définie comme suit :

```
def fibonacci(n: int) -> int:
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Nous avons ici deux cas de base (pour $n = 0$ et $n = 1$) et un cas récursif pour les autres valeurs de n.

Contenu avancé

Il est important de noter que cette implémentation récursive de la suite de Fibonacci n'est pas très efficace pour de grandes valeurs de n, car elle entraîne de nombreux appels redondants. Il existe cependant des méthodes afin de regagner en efficacité. Nous en parlerons plus en détail plus bas dans le document.

1.4 Structure d'une fonction récursive

Une fonction récursive traite généralement deux types de cas en fonction de son ou ses arguments :

- **Cas de base :** On appelle *cas de base* les cas où la fonction ne s'appelle pas elle-même, où elle retourne une valeur simple ou effectue une action simple. Ces cas permettent de terminer la récursion.
- **Cas récursifs :** On appelle *cas récursifs* les cas où la fonction s'appelle elle-même avec des arguments modifiés (généralement plus simples ou plus petits) pour progresser vers un cas de base. Le nombre d'appels récursifs dépend de la nature du problème et de la manière dont les arguments sont modifiés.

Pour que les appels récursifs se terminent correctement, il est crucial que chaque suite d'appels récursifs se termine par un cas de base. Très souvent, cela implique qu'un ou plusieurs arguments de la fonction soient modifiés à chaque appel récursif pour se rapprocher du cas de base.

1.5 Comparaison entre récursivité et itération

La récursivité (utilisation de fonctions récursives) et l'itération (utilisation de boucles) sont deux approches différentes qui peuvent être utilisées de manière interchangeable pour résoudre certains problèmes. En effet, les deux techniques ont la même puissance expressive, ce qui signifie que tout algorithme pouvant être implémenté de manière itérative peut également être implémenté de manière récursive, et vice versa. Cependant, certains problèmes se prêtent mieux à l'une ou l'autre approche en fonction de leur nature.

Au niveau des performances, les fonctions récursives peuvent parfois être moins efficaces que les boucles en raison de la surcharge liée aux appels de fonction et à la gestion de la pile d'appels.

La récursivité reste néanmoins une technique puissante et élégante pour résoudre des problèmes qui ont une structure naturellement récursive, comme nous allons le voir dans les prochaines sections.

Contenu avancé

Parfois, la structure récursive d'un problème amène à des appels répétés aux mêmes sous-problèmes, ce qui peut entraîner une inefficacité. Dans de tels cas, des techniques comme la *mémoïsation* ou la *programmation dynamique* peuvent être utilisées pour optimiser les performances en stockant les résultats des sous-problèmes déjà résolus.
Un exemple classique est le calcul des nombres de Fibonacci, où une approche naïve récursive peut entraîner une explosion du nombre d'appels.

```
def fibonacci(n: int) -> int:
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Un appel à `fibonacci(5)` entraînera les appels suivants :
- `fibonacci(5)` appelle `fibonacci(4)` et `fibonacci(3)`
- `fibonacci(4)` appelle `fibonacci(3)` et `fibonacci(2)`
- `fibonacci(3)` appelle `fibonacci(2)` et `fibonacci(1)`
- `fibonacci(2)` appelle `fibonacci(1)` et `fibonacci(0)`

En comptabilisant les appels, on obtient :
- `fibonacci(5)` : 1 appel
- `fibonacci(4)` : 1 appel
- `fibonacci(3)` : 2 appels
- `fibonacci(2)` : 3 appels
- `fibonacci(1)` : 5 appels
- `fibonacci(0)` : 3 appels

Le nombre total d'appels est donc de 15 pour calculer `fibonacci(5)`.
Grâce à la mémoïsation, on peut améliorer considérablement l'efficacité de cette fonction.

```
memo = {}  # Dictionnaire pour stocker les résultats déjà calculés.
def fibonacci_memo(n: int) -> int:
    if n in memo:  # On vérifie si le résultat est déjà calculé.
        return memo[n]
    if n <= 1:
        result = n
    else:
        result = fibonacci_memo(n - 1) + fibonacci_memo(n - 2)
    memo[n] = result
    return result
```

Avec cette version *mémoïsée*, chaque nombre de Fibonacci n'est calculé qu'une seule fois, réduisant ainsi le nombre total d'appels à `fibonacci_memo(n)` à `n + 1`.

2 Diviser pour régner

La récursivité est particulièrement utile pour résoudre des problèmes qui peuvent être décomposés en sous-problèmes plus petits et similaires. On appelle cette technique d'algorithmique *diviser pour régner*. L'idée principale de cette approche est de diviser un problème en plusieurs sous-problèmes plus petits, de résoudre ces sous-problèmes de manière récursive, puis de combiner les résultats pour obtenir la solution au problème initial.

2.1 Exemple : recherche dichotomique

Un exemple classique de l'approche *diviser pour régner* est la recherche dichotomique. La recherche dichotomique permet de trouver un élément dans une liste triée en divisant continuellement la liste en deux moitiés et en se concentrant sur la moitié qui pourrait contenir l'élément recherché.

Voici une implémentation récursive de la recherche dichotomique en Python :

```
def recherche_dicho(liste: list[int], cible: int, debut: int, fin: int) -> int:
    if debut > fin:
        return None
    milieu = (debut + fin) // 2
    if liste[milieu] == cible:
        return milieu
    elif liste[milieu] < cible:
        return recherche_dicho(liste, cible, milieu + 1, fin)
    else:
        return recherche_dicho(liste, cible, debut, milieu - 1)
```

Dans cet exemple, la fonction `recherche_dichotomique` divise la liste en deux moitiés à chaque appel récursif, et se concentre sur la moitié qui pourrait contenir l'élément recherché.

2.2 Exemple : rendu de monnaie

Un autre exemple d'algorithme *diviser pour régner* est le problème du rendu de monnaie. Le but est de calculer combien de façons différentes on peut rendre une somme donnée avec un ensemble de pièces de monnaie de différentes valeurs. Par exemple, pour rendre 5 francs avec des pièces de 1 franc et 2 francs, on peut le faire de plusieurs façons :

— 5 pièces de 1 franc
— 3 pièces de 1 franc et 1 pièce de 2 francs
— 1 pièce de 1 franc et 2 pièces de 2 francs

Soit un total de 3 façons de rendre 5 francs avec des pièces de 1 franc et 2 francs.

L'idée pour résoudre ce problème est de réfléchir de manière récursive. Commençons par identifier les cas de base : les cas où la réponse est immédiate.

— Si la somme à rendre est 0, il y a exactement une façon de le faire : ne rien rendre.
— Si par contre la somme à rendre est négative ou si nous n'avons plus de pièces disponibles, il n'y a aucune façon de rendre cette somme.

Vient ensuite le cas récursif. Dans ce cas, nous avons une somme strictement positive à rendre et au moins une pièce disponible. Nous avons alors deux options :

— Utiliser la première pièce disponible pour rendre une partie de la somme, puis résoudre le problème pour la somme restante avec toutes les pièces disponibles.

— Ne pas utiliser la première pièce, et résoudre le problème pour la même somme mais avec les pièces restantes (sans la première pièce).

En combinant les résultats de ces deux options, nous obtenons le nombre total de façons de rendre la somme.

Voici une implémentation récursive de cette approche en Python :

```python
def rendu_de_monnaie(somme: int, pieces: list[int]) -> int:
    # Cas de base.
    if somme == 0:
        return 1  # Une façon de rendre la somme 0 (ne rien faire).
    if somme < 0 or not pieces:
        return 0  # Pas de façon de rendre une somme négative ou sans pièces.

    # Cas récursif.
    # Soit on utilise la première pièce.
    avec_piece = rendu_de_monnaie(somme - pieces[0], pieces)
    # Soit on n'utilise pas la première pièce.
    sans_piece = rendu_de_monnaie(somme, pieces[1:])
    return avec_piece + sans_piece
```

3 Structures de données récursives

Dans les deux sections précédentes, nous avons vu la notion de fonctions récursives et l'approche algorithmique *diviser pour régner*. Il s'agit en quelque sorte du pendant procédural de la récursivité. La récursivité peut se retrouver également dans le monde des données, et non plus seulement des calculs, à travers les *structures de données récursives*.

Une structure de données est dite *récursive* si elle est définie en termes d'elle-même. Cela signifie qu'une instance de la structure peut contenir des références à d'autres instances de la même structure.

Les structures de données récursives sont particulièrement utilisées pour représenter des données hiérarchiques ou imbriquées.

Exemple

Si vous avez déjà travaillé avec des documents HTML (pour les pages web), vous avez déjà manipulé une structure de données récursive. En effet, un document HTML est constitué d'éléments qui peuvent contenir d'autres éléments, formant ainsi une hiérarchie d'éléments imbriqués.

3.1 Exemple : listes chaînées

La liste chaînée est l'exemple classique de structure de données récursive, et peut-être l'exemple le plus simple. Une liste chaînée est une collection d'éléments, où chaque élément (appelé *nœud*) contient une valeur et une référence vers le nœud suivant dans la liste. Une liste vide est représentée par une valeur nulle, comme par exemple `None` en Python.

En Python, on pourrait utiliser un tuple de deux éléments pour représenter un nœud d'une liste chaînée non-vide et None pour représenter une liste vide. Voici une définition simple d'une liste chaînée en Python, encodée de cette manière :

```
ma_liste_chaînée = (1, (2, (3, None)))
```

Dans cet exemple, la liste chaînée contient les éléments 1, 2 et 3. Le premier élément est un tuple contenant la valeur 1 et une référence vers le nœud suivant, qui est lui-même un tuple contenant la valeur 2 et une référence vers le nœud suivant, et ainsi de suite jusqu'à ce que le dernier nœud référence `None`.

Pour manipuler une liste chaînée, on utilise souvent des fonctions récursives. Par exemple, voici une fonction récursive pour calculer la somme des éléments d'une liste chaînée :

```
def somme_liste_chaînée(liste: tuple | None) -> int:
    if liste is None:
        return 0
    else:
        valeur, reste = liste
        return valeur + somme_liste_chaînée(reste)
```

Dans cet exemple, la fonction `somme_liste_chaînée` s'appelle elle-même pour calculer la somme des éléments restants de la liste chaînée.

Contenu avancé

Le type qui correspond à une liste chaînée d'entiers peut être défini en Python (à partir de la version 3.12) comme suit :

```
type LinkedList[T] = tuple[T, LinkedList[T]] | None
```

Ici, `LinkedList[T]` est un type générique qui peut contenir des éléments de type `T`. Une liste chaînée est soit un tuple contenant une valeur de type `T` et une référence vers une autre liste chaînée de type `LinkedList[T]`, soit `None` pour représenter une liste vide.

3.2 Exemple : arbres

Une autre structure de données récursive couramment utilisée est l'*arbre*. Alors qu'un nœud d'une liste chaînée ne peut avoir qu'un seul nœud enfant (le nœud suivant), un nœud d'un arbre peut avoir plusieurs nœuds enfants. Un arbre est constitué de nœuds, où chaque nœud contient une valeur et une *liste* de références vers ses nœuds enfants. Un arbre vide est souvent représenté par une valeur nulle, comme `None` en Python.

Voici un exemple simple d'un arbre en Python :

```
mon_arbre = (1, [(2, []), (3, [(4, [])])])
```

Dans cet exemple, l'arbre a pour *racine* le nœud avec la valeur 1. Ce nœud a deux enfants : le nœud avec la valeur 2 (qui n'a pas d'enfants) et le nœud avec la valeur 3 (qui a un enfant, le nœud avec la valeur 4).

Comme pour les listes chaînées, on utilise souvent des fonctions récursives pour manipuler des arbres. Par exemple, voici une fonction récursive pour calculer la somme des valeurs de tous les nœuds d'un arbre :

```
def somme_arbre(arbre: tuple | None) ->
    if arbre is None:
        return 0
    else:
        valeur, enfants = arbre
        sommes_enfants = [somme_arbre(enfant) for enfant in enfants]
        return valeur + sum(sommes_enfants)
```

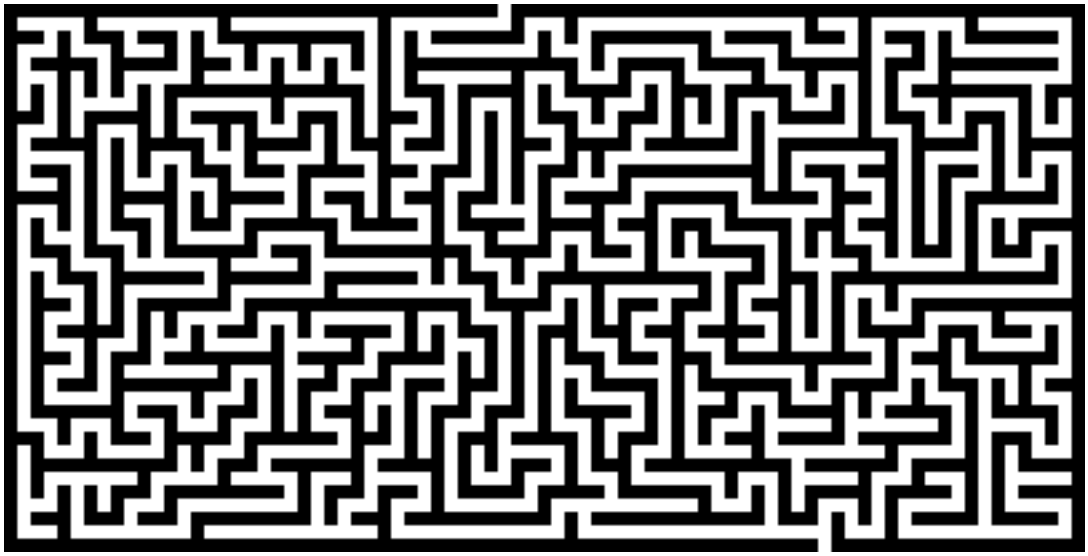
Les arbres sont très utiles pour représenter des données hiérarchiques, comme par exemple les systèmes de fichiers et dossiers, ou les structures de documents XML ou JSON. On retrouve également des structures arborescentes en dehors de l'informatique, comme par exemple les arbres phylogénétiques en biologie ou les organisations hiérarchiques dans les entreprises. Les arbres permettent de modéliser efficacement ces relations hiérarchiques.

Notes de cours
Semaine 7

Cours Turing

Introduction

Cette semaine, nous allons voir comment, à l'aide de Python, générer des images de labyrinthes (pseudo)aléatoires telles que celle ci-dessous.



Ce projet, plus pratique, vous permettra d'appliquer les concepts vus ces dernières semaines ainsi que de découvrir une autre façon de manipuler des images en Python, assez différente de celle vue avec *PyTamaro*.

Dans un premier temps, nous allons nous intéresser à la bibliothèque *Pillow* et à son module `PIL`, un module qui nous permettra de créer et manipuler des images. Dans un second temps, nous aborderons la méthode de génération des labyrinthes.

1 Les images avec `PIL`

Le module `PIL`, de la bibliothèque *Pillow*, permet la création, la manipulation, la sauvegarde et l'affichage d'images *matricielles*. Pour le reste de cette section, nous allons découvrir les bases de Pillow. Cependant, le but de ce document n'est pas d'être exhaustif, mais simplement de fournir une introduction. Une documentation complète du module est disponible sur le site de la bibliothèque Pillow (en anglais).

Images matricielles Comme énoncé plus haut, Pillow permet la manipulation d'images matricielles. Les images matricielles sont des représentations d'images sous la forme d'une matrice de pixels. Une matrice est simplement un tableau, une grille, à deux dimensions. Les éléments de cette matrice sont ce qu'on appelle des pixels (pour *picture element*, littéralement *élément d'image* en anglais). Chaque pixel a donc une position dans l'image et indique la couleur à cet emplacement.

Code couleur Pour indiquer la couleur d'un pixel, on utilisera un *code couleur*. Les différents codes couleurs permettent d'identifier, de décrire, un ensemble de couleurs. Il en existe plusieurs, mais celui que nous allons utiliser est le code RGB (pour *red, green, blue*, soit *rouge, vert, bleu* en anglais). En RGB, on décrit une couleur en fonction de l'intensité de ses composantes de rouge, de vert et de bleu. Chaque composante a une valeur entre 0 et 255. Ainsi, chaque couleur représentable est associée à un triplet de valeurs. On aura par exemple $(0, 0, 0)$ pour le noir, et $(255, 255, 255)$ pour le blanc. Le rouge sera $(255, 0, 0)$, alors que le bleu $(0, 0, 255)$. Quant au code $(185, 237, 234)$, il correspond à un joli bleu ciel. Ce code couleur permet de représenter $256^3 = 16'777'216$ de couleurs différentes. Il existe de nombreux outils, en ligne ou à installer, qui permettent de sélectionner des couleurs et d'en obtenir le code RGB.

1.1 Import du module

Pour avoir accès à ce module `PIL` dans un programme Python, il faut l'*importer*. Pour importer un module, on utilise le mot-clé `import`. Dans notre cas précis, nous sommes uniquement intéressés par une unique définition : la *class* Image. Pour cela, on utilisera l'instruction suivante :

```
from PIL import Image
```

Les instructions d'import de modules sont généralement situées au tout début du fichier. Cela permet de facilement retrouver quels sont les modules utilisés par un programme et ses éventuelles dépendances à d'autres fichiers.

Installation de Pillow Si l'exécution de l'instruction donne lieu à une erreur de type `ModuleNotFoundError`, c'est que la bibliothèque *Pillow* n'est pas installée. En principe, la bibliothèque Pillow est déjà pré-installée dans l'environnement de programmation proposé et cette étape d'installation n'est pas nécessaire.

Pour installer la bibliothèque Pillow, il suffit de passer par le gestionnaire de paquets de Python, appelé `pip`. Pour cela, on exécutera la commande suivante dans un terminal :

```
pip install Pillow
```

Suivant votre installation Python, il se peut que l'utilitaire `pip` soit connu sous un autre nom, comme `pip3` par exemple. Dans ce cas, il vous faudra juste indiquer le bon nom à la place de `pip` dans la commande indiquée ci-dessus. Une fois la bibliothèque Pillow installée, le module `PIL` devrait être accessible depuis Python.

1.2 Création d'une image

Pour créer une image, on utilise la méthode `new` de la classe `Image`, comme suit :

```
largeur = 400  # Largeur en pixels
hauteur = 200  # Hauteur en pixels
couleur_fond = (181, 31, 31)  # Rouge groseille

# Création d'une image
mon_image = Image.new("RGB", (largeur, hauteur), couleur_fond)
```

La méthode `new` prend trois arguments :

1. Premièrement, un mode de couleur. Nous utiliserons ici toujours le mode RGB.
2. Deuxièmement, les dimensions de l'image, sous la forme d'un tuple de deux éléments (largeur et hauteur).
3. Finalement, et de façon optionnelle, le code d'une couleur (en mode RGB) pour le fond de l'image.

La méthode retourne comme valeur l'image nouvellement créée. Pour être précis, la valeur retournée est ce que l'on appelle une *instance* de la classe `Image`.

1.3 Ouverture d'un fichier

Il est aussi possible d'ouvrir une image depuis un fichier. Pour cela, on utilisera la méthode `open` de `Image`.

```
# Lecture d'une image
mon_image = Image.open("nom_du_fichier.ext")
```

Pour s'assurer que l'image soit bien en mode RGB, on peut suivre l'ouverture de l'image par un appel à la méthode `convert`, comme ceci :

```
# Ouverture et conversion d'une image
mon_image = Image.open("nom_du_fichier.ext").convert("RGB")
```

En effet, certaines images, comme les images avec de la transparence ou certaines images en noir et blanc, n'utilisent pas le mode RGB pour les couleurs. Avec l'appel à `convert`, on s'assure que les couleurs seront bien représentées selon le mode RGB.

1.4 Affichage d'une image

On utilisera la méthode `show` pour afficher une image.

```
# Affichage d'une image
mon_image.show()
```

Notez que cette méthode ouvre l'image dans un visualiseur d'images externe. Dans l'environnement de programmation proposé, un bug fait que l'image ne s'affiche pas et qu'une erreur est levée. Pour visualiser votre image, faites plutôt appel à la méthode `save` pour enregistrer l'image dans un fichier, comme expliqué ci-dessous.

1.5 Sauvegarde d'une image

On utilisera la méthode `save` pour enregistrer une image.

```
# Sauvegarde d'une image dans un fichier
mon_image.save("mon_nom_de_fichier.png")
```

Notez qu'on utilisera généralement l'extension `png` pour sauvegarder nos images. Il est possible d'utiliser d'autres formats de fichiers, mais parfois ces formats utiliseront des techniques de compression de données avec pertes qui réduisent la qualité de l'image.

1.6 Dimensions d'une image

La largeur et la hauteur d'une image sont accessibles via les champs `width` et `height`. Il est aussi possible d'utiliser la méthode `size` afin d'obtenir un tuple de la largeur et de la hauteur.

```
print(mon_image.width)  # Affiche la largeur
print(mon_image.height)  # Affiche la hauteur
print(mon_image.size)  # Affiche la paire (largeur, hauteur)
```

1.7 Lecture d'un pixel

On utilisera la méthode `getpixel` afin d'obtenir la couleur d'un pixel de l'image. La méthode prend en unique argument la position du pixel dans l'image sous la forme d'une paire (x, y). La composante x indique la position sur l'axe horizontal, en nombre de pixels depuis la gauche. La composante y indique la position sur l'axe vertical, en nombre de pixels depuis le haut (et non le bas).

```
# Couleur du pixel en haut à gauche
couleur_pixel = mon_image.getpixel((0, 0))

 # Couleur du pixel en haut à droite
couleur_pixel = mon_image.getpixel((mon_image.width-1, 0))
```

La position du pixel en haut à gauche de l'image est $(0, 0)$. Comme l'on compte à partir de 0 sur les deux axes, la valeur maximale pour la position sur l'axe horizontal est la largeur de l'image *moins 1* et la valeur maximale sur l'axe vertical la hauteur *moins 1*.

Souvent, il est utile de décomposer une couleur en ses trois composantes. Pour cela, on utilise généralement en Python la syntaxe de déconstruction de tuple :

```
couleur_pixel = mon_image.getpixel((x, y))
rouge, vert, bleu = couleur_pixel

# Ou plus directement...
rouge, vert, bleu = mon_image.getpixel((x, y))
```

1.8 Écriture d'un pixel

La méthode `putpixel` permet de modifier la couleur d'un pixel de l'image. La méthode prend deux arguments : la position et la couleur.

```
couleur = (0, 0, 0)  # Noir
mon_image.putpixel((x, y), couleur)
```

Notez que la méthode modifie l'image et donc que les images telles qu'implémentées par le module `PIL` sont *mutables*.

1.9 Redimensionner une image

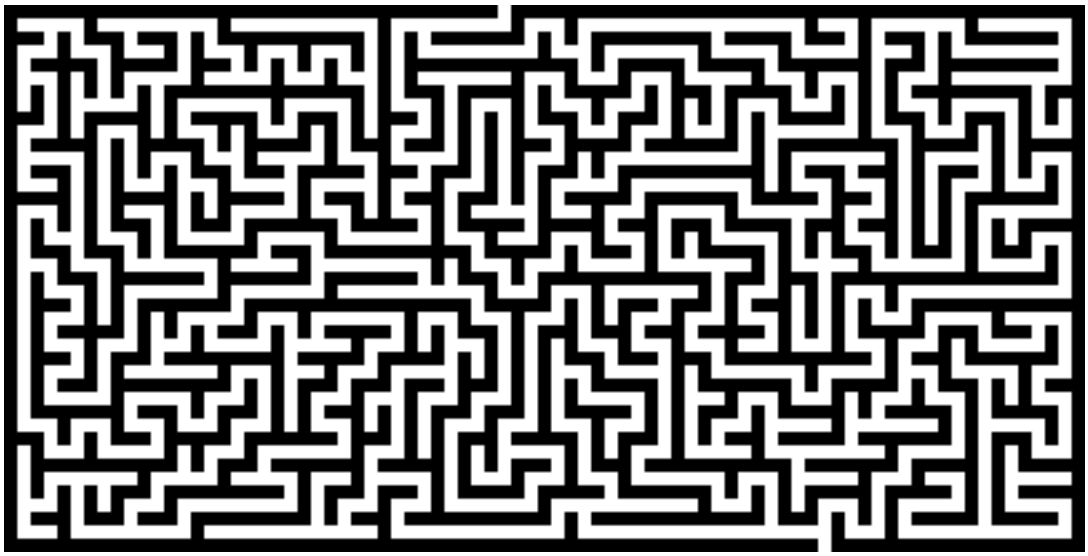
La méthode `resize` permet d'obtenir la copie d'une image aux dimensions données.

```
# Zoom par 5 dans une image
largeur = mon_image.width * 5
hauteur = mon_image.height * 5
ma_copie = mon_image.resize((largeur, hauteur), Image.BOX)
```

Notez que la méthode `resize` ne modifie pas l'image mais en retourne une copie. L'argument `Image.BOX` permet de faire en sorte de ne pas flouter l'image lors du redimensionnement. En effet, par défaut, lors du redimensionnement une méthode est utilisée pour éviter l'aspect net voire *pixelisé* de l'image agrandie. Or, dans notre cas, nos cherchons à préserver cette apparence très nette.

2 Génération d'un labyrinthe

Nous allons maintenant nous intéresser à la création d'un générateur de labyrinthes en Python. Le labyrinthe sera constitué de chemins blancs et de murs noirs, comme ci-dessous :



Le but de ce mini-projet sera de créer un programme qui permettra à l'utilisateur de spécifier des dimensions et d'obtenir un labyrinthe de la taille appropriée sous la forme d'une image. Pour ce mini-projet, nous allons représenter les labyrinthes à l'aide d'une image de Pillow que vous pourrez sauvegarder dans un fichier.

Dans le but de simplifier votre implémentation, nous vous conseillons de séparer la partie interface avec l'utilisateur (demande des dimensions, affichage/sauvegarde de l'image) de la logique de création du labyrinthe. Pour cette dernière, nous vous conseillons d'implémenter une fonction nommée `creer_labyrinthe` qui prendra en entrée les dimensions du labyrinthe et qui retournera une image. L'image retournée contiendra le labyrinthe. Dans cette image, la largeur des murs et des couloirs sera de 1 pixel. Vous serez ensuite libre de redimensionner l'image avant de l'afficher ou de l'enregistrer.

```
def creer_labyrinthe(largeur, hauteur):
    pass
```

Cette fonction sera chargée, comme son nom l'indique, de créer le labyrinthe de toutes pièces. Le processus de création du labyrinthe sera découpé en quatre étapes. Lors de la première étape, nous allons créer le labyrinthe entièrement noir. Dans une deuxième étape, nous allons y établir des petites pièces séparées, entièrement entourées de murs. Dans un troisième temps, nous allons retirer des murs du labyrinthe, de façon aléatoire, afin de s'assurer que toutes les pièces du labyrinthe sont connectées entre elles. Finalement, nous ajouterons une entrée et une sortie au labyrinthe.

2.1 Création de l'image noire

Initialement, nous allons créer un labyrinthe entièrement composé de murs. Pour cela, il vous faudra créer une image noire aux dimensions données. Assurez-vous avant toute chose que les dimensions données pour le labyrinthe soient bien les deux impaires. En effet, comme nous allons alterner entre mur et couloir, et qu'il y a un mur au début et à la fin, il est important que la largeur et la hauteur soient impaires. Si ce n'est pas le cas, vous pouvez lancer une erreur avec l'instruction suivante :

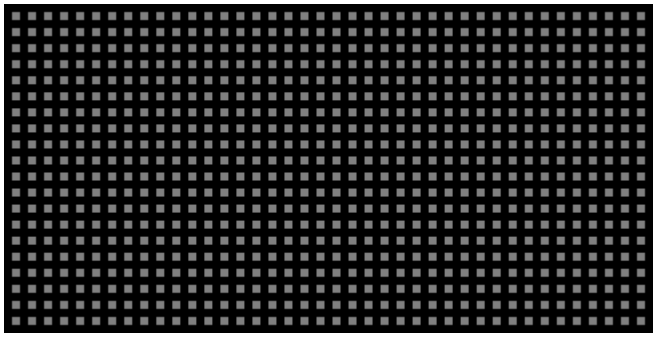
```
raise ValueError("Dimensions incorrectes")
```

Dans le code de votre interface, vous pourrez gérer le cas où l'utilisateur entre des dimensions incorrectes. À la fin de cette étape, le labyrinthe doit ressembler à un grand rectangle noir.

2.2 Création des pièces

Dans un second temps, nous allons peupler de pièces ce labyrinthe noir. Pour cela, nous allons colorer en gris tous les pixels de l'image à des positions (x, y) où x et y sont impaires. La couleur grise (par exemple $(128, 128, 128)$) indique que la pièce n'est pas encore connectée au reste du labyrinthe. La couleur sera ensuite changée à la prochaine étape.

À la fin de cette étape, le labyrinthe doit ressembler à ceci :



2.3 Connexion entre les pièces

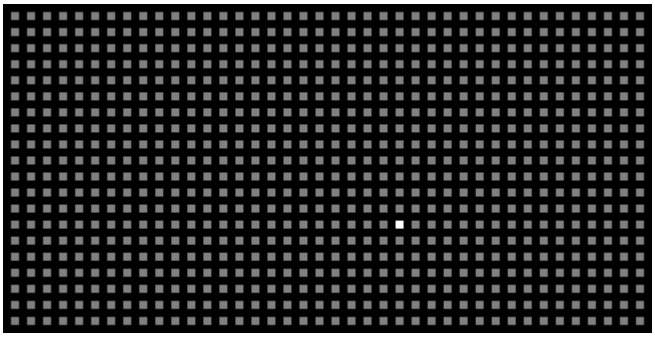
Arrive ensuite la partie principale de la génération du labyrinthe : la connexion des différentes pièces. Pour cela, sélectionnez aléatoirement une des pièces du labyrinthe comme position de départ.

2.3.1 Choix de la position de départ

Pour le choix de la pièce de départ, la fonction `randint` du module `random` pourra vous être utile. La fonction prend deux arguments, une borne inférieure et une borne supérieure (deux entiers) et retourne un nombre entier aléatoire entre ces deux bornes. Notez que les deux bornes sont inclusives.

Assurez-vous aussi de bien choisir une position de départ x et y où à la fois x et y sont impairs. Rappelez-vous que tous les nombres impairs peuvent être écrits sous la forme $2n + 1$, où n est un nombre entier.

Mémorisez cette position de départ et colorez-là en blanc dans l'image.

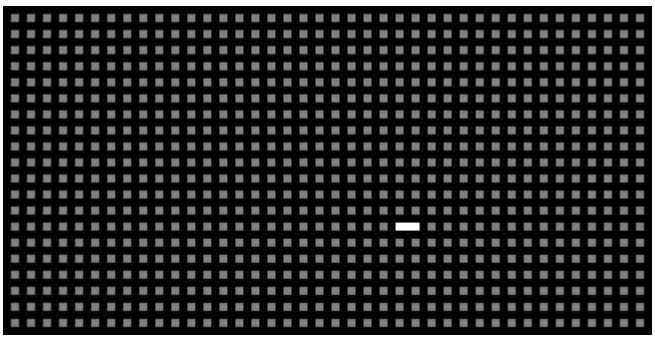


2.3.2 Déplacement à une pièce adjacente

Ensuite, depuis la pièce courante, sélectionnez au hasard une pièce adjacente non-connectée au reste du labyrinthe (une pièce grise), puis faites sauter le mur entre les deux pièces (en coloriant le mur et la nouvelle pièce en blanc).

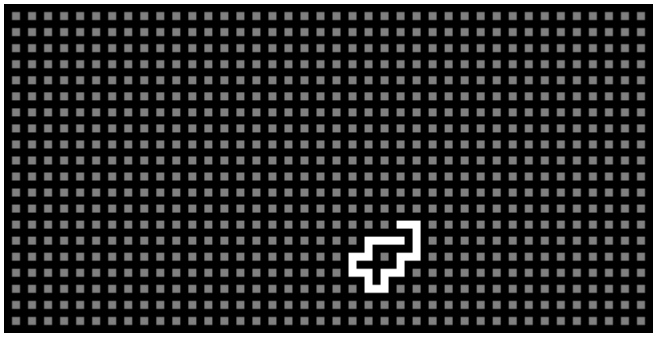
Pour cela, nous vous conseillons de concevoir une fonction Python qui prend en argument l'image et la position d'une pièce et qui retourne la position de toutes les pièces adjacentes de couleur grise. Attention à bien traiter le cas des positions en bordure du labyrinthe.

Une fois cette liste obtenue, la fonction `choice` du module `random` vous permettra de choisir une pièce adjacente au hasard pour la visiter. Le labyrinthe obtenu après un unique déplacement devrait ressembler à ceci :



2.3.3 Répétition

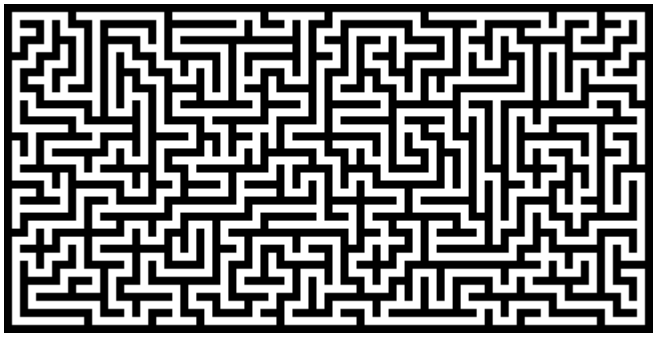
Ensuite, on répète le processus de sélection d'une pièce adjacente non-connectée, cette fois depuis la pièce nouvellement connectée. On enchaîne les répétitions, choisissant à chaque fois une nouvelle pièce voisine depuis la dernière pièce connectée, et ce jusqu'à ce que l'on arrive à une pièce où toutes les pièces adjacentes sont connectées (un cul-de-sac), comme dans l'image ci-dessous.



2.3.4 Rebrousser chemin

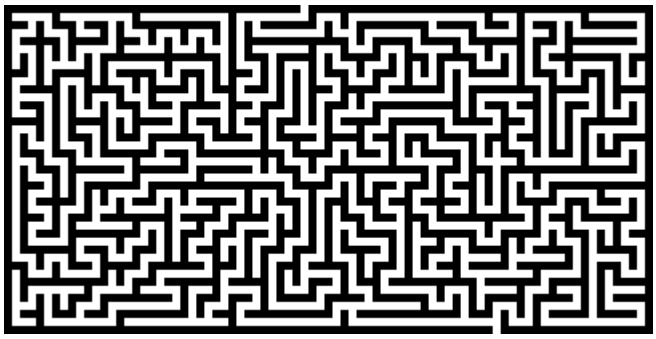
Lorsque l'on arrive à un cul-de-sac, il faudra rebrousser chemin. Pour cela, il vous faudra maintenir un chemin depuis la position de départ, par exemple sous la forme d'une liste Python. Il est aussi possible d'utiliser la récursivité (dans ce cas, la pile d'appels de fonctions tiendra le rôle du chemin).

Au départ, ce chemin sera vide, mais à chaque fois que l'on se déplace dans une nouvelle pièce, il faudra ajouter à la fin de ce chemin la position de la pièce que l'on quitte. Lorsque l'on arrive à un cul-de-sac, on se déplacera à la dernière pièce du chemin (la pièce d'où l'on était arrivé). Cette dernière sera alors retirée du chemin et l'on réessaiera le processus de sélection d'un voisin depuis cette pièce. La pièce aura certes déjà été visitée, mais elle aura peut-être encore des cellules adjacentes à visiter ! La génération du labyrinthe s'arrête lorsqu'il n'y a plus de pièces adjacentes non-connectées et que le chemin est vide. Le résultat à la fin de cette étape doit ressembler à ceci :



2.4 Ajouter une entrée et sortie

Finalement, on ajoutera une entrée sur le haut du labyrinthe ainsi qu'une sortie sur le bas du labyrinthe. La position sur l'axe horizontal est aléatoire mais doit permettre de connecter une pièce à l'extérieur du labyrinthe. Le résultat attendu est le suivant :

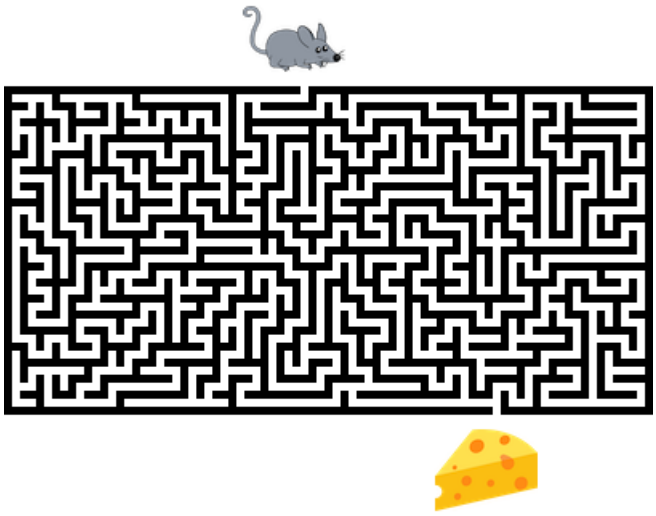


Une fois cette dernière étape implémentée, votre générateur de labyrinthe est terminé, félicitations !

2.5 Idées d'améliorations

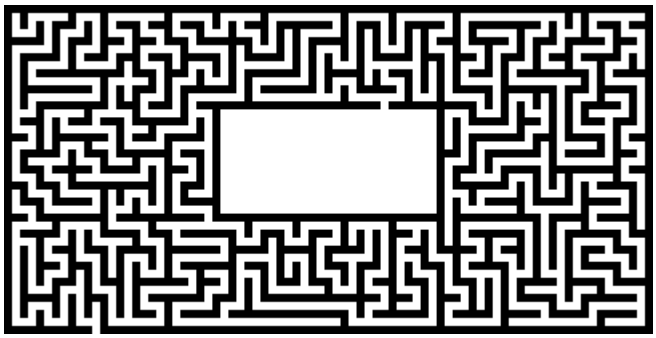
Si le coeur vous en dit, voici quelques idées d'améliorations pour votre générateur de labyrinthe :

Ajout d'images à l'entrée et à la sortie du labyrinthe Après avoir généré votre labyrinthe, vous pouvez y ajouter des éléments décoratifs à l'entrée et à la sortie. Par exemple, dans l'image ci-dessous, on a ajouté une souris à l'entrée et un fromage à la sortie.

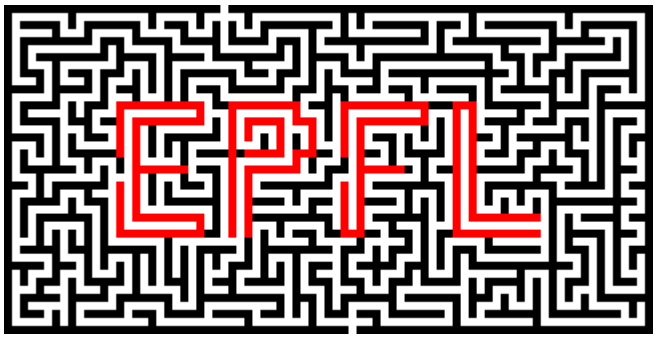


Notez que le module `PIL` dispose de fonctionnalités pour coller des images dans d'autres images. N'hésitez pas à consulter la documentation de la classe `Image`.

Zone vide et entrée au centre du labyrinthe Une variante pourrait être de faire commencer le labyrinthe en son centre, dans une plus grande zone vide, comme dans l'exemple ci-dessous.



Ajout de murs incassables De base, dans le labyrinthe, tous les murs internes sont susceptibles d'être supprimés. Une amélioration possible serait de pouvoir spécifier que certains murs sont incassables, en les colorant par exemple en rouge, comme dans l'exemple ci-dessous.



Reste le problème de savoir comment spécifier quels murs sont incassables !

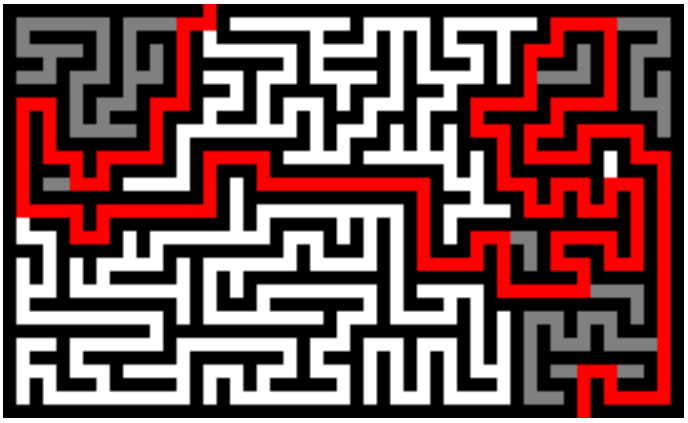
Notes de cours
Semaine 8

Cours Turing

1 Résolution de labyrinthes

La semaine dernière nous avons vu comment générer des labyrinthes sous forme d'images en Python. Cette semaine nous allons voir, sous la forme d'un projet, comment résoudre ces mêmes labyrinthes, c'est-à-dire trouver un chemin de l'entrée à la sortie du labyrinthe.

Ci-dessous est présenté un exemple de ce que l'on cherche à obtenir aujourd'hui. Le chemin de l'entrée à la sortie est affiché en rouge, les zones explorées mais non retenues en gris.



Pour cela, nous allons voir deux approches de résolution :

1. le parcours en profondeur, et
2. le parcours en largeur.

Après ces deux approches, nous aborderons des généralisations de ce problème dans le cadre des *graphes* et des algorithmes de plus court chemin.

1.1 Bases communes

Avant de plonger dans la découverte des différentes approches, il nous faut poser quelques bases communes aux deux approches.

1.1.1 Nœud

Tout d'abord, intéressons-nous au concept de nœud. La semaine dernière, nous avions le concept de pièces reliées par des couloirs, avec une distinction entre les pièces et les couloirs. Dans l'étape du jour, nous n'aurons pas besoin de faire cette différence, et pour cela nous utiliserons juste le concept de *nœud*.

Les différentes positions accessibles dans le labyrinthe seront des nœuds, *reliés entre eux* si l'on peut se déplacer directement de l'un à l'autre.

Quant aux murs, ils ne seront pas représentés par des nœuds. Deux nœuds séparés par un mur ne seront pas reliés entre eux.

En termes d'implémentation, chaque nœud correspondra à exactement un pixel de l'image du labyrinthe. Au départ, tous les nœuds du labyrinthe seront représentés par les pixels blancs de l'image. On identifiera donc un nœud grâce à ses coordonnées x et y.

1.1.2 Couleur des nœuds

La couleur du pixel associé à un nœud nous servira à indiquer si le nœud a déjà été visité ou encore s'il fait partie du chemin jusqu'à la sortie. Nous utiliserons un pixel gris pour indiquer que le nœud a été visité lors de la recherche mais qu'il ne mène pas à la sortie. Au final, nous utiliserons la couleur rouge pour indiquer les nœuds sur le chemin vers la sortie. La couleur blanche sera réservée aux nœuds qui n'ont pas été visités.

```
def est_libre(image, noeud):
    return image.getpixel(noeud) == (255, 255, 255)
```

1.1.3 Voisins et voisins non-visités

On appelle les nœuds situés directement au-dessus, au-dessous, à gauche ou à droite d'un nœud ses voisins. Notez que, contrairement à l'étape précédente, les voisins sont directement adjacents et ne sont pas séparés par un couloir.

Nos deux méthodes de résolution auront besoin d'une fonction pour déterminer les voisins *non-visités* d'un nœud. Les voisins non-visités sont représentés par des pixels blancs dans l'image du labyrinthe. Il sera donc intéressant d'implémenter une fonction pour donner la liste des voisins non-visités d'un nœud. Attention à la gestion des bordures de l'image.

```
def voisins_non_visites(image, noeud):
    voisins = []
    # À compléter
```

1.1.4 Nœud de départ

Dans les labyrinthes que l'on cherchera à résoudre, il y aura toujours un unique nœud de départ situé sur le haut du labyrinthe. Il sera intéressant d'implémenter une fonction pour trouver les coordonnées x et y du nœud de départ à partir de l'image du labyrinthe.

```
def noeud_depart(image):
    # À compléter
```

1.1.5 Dessiner un chemin

Lorsque nous aurons trouvé un chemin du départ à l'arrivée dans le labyrinthe, il faudra indiquer ce chemin en rouge dans l'image. Pour cela, vous pourrez utiliser la fonction suivante :

```
def dessiner_chemin(image, chemin):
    for noeud in chemin:
        image.putpixel(noeud, (255, 0, 0))
```

1.2 Parcours en profondeur

Le parcours en profondeur (*depth-first search* en anglais) est une méthode de parcours des labyrinthes et plus généralement de parcours dans ce que l'on appelle des *arbres*. Nous aurons l'occasion de parler des arbres en plus de détails tout prochainement dans la suite du cours.

Le parcours en profondeur est une méthode qui consiste à explorer les chemins jusqu'au bout avant de tenter d'explorer d'autres chemins. À chaque étape d'un chemin, à chaque nœud, on choisit arbitrairement un nœud voisin non-visité pour s'y déplacer. Au bout du chemin, soit on arrive à la sortie et on arrête le parcours, soit on tombe sur un cul-de-sac. Dans le cas d'un cul-de-sac, on rebrousse chemin jusqu'au dernier nœud avec des voisins encore non-visités. Notez que cette méthode d'exploration des labyrinthes est très similaire à celle employée la semaine dernière pour générer le labyrinthe.

En termes d'implémentation en Python, il vous faudra implémenter une fonction nommée `parcours_en_profondeur` qui prend deux arguments :

1. l'image du labyrinthe, et
2. la position de départ.

La fonction devra retourner, sous la forme d'une liste, les positions des nœuds sur le chemin du départ à l'arrivée. Les positions de départ et d'arrivée devront être contenues dans cette liste. L'image pourra être modifiée par la fonction, pour, par exemple, indiquer quel nœuds ont été visités.

```
def parcours_en_profondeur(image, pos_depart):
    x, y = pos_depart
    chemin = []
    # À compléter
    return chemin
```

Il est possible de l'implémenter à l'aide d'une boucle, auquel cas il faudra maintenir une liste Python pour contenir le chemin parcouru depuis le noeud de départ jusqu'à la cellule courante. Il est aussi possible d'implémenter cette fonction de manière récursive. Une fois le chemin retourné, vous pourrez utiliser la fonction `dessiner_chemin` afin de colorier ce chemin en rouge.

1.3 Parcours en largeur

Le parcours en largeur *breadth-first search* est une autre méthode de parcours qui explore les noeuds à des distances de plus en plus grandes du nœud de départ. Avant d'explorer un nœud à une distance $d + 1$ de la position de départ, tous les nœuds à distance d doivent avoir été explorés.

Les files Afin de réaliser un parcours en profondeur, on s'aide généralement d'une *file* (*queue* en anglais). Une file est une structure de données, une collection de valeurs, qui permet de réaliser efficacement deux opérations de base :

1. L'ajout d'un ou plusieurs éléments en fin de file.
2. Le retrait d'un élément en début de file.

Les files sont des structures de données dites FIFO (pour *first-in first-out*, littéralement *premier dedans, premier dehors*). Elles permettent de traiter les éléments dans l'ordre où ils sont insérés, comme une file d'attente au magasin ou à la cafétéria.

Notez que, à contrario, les listes Python permettent de faire efficacement des traitements dans l'ordre LIFO (pour *last-in, first-out*, *dernier dedans, premier dehors*). En effet, pour les listes Python, l'ajout et le retrait à la fin de la liste sont efficacement effectués. Pour les files, on peut ajouter efficacement à la fin et retirer efficacement au début.

En Python, il est possible d'utiliser très simplement les files à l'aide du module `collections`. Le module exporte une collection appelée `deque` (pour *double-ended queue*). Cette collection représente une file améliorée qui permet l'ajout et le retrait efficace des deux côtés de la file.

```
from collections import deque

ma_file = deque()  # Création d'une file
ma_file.append(1)  # Ajout d'un élément
ma_file.extend([2, 3, 4])  # Ajout de plusieurs éléments
elem = ma_file.popleft()  # Retrait du premier élément
```

Description du parcours en profondeur Le parcours en profondeur se base sur l'utilisation d'une file de nœuds. Initialement, la file ne contiendra que le nœud de départ. Puis, de manière répétée et tant que la file n'est pas vide, on retirera le premier élément de la file, on vérifiera que cet élément n'est pas le nœud d'arrivée, puis on ajoutera en fin de file tous les voisins non-visités du nœud. Si l'on tombe sur le nœud d'arrivée, le parcours se termine.

Cette manière de fonctionner, très simple, ne permet cependant pas directement de reconstruire un chemin du départ à l'arrivée. En effet, on perd trace du chemin qui mène à un nœud une fois ce nœud ajouté à la file. Pour reconstruire le chemin, il nous faudra nous aider d'une

autre structure de données. Cette structure additionnelle nous permettra de noter, pour chaque nœud visité (autre que le nœud de départ), le nœud par lequel on était arrivé. Pour cela, la collection la plus adaptée sera un dictionnaire.

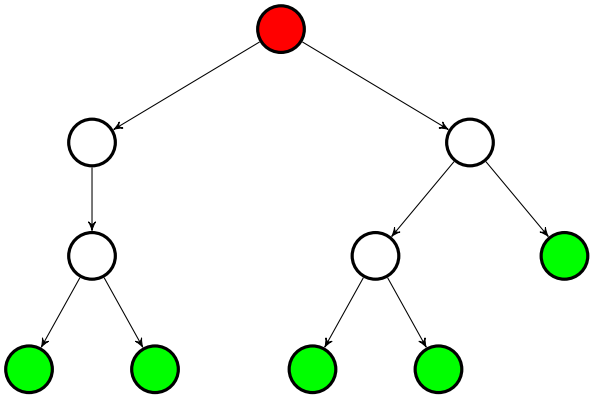
À chaque fois que l'on ajoute un nœud à la file, on ajoutera aussi au dictionnaire le nœud par lequel on y accède. Une fois le nœud d'arrivée trouvé, il suffira de consulter ce dictionnaire afin de remonter le chemin inverse.

Comparaisons avec la recherche en profondeur La recherche en profondeur et la recherche en largeur sont deux approches très similaires. D'ailleurs, en changeant un unique appel de méthode, vous devriez être capable de transformer votre implémentation de la recherche en largeur en une recherche en profondeur !

La recherche en largeur a cependant l'avantage de retourner le plus court chemin même en présence de boucles dans les labyrinthes. Ce cas de figure ne se présente pas encore, mais nous l'étudierons la semaine prochaine !

2 Les arbres

Les labyrinthes que vous avez générés la semaine passée et que nous avons résolus aujourd'hui sont des exemples d'un concept plus abstrait que l'on appelle les *arbres*. Un arbre est une structure de données formée de nœuds connectés de manière hiérarchique. Voici la représentation graphique d'un arbre.



Il existe un vocabulaire assez riche pour parler des différents éléments d'un arbre. Ci-dessous, nous allons présenter brièvement quelques-uns de ces termes.

Nœuds Les arbres sont composés de *nœuds*. Dans le schéma ci-dessus, ils sont représentés par des cercles. Dans le cas des labyrinthes, les nœuds de l'arbre correspondent aux différentes zones où il est possible de se déplacer.

Racine L'arbre a une unique *racine* (en rouge dans le schéma), un unique nœud qui est le point d'entrée dans l'arbre. Dans le cas des labyrinthes, il s'agissait du nœud de départ.

Arêtes Les nœuds d'un arbre peuvent être reliés à d'autres nœuds par le biais d'une *arête*. Les arêtes vont toujours d'un nœud à un autre nœud plus bas dans l'arbre, et ainsi ne forment jamais de *cycles* (de boucles).

Parent et enfants On appelle les nœuds directement en dessous d'un nœud et reliés à celui-ci par une arête les *enfants* du nœud. Selon la même métaphore, on appelle le *parent* d'un nœud le nœud dont il est l'enfant. Chaque nœud, à part la racine, a un unique parent.

Feuilles Certains nœuds n'ont pas d'enfants, on appelle ces nœuds des *feuilles* (en vert dans le schéma). Dans le contexte des labyrinthes, le nœud à la sortie est une feuille, tout comme les nœuds qui sont dans un cul-de-sac.

Étiquettes Les arbres peuvent aussi contenir des valeurs. Ces valeurs peuvent être liées aux nœuds comme aux arêtes. On appelle ces valeurs liées aux éléments de l'arbre des *étiquettes*.

Applications des arbres

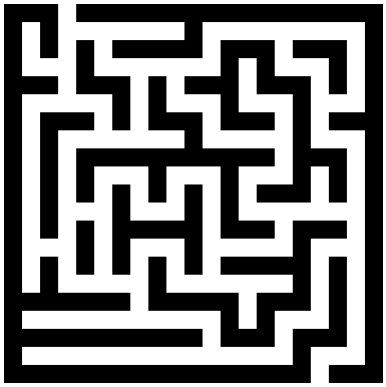
Les arbres ont de nombreuses applications en informatique. Ils permettent de représenter des données hiérarchiques comme l'on retrouve par exemple dans les systèmes de fichiers (avec dossiers, sous-dossiers, etc.) ou l'organisation de grandes entreprises. Les arbres sont aussi au cœur d'algorithmes, comme par exemple l'algorithme de Huffman pour la compression de données. Ils se retrouvent aussi dans le domaine de l'intelligence artificielle sous la forme d'arbres de décision. Comme nous le verrons aussi à la toute fin du cours, les arbres peuvent aussi servir à représenter les expressions d'un langage de programmation et même la structure d'un programme.

Les deux méthodes de parcours que l'on a vues aujourd'hui sont applicables à n'importe quel arbre, et même à des structures plus générales appelées *graphes*.

3 Plus court chemin

Jusqu'à présent, nous avons vu comment générer des labyrinthes et comment y trouver son chemin. Nous venons de voir que les labyrinthes en question sont des instances d'un concept plus abstrait : celui des arbres. Un arbre est un ensemble de nœuds connectés de manière strictement hiérarchique. Dans ce genre de structures, il n'y a toujours qu'un unique chemin entre deux nœuds.

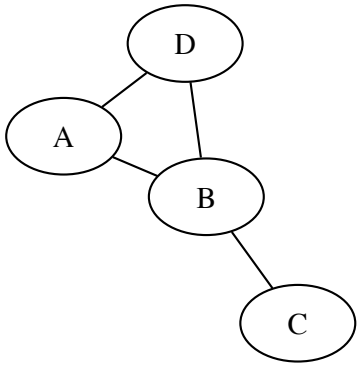
Pour cette dernière partie du module, nous allons considérer une généralisation de ce problème. Nous allons nous intéresser au cas où les labyrinthes ne forment plus simplement des arbres mais des structures plus générales. Par exemple, observez le labyrinthe ci-dessous et notez qu'il y a plus d'un chemin vers la sortie.



Pour cela, nous allons étudier une méthode, l'algorithme A* (*A star*), pour non seulement trouver efficacement un chemin dans de telles structures, mais aussi garantir qu'il s'agit du chemin le plus court. Les labyrinthes sur lesquels nous allons travailler sont des exemples d'un concept plus général, celui des *graphes*. Cet algorithme A* est très utilisé en pratique, par exemple dans les jeux vidéo pour permettre aux personnages non-joueurs de se déplacer efficacement dans leur environnement.

3.1 Graphes

Tout comme un arbre, un graphe est une collection de nœuds reliés par des arêtes. Chaque arête relie exactement deux nœuds. Notez qu'il n'y a pas forcément d'arête entre chaque paire de nœuds. Voici un exemple de graphe.

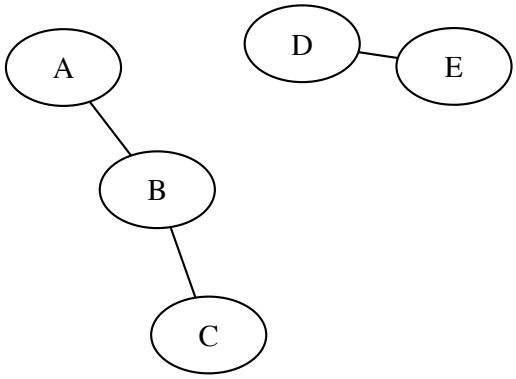


Remarquez que les arbres que nous avons vus la semaine dernière sont aussi des graphes. Les arbres sont juste des graphes avec des contraintes additionnelles.

Si l'on applique ce concept de graphes à des labyrinthes, chaque pièce du labyrinthe correspond à un nœud. On considère que deux nœuds sont reliés par une arête si les deux pièces sont voisines dans le labyrinthe.

3.2 Chemins

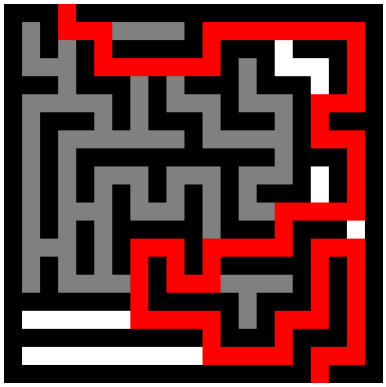
Un chemin dans un graphe est une séquence de nœuds où chaque paire de nœuds successifs sont reliés par une arête. Contrairement aux arbres, il n'y a pas toujours un unique chemin entre deux nœuds dans les graphes en général. Dans le graphe présenté plus haut, A - B - C est un premier chemin entre A et C, et A - D - B - C en est un deuxième. Notez que parfois, pour certains graphes, il n'y aura pas forcément de chemin entre chaque paire de nœuds, comme par exemple entre les nœuds A et E dans le graphe suivant.



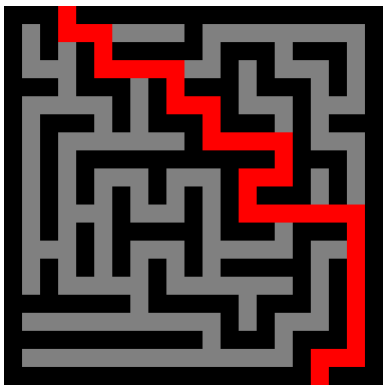
3.3 Comportement des méthodes de parcours

Jusqu'à présent, nous avons vu deux méthodes de parcours des arbres : le parcours en profondeur et le parcours en largeur. Ces méthodes de parcours, bien qu'abordées dans le cadre des arbres, peuvent être appliquées à des graphes plus généraux.

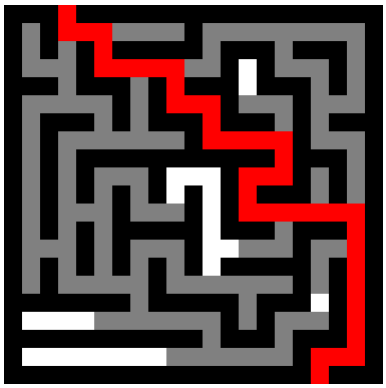
Parcours en profondeur La méthode de parcours en profondeur, qui explore les chemins jusqu'au bout et remonte en arrière en cas de cul-de-sac, peut s'appliquer à des graphes mais ne garantit pas de tomber sur le chemin le plus court. Il se peut que le chemin trouvé soit plus long que le plus court chemin possible. Ci-dessous par exemple est présenté le chemin trouvé par le parcours en profondeur sur le labyrinthe présenté plus tôt. Il s'agit bien d'un chemin de l'entrée à la sortie, mais le chemin n'est pas optimal.



Parcours en largeur La méthode d'exploration en largeur, quant à elle, garantit de retourner le chemin le plus court. Cependant, pour ce faire, la méthode requiert généralement d'explorer une grande partie du labyrinthe. Ci-dessous par exemple est présenté le chemin trouvé par le parcours en largeur sur le même labyrinthe. Notez que dans ce cas toutes les cases du labyrinthe ont dû être explorées avant d'arriver à ce chemin.



Afin de trouver un chemin efficacement, nous allons étudier ensemble un algorithme appelé A* (prononcé *A star*). L'algorithme A* permet de retrouver le plus court chemin dans un graphe de manière plus efficace que le parcours en largeur car il priorise l'exploration des nœuds sur des chemins prometteurs (c'est-à-dire des chemins courts et qui amènent proche de la sortie). Observez ci-dessous le résultat obtenu par cette méthode d'exploration.



Cette méthode nous permettra d'obtenir le plus court chemin en explorant moins du labyrinthe qu'avec un parcours en largeur. Notez que cette méthode requiert de pouvoir estimer si l'on s'approche ou l'on s'éloigne d'un objectif, par exemple en mesurant la distance qui sépare le nœud de la sortie sans tenir compte des murs.

L'algorithme A* est très similaire à la méthode de parcours en largeur et se base aussi sur l'utilisation d'une file. Cependant, l'algorithme A* utilise une *file de priorité* dans laquelle les éléments sont traités en fonction de leur niveau de priorité.

3.4 File de priorité

Tout comme les listes et les files, une file de priorité est une collection de valeurs. Une priorité est cependant associée à chaque valeur. Ces priorités indiquent l'ordre dans lequel les éléments doivent être traités. Plus la priorité d'un élément est basse, plus l'élément doit être traité rapidement. Les files de priorité supportent les deux opérations de base suivantes :

1. Ajout d'un élément dans la file à une valeur de priorité donnée.
2. Retrait de l'élément le plus prioritaire. Contre-intuitivement, l'élément le plus prioritaire est celui avec *la plus faible* valeur de priorité.

Utilisation en Python La librairie standard de Python inclut un module pour les files, et notamment les files de priorité. Il s'utilise comme suit.

```
from queue import PriorityQueue  # Import de la classe

ma_file = PriorityQueue()  # Création d'une file de priorité

ma_file.put((1, "A"))  # Ajout d'un élément (priorité 1)
ma_file.put((0, "B"))  # Ajout d'un élément (priorité 0)
ma_file.put((4, "C"))  # Ajout d'un élément (priorité 4)

print(ma_file.get())  # Affiche (0, "B")
print(ma_file.get())  # Affiche (1, "A")
print(ma_file.get())  # Affiche (4, "C")
```

3.5 L'algorithme A*

L'algorithme A* est relativement simple. Il opère à l'aide d'une file de priorité des nœuds. L'algorithme s'exécute de façon itérative. À chaque itération, le nœud de plus faible priorité est retiré de la file et est ensuite traité. Pour traiter un nœud, on regarde premièrement s'il ne s'agit pas du nœud d'arrivée. Si c'est le cas, on reconstruit le chemin parcouru (comme pour le parcours en largeur de la semaine dernière) et on termine la recherche. Dans le cas où il ne s'agit pas du nœud d'arrivée, on insère dans la file tous les voisins du nœud qui soit n'ont jamais été atteints, soit sont atteints par un chemin plus court que précédemment. On note aussi la distance à laquelle le nœud peut être atteint depuis l'entrée et le nœud qui y mène (afin de pouvoir reconstruire le chemin à la fin). En terme de code, la structure de l'algorithme est la suivante :

```
def a_star(graphe, depart, arrivee):
    # File de priorité pour le traitement des noeuds
    file_noeuds = PriorityQueue()
    file_noeuds.put((0, depart))

    origines = {}  # Origine de chaque nœud atteint
    origines[depart] = None
    distances = {}  # Distances des noeuds atteints depuis le départ
    distances[depart] = 0

    while not file_noeuds.empty():
        # Récupération du prochain noeud
        _, noeud = file_noeuds.get()

        # Vérification de si on a atteint l'arrivée
        if noeud == arrivee:
            chemin = []
            while noeud is not None:
                chemin.append(noeud)
                noeud = origines[noeud]
            chemin.reverse()
            return chemin

        # Ajout des voisins s'ils sont plus rapidement atteignables
        for voisin in voisins(image, noeud):
            # Calcul de la distance du voisin au départ
            d = distances[noeud] + distance(graphe, noeud, voisin)
            if voisin not in distances or d < distances[voisin]:
                distances[voisin] = d
                origines[voisin] = noeud
                priorite = d + estimation(graphe, voisin, arrivee)
                file_noeuds.put((priorite, voisin))
```

Reste à déterminer les priorités auxquelles les nœuds sont insérés dans la file. La priorité de chaque voisin consiste en la somme de deux valeurs :

1. Le distance du nœud depuis le nœud de départ. Cette distance peut être calculée facilement en maintenant un dictionnaire avec la distance effective depuis le départ pour chaque nœud inséré dans la file. Au moment de l'insertion dans la file, il est facile de calculer cette valeur.
2. Une estimation de la distance qui sépare le voisin de l'arrivée. Pour que le chemin retourné soit le plus court, il faut s'assurer que cette estimation ne surestime jamais la véritable distance.[1] On appelle ce genre de fonctions des *heuristiques*. Plus bas sont listées différentes heuristiques applicables dans le cadre de cet algorithme, plus ou moins appropriées selon le contexte.

Notez que dans le cas où l'heuristique choisie indique toujours 0 comme estimation de la distance au nœud d'arrivée, on retombe sur l'algorithme de parcours en largeur.

3.5.1 Distance de Manhattan

Dans notre cas, comme on peut uniquement se déplacer à gauche, à droite, en haut ou en bas, il est relativement simple d'estimer le nombre minimal d'étapes à parcourir dans le meilleur des cas. Il s'agit simplement de la distance horizontale additionnée à la distance verticale.

$$d((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|$$

On appelle cette distance la *distance de Manhattan*, en référence à l'organisation quadrillée de cet arrondissement de New York. La distance de Manhattan constitue une bonne heuristique dans le cas de nos labyrinthes. En effet, elle ne surestime jamais la distance réelle et y est même égale dans certains cas.

3.5.2 Autres heuristiques

Notez qu'il existe d'autres distances que l'on peut utiliser comme heuristiques, c'est-à-dire comme approximations de la distance réelle. Par exemple, la distance euclidienne, qui permet de mesurer la distance à vol d'oiseau, est une heuristique adéquate.

$$d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Il y a aussi des distances intéressantes dans le cas de déplacements dans une grille avec des déplacements en diagonale. Par exemple, la distance de Tchebychev, qui mesure la plus grande des distances sur un seul axe et qui permet de mesurer le nombre de cases à traverser si les sauts en diagonale sont autorisés (et comptent pour une distance de 1).

$$d((x_1, y_1), (x_2, y_2)) = \max(|x_1 - x_2|, |y_1 - y_2|)$$

1. Pour être exact, afin de s'assurer de ne pas devoir visiter à répétition un même nœud, il faut aussi que la fonction soit *monotone* : pour chaque nœud, la distance estimée à l'arrivée doit être plus petite ou égale à la somme de la distance effective à un voisin additionnée à la distance estimée du voisin à l'arrivée. Les fonctions proposées ont toutes cette propriété.

Dans le cas où l'on autorise les sauts en diagonale mais que l'on souhaite les comptabiliser de manière géométrique, on utilisera la distance suivante :

$$d((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2| + (\sqrt{2} - 2) \cdot \min(|x_1 - x_2|, |y_1 - y_2|)$$

Le choix de la distance, plus ou moins précise, influencera le nombre de nœuds qui seront visités par l'algorithme A*. Toutefois, le choix de la distance n'a pas d'influence sur l'optimalité du résultat.

4 Idées d'extensions

Une fois l'algorithme A* implémenté dans le cadre des labyrinthes, il vous sera en principe simple de l'appliquer à d'autres contextes. Dans cette section, nous explorerons deux de ces différents contextes :

1. Le plus court chemin sur une carte avec des obstacles et la possibilité de se déplacer en diagonale.
2. Le plus court chemin sur une carte avec des cases à différentes hauteurs.

4.1 Première extension

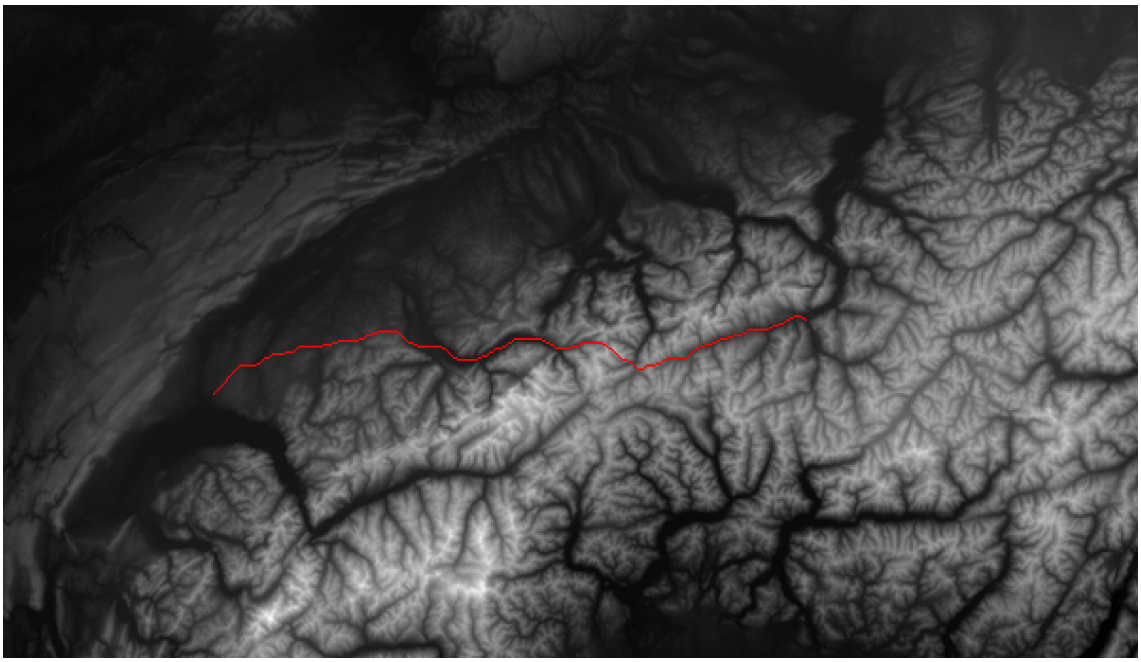
Pour la première extension, concevez un programme qui prend en entrée une image et qui y recherche un chemin en évitant les zones infranchissables. Les pixels blancs indiquent des zones qu'il est possible de visiter, les pixels noirs des zones infranchissables. Dans cette image, il faudra trouver un pixel rouge (le nœud de départ) et un pixel vert (le nœud d'arrivée). Trouvez ensuite le chemin le plus court entre ceux deux nœuds. Notez en gris les nœuds visités.

Pour cet exercice, nous considérerons qu'il est possible de se déplacer à gauche, à droite, en haut et en bas, mais aussi en diagonale. Alors que les déplacements verticaux et horizontaux auront un coût de 1 unité, les déplacements en diagonale auront un coût de $\sqrt{2}$.

Vous trouverez des images d'exemple sur Moodle. Vous êtes aussi libres de créer vos propres cartes et d'apporter d'autres améliorations ! On pourrait par exemple imaginer avoir des zones de couleurs différentes dans lesquelles on progresse plus vite ou plus lentement. Par exemple, on pourrait représenter des marais en vert et des routes pavées en jaune. Il serait alors logique qu'un déplacement par la route prenne moins de temps et qu'un déplacement au travers d'un marais en prenne plus.

4.2 Deuxième extension

Pour cette deuxième extension, faites en sorte de trouver le plus court chemin sur une carte d'élévation. En entrée, votre programme prendra une image sur laquelle l'élévation sera indiquée par un niveau de gris. Plus une zone sera élevée, plus elle sera proche du blanc. Au contraire, plus une zone est basse, plus elle sera proche du noir. Dans le cadre de cet exercice, on considérera que la somme des trois composantes de la couleur (le rouge, le vert et le bleu) d'un pixel représente la hauteur d'une case.



Faites en sorte de demander à l'utilisateur la position du point de départ et du point d'arrivée dans l'image et de retourner le chemin avec le plus petit coût du départ à l'arrivée. Le coût d'un chemin sera comptabilisé comme ceci :

1. 1 pour un déplacement d'une case à l'horizontale ou à la verticale sur le chemin.
2. $\sqrt{2}$ pour un déplacement d'une case en diagonale sur le chemin.
3. $0,05$ par unité de hauteur de différence d'une case à l'autre sur le chemin.

Comme pour l'extension précédente, vous trouverez des images d'exemple sur Moodle. Vous êtes aussi libre de créer vos propres cartes à l'aide de l'outil `https://tangrams.github.io/heightmapper` par exemple.