

IFOSUP

Institut de Formation Supérieure
Ville de Wavre

Algorithmique & Programmation

Cédric Vanconingsloo

Table des matières

1. Introduction	1
1.1. Définition	1
1.2. À quoi sert un algorithme ?	3
1.3. Les outils utilisés	3
2. Installation des outils	4
2.1. Python	4
2.2. uv et ruff	4
2.3. PyCharm	5
2.4. VS Code ou Fleet	5
3. Flowcharts	6
3.1. Symbolique	6
4. Algorithmique de base en Python	7
4.1. Délimiteurs	7
4.1.1. Instructions et commentaires	7
4.2. Les mots clés	7
4.3. Les fonctions à connaître dès le départ	8
4.4. Exemple : Hello World	8
5. Les valeurs littérales	9
5.1. Nombres	9
5.2. Chaînes de caractères	9
5.3. Caractère d'échappement et de contrôle	9
6. Les variables	10
6.1. Principe	10
6.1.1. Comment nommer une variable ?	10
6.2. Types et déclaration	10
6.2.1. Transtypage	12
6.3. Saisie et affichage	12
6.3.1. La fonction <code>print</code>	12
6.3.2. La fonction <code>input</code>	12
6.3.3. Formatage des chaînes de caractères	13
6.4. Les constantes	13
7. Opérateurs et calculs	14
7.1. Les opérateurs arithmétiques	14
7.2. Les opérateurs de comparaison	15
7.3. Les opérateurs booléens	15
7.3.1. L'opérateur <code>not</code>	15
7.3.2. L'opérateur <code>and</code>	15
7.3.3. L'opérateur <code>or</code>	15
7.3.4. L'opérateur <code>xor</code>	16
7.3.5. Les opérateurs de comparaison	16
7.4. Les opérateurs d'affectation	16
8. Structure conditionnelle	17
8.1. Cas particulier	17
8.2. L'instruction <code>if</code>	17
8.2.1. Forme simple	17
8.2.2. Forme complexe	18

8.2.3. Embranchements multiples	18
8.3. Tests imbriqués	18
9. Structures itératives	21
9.1. But d'une boucle	21
9.2. La boucle <i>indéfinie</i> (<i>while</i> – tant que)	22
9.3. La boucle <i>finie</i> (<i>for... in</i>)	23
9.4. Quel type de boucle à utiliser et quand ?	24
9.5. Boucles imbriquées	24
9.6. Casser une boucle (<i>break</i> et <i>continue</i>)	25
10. Fonctions	26
10.1. Les fonctions	26
10.1.1. Exemple	27
10.2. Analyse fonctionnelle	27
10.3. Le passage de paramètres	28
10.3.1. Paramètres par défaut	28
10.3.2. Paramètres nommés	29
10.3.3. Paramètres multiples	29
10.4. Les docstrings	29
10.5. Les fonctions récursives	30
10.5.1. La factorielle	30
11. Les tableaux	32
11.1. Les tableaux en Python	32
11.1.1. Notion d'objet et de classe	32
11.1.2. Tableaux dynamiques et listes	33
11.1.3. Tableaux multidimensionnels	34
12. Dictionnaires, tuples, ensembles	36
12.1. Les tuples	36
12.1.1. Accès aux données	36
12.2. Les dictionnaires	36
12.2.1. Ajout et mise à jour d'élément	37
12.2.2. Accès aux éléments	37
12.2.3. Suppression d'éléments	37
12.3. Les ensembles	37
12.3.1. Ajout et mise à jour de données	38
12.3.2. Suppression d'élément	38
13. Les strings	39
14. Itérateurs et Générateurs	40
14.1. Itérateur	40
14.2. Générateur et fonction génératrice	40
14.2.1. Cas d'utilisation	40
14.2.2. Fonction génératrice	40
14.3. Liste ou générateur ?	41
14.4. Les compréhensions de listes	42
15. Les fonctions intégrées	43
16. Les modules	46
16.1. Les modules standard de Python	46
16.1.1. La directive <i>import</i>	46

16.1.2. La directive <code>from ... import</code>	46
16.2. Nommage des modules	47
16.3. Créer un module	47
17. La gestion des fichiers	48
17.1. Les fichiers texte	48
17.2. Les fichiers binaire	48
17.3. Les accès au fichier	48
17.3.1. L'accès séquentiel	48
17.3.2. L'accès direct	49
17.3.3. L'accès indexé	49
17.4. Les enregistrements	49
17.5. Accéder à un fichier	49
17.6. Traiter un fichier en Python	50
17.6.1. Lire un fichier texte	50
17.6.2. Écrire dans un fichier texte	50
17.7. Pathlib	51
17.8. Pickle	51
18. Gestion des exceptions	52
18.1. Lever une exception	52
18.2. Pourquoi lever une exception ?	52
18.3. Capturer une exception	53
18.3.1. La clause <code>else</code>	53
18.3.2. La clause <code>finally</code>	54
19. Les Tests unitaires avec Pytest	55
19.1. Introduction	55
19.1.1. Qu'est-ce qu'un <i>test unitaire</i> ?	55
19.2. Pytest	55
19.2.1. Installer Pytest	55
19.2.2. Conventions de nommage	55
19.2.3. Écriture d'un test	55
19.3. Création d'un projet en TDD	56
19.3.1. Notre premier test	56
19.3.2. Les fixtures	57
19.3.2.1. Écrire les fixtures	58
19.3.3. Le <i>monkeypatching</i>	60
19.3.4. La paramétrisation	61

1. Introduction

1.1. Définition

Algorithmique L'algorithmique est la *science et la production d'algorithmes*.

Algorithme Un algorithme est un *ensemble d'instructions permettant de résoudre un problème*. Le mot **algorithme** vient du mathématicien *Al-Khwârizmi*. On peut voir un algorithme comme une *recette de cuisine*.

Par exemple, si l'on se pose le problème suivant: *comment cuire une omelette*? Quelles sont les étapes (les plus détaillées possibles) qui vont répondre à cette question? Déjà, nous avons en notre possession des outils et des ingrédients déjà conçus pour répondre à une partie du problème: des œufs, une poêle, une cuisinière, du sel, du poivre. Nous n'avons pas à les recréer (même si on peut le faire!). Voici une solution (il peut y en avoir d'autres, vous faites votre omelette à votre goût...):

- Prendre un bol et y casser des œufs.
- Battre les œufs dans le bol à la fourchette.
- Saler et poivrer la préparation.
- Mettre un peu de beurre dans la poêle.
- Verser la préparation dans la poêle chaude.
- Laisser cuire la préparation quelques minutes.
- Servir la préparation dans une assiette.

Voilà, nous avons écrit notre premier *algorithme*! Mais la solution peut être plus ou moins complète, en fonction des outils que nous avons en notre possession.

En informatique, ces différents outils se nomment des *frameworks*. Ces frameworks dépendent aussi du langage de programmation utilisé.

Framework Un framework (ou *infrastructure logicielle* en français) est un ensemble de composants qui servent à créer plus facilement un logiciel sans devoir en coder les éléments constitutifs. Un framework ne doit pas être confondu avec une bibliothèque logicielle (*library*)! Par contre, un framework est constitué de différentes *libraries*.

Library Une library (*bibliothèque logicielle*, aussi appelée à tort *librairie*) est une collection de fonctions spécialisées et prêtes à l'emploi. Les *librairies* sont plus couramment nommées les *dlls*.

Pour concevoir un algorithme, nous allons utiliser une *machine de Turing*. Un ordinateur est une machine de Turing, mais nous allons utiliser une autre machine, bien plus puissante: notre **cerveau**.

Machine de Turing Une machine de Turing est un modèle *abstrait* mathématique. Une machine de Turing comporte 4 éléments:

1. Un **ruban infini**, divisé en cases, dans lesquelles nous allons écrire des symboles (du code).
2. Une **tête de lecture/écriture** qui va lire ou écrire les symboles sur le ruban, avant de se déplacer sur ce ruban.
3. Un **registre d'état** qui mémorise l'état courant de la machine après traitement du symbole. Le registre d'état possède toujours un état de départ et un état d'arrivée.

4. Une **table d'action** qui indique à la machine quel symbole écrire sur le ruban, comment déplacer la tête de lecture/écriture, quel état appliquer à la machine en fonction du symbole lu et l'état courant de la machine. Si aucune action n'existe pour un symbole et un état courant, la machine s'arrête.

Nous sommes également une machine de Turing!. Nous possédons une mémoire qui nous permet d'imaginer un ruban infini de cases (et nous pouvons aussi nous aider de papier...). Notre tête de lecture/écriture est le couple œil/main (pour lire et écrire sur ledit papier). Notre cerveau nous sert de registre d'état, et notre alphabet nous sert de table d'action!

Pseudo-code Le pseudo-code est une façon de décrire un algorithme, sans faire référence à un langage particulier. Il est à mi-chemin entre le langage courant et un langage de programmation. Dans notre cas, nous écrirons nos pseudo-codes en **anglais**.

Flowchart Un flowchart (ou *algorithme*, *organigramme de programmation*) est une représentation graphique des opérations et décisions effectuées par un programme.

Python Le Python est un langage de programmation conçu pour être simple, laissant beaucoup de latitude au développeur, mais en prenant à sa charge les problématiques de bas niveau. Python est un **langage de programmation interprété, multiparadigmes, de haut niveau à typage dynamique fort**. Sa syntaxe est minimaliste et claire, ressemblant énormément au langage naturel (en langue anglaise).

De plus, Python est un langage *interprété*, ce qui signifie qu'il n'a pas besoin d'être *compilé*. L'avantage d'un programme interprété est qu'il est compatible sur n'importe quel OS et le programme est directement modifiable avec un éditeur de texte. Mais un programme interprété sera plus lent et nécessite un *interpréteur* qui fonctionnera dans une *machine virtuelle*.

À l'inverse, un programme *compilé* nécessite un **compilateur**, c.-à-d. un outil qui convertit le code source en un *bytecode*, qui sera lisible directement par le système d'exploitation installé. Un programme compilé est donc plus rapide, mais dépendant d'un système d'exploitation et chaque changement dans le code nécessitera une nouvelle compilation. Ce type de programme étant plus proche du langage machine, il est aussi plus optimisé. Le C et le C++ sont des exemples de langages compilés.

Python est également un langage *compilé*. Mais comment peut-il être un langage *compilé et interprété*? C'est simple: lors de la première exécution d'un programme écrit en Python, il est **interprété**. Dans le même temps, le résultat de l'interprétation (qui est un résultat compilé) est stocké (dans un fichier **pyc**). Si on rappelle le programme *et qu'il n'a pas été modifié*, Python chargera la copie compilée pour que le traitement soit plus rapide.

Paradigme un paradigme est une façon de coder, souvent liée à un langage de programmation particulier. Les paradigmes les plus employés sont:

- Le paradigme **impératif**: on écrit un programme en séquences exécutables par l'ordinateur (comme l'assembleur);
- Le paradigme **procédural**: évolution de l'impératif dans lequel on crée un ensemble de *procédures* et de *fonctions*, mais le programme continue de se dérouler d'un début à une fin. Le **C** est un langage procédural;
- Le paradigme **orienté objet**: un programme est composé d'objets indépendants mis en interaction. Le **Java** est un langage orienté objet;

- Le paradigme **fonctionnel** : un programme est écrit comme une fonction mathématique. Le **Kotlin** est un langage fonctionnel (et objet).
- Le paradigme **évènementiel** : un programme réagit en fonction de différents évènements (déplacement de souris, clic sur un bouton...);
- Le paradigme **concurrent** : on tient compte de l'exécution en parallèle de plusieurs processus au sein d'un même programme;
- ...

Python gère ces différents paradigmes. Le programmeur peut les mélanger aisément, et se mélanger les pinceaux également...

1.2. À quoi sert un algorithme ?

De nombreux programmeurs autodidactes vont certainement vous dire que **« L'algorithmique, ça sert à rien »**. Vous allez aussi sûrement me dire **« On s'en tape de L'algorithmique, on veut coder en Python ! »**. Alors, **oui**, nous ferons de L'algorithmique en Python, et **non**, L'algorithmique va nous permettre de voir plus facilement les différents cas d'utilisation de notre programme.

Un algorithme bien établi et fonctionnel pourra être réécrit simplement dans un langage de programmation (par exemple, Python). Il existe plusieurs algorithmes menant au même résultat, mais certains sont meilleurs que d'autres.

La maîtrise de L'algorithmique est une des conditions de la réussite d'un projet en programmation. Avec de l'expérience, l'algorithmique vous mènera à travers des mécanismes de pensée qui vous permettront d'optimiser les traitements à programmer. Vous coderez directement en Python, mais vous *penserez* en algorithmique.

1.3. Les outils utilisés

Nous allons utiliser différents outils pour écrire nos algorithmes : Nous utiliserons le framework **Python** pour implémenter nos algorithmes, ainsi que **Fleet**, **PyCharm** ou **VS Code** pour écrire nos fichiers Python. Toutefois, un simple éditeur de texte fait aussi bien l'affaire !

Nous utiliserons aussi **Rye**, une suite d'outils bien pratique pour le développement Python.

2. Installation des outils

2.1. Python

Téléchargez Python à l'adresse suivante: <https://www.python.org>. Installez-le suivant les instructions qui apparaîtront à l'écran.

Note

Vous pouvez commencer par installer **uv** (<https://astral.sh>). Il permet d'installer automatiquement le bon environnement Python.

2.2. uv et ruff

uv et **ruff** sont deux outils en ligne de commande permettant de gérer aisément vos projets Python, développés par *astral* dans le langage Rust, réputé pour sa vitesse d'exécution et sa fiabilité.

uv est un gestionnaire de projets et de paquets. Il permet de créer et de déployer rapidement une application écrite en Python.

Sans entrer dans les détails, voici quelques commandes utiles à la gestion d'un projet :

uv python install [python_version] : Installe la version de python sélectionnée (ou la dernière par défaut). Il est donc possible de travailler avec plusieurs versions de python, indépendamment des projets.

Ajoutez `--preview` à la commande pour pouvoir utiliser Python sans environnement virtuel.

uv init [project] : Crée le dossier [project] et son environnement virtuel (venv).

uv add [dep] : Ajouter une dépendance au projet en cours.

uv add --dev [dep] : Ajouter une dépendance de développement (comme PyTest pour les tests).

uv sync : Synchroniser les dépendances.

uv remove [dep] : Supprime une dépendance.

uv run [tool] : Lance l'outil sélectionné (pratique pour PySide...).

ruff est un *linter* et un formateur de code pour Python. Il analysera votre code en temps réel pour vous indiquer les erreurs courantes **avant** de lancer l'application.

Pour installer **ruff** dans un projet python, un simple

```
uv add --dev ruff
```

suffit. Je recommande Toutefois de l'installer *globalement* sur le système, via la commande

```
uv tool install ruff
```

Une fois installé, cet outil fournira deux nouvelles commandes :

ruff check [path] : pour vérifier votre code.

ruff format [path] : pour formater votre code.

ruff gère pas moins de 800 (!) règles Python, adaptable via le fichier de projet *pyproject.toml*.

Note

Il est recommandé de lire la documentation sur ces outils. Nous les utiliserons en classe.

2.3. PyCharm

PyCharm est un **IDE** (*Integrated Development Environment* – Environnement de Développement Intégré), c.-à-d. un environnement de travail composé d'une série d'outils permettant de coder rapidement et efficacement. La société russe **JetBrains** fournit les meilleurs IDE pour différents langages, mais tous dépendent essentiellement d'IntelliJ IDEA, développé pour coder du Java. PyCharm est une modification d'IntelliJ IDEA pour coder du Python. Notez aussi que vous pouvez utiliser le plugin Python dans IntelliJ IDEA.

PyCharm existe en deux versions, la *community* et la *professional*. La version *community* est gratuite et open source. Par contre, la version *professional*, disposant de plus d'outils, est payante. Mais la version *community* pourvoira à nos besoins.

Vous trouverez PyCharm à cette adresse : <https://jetbrains.com/fr-fr/pycharm>.

2.4. VS Code ou Fleet

VS Code est l'éditeur de code léger et extensible de Microsoft. Il dispose de nombreuses extensions pour différents langages de programmation, dont Python. Téléchargez-le à l'adresse <https://code.visualstudio.com>.

Fleet est l'équivalent de VS Code, créé par JetBrains. Il est encore récent mais il est prometteur. Il est disponible à l'adresse <https://jetbrains.com/fr-fr/fleet>.

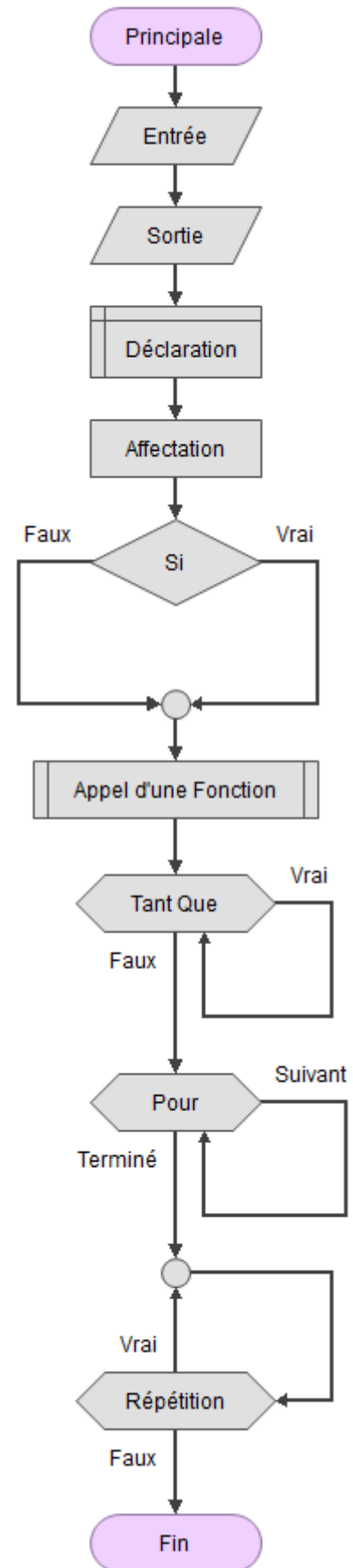
3. Flowcharts

3.1. Symbolique

Un flowchart (ou *algorithme*) est un dessin composé d'une série de symboles indiquant le comportement d'un algorithme. Chaque symbole possède sa propre signification :

1. Les *terminaux*, représentés par des **rectangles arrondis**, indiquent le début et la fin d'un algorithme.
2. Les *entrées-sorties*, représentés par des **parallélogrammes**, indiquent des éléments interactifs avec l'utilisateur.
3. La *déclaration d'une variable*, représentée par un **rectangle barré**, indique la création d'une variable (logique...).
4. L'*affectation d'une variable*, représentée par un **rectangle**, indique l'utilisation d'une variable (précédemment créée).
5. La structure *si*, en **losange**, permet d'établir une *condition*. Notez la présence de deux branches : l'algorithme suit une des branches en fonction de la condition (soit *vraie*, soit *fausse*).
6. L'*appel d'une fonction*, en **rectangle avec des lignes parallèles**, indique que l'algorithme fait appel à une fonction (externe ou écrite par le programmeur).
7. Les structures *répétitives* (les boucles) sont dessinées sous forme d'**hexagone allongé**. Il existe trois types de boucles :
 1. La boucle *Tant que*, qui va boucler **tant que la condition est vraie**. Cette boucle est dite *infinie*.
 2. La boucle *Pour*, qui va effectuer **un nombre précis de tours**. Cette boucle est dite *finie*.
 3. La boucle *Faire tant que*, qui va boucler **jusqu'à ce que la condition soit vraie**. Cette boucle est dite *infinie* et est l'exacte inverse de la boucle *Tant que*.

Nous verrons les structures conditionnelles et les répétitives dans leurs chapitres dédiés.



4. Algorithmique de base en Python

4.1. Délimiteurs

4.1.1. Instructions et commentaires

Une *instruction* est un texte formel permettant au développeur de déterminer une action à mener par son algorithme. Dans certains langages, tels que le Java ou le C, une instruction se termine par un point-virgule (;). Il est possible d'écrire tout son code source sur une seule ligne, mais c'est complètement illisible !

En Python, le type d'écriture est très structuré : **on ne peut écrire qu'une instruction par ligne**. Une instruction commence au début de ligne et se termine par un passage à la ligne. Le point-virgule (;) de séparation des instructions, même s'il existe en Python, est optionnel.

Note

Si toutefois vous souhaitez écrire deux instructions sur la même ligne, vous **devez** les séparer par le point-virgule ! Cependant, comme je l'ai expliqué plus haut, cela rend le code illisible.

Un *commentaire* est un texte qui ne sera pas interprété par Python. Il s'écrit en plaçant un dièse (#). S'il est placé en début de ligne, toute la ligne est alors ignorée.

Note

Il est très important de prendre l'habitude de commenter son code, même si Python est un langage suffisamment clair pour être *auto documenté*. En effet, sans commentaires, si vous (ou quelqu'un d'autre) relisez votre code plus tard, vous ne saurez plus exactement *pourquoi* vous avez conçu votre algorithme de telle manière !

À l'inverse, ne tombez pas dans l'excès ! Il est inutile de commenter *chaque* ligne de code, surtout s'il est trivial !

Une *documentation* est un commentaire un peu particulier. Elle permet d'expliquer un fichier ou une fonction et sera utilisée par les IDE. Pour créer une documentation, il faut placer en entête de fichier (ou sous la définition de la fonction) une **paire de triple double guillemets** (""" """). Cette écriture permet d'écrire de la documentation **en bloc**. Tout ce qui sera englobé par les guillemets seront considérés comme documentation.

Dans certains cas, une instruction peut être tellement longue que, par lisibilité, il soit nécessaire de la fractionner. Pour cela, on désactive le passage à la ligne en plaçant au niveau de la coupure un *backslash* \.

4.2. Les mots clés

Python contient uniquement **33** mots-clés permettant de structurer des algorithmes. Chaque mot-clé a une signification particulière qui ne peut pas être modifiée par le développeur. Dans ces 33 mots-clés, 30 sont des instructions et 3 sont des éléments spéciaux.

and	as	assert	break	class	continue	def	del	elif	else
except	finally	for	from	global	if	import	in	is	lambda
nonlocal	not	or	pass	raise	return	try	while	with	yield

Les 3 éléments spéciaux sont *None*, représentant le vide, *True* et *False* sont les valeurs booléennes.

4.3. Les fonctions à connaître dès le départ

Une *fonction* est un algorithme écrit pour réaliser une opération bien spécifique. En Python, une fonction s'écrit **toujours** avec un nom de fonction, suivi d'un couple de parenthèses. Par exemple `print(...)` ou `help(...)`. Le contenu des parenthèses peut être vide ou contenir des *paramètres*. Nous verrons tout cela ultérieurement.

La première fonction que l'on va utiliser est la fonction `print(...)`. Cette fonction va IMPRIMER des affichages sur la sortie standard (la console).

La seconde fonction que l'on va utiliser est la fonction `help(...)`. Cette fonction va nous AIDER en affichant la syntaxe d'écriture d'une fonction. Par exemple, tapez dans la console python: `help(print)` pour afficher la syntaxe de la fonction `print`.

La troisième fonction que l'on va utiliser est la fonction `input(...)` -> `str`. Cette fonction permet de LIRE une entrée au clavier. Le résultat de cette fonction sera **toujours** du texte. Nous verrons plus loin comment transformer du texte en nombre.

4.4. Exemple : Hello World


Écrivez un programme Python qui affiche *Hello World* à l'écran.

Ouvrez votre éditeur de texte préféré et tapez le code suivant :

```

1  """
2  Hello World
3  __author__ = "VCO"
4  """
5  print("Hello World")

```

 Python

5. Les valeurs littérales

Une valeur littérale signifie que l'on va utiliser « directement » la valeur. Par exemple, le nombre **2** représente toujours lui-même et rien d'autre. C'est une « constante » car sa valeur ne change pas. Les valeurs littérales correspondent à des nombres ou des chaînes de caractères.

Note

En Python, les valeurs littérales sont en réalité des *objets* sur lesquels on peut appliquer certaines méthodes. Nous les verrons plus tard.


5.1. Nombres

Les nombres en Python sont principalement divisés en deux types : les **entiers** et les **flottants**. Un nombre entier (*int*) est par exemple **2**. Un flottant (*float*) est un nombre à virgule, par exemple **5.12** ou **5.12E-4**. La notation scientifique (*E*) indique des puissances de 10.

5.2. Chaînes de caractères

Une chaîne de caractères est une *suite* de caractères. Les chaînes de caractères peuvent être entourées de guillemets simple (' '), de guillemets doubles (" "), de guillemets triples (''' ''') ou de guillemets triple doubles (""" """). Les guillemets triples vous permettront d'écrire du texte sur plusieurs lignes.

```
1 '''
2 Ceci est une chaîne multi lignes. Ceci est la première ligne.
3 Ceci est la seconde ligne.
4 '''
```

 Python

5.3. Caractère d'échappement et de contrôle

Supposons que vous souhaitez avoir une chaîne de caractères contenant un simple guillemet ('). Si vous déclarez votre chaîne comme ceci : `'comment t'appelles-tu ?'`, Python ne va pas comprendre. Pour lui, le texte s'arrêtera aux guillemets `'comment t'` et le reste sera interprété comme un code erroné.

Pour indiquer à Python que le guillemet n'est pas la fin de la chaîne, nous pouvons remplacer les guillemets encadrants par des guillemets double (`"comment t'appelles-tu ?"`), soit en **échappant** le caractère. Pour échapper un caractère, il faut le précéder d'un backslash (\).

De la même manière, certains caractères échappés ont un comportement spécial :

\n Permet de forcer un passage à la ligne dans une chaîne de caractères.

\t Permet d'insérer une tabulation dans la chaîne de caractères.

\r Permet de revenir au début de la ligne.

Il existe d'autres caractères d'échappement, mais nous les verrons en temps utiles.

6. Les variables

6.1. Principe

Pour comprendre la notion de variable, il faut savoir comment la mémoire de l'ordinateur fonctionne. Nous pouvons voir la RAM comme une succession de cases numérotées en **hexadécimal**. Pour pouvoir stocker des valeurs dans ces cases mémoires, il faut connaître leur *adresse*. Seulement, les adresses mémoire ne sont pas figées. Pour cela, nous allons apposer des *étiquettes* sur ces cases, afin de les identifier. C'est ce qu'on appelle une **variable**.

En informatique, une variable est l'association d'une étiquette à une valeur. La variable représente la valeur et se substitue à elle. De plus, la variable, comme son nom l'indique, est *variable* dans le temps.

Une *variable* fait exactement ce que son nom indique. Leur valeur peut changer et vous pouvez stocker n'importe quoi dans une variable. C'est juste un endroit où l'on range l'information dans la mémoire de l'ordinateur.

Quand on attribue une valeur à une variable, on dit qu'on l'**affecte**. Le symbole de l'affectation en Python est le symbole `=`.

6.1.1. Comment nommer une variable ?

On peut donner n'importe quel nom à une variable, mais en respectant quelques règles :

- Le nom de la variable doit expliquer son but.
- Le nom de la variable ne doit pas être trop long.
- Le nom de la variable ne doit pas commencer par un chiffre.
- Le premier caractère du nom doit être une lettre de l'alphabet ou un underscore (`_`).
- Le reste du nom peut être composé de lettres ou de chiffres.
- Vous ne pouvez pas utiliser un mot-clé comme nom de variable.
- **Attention !** Python fait la différence entre les majuscules et les minuscules ! Ainsi, `mavar` et `MaVar` sont deux variables différentes !

6.2. Types et déclaration

Pour exister, une variable doit être **déclarée**, c.-à-d. qu'il faut indiquer comment elle s'appelle et ce qu'elle doit contenir. Pour déclarer une variable en Python, *il suffit de lui donner un nom*, puis de lui *affecter une valeur* avec le signe `=`. Inutile de la typer, même s'il est possible de le faire (`age: int = 42`)

Le typage d'une variable est l'indication du contenu de cette variable.

Type de donnée	Type	Plage de valeurs	Type Python
Byte	char	0 à 255	non
Entier simple signé	int	-32768 à +32767	oui
Entier simple non signé	uint	0 à 65535	non
Entier long signé	long	-2 147 483 648 à +2 147 483 647	non
Réel simple	float	beaucoup!	oui
Réel double	double	beaucoup plus, plus précis	non
Caractère	char	un caractère alphanumérique	oui
Texte*	str	une chaîne de caractères	oui
Booléen	bool	Vrai ou Faux (0 ou 1)	oui

Note

Le type **string** n'est pas un type de base, mais il est tellement utilisé qu'il a sa place dans ce tableau.

Note

Python ne gère pas les nombres non signés.

Pour connaître le type d'une variable, nous pouvons utiliser la fonction `type(...)`.

Note


Python est un peu particulier : le typage ne sert à rien, car Python déduit tout seul le type de données. Toutefois, typer une variable permet au développeur (et surtout à l'IDE) de prévenir en cas de problème. En fait, Python permet d'écrire votre code **comme vous le souhaitez**.

En Python, il n'est pas nécessaire de déclarer une variable dès le début du programme, mais uniquement quand on en a besoin.

```

1  """
2  affVar
3  __author__ = "VCO"
4  """
5  name: str = "Ella Stick" # explicit typing
6  age = 24 # implicit typing
7  tax: float # not needed but may be usefull later
8  tax = 0.06
9
10 type(name) # class 'str'
11 type(age) # class 'int'
12 type(tax) # class 'float'

```

 Python

Ne tenez pas compte du mot-clé `class`, nous verrons ça beaucoup plus tard. Si vous souhaitez supprimer une variable qui n'est plus utilisée, utilisez la fonction `del`. Exemple :

```
del name.
```


6.2.1. Transtypage

Parfois, il peut être nécessaire de changer le type d'une variable. On appelle ça le *transtypage*. Mais le transtypage peut parfois faire perdre des informations. Par exemple, quand on passe un `FLOAT` (12.4) dans un `INT`, on va perdre la partie décimale (0.4).

Pour transtyper, nous allons utiliser des fonctions spécifiques :

```
1 str(...) # cast to string
2 int(...) # cast to integer
3 float(...) # cast to float
```


 Python

6.3. Saisie et affichage

6.3.1. La fonction `print`

Pour rappel, pour afficher du texte dans la console, utilisez la fonction `print(...)`. Cette fonction permet de faire un peu plus que simplement afficher du texte. Si vous regardez dans l'aide de la fonction (`help(print)`), vous verrez qu'elle accepte plusieurs *paramètres* (un paramètre est un élément à l'intérieur de la parenthèse.)

```
1 print("hello", "world")
```


 Python

6.3.2. La fonction `input`

La fonction `input(...)` va nous permettre d'attendre que l'utilisateur entre une donnée puis valide en tapant sur la touche *enter*. Nous pouvons invoquer la fonction `input()` sans paramètres, mais pour cela, nous devons au préalable faire un `(...)` pour spécifier à l'utilisateur ce qu'il doit entrer.

Cependant, la fonction `input(...)` de Python possède une fonction un peu spéciale : on peut entrer le *prompt* en paramètre de la fonction. C'est d'ailleurs ainsi que nous allons l'utiliser.


```
1 #pseudocode method
2 print("Entrez une valeur entre 1 et 10: ")
3 value = input()
4
5 #pythonic method, shorter
6 value = input("Entrez une valeur entre 1 et 10: ")
```

 Python

Note

Attention, une entrée au clavier sera toujours de type ***string***. Si vous demandez un nombre à l'utilisateur, il faudra le *transtyper*.

```
1 #cast
2 int_value = int(input("Entrez un nombre entier"))
3 float_value = float(input("Entrez un nombre réel"))
4 str_value = input("Entrez un texte") #already a string !!
```

 Python


Pour le moment, considérons que l'utilisateur fait ce qu'on lui demande. Plus tard, nous devrions vérifier que l'utilisateur a bien entré le type de valeur demandé.

6.3.3. Formatage des chaînes de caractères

Nous avons parfois besoin de fabriquer des chaînes de caractères à partir des variables. Dans ce cas, il existe plusieurs méthodes : une ancienne (avant Python 3.6) et une moderne.

L'ancienne version nous allons utiliser la méthode `format`. Une chaîne de caractères peut utiliser certaines spécifications et la méthode `format` va remplacer les spécifications par une variable. Par exemple :


```
1 name = "Ella Stick"
2 age = 24
3
4 print("Bonjour {0}, vous avez {1} ans".format(name,age))
```

 Python

Nous utilisons `{0}` et `{1}` qui correspondent aux emplacements des variables `name` et `age`. Ensuite, dans la méthode `format` (notez que l'on appelle cette méthode directement après les guillemets fermants), nous indiquons dans l'ordre les variables à afficher.

La nouvelle version Depuis Python 3.6, nous pouvons utiliser un moyen plus court, les *f-strings*. Une f-string est une chaîne de caractères précédée de la lettre **f**.

```
1 name = "Ella Stick"
2 age = 24
3 print(f"Bonjour {name}, vous avez {age} ans")
```

 Python

6.4. Les constantes

En Python, les constantes ***n'existent tout simplement pas***. On utilisera à la place des variables dont on évitera de modifier la valeur, ce qui peut évidemment poser des problèmes. Toutefois, nous noterons les constantes en MAJUSCULES.

7. Opérateurs et calculs

Attardons-nous sur un aspect important de l'algorithmique qui est l'utilisation des **opérateurs**. Comme son nom l'indique, un *opérateur* est utilisé pour réaliser une opération. Nous avons déjà vu un opérateur particulier, l'opérateur d'affectation (=). Il existe d'autres opérateurs, qui servent aux calculs, aux comparaisons, au groupage, à la rotation de bits...

7.1. Les opérateurs arithmétiques


Il existe deux types d'opérateurs: les opérateurs **binaires**, qui nécessitent d'avoir un opérande à gauche et à droite, et les opérateurs **unaires** nécessitant un seul opérande. Le résultat d'une opération mathématique donnera le même type que le type des opérandes, à deux exceptions près : le résultat d'une division donnera **toujours** un *réel*, ainsi que le résultat d'une opération entre un entier et un réel (ce qui est logique, un entier est le cas particulier de l'ensemble des réels).

Nom	Symbole
Addition	+
Soustraction	-
Multiplication	*
Exponentiation	**
Division réelle	/
Division entière	//
Modulo	%

L'ordre de priorité des opérateurs est le même qu'en arithmétique. Il est à noter que la plupart des opérateurs peuvent être **surchargés**, c.-à-d. qu'ils peuvent être réécrits pour effectuer autre chose que leur fonction de base.

Exemples :

```
1 print(7 // 2 ) # 3
2 print(7%2) #1
```

 Python

Les opérateurs unaires s'utilisent devant un nombre. Ils laissent le nombre tel quel (+), le transforme en son opposé (-) et peuvent s'enchaîner. L'opérateur ~ permet d'inverser le signe d'un nombre.

+	-	~
---	---	---

Opérateurs unaires

Note

Dans certains langages, il existe des opérateurs d'*incrément* (++) et de *décrément* (--), qui peuvent être placés avant ou après un nombre. Cette notation est jugée inutile en Python.

Enfin, le signe = peut être considéré comme un opérateur. Il permet d'affecter une valeur à une variable, mais **il n'est pas possible de le surcharger**.

7.2. Les opérateurs de comparaison

Les opérateurs de comparaison sont des opérateurs binaires permettant de comparer deux éléments. L'ordinateur va évaluer la comparaison et répondra un résultat booléen, soit TRUE, soit FALSE. Les opérateurs de comparaison sont :

Nom	Symbole	Exemple
Égalité	==	a == 3
Inégalité	!=	a != 3
Infériorité stricte	<	a < 3
Infériorité inclusive	<=	a <= 3
Supériorité stricte	>	a > 3
Supériorité inclusive	>=	a >= 3

7.3. Les opérateurs booléens

Les opérateurs booléens sont des opérateurs spécifiques qui permettent de faire des combinaisons binaires. Comment faire si nous avons besoin d'effectuer plusieurs comparaisons, sachant que chaque comparaison peut répondre TRUE ou FALSE ? Il faut utiliser des opérateurs booléens pour combiner ces résultats.

Il existe 4 opérateurs booléens, produisant des résultats différents. Les opérateurs booléens sont régis par la logique booléenne (conçue par G. Boole). Ces différents opérateurs booléens peuvent être facilement compris en utilisant des **tables de vérités**.

7.3.1. L'opérateur **not**

L'opérateur **not** est l'un des plus simples à comprendre : il **inverse** le résultat qu'on lui donne.

a	$\neg a$
0	1
1	0

7.3.2. L'opérateur **and**

L'opérateur **and** indique que **si** la première assertion **et** que la seconde assertion sont *vraies*, **alors** le résultat sera *vrai*.

a	b	$a \wedge b$
0	0	0
1	0	0
0	1	0
1	1	1

7.3.3. L'opérateur **or**

L'opérateur **or** indique que **si** la première assertion **ou** que la seconde assertion soit *vraie*, **alors** le résultat sera *vrai*.

a	b	$a \vee b$
0	0	0
1	0	1
0	1	1
1	1	1

7.3.4. L'opérateur xor

L'opérateur xor est un opérateur un peu particulier. Il indique que **si** la première **ou** la seconde assertion soit *vraie*, **mais pas les deux en même temps**, alors le résultat sera *vrai*.

a	b	$a \oplus b$
0	0	0
1	0	1
0	1	1
1	1	0

7.3.5. Les opérateurs de comparaison

>	>=	<	<=	!=	==	in	not in	is
---	----	---	----	----	----	----	--------	----

Les opérateurs de comparaison

~	&		>>	<<	^
---	---	--	----	----	---

Les opérateurs booléens bits à bits

7.4. Les opérateurs d'affectation

Il existe d'autres opérateurs dits d'**affectation** qui modifient la variable courante.

+=	-=	*=	**=	/=	//=	%=	&=		^=	<<=	>>=
								=			

Opérateurs d'affectation

Ces opérateurs font en réalité deux choses en même temps. Par exemple `a**=2` signifie que le contenu de la variable **a** sera mis au carré, puis le résultat sera réaffecté dans la variable **a**.

8. Structure conditionnelle

Dans le chapitre précédent, nous avons vu qu'il existait des *opérateurs de comparaison*. Ces opérateurs et ces expressions booléennes trouvent toute leur utilité dans les *structures conditionnelles*. Une structure conditionnelle va permettre d'effectuer une action ou une autre selon le résultat d'une expression évaluée comme vraie ou fausse.

Grâce aux opérateurs booléens, l'expression peut être composée : $a = 1$ or $(b * 3 = 6)$ and $c > 10$ est une expression tout à fait valable.

Imaginons un algorithme permettant de concevoir un petit jeu, celui du **c'est plus, c'est moins**. L'ordinateur choisit un nombre entre 1 et 100 au hasard et le joueur doit tenter de le deviner. L'ordinateur doit indiquer au joueur si le nombre à deviner est plus grand ou plus petit que le nombre donné par le joueur. Nous voyons ici qu'il faut un test conditionnel.

Une condition est donc une *assertion* (une *affirmation*) dont la réponse sera soit VRAIE OU FAUSSE.

Il est aussi possible que la condition soit une variable booléenne.

8.1. Cas particulier

Attention ! Selon certains langages, la condition suivante $1 \leq n \leq 100$ (un nombre n entre 1 et 100) peut provoquer des résultats incohérents, voire tout à fait erronés. Les opérateurs ont un ordre de priorité, et pour les ordres de priorités équivalents, l'expression est évaluée de gauche à droite.

```
1 BEGIN
2     VAR a: INT <- 150
3     WRITE 1 <= a <= 100
4 END
```

Quels seraient les résultats affichés ? Tout va dépendre du langage utilisé. Comme les expressions sont évaluées de gauche à droite, la première affirmation sera évaluée comme tel : $1 \leq a$ est `TRUE`, puis `true` ≤ 100 peut poser un problème. En effet, il est difficile de comparer un booléen avec un entier (même si dans la plupart des langages, la valeur `TRUE` vaut `1` en entier, ce qui signifie que la suite de l'assertion sera `TRUE` aussi, alors que ce n'est pas le résultat attendu !)

Il faudra alors découper cette information en deux affirmations distinctes et les combiner : $1 \leq n \wedge n \leq 100$.

Note


Python est un des rares langages qui traite correctement ce cas particulier !

8.2. L'instruction `if`

En Python, la structure conditionnelle est l'instruction `if... [elif]...[else]`. Elle peut s'écrire de plusieurs façons :

8.2.1. Forme simple


```
1 if assert:
2     #code
```

 Python

Notez ici que est l'expression booléenne de condition. **Si** la condition est vraie, **alors** le bloc d'instructions sera exécuté. Dans le cas contraire, le bloc est ignoré et le programme continue.

Exemple :


```
1 """
2 Write an algorithm that displays the absolute value of a number.
3 __author__ = "VCO"
4 """
5 value = int(input("Entrez un nombre positif ou négatif"))
6 if value < 0:
7     value = ~value
8 print("La valeur absolue est ",value)
```

 Python

8.2.2. Forme complexe

La forme complexe n'a de complexe que le nom. La forme complexe peut aussi exécuter du code si la condition est **FALSE**.

```
1 if assert:
2     #code if true
3 else:
4     #code if false
```


 Python

Si la condition est **TRUE**, le bloc d'instruction sous le **if** sera exécuté, ce qui ne diffère pas de la forme simple. Par contre, si la condition est **FALSE**, c'est le code situé sous la condition **else** qui sera exécuté.

8.2.3. Embranchements multiples

L'embranchement multiple est une structure qui permet de disposer de plusieurs **voies**. En effet, il se peut parfois que la première assertion soit **FALSE** sans pour autant qu'il faille exécuter la section **else**. Nous allons utiliser un mot clé supplémentaire : **elif**.

```
1 if assert1:
2     #code if assert1 is true
3 elif assert2:
4     #code if assert2 is true
5 else:
6     #code if false
```

 Python

Il est à noter que l'on peut avoir autant de structures **elif** que l'on souhaite, pour un maximum de 255 branchements.

8.3. Tests imbriqués

Soit le problème suivant : « Nous souhaitons afficher le lendemain d'une date entrée sous forme de 3 valeurs : le jour, le mois et l'année. »

Il faudra donc gérer :

- les changements de mois ;
- le nombre de jours dans le mois ;
- le changement d'année ;

- les années bissextiles pour le mois de février. Une année est bissextile si elle est divisible par **4** et par **400** mais non divisible par **100**.

Nous avons donc plusieurs conditions à tester et certaines sont imbriquées.

```

1  """
2  tomorrow
3  __author__ = vco
4  """
5  flag = False # no need to type variables!
6  day = int(input("Entrez le jour: ")) # input returns a string. We need to cast value
   to int by explicit cast
7  month = int(input("Entrez le mois: "))
8  year = int(input("Entrez l'année: "))
9
10 day +=1
11
12 if month == 1 or \
13     month == 3 or \
14     month == 5 or \
15     month == 7 or \
16     month == 8 or \
17     month == 10 or \
18     month == 12:
19     if day > 31: # new block of code inside if => indent this block!
20         flag = True # new block
21 elif month == 2: # notice indentation. Aligned to first "if" block
22     if (year%4) == 0 and ((year%400) == 0 or (year%100)>0):
23         if day > 29:
24             flag = True
25         elif day > 28:
26             flag = True
27 else:
28     if day > 30:
29         flag = True
30
31 if flag:
32     day = 1
33     month += 1
34
35 if month > 12:
36     month = 1
37     year += 1
38
39 print(f"Le lendemain de la date entrée est le {day}/{month}/{year}")

```

Détaillons un peu ce code (notre premier véritable algorithme !). À la ligne **5**, nous créons une variable booléenne qui nous permettra d'indiquer quand la date doit se décaler au mois prochain. Cela nous évite de dupliquer du code.

Note

Dans tous les cas, on essaiera d'éviter la **duplication de code**.

À la ligne **12**, nous avons notre première condition multiple. Nous détectons ainsi si le mois entré est un mois de 31 jours.

À la ligne **19**, nous avons notre première structure imbriquée. Si le jour dépasse 31, nous positionnons le drapeau à `TRUE` pour effectuer un traitement ultérieur.

À la ligne **31**, nous voyons l'utilisation de notre `.` Puisqu'il s'agit d'une variable booléenne, il est inutile de tester si elle est vraie ou fausse, car cette variable contient la réponse !

9. Structures itératives

Les boucles font partie des grandes structures de base de l'algorithmique, avec des structures conditionnelles. Une fois ces structures maîtrisées, le reste du cours ne sera qu'une dérivation de ces structures de base.

Une structure *itérative* (ou *répétitive*) est une séquence d'instructions destinées à être exécutées plusieurs fois. Selon le type de boucle, le bloc de code va être répété un nombre fini de fois, ou selon une certaine condition.

Note

La boucle est un élément très simple à première vue, pourtant elle peut être extrêmement piègeuse ! Une simple erreur de condition peut amener une boucle infinie, la calamité des développeurs. À l'inverse, si la condition de sortie est atteinte avant d'exécuter la boucle, celle-ci ne sert à rien.

9.1. But d'une boucle

Prenons le cas d'une saisie au clavier (une lecture), où par exemple, le programme pose une question à laquelle l'utilisateur doit répondre par *Oui* (o) ou *Non* (n). Mais tôt ou tard, l'utilisateur, facétieux ou maladroit, risque de taper autre chose que la réponse attendue. Dès lors, le programme peut planter soit par erreur d'exécution (parce que le type de réponse ne correspond pas au type attendu), soit par une erreur fonctionnelle (il se déroule normalement jusqu'au bout, mais en produisant des résultats fantaisistes).

Alors, dans tout programme un tant soit peu sérieux, on met en place ce que l'on appelle un *contrôle de saisie* afin de vérifier que les données entrées au clavier correspondent bien à celles attendues par l'algorithme.

À vue de nez, on pourrait essayer avec un `if`.

```
1 '''
2 coffee
3 __author__ = "VCO"
4 '''
5 rep = input("Voulez-vous un café? (o/n): ")
6 if rep != "o" and rep != "n":
7     print("Saisie erronée. Recommencez.")
8     rep = input("Voulez-vous un café? (o/n)")
```

Python

C'est impeccable. Du moins tant que l'utilisateur a le bon goût de ne se tromper qu'une seule fois et d'entrer une valeur correcte à la deuxième demande. Si l'on veut également bétonner en cas de deuxième erreur, il faudrait ajouter un `if`, et ainsi de suite. On pourrait ajouter dès lors des centaines de `if` et écrire un algorithme aussi lourd qu'une blague des Grosses Têtes. On n'en sortira pas, il y aura toujours moyen qu'un acharné flanque le programme par terre.

La solution consistant à aligner des `if` en pagaille est donc une impasse. La seule issue est de poser une structure *itérative* qui répète la demande de saisie *tant que l'utilisateur obstiné ne tape pas sur o ou sur n*. Voyons ce type de boucle.

En Python, il n'y a que deux structures itératives, le `while` et le `for...in`.

9.2. La boucle *indéfinie* (**while** – tant que)

En français, nous aurions dit : **Tant que** l'utilisateur ne tape pas sur **o** ou **n**, on répète la demande. En algorithmique, on fait *presque* pareil.

En Python, la boucle **while** exécute un bloc d'instructions **tant que la condition est vraie**. Néanmoins, la boucle **while** peut avoir une clause optionnelle **else**.

```
1 '''
2 coffee
3 __author__ = "VCO"
4 '''
5 rep = ""
6 while not (rep == "o" or rep == "n"): # we can also write while rep != "o" and rep !=
  "n" but I prefer this way.
7     rep = input("Voulez-vous un café? (o/n): ")
```

Notez que l'instruction **while** se termine par un double-point (:) et un passage à la ligne indentée pour former le bloc d'instructions.

Si l'assertion de boucle vérifiée est **vraie**, la boucle exécute le bloc d'instructions. Ensuite, il remonte au **while** et un nouveau tour de boucle démarre.

Remarquez aussi, en ligne 5, que nous avons affecté une variable **avant** la boucle, servant à la condition. Cette affectation doit avoir pour résultat de **provoquer l'entrée obligatoire dans la boucle**. L'affectation doit pouvoir faire en sorte que l'assertion soit **vraie** pour déclencher le premier tour de la boucle.

Dans notre exemple, on peut donc affecter à n'importe quelle valeur, hormis **o** ou **n**, car dans ce cas, l'exécution sauterait la boucle, et ne serait pas du tout lue au clavier.

Nous pouvons également affecter à le résultat d'une **demande préalable** via un **input**. Il faut remarquer que les deux solutions (lecture initiale de la variable en dehors de la boucle ou affectation forçant le passage dans celle-ci) rendent toutes deux l'algorithme satisfaisant, mais présentent une différence assez importante dans leur structure logique.

En effet, si l'on choisit d'effectuer une lecture préalable de la variable, la boucle ultérieure sera exécutée uniquement dans l'hypothèse d'une mauvaise saisie initiale. Si l'utilisateur saisit une valeur correcte à la première demande de , l'algorithme passera outre la boucle sans entrer dedans.

En revanche, avec la deuxième solution (celle d'une affectation préalable de la variable), l'entrée dans la boucle est forcée, et l'exécution de celle-ci est rendue obligatoire **au moins une fois** à chaque exécution du programme.

Du point de vue de l'utilisateur, cette différence est tout à fait mineure ; et à la limite, il ne la remarquera même pas. Mais du point de vue du programmeur, il importe de bien comprendre que les cheminements des instructions ne seront pas les mêmes dans un cas et dans l'autre.

Le fait de forcer l'entrée dans une boucle permet d'utiliser une structure algorithmique de boucle qui n'existe pas en Python : la boucle **Do While**. Ce type de boucle permet de passer **au moins une fois** dans la boucle **avant** de faire la vérification.

Note

Attention à la condition ! Il n'y a rien de plus frustrant que d'écrire une structure `while` dans laquelle la condition n'est **jamais vraie**. Le programme ne rentre alors jamais dans la superbe boucle sur laquelle vous avez tant sué !

Mais la faute symétrique est au moins aussi désopilante. Elle consiste à écrire une boucle dans laquelle la condition n'est **jamais fausse** ! L'ordinateur tourne alors dans une boucle comme un dératé et n'en sort plus. Seule solution, quitter le programme avec un démonte-pneu ou un bâton de dynamite. La **boucle infinie** est une des hantises les plus redoutées du programmeur. C'est un peu comme le verre baveux, le poil à gratter ou le bleu de méthylène : c'est éculé, mais ça fait toujours rire.

Cette faute de programmation grossière – mais fréquente – ne manquera pas d'égayer l'ambiance collective de la formation... et accessoirement d'étancher la soif proverbiale de vos enseignants.

Cependant, la boucle infinie est parfois utile, quand elle est sagement maîtrisée.


9.3. La boucle *finie* (`for... in`)

Il arrive très souvent qu'on ait besoin d'effectuer un nombre déterminé de passages dans une boucle. Or, à priori, notre structure `while` ne sait pas à l'avance combien de tours de boucle elle va effectuer (puisque le nombre de tours dépend de la valeur d'un booléen). Une solution serait d'utiliser un *compteur*. C'est pourquoi une autre structure de boucle est à notre disposition.

L'instruction `for...in` est une instruction pour réaliser des boucles qui itèrent sur une *séquence d'objets*, c.-à-d. passe en revue chaque élément d'une séquence.

Pour pouvoir utiliser cette instruction, il va nous falloir une *séquence*. Nous verrons les séquences plus tard, mais retenez pour l'instant qu'il existe la fonction `range(x,y)` qui va créer une séquence de chiffres de x à y .

```
1 '''
2 inside
3 __author__ = "VCO"
4 '''
5 for first in range(15):
6     print("Il est passé par ici")
```

 Python

Ici aussi, l'instruction se termine par un double point. Comme pour la boucle `while`, l'instruction `for` peut avoir une clause optionnelle `else`.

Note

Pour spécifier un pas spécial, nous pouvons l'ajouter dans la fonction `range(x,y,step)`. Le dernier argument est l'indication de pas.

Note

Attention ! La fonction `range(x,y)` va produire une séquence allant de x à y .

Insistons : la structure `for...in` n'est pas du tout indispensable ! On pourrait fort bien programmer toutes les situations de boucle uniquement avec un `while`. Le seul intérêt du `for...in` est d'épargner un peu de fatigue au développeur, en lui évitant de gérer lui-même la progression de la variable qui lui sert de compteur ou de pointeur sur un élément de séquence.

9.4. Quel type de boucle à utiliser et quand ?

La structure `while` est employée dans les situations où l'on doit procéder à un traitement systématique sur les éléments d'une collection **dont on ne connaît pas à l'avance la quantité**, comme :

- le contrôle d'une saisie,
- la gestion des tours d'un jeu (tant que la partie n'est pas finie, on recommence),
- la lecture des enregistrements d'un fichier de taille inconnue,
- ...

Les structure `for...in` sont employées dans les situations où l'on doit procéder à un traitement systématique sur les éléments d'une collection **dont on connaît à l'avance la quantité**.

Nous verrons dans les chapitres suivants des séries d'éléments nommés *tableaux*, *séquences* et *chaînes de caractères*. Selon les cas, le balayage systématique de tous les éléments de ces séries pourra être effectué par un `for...in`

9.5. Boucles imbriquées

De même que les poupées russes contiennent d'autres poupées russes qui contiennent d'autres poupées russes qui..., de même qu'une structure `if` peut contenir d'autres `if`, une boucle peut tout à fait contenir d'autres boucles (y'a pas de raison...).

```
1  ''' inside
2  __author__ = "VCO"
3  '''
4  for i in range(15):
5      print("Il est passé par ici")
6      for j in range(6):
7          print("Il repassera par là")
```

Dans cet exemple, le programma écrira une fois **Il est passé par ici**, puis six fois de suite **Il repassera par là**, avant de recommencer. À la fin, il y aura donc eu $15 * 6 = 90$ passages dans la seconde boucle.

Pourquoi imbriquer des boucles ? Pour la même raison que l'on imbrique des tests. Une boucle, c'est un traitement systématique, un examen d'une série d'éléments un par un (par exemple, « prenons les étudiants de cette classe un par un »). Eh bien, on peut imaginer que pour **chaque** étudiant, on doit procéder à un examen systématique d'autre chose (par exemple, « prenons chacun des cours suivis »). Voilà un exemple typique de boucle imbriquée : on devra programmer une boucle principale (celle qui prend les étudiants un par un), et à l'intérieur, une boucle secondaire (celle qui prend les cours suivis.)

Dans la pratique de la programmation, la maîtrise des boucles imbriquées est nécessaire, même si ce n'est qu'une fraction des connaissances à assimiler.

9.6. Casser une boucle (**break** et **continue**)

Dans certains cas, il est nécessaire de **casser** l'exécution d'une boucle, afin d'en sortir avant la fin de son traitement. Il existe pour cela deux commandes :

- la commande **break** casse la boucle directement et sort de celle-ci, par exemple pour arrêter l'exécution d'une instruction qui boucle, même si la condition de la boucle n'est pas devenue **FALSE** ou si la séquence d'éléments n'est pas consommée. Une chose importante est à noter : Dans le cas d'un **break**, la clause optionnelle **else** des boucles **n'est pas exécutée**.
- la commande **continue** ramène au début de la boucle en annulant le traitement en cours.

```
1 '''
2 break
3 __author__ = "VCO"
4 '''
5 for i in range(10):
6     if i == 2:
7         break
8     else:
9         print(i)
```

```
1 '''
2 continue
3 __author__ = "VCO"
4 '''
5 for i in range(10):
6     if i == 2:
7         continue
8     else:
9         print(i)
```

10. Fonctions

Un programme, surtout s'il est long, a toutes les chances de devoir procéder plusieurs fois aux mêmes traitements de données, à différents endroits de son déroulement. Par exemple, la saisie d'une réponse par oui ou par non et le contrôle y afférant, peut être utilisé plusieurs fois dans un programme.

Bien entendu, on pourrait réécrire plusieurs fois ce bout de code, à différents endroits, mais cela pose un problème : imaginez que votre code comporte un bug non détecté à l'origine. Si ce code est dupliqué plusieurs fois dans votre programme, il faudra corriger plusieurs fois ce bug. Et si votre programme fait plusieurs milliers de lignes, retrouver ce bout de code sera particulièrement fastidieux.

Nous allons donc opter pour une stratégie plus efficace en termes de programmation. Nous allons **éviter les doublons de code** ! Le code ne sera écrit qu'une fois et pourra être réutilisé n'importe où dans le programme. Et l'on peut même faire mieux ! On peut rassembler nos bouts de code dans une *bibliothèque de fonctions*, aussi appelé *module*, afin qu'ils soient réutilisables dans **tous** nos programmes.

Le programme devient **modulaire**, et il suffira de faire une modification au bon endroit pour que cette modification soit effective dans toute l'application.

De plus, lorsqu'un programme est très long, il est plus lisible et maintenable de le découper en plusieurs fonctions réutilisables.

Nous allons voir comment créer des *sous-programmes*. En algorithmique, nous allons définir deux types de sous-programmes : les **procédures**, qui sont des sous-programmes qui effectuent un traitement **sans renvoyer de résultat** et les **fonctions**, qui sont similaires aux procédures, mais qui **renvoient un résultat**.

Note

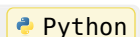
Bien entendu, Python simplifie un peu tout ça... Il n'y a **pas de procédure en Python**, mais uniquement des **fonctions** ! Nous pouvons toutefois écrire des fonctions qui ne renvoient aucun résultat.

10.1. Les fonctions

Une fonction est une portion de code dont le rôle est d'effectuer un traitement et de renvoyer une réponse au programme principal. Reprenons notre exemple ci-dessus, le traitement de l'affichage d'une question de type Oui ou Non.

En Python, pour créer une fonction, nous utiliserons la syntaxe `def fonction_name(args)`.

```
1 """
2 Yes or No
3 __author__ = "VCO"
4 """
5 def yes_or_no() -> str:
6     text = ""
7     while text != "oui" and text != "non":
8         text = input("Tapez oui ou non: ")
9     return text
```



Expliquons ce bout de code. On définit la fonction `yes_or_no()` avec un couple de parenthèses. Les parenthèses sont importantes, ce sont elles qui définissent la fonction (et le passage de paramètres, voir plus loin). On donne également le type de retour de la fonction (optionnel, mais vivement conseillé). Dans ce cas, la fonction renverra du texte.

Le reste du code est trivial. On boucle tant que **oui** ou **non** n'est pas entré. En avant-dernière ligne, remarquez la présence du mot-clé `return` qui renvoie le résultat de la fonction (ici, la valeur de la variable `text`).

10.1.1. Exemple

```
1  """
2  Exemple of code
3  __author__ = "VCO"
4  """
5  def yes_or_no() -> str:
6      """ see above """
7
8  print("Êtes-vous marié ?")
9  rep1 = yes_or_no()
10 print("Avez-vous des enfants ?")
11 rep2 = yes_or_no()
```

10.2. Analyse fonctionnelle

Le plus compliqué en programmation, c'est de faire de l'**analyse fonctionnelle**. Il faut avoir le réflexe de *constituer systématiquement les fonctions adéquates quand on doit traiter un problème donné* et savoir découper son algorithme en différentes fonctions pour le rendre léger, lisible et performant. On parlera alors de **factorisation de code**.

La phase de conception d'une application qui suit l'analyse et qui précède l'algorithmique, donc la partie qui s'occupe du découpage en modules de code, s'appelle l'**analyse fonctionnelle** du problème.

Pour concevoir une application, nous allons passer par différentes étapes :

1. On identifie le problème à traiter, en inventoriant les fonctionnalités nécessaires, les tenants et aboutissants, les règles implicites et explicites... C'est l'**analyse**.
2. On procède au découpage de l'application en fonctions *spécialisées*. C'est l'**analyse fonctionnelle**.
3. On détaille le fonctionnement de chaque fonction. C'est l'**algorithmique**.
4. On procède sur machine à l'écriture du code source dans le langage voulu. C'est la **programmation**.

Note

Une fonction se doit d'être *spécialisée*. Elle n'est écrite que pour réaliser un **traitement spécifique**, le plus concis possible. Le piège est d'écrire des fonctions qui font trop de choses différentes, ce qui n'est pas très **KISS**...

10.3. Le passage de paramètres

Un *paramètre* est une variable qui n'existe que dans le corps de la fonction dans laquelle elle est spécifiée. Elle permet de faire passer une valeur quelconque du programme à la fonction.


Reprenons l'exemple ci-dessus avec la fonction `yes_or_no()`. Dans l'algorithme principal, nous affichons un message avant chaque appel de la fonction. C'est aussi un élément répétitif que l'on peut automatiser au sein de la fonction.

La fonction `yes_or_no()` doit pouvoir afficher un message quelconque avant de faire la demande à l'utilisateur de taper oui ou non. Cela implique que la fonction doit savoir qu'elle va recevoir une donnée ainsi que son type pour pouvoir la traiter. En appelant la fonction, il faudra désormais préciser quel message elle doit afficher avant de lire la réponse.

Note

Si vous vous dites que ça ressemble à la fonction `input()` en Python, vous avez totalement raison !

```
1 """
2 Yes or No v2
3 __author__ = "VCO"
4 """
5 def yes_or_no(message:str) -> str:
6     text = ""
7     while text != "oui" and text != "non":
8         text = input(message + " [oui/non]: ")
9     return text
```


 Python

Bien entendu, on peut passer un nombre illimité d'arguments. Les paramètres d'une fonction sont spécifiés à l'intérieur de la paire de parenthèses de la définition de la fonction, séparées par des virgules. Pour rappel, il n'est pas utile de typer les paramètres en Python, même si c'est vivement conseillé pour mieux comprendre la fonction.

10.3.1. Paramètres par défaut

Pour certaines fonctions, nous voudrions parfois que certains arguments soient *optionnels* et utilisent des valeurs par défaut si l'utilisateur ne les précise pas. On peut spécifier des valeurs par défaut en ajoutant au nom du paramètre dans la définition de la fonction un `=` suivi de la valeur par défaut. La valeur par défaut doit être immuable. Les paramètres par défaut sont toujours situés **en dernier dans la liste des paramètres**.

```
1 def repeat(msg:str, time:int = 1) -> None:
2     print((msg + " ") * time)
3
4     repeat("Hello")
5     # Hello
6
7     repeat("Hello",5)
8     # Hello Hello Hello Hello Hello
```

 Python

10.3.2. Paramètres nommés

Si nos fonctions disposent de nombreux paramètres par défaut et que l'on veut en positionner seulement quelques-uns, on peut donner des valeurs en *nommant* explicitement ces paramètres. Cela a aussi pour avantage de passer les paramètres dans n'importe quel ordre, apportant également une certaine lisibilité.


```
1 def connect_to_db(user:str = "root", pass:str = "", server:str
  = "localhost", port:int = 3342):
2     """
3     Connection to a MySQL server
4     """
5
6 connect_to_db(user="vco", pass="nopass")
7 #or
8 connect_to_db(server="myserver", user="vco")
```

 Python

10.3.3. Paramètres multiples

Autre subtilité du langage, si nous souhaitons passer un nombre inconnu d'arguments (idéalement de même type), nous allons préfixer le paramètre d'une étoile *. Tous les éléments passés en paramètres seront alors regroupés dans un *tuple*. Si nous préfixons de deux étoiles **, les arguments seront regroupés dans un *dictionnaire*.

```
1 def some_useful_function(a:str="", *b, **c):
2     print(a)
3
4     for item in b: # b is a tuple
5         print(b)
6
7     for key, value in c.items(): # c is a dictionary
8         print(key, "=", value)
9
10 some_useful_function("Hello", 12, 24, 34, key1=val1, key2=val2)
```

 Python

Note


Python est un des seuls langages qui permet d'écrire des fonctions renvoyant **plusieurs** résultats. Il suffit de retourner un *tuple*.

10.4. Les docstrings

Python dispose d'une fonctionnalité intéressante nommée *docstrings*, pour *documentation strings*. Les docstrings sont un outil important à utiliser systématiquement, car cela vous aide à documenter le programme. La *docstring* permet de documenter le code (et en particulier les fonctions). Il est même possible de générer une documentation complète à partir des sources.

Une *docstring* s'écrit en plaçant des triples guillemets (""" "" ou """" """) juste après la définition de la fonction.

```
1 def connect_to_db(user:str = "root", pass:str = "", server:str =
  "localhost", port: int = 3342):
```

 Python

```

2 """
3 Connection to a MySQL server
4 """

```

Note

En écrivant une docstring dans une fonction, on **déclare** cette fonction, même si le corps de la fonction est vierge !

Note

Depuis le début de ce syllabus, la plupart des algorithmes ont des *docstrings* !

Il existe plusieurs formes d'écriture des *docstrings* : la *docstring* en une ligne, et la *docstring* multilignes, plus complète (et généralement écrite en ReST). Les *docstrings* sont affichées via la commande `help()`. Par convention, la documentation est écrite en anglais.

10.5. Les fonctions récursives

Une fonction **récursive** est une fonction qui peut s'appeler *elle-même*. En clair, dans une fonction, on va rappeler la fonction. Un exemple de récursivité est l'utilisation de la *factorielle*. Nous la détaillerons juste après.

Il existe deux types de récursivité :

- la récursivité *simple* : la fonction s'appelle elle-même (comme la factorielle) ;
- la récursivité *croisée* : deux fonctions s'appellent l'une l'autre. La première fonction appelle la seconde, qui en retour appelle la première.

10.5.1. La factorielle

Le concept de récursivité étant complexe à appréhender, prenons l'exemple de la **factorielle**. Nous l'avons dit plus haut, la formule de la factorielle est $n! = \prod_{1 \leq n \leq n-1} \times n = 1 \times 2 \times 3 \times \dots \times (n-1) \times n$. En d'autres termes, la factorielle est le **produit de toutes les valeurs inférieures (0 exclu) ou égales à n**. Si n vaut **5**, la factorielle vaut dès lors : $1 \times 2 \times 3 \times 4 \times 5 = 120$.


Pour calculer cette factorielle, prenons là à l'envers : pour faire la factorielle de 5 (5!), il suffit de faire $5 \times 4 \times 3 \times 2 \times 1$. Or, $4 \times 3 \times 2 \times 1$ est la définition de 4!. On peut donc écrire que $5! = 5 \times 4!$. De la même manière, $4! = 4 \times 3!$, etc. On y voit donc un motif de **récursivité**.

Nous pouvons facilement écrire l'algorithme de la factorielle :

```

1 def factorial(n:int) -> int:
2     ''' factorial '''
3     return n * factorial(n-1)


```

 Python

La fonction `factorial` va s'appeler elle-même pour résoudre le problème. Le souci, puisque la fonction s'appelle elle-même, **elle va tourner en boucle infinie !!** (Pas tout à fait en pratique, Python va s'en rendre compte et générer une erreur).

Pour pouvoir s'arrêter, la fonction récursive doit **avoir une condition d'arrêt**, qui va renvoyer une valeur *finie* non générée par l'appel de la fonction. Dans le cas de la factorielle, la condition d'arrêt est $n = 1$.

```
1 def factorial(n:int) -> int:
2     ''' factorial '''
3     if n == 1 :
4         return 1
5     else:
6         return n * factorial(n-1)
```

 Python

En règle générale, on utilisera une fonction récursive pour diviser un problème complexe à résoudre en problèmes plus petits qui sont plus faciles à résoudre. On s'en servira principalement pour les tris, le parcours des arbres ou les recherches binaires.

La récursivité est plus coûteuse en mémoire que l'utilisation d'une simple boucle, car chaque appel de la fonction nécessite de stocker dans la pile mémoire un *point de retour* dans la fonction appelante. Afin de ne pas saturer cette mémoire, Python possède une limite d'appels récursifs (**1000** par défaut). Néanmoins, la récursivité est un outil parfois plus naturel et plus lisible que le traitement par boucles, surtout si le traitement est complexe, comme le parcours des arbres binaires.

11. Les tableaux

Jusqu'à présent, les types de données rencontrés sont nommés des *scalaires*, mis à part les **strings**. Un *scalaire* est un type de donnée qui ne représente qu'une seule valeur d'un seul type à la fois.

Un **string** n'est pas un scalaire. Il s'agit d'une **suite ordonnée de caractères**, même si les langages de programmation offre un type particulier pour ça (les justement).

Prenons un exemple : « Nous souhaitons écrire un programme qui calcule la moyenne des notes des étudiants de la classe. »

Pour le moment, la seule possibilité est de créer **une variable par note et par étudiant**. S'il y a 30 étudiants par classe, et 12 notes par étudiants, nous devrions créer **360 variables !**

De plus, cela pose un autre problème : comment pourrions-nous faire si le nombre d'étudiants ou de notes change ? Nous n'allons pas réécrire tout le programme ! (À l'époque, c'est ce que les scientifiques faisaient, en perçant leur bande de papier...).

L'idéal serait d'avoir une variable qui serait composée de plusieurs **cases** accessibles. Chaque case correspondrait à une note. C'est le rôle du **tableau**.

Un **tableau** (ou **array**) est un ensemble de valeurs représenté par un nom de variable dont chaque case est identifiée par un n°, nommé un *indice*. En Python, un élément de tableau est représenté par le nom de la variable auquel on accole l'indice entre crochets :
`note[0] = 15.`

Note

Un tableau n'est pas un type de données, mais une liste d'éléments d'un type donné. De plus, la numérotation des indices d'un tableau commence à **0**.

11.1. Les tableaux en Python

En Python, il n'y a pas vraiment de tableaux. On parlera plus facilement de *séquences*. Il existe plusieurs types de séquences : les chaînes de caractères, les listes, les tuples, les dictionnaires et les ensembles. Parmi ces séquences, on distinguera les séquences **mutables**, que l'on peut modifier directement sans changer son adresse en mémoire (tels que les listes, les dictionnaires et les ensembles) et les séquences **immuables**, impossible à modifier sauf en créant une nouvelle séquence (tels que les strings et les tuples).

11.1.1. Notion d'objet et de classe

Python est un langage à paradigme orienté objet (notamment). Même si nous reviendrons en détails sur la notation orientée objet, nous avons désormais besoin de quelques notions utiles pour l'utilisation des séquences.

Quand nous créons une variable `i: int = 0`, nous créons en réalité un **objet i** de **classe int**.

Une classe peut avoir des *méthodes*, c.-à-d. des fonctions définies pour être utilisées avec les objets de cette classe. Par exemple, pour la classe `lst = list()`, il existe une méthode `lst.append(...)` qui nous permettra d'ajouter des éléments en fin de liste. Notez que l'appel de la méthode de classe se fait en mettant un point (.) après l'objet, suivi de la méthode à appeler.

11.1.2. Tableaux dynamiques et listes

En Python, les tableaux sont *dynamiques*, c.-à-d. que l'on ne connaît pas à l'avance le nombre d'éléments que devra comporter le tableau. On pourrait déclarer un tableau avec un nombre conséquent d'éléments (genre 10 000 ou même 100 000 éléments), mais cela monopoliserait d'énormes quantités de mémoire pour rien et cette solution ne serait pas fiable de toute façon. Ce type de tableau est appelé une **liste**.

Pour créer une liste en Python, nous pouvons la déclarer de deux façons :


```
1 lst = list() # calling class #OR
2 lst = [] # shortest way
```

 Python

Nous indiquons ainsi que la liste `lst` contiendra un tableau à une dimension. Notez que la taille du tableau n'est pas précisée au moment de sa déclaration.


Nous pouvons aussi déclarer un tableau avec des valeurs prédéfinies en les séparant par des virgules à l'intérieur des crochets, ce qui est la déclaration la plus proche d'un tableau statique :

```
1 lst = [2,12,48,36,58,74]
```

 Python


Ou encore via un **générateur** pour créer des *listes en compréhension* (voir plus loin).

```
1 lst = [i for i in range(7)]
2 # [1,2,3,4,5,6]
```

 Python

L'accès aux données à l'intérieur de la liste se fait en donnant le numéro de l'indice entre crochets. Pour connaître la taille de la liste, nous allons utiliser la fonction `len(lst)` en passant en paramètre la liste à mesurer. Par exemple :

```
1 lst = [2,12,48,36,58,74]
2 print(lst[2]) # calling third element
3 # 48
```

 Python


Note

L'indice qui sert à désigner les éléments peut être un scalaire, une variable scalaire ou une expression calculée. Dans un tableau, la valeur d'un indice doit **toujours** :

- être égale au moins à 0. Dans la plupart des langages, le premier indice est 0. Ce qui signifie que `note[11]` est la 12^e valeur !
- être un nombre entier.
- être inférieure ou égale au nombre d'éléments du tableau (-1 car on commence de 0). Si vous dépassez le nombre de cases déclarées, le programme déclenchera une erreur.


De même, on peut parcourir la liste **à l'envers** en mettant un indice *négatif* et ainsi demander un élément à la fin du tableau (sans savoir sa taille !) :

```
4 print(lst[-1])
```

 Python

Mais ce n'est pas tout ! Python est également capable de *slicer* une liste, c.-à-d. d'extraire les éléments d'une liste via l'utilisation du double point (:). Il permet de séparer l'indice de début et celui de fin **en créant une nouvelle liste** :

```
6 print(lst[:3]) # calling first three elements
7 # [2,12,48]
8 print(lst[2:5]) # calling elements between [2 and [5
9 #[48,36,58]
10 print(lst[3:]) # calling last three elements
11 #[36,58,74]
12 print(lst[::2]) # calling even elements
13 #[2,48,58]
```


 Python

Bien entendu, on peut faire tout ceci indifféremment en utilisant des indices et un pas pouvant être positifs ou négatifs.

Attention ! On part du premier élément de la liste et on se déplace dans le sens du pas. Si l'indice d'arrivée est dans la direction opposée, la liste obtenue sera vierge.

L'indice d'un tableau permet également de remplacer la valeur à cet endroit :

```
10 lst[2] = 90
11 print(lst[2])
12 #90
13 list[3:5] = [15,30,45]
14 print(lst)
15 # [2,12,90,15,30,45,74]
```

 Python

On peut supprimer un élément de la liste via `del list[2]`. Attention avec `del`, ça détruit la case entière, réduisant la taille de la liste.

Pour dupliquer une liste, nous ne pouvons **pas** faire `lst_a = lst_b` (Faites-moi confiance pour le moment, nous verrons pourquoi plus tard). Pour la dupliquer correctement, le plus simple est d'utiliser la méthode `lst_a = lst_b.copy()` ou `lst_a = list(lst_b)`

11.1.3. Tableaux multidimensionnels


Prenons le cas de la modélisation d'un jeu de dames. Un jeu de dames se joue sur un damier carré de 8*8 cases, soit 64 cases en tout. Avec les outils que nous avons abordés jusqu'ici, le plus simple serait bien évidemment de modéliser notre plateau de jeu sous forme de tableau. Chaque case du tableau contiendra une valeur booléenne pour indiquer ou non la présence d'un pion. Chaque case du tableau sera numérotée, par exemple comme ceci :

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Si l'on se souvient des règles des dames, un pion ne peut se déplacer qu'en diagonale, sur une des 4 cases adjacentes. Nous pourrions bien évidemment nous en sortir avec un tableau unidimensionnel. Plaçons un pion sur la case **29**. Sur quelles cases pourrions-nous déplacer le pion ? Avec un tableau unidimensionnel, ça risque d'être compliqué à calculer. Nous voyons pourtant rapidement sur le dessin ci-dessus les cases **20,22,36** et **38**. Nous pouvons le faire facilement, car nous avons travaillé d'instinct en **deux dimensions**. Alors, pourquoi ne pas créer un tableau en 2D également ?

Pour créer un tableau à deux dimensions, nous devons créer **une liste dans une liste**.

```
1 lst = [[]]
2 # with comprehension list
3 lst = [[_ for i in range(8)] for j in range(8)]
```

 Python

Quel que soit la dimension du tableau, pour chaque dimension, nous ajouterons une paire de crochets. La taille de chaque dimension pourra être récupérée via la fonction [len](#).


12. Dictionnaires, tuples, ensembles

12.1. Les tuples

En Python, nous utilisons des **tuples** (*n-uplets* en français) pour stocker des données similaires à une liste. La principale différence entre un tuple et une liste est que le tuple est **immuable**. Le tuple contiendra un nombre fini (de par son immuabilité) d'éléments. On se servira souvent de tuples pour renvoyer plusieurs valeurs, représenter différentes données complexes (telles qu'un point en cartésien (x,y) ou des couleurs au format `rgb(r,g,b)`).

La création d'un tuple se fait comme une liste ou un dictionnaire, en plaçant des variables, des scalaires ou des objets entre parenthèses :

```
1 red = (255,0,0)
```


 Python

On peut créer un tuple vide simplement en écrivant un couple de parenthèses : `empty_tuple = ()`. Chose non utile, puisque le tuple est *non modifiable* !

12.1.1. Accès aux données

Un tuple se comporte comme un tableau, c.-à-d. qu'il est *indexé*. On accède aux données via leur index démarrant à 0 :


```
1 languages = ("Python","Kotlin","Rust")
2 print(languages[2])
3 # Rust
```

 Python

12.2. Les dictionnaires


Un **dictionnaire** en Python est une collection d'éléments qui permet de stocker des informations sous la forme de paire *clé:valeur*. La création d'un dictionnaire est similaire à celle d'un tableau, à la différence que l'on utilise des *accolades*.

```
1 dic = {} # empty dictionary
2 dic = dict() # same as above
```

 Python

Nous pouvons pré-charger un dictionnaire en plaçant les paires de *clés:valeurs* séparées par des virgules entre les accolades :

```
1 capitals = {
2     "Belgique": "Bruxelles",
3     "France": "Paris",
4     "Allemagne": "Berlin"
5 }
```

 Python

Dans l'exemple ci-dessus, nous avons un dictionnaire contenant trois éléments, où *Belgique* est la **clé** et *Bruxelles* la valeur associée à cette clé.

Note


Les clés du dictionnaire se doivent d'être *immuables*, comme des tuples, du texte, des integer....

Les clés du dictionnaire se doivent d'être également *uniques*. Toute duplication de clé effacera la précédente.

12.2.1. Ajout et mise à jour d'élément


Pour ajouter un élément à un dictionnaire, nous pouvons passer une clé *non existante* entre crochets ainsi que la valeur à ajouter :

```
1 capitals["Italie"] = "Rome"
```

 Python

La mise à jour d'un élément se fait de la même manière :

```
1 capitals["Belgique"] = "Wavre"
```

 Python

```
2 capitals.update("Belgique") = "Wavre" # same as above
```


Note

La méthode `update()` permet d'ajouter et de mettre à jour une valeur.

12.2.2. Accès aux éléments

Nous pouvons accéder aux valeurs d'un dictionnaire en passant la clé entre des crochets, comme l'indice d'un tableau. Nous pouvons aussi utiliser la méthode `get()` du dictionnaire :

```
1 print(capitals["Belgique"])
```

 Python


```
2 print(capitals.get("Belgique")) # same as above
```

Nous pouvons également utiliser la méthode `popitem()` qui permet de récupérer le *dernier élément inséré*.

12.2.3. Suppression d'éléments

Nous pouvons utiliser le mot-clé `del` pour supprimer un élément du dictionnaire, ou la méthode `pop()` :


```
1 del capitals["Belgique"]
```

 Python

```
2 capitals.pop("Belgique")
```

Nous pouvons aussi *vider* l'entièreté du dictionnaire via la méthode `clear()`.

```
1 capitals.clear()
```


 Python

12.3. Les ensembles

Un **set** (*ensemble* en français) est *une collection de données uniques*, signifiant que les éléments à l'intérieur d'un set **ne peuvent pas être dupliqués**.

La création d'un **set** utilise la même graphie qu'un dictionnaire, avec des *accolades*. Chaque élément est séparé par des virgules.

```
1 ids = {17,34,78,52}
```

 Python

Note


Dans un **set**, il n'y a pas de couple clé:valeur

Note

Un **set** n'est pas ordonné ! Ne soyez donc pas étonné si l'affichage est différent de l'insertion des éléments.

Pour créer un set vide, nous ne pouvons pas utiliser d'écriture rapide, sinon nous créerions un dictionnaire. La seule possibilité est d'utiliser la fonction `set()` :

```
1 empty_set = set()
```


 Python

12.3.1. Ajout et mise à jour de données

Les ensembles sont **mutables** (mais doivent contenir des données *immuables*), mais de façon non ordonnée, ce qui signifie qu'il n'y a pas d'index. On ne peut donc pas indexer ou *slicer* un ensemble.

Pour ajouter un élément dans le set, nous utiliserons la méthode `add()` :


```
1 ids.add(44)
```

 Python

La mise à jour est un peu plus complexe à comprendre, de par les limitations induites ci-dessus. En effet, **comment** mettre à jour un ensemble s'il n'y a pas d'index ?

Si l'on souhaite mettre à jour l'ensemble, nous allons passer une *séquence* d'éléments. Chaque élément non existant à la liste sera ajouté à l'ensemble. La méthode `update()` du set permet donc plus un ajout *massif* de données qu'une réelle mise à jour de celles-ci.


```
1 ids.update((89,77,21))
2 print(ids)
3 # {17,34,78,52,44,89,77,21}
```

 Python

12.3.2. Suppression d'élément

La suppression d'un élément se fait via la méthode `discard()` :

```
1 ids.discard(44)
```

 Python

Note

On peut faire plein de choses avec des ensembles, notamment en mathématiques... Mais ce ne sera pas vu dans le cadre de ce cours.


13. Les strings

Un **string** en Python est *une séquence de caractères*. Techniquement, un **string** est un *tableau immuable de caractères*. Toutefois, on utilise un type spécial (`str`) pour le définir, car il est massivement utilisé. On en a besoin très tôt sans toutefois utiliser toutes ses possibilités.

Un **string** est entouré par un couple de guillemets simples, doubles ou triples. Les guillemets simple et double fonctionnent de la même manière. Les triples guillemets sont utilisés pour de gros blocs de texte.

Puisqu'il s'agit d'une liste de caractères, on peut accéder à un caractère en passant un *index* :

```
1 text = "Hello World !"
2 print(text[6])
3 # W
```

 Python

Tout ce qu'on a vu dans l'utilisation des tableaux (index, slicing,...) est donc valable pour les strings.


14. Itérateurs et Générateurs

14.1. Itérateur

Un *itérateur* est un outil permettant de parcourir une séquence d'éléments. Les **collections** en Python, tels que les listes, les ensembles, les tuples, les dictionnaires et les chaînes de caractères sont des **objets itérables**. Nous pouvons les utiliser dans une boucle `for`. Il est tout à fait possible d'écrire un itérateur, mais cela implique d'écrire du code orienté objet, ce qui dépasse le cadre de ce cours. L'avantage d'un **itérateur** est qu'il consomme très peu de ressources mémoire et processeur.

Pour créer un itérateur, nous pouvons utiliser `iter()` sur un élément itérable. Pour parcourir cet itérateur, nous utiliserons la fonction `next()`.

```
1 text = "Hello Python"
2 it = iter(text) # making iterator
3 print(next(it))
4 # H
```

 Python

La fonction `next()` va renvoyer le **prochain élément** de l'itération, qui sera consommé. Lors d'un prochain appel à `next(it)`, ce sera l'élément suivant dans l'itérateur qui sera consommé, et ce jusqu'à la fin.

14.2. Générateur et fonction génératrice

Un *générateur* est une catégorie particulière d'itérateurs. Un générateur va créer à la demande non pas une **séquence d'éléments**, mais **un seul élément à la fois !**. Il existe plusieurs types de générateurs, mais nous allons nous focaliser sur la *fonction génératrice*, la plus simple à aborder mais tout aussi efficace.

14.2.1. Cas d'utilisation

Comme le fonctionnement des Python Generators est basé sur le principe d'une **lazy evaluation** (*évaluation paresseuse*) et qu'ils n'évaluent les valeurs qu'en cas d'absolue nécessité, les fonctions génératrices se prêtent particulièrement bien au travail avec d'importants volumes de données.

Une fonction normale commencerait par charger l'ensemble du contenu de votre fichier dans une variable en mémoire. Si vous travaillez avec d'importants volumes de données, votre mémoire locale pourrait ne pas suffire et le processus se solderait donc par le message `MemoryError`. Les fonctions génératrices vous permettent d'éviter facilement les problèmes de ce type, car elles lisent votre fichier ligne par ligne. Le mot-clé `yield` renvoie la valeur nécessaire au moment où vous en avez besoin, puis interrompt l'exécution de la fonction jusqu'à son prochain appel pour lire une autre ligne du fichier.


14.2.2. Fonction génératrice

Une *fonction génératrice* est une simple fonction en Python, mais avec une toute petite différence. Au lieu d'utiliser le mot clé `return` pour renvoyer le résultat de la fonction, une fonction génératrice utilisera le mot clé `yield`.

Pour comprendre la différence, revenons sur `return`. Ce mot clé est utilisé pour transmettre les valeurs calculées par la fonction au reste du programme. Dès que `return` est atteint, la fonction est interrompue et toutes ses variables constitutives sont détruites. En rappelant la fonction, elle recommence depuis le début.

Dans le cas de `yield`, le fonctionnement est légèrement différent : `yield` va transmettre le résultat, mais la fonction est **suspendue**, comme mise en attente. En rappelant la fonction génératrice, elle **reprend son fonctionnement après le `yield`**.

```
1 ''' Infinite generator
2 __author__ = "VCO"
3 '''
4 def infinite_sequence():
5     num = 0
6     while True:
7         yield num
8         num +=1
```

 Python

Expliquons rapidement le fonctionnement. Nous créons notre fonction normalement, à ceci près que nous utilisons `yield`. La variable `num` est initialisée et une boucle infinie est démarrée. Ensuite, nous **capturons** la valeur de `num` (0) et la fonction *se met en attente du prochain appel*. Lors du prochain appel, la fonction reprend **après le `yield`** en incrémentant `num` et la boucle recommence jusqu'au `yield`.


Toutefois, un générateur **reste une fonction**, et une fonction se termine avec le mot clé `return`. On utilisera `return` pour indiquer la fin de la génération. On peut néanmoins l'ignorer si le générateur fonctionne sur un ensemble fini.

14.3. Liste ou générateur ?

Quand doit-on choisir une liste ou un générateur, puisque dans les deux cas on retrouve tous les éléments d'une collection ? Voici quelques éléments de réponse.

- Les listes prennent plus de place en mémoire qu'un générateur, car **tous les éléments de la liste sont créés en même temps**. Dans le cas d'un générateur, l'élément n'existe que lorsqu'il est *généré* et n'existe plus lors de la génération suivante.
- Les générateurs peuvent être **infinis**, contrairement à une liste qui est naturellement *finie*.

```
1 def infinite():
2     n = 0
3     while True:
4         yield n
5         n += 1
```

 Python

- Les générateurs **ne sont pas indexables**, contrairement aux listes (du fait que le générateur ne génère pas tous les éléments en une fois).
- Les générateurs **ont une empreinte mémoire plus faible**.

L'intérêt d'utiliser un générateur est donc dans le traitement systématique d'éléments qui ne doivent pas être conservés en mémoire, comme la lecture d'un fichier texte et son traitement ligne par ligne, sans devoir charger au préalable **toutes les lignes du fichier**.


Note

Une fonction génératrice permet d'écrire des *listes en compréhension*.

14.4. Les compréhensions de listes


Une *compréhension de liste* est un moyen de créer une liste à partir d'une itération. C'est une façon Pythonique compacte qui peut sembler compliqué à comprendre au premier abord, mais en les décomposant, ce n'est pas si compliqué que ça. Il s'agit de créer une boucle **for** sur une collection dans une syntaxe plus concise. Par exemple, si nous souhaitons créer un tableau contenant les carrés des 10 premiers chiffres, nous pourrions écrire :

```
1 squares = [] # dynamic list
2 for x in range(10):
3     squares.append(x**2)
4 # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

 Python


Pour chaque élément **x** allant de [0 à 10[, nous ajoutons au tableau *square* l'opération **x**2**. Avec la compréhension de liste, nous pouvons affecter le même algorithme selon une écriture plus concise :

```
1 squares = [x**2 for x in range(10)]
2 # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

 Python


L'intérêt des compréhensions de liste est aussi de pouvoir construire d'autres listes à partir d'une liste existante :

```
1 words = ["Hello", "World", "!"]
2 counts = [len(word) for word in words]
3 #[4, 5, 1]
```

 Python


Les compréhensions de listes peuvent également **filtrer** des éléments en ajoutant une structure **if** dans l'itération :

```
1 even_squares = [x**2 for x in range(10) if x%2 == 0]
2 #[0, 4, 16, 36, 64]
```

 Python


En version *longue*, nous aurions pu écrire :

```
1 even_squares = []
2 for x in range(10):
3     if x%2 == 0:
4         even_squares.append(x**2)
```

 Python

De même, il est également possible de créer une compréhension de liste sur des tableaux multi dimensionnés. Il suffit d'écrire une compréhension pour chaque dimension.

```
1 words = ["Hello", "World"]
2 letters = [char for word in words for char in word]
3 # ["H", "e", "l", "l", "o", "W", "o", "r", "l", "d"]
```

 Python

15. Les fonctions intégrées

Python d'une série de fonctions et de types *intégrés* (*builtins*) qui sont tout le temps disponible. Nous ne reprendrons ici que les plus utiles.


all(iterable) Cette fonction retourne `True` si **tous** les éléments de l'itérable sont `True` ou si l'itérable est vide. Très pratique pour tester une liste en compréhension !

```
1 all(i%2 == 0 for i in [3,4,6])
2 # False
3
4 all(i%2 == 0 for i in [2,4,6])
5 # True
```

 Python


any(iterable) Cette fonction retourne `True` si **un** des éléments de l'itérable est `True`. Si l'itérable est vide, `any()` renverra `False`.

```
1 any(i%2 for i in [3,4,6])
2 # True
```

 Python


divmod(a,b) Cette fonction renvoie un tuple de deux valeurs contenant le quotient **et** le reste d'une division entière. Cela équivaut à faire `a // b` ; `a % b` en même temps.

```
1 divmod(4,2)
2 # (2,0)
```

 Python


enumerate(iterable, start=0) Cette fonction retourne un tuple contenant un compteur (démarrant à *start*) associé à chaque valeur de l'itérable.

```
1 lst = ["Hello", "World", "!"]
2 list(enumerate(lst, start=1))
3 # [(1, "Hello"), (2, "World"), (3, "!")]
```

 Python

filter(function, iterable) Cette fonction construit un **itérateur** depuis les éléments de l'itérable pour lesquels la *function* est `True`.


```
1 def is_even(element:int) -> bool
2     return element%2 == 0
3
4 list(filter(is_even, [1,3,4,6,8,9]))
5 # [4,6,8]
```

 Python

id(object) Cette fonction retourne l'**identifiant** d'un objet. Cet identifiant est garanti comme étant unique et constant pour chaque objet durant sa durée de vie. Cet identifiant correspond à l'adresse mémoire de l'objet.


input(prompt str): Cette fonction écrit sur la sortie standard le *prompt* éventuel et attend l'entrée d'une ligne sur l'entrée standard. Cette ligne est convertie en

```
1 str
```

 Python


len(sequence) Cette fonction renvoie la **longueur** de la *séquence* passée en paramètre.


```
1 len("Hello World")
2 # 11
```

 Python


list(iterable), dict(iterable), tuple(iterable), set(iterable) Ces fonctions construisent une liste, dictionnaire, tuple ou set en fonction d'un *iterable*, ou une séquence vide si l'*iterable* n'est pas fourni.

```
1 lst = ["Hello", "World", "!"]
2 list(enumerate(lst, start=1))
3 # [(1, "Hello"), (2, "World"), (3, "!")]
```


 Python

map(function, iterable, *iterables) Cette fonction renvoie un **itérateur** qui applique la *function* sur **chaque élément de la collection**. Si d'autres *iterables* sont passés, la *function* doit avoir autant de paramètres que d'*iterables*.

```
1 def double_is_even(num: int) -> int:
2     return num*2 if num % 2 == 0 else num
3
4 numbers = [1,2,3,4,5]
5 result = list(map(double_is_even, numbers))
6 # [1,4,3,8,5]
```

 Python

```
1 def sum_numbers(x: int, y: int) -> int:
2     return x + y
3
4 numbers_1 = [1,2,3,4,5]
5 numbers_2 = [10,20,30,40,50]
6
7 result = list(map(sum_numbers, numbers_1, numbers_2))
```

 Python

min(iterable) et max(iterable) Ces fonctions renvoient respectivement la **plus petite valeur** et la **plus grande valeur** de l'*iterable* fourni.

next(iterable) Cette fonction renvoie l'**élément suivant** d'un itérable.

pow(base,exp,mod=None) Cette fonction renvoie la $base^{exp}$ si un *mod* est fourni, cela calcule le modulo (similaire à $base^{**}exp \% mod$) de façon plus rapide et précise.

print(*objects, sep=< < , end= >\n >, file=None, flush=False) Cette fonction permet d'imprimer dans un terminal les *objects* séparés par des *sep* et suivi du terminateur de ligne *end*. Pour redéfinir les autres arguments, il faut les nommer. Tous les arguments sont convertis en *str*.

reversed(sequence) Cette fonction renvoie la séquence passée en paramètre **renversée**.

round(number, ndigits=None) Cette fonction renvoie le *number* arrondi avec une précision de *ndigits* chiffres après la virgule. Si *ndigits* est omis, la fonction renvoie l'entier le plus proche.

sorted(iterable, key=None, reverse=False) Cette fonction renvoie une liste triée depuis un *iterable*. Elle possède deux arguments nommés, *key* et *reverse*. *Key* spécifie une fonction à un argument utilisé pour extraire une clé de comparaison de chaque élément de l'*iterable*. *Reverse* permet d'inverser l'ordre de tri.

sum(iterable,start=0) Cette fonction additionne les éléments d'un itérable à partir de la valeur *start*.

zip(*iterables, strict=False) Cette fonction itère sur plusieurs *iterables* en parallèle, en regroupant chaque élément de chaque *iterable* dans un tuple. Voyons cela comme une **tirette**. Les itérables sont imbriqués les uns dans les autres. La fonction s'arrête à l'itérable le plus petit.

16. Les modules

Nous avons vu comment on peut réutiliser du code dans notre programme en définissant des fonctions. Mais dans de nombreux cas, vous aurez conçu des fonctions tellement puissantes que vous voudriez les réutiliser dans d'autres applications. Pour ce faire, nous allons créer des **modules**.

L'autre avantage de l'utilisation des modules est de pouvoir structurer votre programme en différents éléments constitutifs. Par exemple, un module **views** qui ne traitera que de la partie graphique de votre programme.

Un module est un fichier Python qui contient **uniquement des fonctions et des constantes**. Il peut être importé par un autre programme pour utiliser ses fonctionnalités. De la même manière, Python possède déjà une série de modules que l'on peut importer et utiliser.

16.1. Les modules standard de Python

Mous n'allons pas lister ici tous les modules de Python. Il y en a trop et on peut en installer des milliers d'autres. Nous allons seulement voir comment les importer. L'importation d'un module se fait de plusieurs manières : soit on importe **tout** le module, soit on importe uniquement ce dont on a besoin. Les importations se situent généralement en début de script. Voyons d'abord comment importer un module.

16.1.1. La directive `import`

```
1  """
2  __author__ = "VCO"
3  """
4  import random as rdm
5
6  def die(max_die: int) -> int:
7      ''' roll a die '''
8      return rdm.randrange(1,max_die+1)
9
10 die(6)
```

Analysons ce bout de code. En première ligne, nous importons le module **random**, que nous renommons **rdm** via le mot-clé `as` (optionnel, mais pour éviter d'écrire trop de texte...). Ce module possède une série de fonctions pour générer des nombres aléatoires. Python sait où chercher ce module, car c'est un module système.

À la ligne 5, nous utilisons le module **rdm** pour avoir accès à la fonction `randrange()` qui renvoie un nombre entier situé entre deux valeurs. Nous aurions pu utiliser `randint()` également.

Cependant, importer un module est relativement coûteux en ressources, car Python doit interpréter **tout** le module pour le charger en mémoire. Mais Python va ruser un peu pour être plus rapide. Il va créer des fichiers **compilés** (les `.pyc`) !


16.1.2. La directive `from ... import`

Si nous n'avons pas besoin d'importer tout un module, nous pouvons en importer qu'une partie via la directive `from import`. Python va **directement** importer la fonction dans notre programme, comme si nous l'avions écrite nous-même.

```

1 __author__ = "VCO"
2 from random import randrange
3
4 def die(max_die: int ) -> int:
5     ''' roll a die '''
6     return randrange(1,max_die+1)
7
8 die(6)

```

 Python

16.2. Nommage des modules

Chaque module porte un nom et les instructions dans un module peuvent retrouver le nom du module. Cela peut être pratique pour déterminer si le module est autonome ou s'il est importé. Comme dit précédemment, quand un module est importé pour la première fois, le code du module est exécuté. Nous pouvons utiliser ce concept pour modifier le comportement du module selon que le programme s'exécute seul ou s'il est importé à partir d'un autre module. Ce résultat est obtenu avec l'attribut `__name__` du module (notez les doubles underscores de part et d'autre).

16.3. Créer un module

Créer un module est excessivement facile, puisque sans vous en rendre compte, vous l'avez déjà fait. Chaque programme Python est aussi un module ! À partir du moment où vous avez donné un nom au fichier `.py`, ce qui est normalement le cas en enregistrant votre travail, vous avez créé un module du même nom. Par convention, quand on écrit un module, on ajoutera deux variables, `__author__` et `__version__` pour indiquer l'auteur et la version du module.

L'appel de votre module est identique à celui d'un module standard. Attention cependant, vos modules doivent être enregistrés dans le même répertoire que votre application. Vous pouvez créer des sous-dossiers (c'est même recommandé) et dans ce cas, l'appel de votre module doit être adapté. Le sous-dossier qui contient vos modules porte lui aussi un nom particulier, le **package**.

Vous commencez à voir comment on organise un programme. Les variables sont intégrées dans des fonctions. Ces fonctions sont intégrées dans des modules. Ces modules sont dans des packages. Un *package* est un dossier contenant les différents modules ainsi qu'un fichier spécial nommé `__init__.py` qui est un fichier vide. Il est juste là pour indiquer à Python que le dossier est considéré comme un package (pas que..., mais nous ne le verrons pas ici. **TGCM !**).

17. La gestion des fichiers

Un fichier est un ensemble d'informations généralement structurées d'une certaine manière, stocké de manière pérenne sur un support moins volatile que la mémoire de l'ordinateur.

Un fichier se distingue des autres par quelques attributs dont son nom et sa catégorie. Ils se distinguent aussi entre eux par l'organisation de leurs données, ce qui définit leur format. Il existe des fichiers texte, mp3, jpg, wav...

En général, deux catégories de fichiers sont distinguables :

- Les fichiers texte, organisés sous forme de lignes successives.
- Les fichiers binaire, contenant des données brutes.

Dès lors, quel format utiliser ? Tout dépend en fait de ce que vous souhaitez faire. Les fichiers texte sont tout aussi efficaces que les fichiers binaires.

17.1. Les fichiers texte

- Les fichiers texte sont utilisés pour stocker des données structurées.
- Ces données structurées peuvent être du texte ou des nombres, du moment que le contenu soit converti en texte lisible.
- Les enregistrements sont représentés sous forme de lignes.
- Chaque ligne représente une structure d'enregistrement, selon un format fixe ou délimité (comme les .csv, par exemple).
- Un fichier texte doit rester lisible et modifiable par n'importe quel éditeur de texte.
- Un fichier texte doit être lu ligne par ligne.
- Un enregistrement ne s'ajoute qu'en fin de fichier. Pour modifier ou insérer un enregistrement, il faudra parfois réécrire d'entièreté du fichier.

17.2. Les fichiers binaire

- Les fichiers binaires peuvent stocker n'importe quoi, structurés ou non.
- Toutes les données sont représentées sous forme binaire. Les données sont écrites comme si elles sortaient brutes de la mémoire. Il se peut que l'on reconnaisse des chiffres ou du texte, mais c'est tout à fait fortuit.
- La structure des enregistrements est dépendante de l'interprétation du programme. Les enregistrements peuvent être de longueur fixe, mais collés les uns aux autres sans passage à la ligne.
- Un fichier binaire ne doit pas être ouvert ou modifié par un programme autre que celui qui l'a créé pour ne pas corrompre les données.
- Le fichier peut être lu octet par octet, ou bloc par bloc, ou entièrement, depuis n'importe quelle position, puisque c'est vous qui définissez sa structure. Idem pour les enregistrements.

17.3. Les accès au fichier

17.3.1. L'accès séquentiel

Un fichier **séquentiel** permet d'accéder aux données dans leur ordre d'écriture. Pour accéder par exemple au 1000^e enregistrement, il faudra d'abord lire les 999 premiers. Les fichiers texte sont des fichiers à accès séquentiel. Un fichier binaire peut aussi être lu en séquentiel.

17.3.2. L'accès direct

Un fichier en **accès direct** (aussi appelé **accès aléatoire**) est un fichier sur lequel on peut sauter directement à l'endroit désiré. Pour un fichier texte, cela peut être via un n° de ligne. Pour un fichier binaire, c'est la position de l'octet souhaité.

17.3.3. L'accès indexé

Dans un fichier **indexé**, les enregistrements sont identifiés par un *index*. La connaissance de cet index permet d'accéder directement à l'enregistrement correspondant à cet index. Les enregistrements sont souvent placés les uns à la suite des autres dans le fichier, comme en séquentiel. Les index sont, eux, placés dans un tableau d'index, avec pour chacun d'entre eux la position de l'enregistrement correspondant dans le fichier. Un fichier indexé est le mélange entre un accès séquentiel et un accès direct.

Les index peuvent être totalement indépendants, mais bien souvent, vous entendrez parler de **fichier séquentiel indexé** : le fichier est indexé, mais depuis un index, il est possible de lire successivement tous les enregistrements suivants. Les index peuvent être *chaînés*. Nous ne verrons pas les chaînages dans ce syllabus.

17.4. Les enregistrements

Si les fichiers sont structurés en enregistrements, il existe néanmoins deux types de structures : les structures **délimitées**, ou à **largeur de champs fixe**.

Une structure délimitée contient des champs séparés par un caractère particulier donné *délimiteur*. Ce sont souvent le point-virgule (;), la virgule (,) ou le double-point (:) qui sont souvent utilisés.

Prenons l'exemple d'un fichier **csv**. La manipulation d'un tel fichier semble assez évidente : il suffit de lire une ligne, puis de la découper champ par champ selon les délimiteurs. Cependant, le traitement de découpage nécessite un algorithme plus complexe qu'il n'y paraît et qui consomme pas mal de ressources.

Pour les enregistrements à **largeur fixe**, il n'y a pas de délimiteurs. Chaque champ a une longueur prédéfinie et occupe toute cette longueur, quitte à être complétée par du *padding*. Les champs sont collés les uns aux autres, en un seul bloc.

Cependant, les formats à largeur fixe consomment plus de mémoire (à cause du *padding*), contrairement aux formats délimités. Mais la récupération des données est plus simple, car on connaît à l'avance la taille de chaque champ et donc les positions de découpe des enregistrements. Il faudra juste penser à supprimer le *padding*, avec une fonction `trim`, par exemple.

17.5. Accéder à un fichier

Pour travailler avec un fichier, il faut impérativement suivre un certain ordre :

Ouvrir le fichier, c-à-d. indiquer sur quel fichier on opère.

Traiter le fichier, le lire ou y écrire, selon le but recherché.

Traiter le fichier, quand le traitement est effectué.

Il existe trois modes d'ouverture d'un fichier, en fonction de nos besoins :

L'ouverture en lecture qui nous donne un accès en **lecture seule** au fichier. Nous n'avons pas le droit d'y écrire.


L'ouverture en écriture qui nous donne un accès en **lecture et en écriture**. nous pouvons y écrire où l'on veut, au risque de faire des bêtises.

L'ouverture en ajout qui nous donne un accès en **écriture seule**. Nous ne pouvons pas lire le fichier, seulement y écrire **en fin de fichier**. Les modifications sont interdites.

17.6. Traiter un fichier en Python

Python étant à l'origine conçu pour réaliser des opérations systèmes, il est naturellement pourvu d'outils très simples pour gérer les fichiers. L'ouverture d'un fichier se fait ainsi :

```
1 with open("myfile.txt","rt") as file:
2     # some instructions
3     file.close() # optional due to context
```

 Python

La variable `file` est une structure qui correspond à une entrée/sortie du descripteur de fichiers. Le fait d'utiliser le mot-clé `with` permet à Python de gérer correctement le fichier et de le fermer de lui-même à la fin du traitement.


Notez aussi dans la commande `open` le deuxième argument. C'est le mode d'ouverture du fichier. Il existe deux groupes de trois modes pour indiquer à Python comment traiter le fichier.

Symbole	Signification	Symbole	Signification
r	lecture seule (par défaut)	t	mode texte (par défaut)
w	écriture	b	mode binaire
a	ajout	+	mise à jour

17.6.1. Lire un fichier texte


Il y a plusieurs méthodes pour lire un fichier selon la manière dont on souhaite le traiter. On peut lire l'entièreté du fichier dans une chaîne de caractères avant de la traiter :

```
1 with open("myfile.txt","rt") as file:
2     content = file.read() # reading whole file, outputs str...
```

 Python

Ou traiter le fichier ligne par ligne (méthode plus couramment utilisée, car consomme moins de ressources) :


```
1 with open("myfile.txt","rt") as file:
2     for line in file:
3         # instructions
```

 Python

17.6.2. Écrire dans un fichier texte

Pour écrire dans un fichier, il suffit d'utiliser la méthode `write` du fichier. Veillez à ouvrir le fichier en écriture ou en ajout. Vous pouvez également écrire plusieurs lignes d'une liste avec `writelines`.

```
1 with open("myfile.txt","wt+") as file:
2     file.write(any_content)
3     file.writelines(any_list)
```

 Python

17.7. Pathlib

`pathlib` est une bibliothèque Python permettant de gérer les chemins des fichiers. Elle a l'avantage de mieux gérer les fichiers indépendamment du système d'exploitation et permet d'ouvrir les fichiers plus simplement et rapidement.

Pour l'utiliser, il suffit d'ajouter en entête du fichier `from pathlib import Path`.

Utilisez l'aide en ligne de Python pour en savoir plus.

17.8. Pickle

`pickle` est une bibliothèque Python permettant de **sérialiser** ou **désérialiser** des objets en Python.

La **sérialisation** est une méthode qui permet de convertir un objet en mémoire en une suite d'octets inscriptible dans un fichier. La **désérialisation** est le phénomène inverse.

Comme pour `pathlib`, il suffit d'ajouter la bibliothèque `pickle` en entête du fichier. Consultez également l'aide de Python pour en apprendre plus.

18. Gestion des exceptions

En programmation, on rencontre deux types d'erreurs fréquentes :

- Les erreurs de syntaxe, dues au développeur ;
- Les erreurs d'environnement, dues à des causes extérieures.


L'utilisation d'un IDE nous permet de nous affranchir des premières. En revanche, pour les secondes, il est nécessaire de mettre en place un système de gestion des exceptions qui indiquera à Python quoi faire en cas d'erreur. Par défaut, Python affichera l'erreur en rouge dans la console et arrêtera le programme. Mais on peut lui demander de faire autre chose !

Les erreurs détectées durant l'exécution d'un programme sont appelées en Python des **exceptions**. Chaque exception va avoir un nom spécifique résultant d'une action erronée précise : par exemple, la `ZeroDivisionError` sera levée si on tente de diviser un nombre par 0.

18.1. Lever une exception


Lever une exception signifie indiquer au programme de générer une erreur en fonction d'un cas particulier. Plutôt que de laisser du code s'exécuter de façon incontrôlée pouvant générer des exceptions par la suite, il sera préférable de prendre les devants et de lever nous-même une exception plus cohérente. Prenons par exemple ce code :

```
1 def average(*args:float) -> float:
2     return sum(args)/len(args)
```

 Python


Cette fonction est facile à comprendre, mais si on l'exécute de façon erronée, elle nous répondra :

```
1 average( )
2 >>> ZeroDivisionError: division by zero
```

 Python

C'est logique, vu qu'il n'y a pas d'arguments passés en paramètres. Cependant, le problème **n'est pas une division par zéro, mais un manque d'arguments !**. Nous allons donc lever une exception plus logique. Nous utiliserons pour ça un nouveau mot clé, `raise`.

```
1 def average(*args:float) -> float:
2     if len(args) == 0:
3         raise TypeError('average expected at least 1 argument, got 0')
4     else:
5         return sum(args)/len(args)
```

 Python

18.2. Pourquoi lever une exception ?

Souvent, les débutants en programmation demandent *pourquoi il faut lever des exceptions ?*, car Python nous indiquera l'erreur de toute façon. Cela est vrai pour les erreurs de **logique**.

Si nous concevons une application se connectant à un serveur distant, par exemple, et que le serveur est éteint, notre application devant se connecter n'y arrivera pas. Donc, pour elle, il s'agira d'une exception et s'arrêtera sur un code d'erreur non contrôlé. Cette partie n'est pas liée à une erreur dans notre code, mais d'un élément extérieur à celui-ci. Lever

une erreur (ou la capturer, nous verrons cela après) permet de **partager la responsabilité** et d'agir autrement.

18.3. Capturer une exception

Capter une exception est l'inverse de la *lever*. On va traiter un **comportement alternatif** du code si un type d'erreur est levé.


Par défaut, quand une erreur est levée, Python affiche une *trace* avec la pile d'appels conduisant à cette erreur.

Lorsqu'on utilise une fonction avec un **code critique** susceptible de générer une erreur, on peut volontairement *ne rien faire*. L'exception va alors se propager jusqu'à ce que *quelque chose* la capture. Souvent ce sera Python qui affichera la trace en rouge (Python *capturera* L'exception).

Le fait de laisser une exception se propager n'est pas forcément une erreur de programmation. Il n'est pas indispensable de capturer une exception si c'est pour en lever une seconde par la suite, autant la laisser se propager.

Pour capturer une exception, nous allons utiliser un bloc `try ... except`. Le code critique sera à l'intérieur de ce bloc.

```
1 def average2(*args:float) -> float:
2     try
3         return sum(args)/len(args)
4     except ZeroDivisionError:
5         return 0
```

 Python


Si aucune erreur n'est détectée par Python pendant l'exécution de ce code, ce qui se situe dans la clause `except` sera ignoré. En revanche, si une exception est levée, la clause `try` sera ignorée et le code dans l'`except` sera exécuté.

La différence avec la fonction `average()` précédente est qu'ici on avertit pas l'utilisateur d'un problème, on effectue une solution alternative.

18.3.1. La clause `else`

Le bloc `try ... except` peut s'agrémenter d'une clause `else`, qui s'exécutera **après** le `try` si **aucune exception n'a été levée**. Il est considéré comme *bonne pratique* de placer le code **critique** dans la clause `try` et le code **non critique** dans la clause `else`.

```
1 """
2 My division
3 __author__ = "VCO"
4 """
5 try:
6     x = int(input("Enter numerator: "))
7     y = int(input("Enter denominator: "))
8     rslt = x/y
9
10 except ValueError: # case if inputs are not numbers
11     print("Value entered are not a number!")
12 except ZeroDivisionError: # case if y == 0
```

 Python


```
13 print("Denominator's value must be different than 0")
14 else:
15 print("Result: ",rslt)
```

18.3.2. La clause **finally**

Le bloc `try ... except` peut aussi s'agrémenter de la clause `finally` en plus du `else`.

La différence avec la clause `else` est que le code sera exécuté **dans tous les cas**, qu'il y ait eu un problème ou non. Cette clause sera très utile pour clôturer certaines opérations, quel que soit l'état du programme.

```
1 """
2 My division
3 __author__ = "VCO"
4 try:
5     x = int(input("Enter numerator: "))
6     y = int(input("Enter denominator: "))
7     rslt = x/y
8
9 except ValueError: # case if inputs are not numbers
10    print("Value entered are not a number!")
11 except ZeroDivisionError: # case if y == 0
12    print("Denominator's value must be different than 0")
13 else:
14    print("Result: ",rslt)
15 finally:
16    print("My program is now terminated.")
17 """
```

 Python

Note

On peut utiliser les différentes clauses indépendamment les unes des autres.

19. Les Tests unitaires avec Pytest

19.1. Introduction

19.1.1. Qu'est-ce qu'un *test unitaire* ?

Selon Wikipedia¹, un **test unitaire** est « une procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel ou d'une portion de programme. »

Ce type de test est notamment utilisé dans la méthodologie *Test Driven Development* (ou *Développement Dirigé par des Tests*).

Pour créer des tests unitaires, nous avons besoin d'un *framework* de tests qui nous permet d'écrire des tests avec facilité. Il en existe plusieurs pour Python, dont la référence en la matière, **unittest**. Toutefois, nous allons voir ici le *framework* **Pytest**.

19.2. Pytest

Pytest est un *framework* de tests basé sur Python. Le principal avantage de Pytest est qu'il permet d'écrire très simplement des tests unitaires, en comparaison avec le framework de référence, **unittest** ou son successeur, **nose**.

Pytest permet d'écrire simplement de petits tests et de s'adapter aux besoins du programmeur. Nous pouvons écrire rapidement des tests les plus simples jusqu'à écrire des tests extrêmement élaborés en employant les fonctionnalités avancées du framework. Il dispose également d'outils préétablis nous aidant dans nos écritures de tests.

Enfin, Pytest peut également lancer les tests écrits avec les frameworks **unittest** et **nose**.

19.2.1. Installer Pytest

Pour installer Pytest, il suffit d'une simple ligne de commande: `pip install -U pytest`, ou (mieux) via **pdm**: `pdm install -dG tests pytest`. Cela installera le framework de test dans Python.


19.2.2. Conventions de nommage

Pytest est capable de découvrir automatiquement les nouveaux tests écrits, sans devoir reprogrammer Pytest à chaque utilisation. Il suffit pour cela de nommer les fichiers Python ainsi que les définitions de fonctions en leur ajoutant **test** comme affixe.

19.2.3. Écriture d'un test

L'écriture d'un test est une simple définition de fonction qui utilise le mot clé `assert` pour vérifier l'état d'un test. Écrivons un simple test pour s'en assurer :

```
1 def upper_text(s:str)->str:
2     return s.upper()
3
4 def test_upper_text():
5     assert upper_text("hello") == "HELLO"
```

 Python

Exécutez le test via l'invite de commande ou le terminal en tapant simplement `pytest` dans votre répertoire projet ou via une configuration de lancement dans PyCharm.

¹https://fr.wikipedia.org/wiki/Test_unitaire

19.3. Création d'un projet en TDD

Le mieux pour comprendre l'utilisation des tests, c'est de faire un mini-projet en TDD : écrivons un jeu de **morpion** !

Note

Le **TDD** (*Test Driven Development*) est une manière d'écrire du code, en **commençant d'abord par écrire les tests**, puis en tentant de résoudre **au plus simple** les tests ainsi écrits (*KISS*).

L'écriture d'un test se fait via une méthode facile à comprendre : le **GWT** (*Given, When, Then*).

Par facilité, nous allons écrire notre programme en plusieurs fichiers :

morpy.py contenant le *moteur* du jeu ;

tui.py contenant l'interface graphique (en console) ;

app.py contenant le script de lancement du projet ;

conftest.py contenant la configuration des différents tests ;

morpy_test.py contenant les tests du *moteur* du jeu ;

tui_test.py contenant les tests de l'interface graphique.

Avant de commencer, définissons les besoins :

- Le jeu du morpion se joue sur un tableau de 3×3 cases.
- Le joueur choisit de jouer soit avec les **X**, soit avec les **O**. L'ordinateur jouera avec l'autre symbole.
- Chaque joueur doit placer son symbole dans une case, dans le but de former une ligne. S'il forme une ligne verticale, horizontale ou diagonale, le joueur gagne.
- L'ordinateur jouera en premier en jouant dans une case vide. Plus tard on lui créera une mini intelligence artificielle.
- Chaque joueur joue à tour de rôle.
- Le jeu s'arrête en cas de victoire ou de match nul.

19.3.1. Notre premier test

Écrivons notre premier test. Pour cela, nous allons écrire notre test en **GWT**. En français : *Étant donné* rien du tout...

Quand la fonction `empty_board()` est utilisée...

Alors nous obtiendrons un plateau 3×3 cases vide.

```

1  import pytest
2
3  import morpy
4
5
6  def test_empty_board():
7      # Given nothing
8      # When we want an empty board
9      # Then we should get an empty board
10     assert morpy.empty_board() == [
11         ["", "", ""],
12         ["", "", ""],
13         ["", "", ""],
14     ]
15

```

Dans morpy_test.py

Expliquons un peu le **GWT** :

- *Given (Étant donné)* définit un **état de départ** ;
- *When (Quand)* définit une **série d'actions à effectuer** ;
- *Then (Alors)* définit le **résultat à obtenir**.

Lançons notre test, qui doit en toute logique être un échec, puisque le code n'est pas encore implémenté. Nous allons répondre à ce test en écrivant le code **le plus simple possible** qui fait en sorte que le test réussisse.

```

1  """
2  Morpy - Game engine
3  __author__ = "VCO"
4  """
5
6  import random
7
8
9  def empty_board() -> list[list[str]]:
10     return [["" for _ in range(3)] for _ in range(3)]

```

Dans morpy.py

Ici, le code est vraiment trivial. Mais si nous avions la possibilité de faire un code plus (trop ?) bête, nous aurions pu écrire d'autres tests venant compléter le(s) test(s) existant(s).

Note


Dans l'idéal, on ne modifie pas un test et on n'en supprime pas. On ajoute des tests de plus en plus poussés pour éviter chaque problème potentiel.

19.3.2. Les fixtures

Écrivons un second test :

Étant donné un tableau vide,
Quand on veut les différentes cases disponibles,
Alors nous obtiendrons une liste allant de 1 à 9.


```
17 def test_find_empty_cells():
18     # Given an empty board
19     empty_board = morpy.empty_board()
20     # When we want to find empty cells
21     # Then we should get a list of empty cells
22     assert morpy.find_empty_cells(empty_board) == [1,2,3,4,5,6,7,8,9]
```

 Python

Dans morpy_test.py

Nous pourrions écrire comme résolution du test :


```
13 def find_empty_cells(board: list[list[str]]) -> list[int]:
14     return list(range(1,10))
```

 Python

Dans morpy.py

En lançant les tests, nous voyons qu'il est résolu ! En effet, le test demande de récupérer une liste allant de **1 à 9**, *peu importe comment on le fait...* Nous allons devoir ajouter un test pour préciser la demande.

```
23 def test_find_empty_cells_in_partial_board():
24     # Given a partial board
25     board = morpy.empty_board()
26     board[0][0] = "X"
27     board[1][1] = "0"
28     # When we want to find empty cells
29     # Then we should get a list of empty cells
30     assert morpy.find_empty_cells(board) == [2,3,4,6,7,8,9]
```

 Python

Dans morpy_test.py

En lançant les tests, nous pouvons remarquer que la fonction `find_empty_cells(board)` ne fonctionne pas correctement. Nous allons la modifier.

19.3.2.1. Écrire les fixtures


Avant de corriger le code pour coller aux tests, nous allons implémenter un autre élément : **la fixture**. Une *fixture* est une fonction particulière qui a pour but d'initialiser une variable ou un objet. Elle permet de créer des configurations de départ (*Given*) réutilisables pour différents tests, **sans modifier le code**. L'autre avantage d'une *fixture* est qu'après avoir été utilisée, elle remet le code **tel qu'il était avant le test**. Nous allons écrire deux configurations de départ **indépendante du code d'origine**. En effet, on est pas forcément sûr que la fonction `empty_board()` fonctionne correctement...

Ce fichier de configuration doit avoir comme nom **conftest.py**. Pytest s'arrangera pour que ce fichier de configuration soit chargé **pour tous les tests**, même s'ils sont répartis dans plusieurs fichiers.

```

1  """
2  Configuration for pytest
3  __author__ = "VCO"
4  """
5
6  import pytest
7
8
9  # Given an empty board
10 @pytest.fixture
11 def empty_board():
12     yield [["_ for _ in range(3)] for _ in range(3)]
13
14
15 # Given a partial board
16 @pytest.fixture
17 def partial_board():
18     board = [["_X", "0", ""], [_, "", ""], [_, "X", ""]]
19     yield board
20
21
22 # Given a full board
23 @pytest.fixture
24 def full_board():
25     board = [["_X", "0", "X"], [_"0", "X", "0"], [_"0", "X", "0"]]
26     yield board
27

```

 Python

Dans conftest.py

Expliquons un peu ce code. Pour définir une *fixture*, il faut importer la bibliothèque `pytest` afin d'utiliser le décorateur *fixture*. Je ne rentre pas dans les détails du décorateur, reprenez qu'en gros, un **décorateur** est un bout de code qui va modifier une fonction (**TGCM !**). Dans notre cas, nous *annotons* nos fonctions en écrivant `@pytest.fixture`. Notre *fixture* est définie.

La *fixture* va créer ou modifier l'état de différentes variables ou objets. Intervient ici un mot clé important : le `yield`. Le `yield` va agir comme un `return` à une exception près : tout le code situé **avant** le `yield` agira comme un **setup**, créant ou modifiant les variables avant de retourner l'état du setup. Tout le code situé **après** le `yield` agira comme un **teardown**, déchargeant les modifications effectuées.

Ça semble un peu complexe, donc résumons. **Avant** le `yield` nous écrivons le code qui va mettre en place les variables et **après** nous écrivons le code qui va supprimer les mises en place.

Pour passer une *fixture* à nos tests, il suffit d'indiquer le nom de la *fixture* en paramètre du test.


```

17 def test_find_empty_cells(empty_board):
18     # Given an empty board
19     # When we want to find empty cells
20     # Then we should get a list of empty cells
21     assert morpy.find_empty_cells(empty_board) == [1, 2, 3, 4, 5, 6, 7, 8, 9]
22
23
24 def test_find_empty_cells_in_partial_board(partial_board):
25     # Given a partial board
26     # When we want to find empty cells
27     # Then we should get a list of empty cells
28     assert morpy.find_empty_cells(partial_board) == [3, 4, 5, 6, 7, 9]
29
30
31 def test_find_no_cells(full_board):
32     # Given a full board
33     # When we want to find empty cells
34     # Then we should get an empty list
35     assert morpy.find_empty_cells(full_board) == []

```

Dans morpy_test.py

Et donc notre code :

```

13 def find_empty_cells(board: list[list[str]]) -> list[int]:
14     empties = []
15     for i, row in enumerate(board):
16         for j, col in enumerate(row):
17             if col == "":
18                 empties.append(i * 3 + j + 1)
19     return empties

```

Dans morpy.py

19.3.3. Le monkeypatching

Ajoutons une nouvelle fonctionnalité : créons une mini IA qui choisira aléatoirement une case parmi les cases vide. Cette case sera convertie en tuple (ligne, colonne). Le problème ici, c'est la notion d'**aléatoire**. Nous pouvons écrire un test afin de vérifier que la case sélectionnée soit bien entre **1** et **9**, ou que la ligne et la colonne soient entre **0** et **2**.

```

29 # Given empty cells
30 @pytest.fixture
31 def empty_cells():
32     yield [1, 2, 3, 4, 5, 6, 7, 8, 9]
33
34
35 # Given partial empty cells
36 @pytest.fixture
37 def partial_empty_cells():
38     yield [1, 4, 6, 7]


```

Dans conftest.py

```

38 def test_ai_choose_random(partial_empty_cells):
39     # Given an empty cells
40     # When ai want to choose a cell
41     # Then we should get a empty cell
42     r, c = morpy.ai_choose_random(partial_empty_cells)
43     assert r in range(3) and c in range(3)

```


 Python

Dans morpy_test.py

```

22 def ai_choose_random(cells: list[int]) -> tuple[int, int]:
23     cell = random.choice(cells)
24     r, c = divmod(cell - 1, 3)
25     return r, c

```

 Python


Dans morpy.py

Un problème se pose quand même : **comment être sûr que la case choisie aléatoirement soit la bonne ?** Nous allons donc « tricher » un peu : nous allons forcer le choix d'une case aléatoire en « imitant » la fonction `random`. Pytest offre une solution : le **monkeypatching**. Sous ce terme curieux, Pytest permet de modifier le comportement d'une fonction **sans en altérer le code original** (ici, la fonction `random`) Écrivons nos tests.

```

46 def test_ai_choose_not_random(monkeypatch, partial_empty_cells):
47     # Given an empty cells
48     # When ai want to choose cell 4
49     # Then we should get a empty cell
50     monkeypatch.setattr("random.choice", lambda cells: 4)
51     cell = morpy.ai_choose_random(partial_empty_cells)
52     assert (1, 0) == cell
53
54
55 def test_ai_choose_not_random2(monkeypatch, partial_empty_cells):
56     # Given an empty cells
57     # When ai want to choose cell 6
58     # Then we should get a empty cell
59     monkeypatch.setattr("random.choice", lambda cells: 6)
60     cell = morpy.ai_choose_random(partial_empty_cells)
61     assert (1, 2) == cell

```

 Python

Dans morpy_test.py

Expliquons ce code : Pytest possède une fonction nommée `monkeypatch` qui permet d'imiter (et de remplacer) le comportement de n'importe quelle fonction.

Nous indiquons via `monkeypatch.setattr()` que nous allons remplacer la fonction `random.choice()` par une fonction anonyme `lambda cells: 4` qui renverra la case **4**.

19.3.4. La paramétrisation

Nous devrions tester toutes les possibilités pour vérifier si toutes les cases sont bien trouvées. Pour cela, nous devrions écrire **9** fois le même test. Ici aussi, Pytest nous permet, via un autre décorateur, de passer différentes valeurs à tester.

```

64 @pytest.mark.parametrize(
65     "number, expected",
66     [
67         (1, (0, 0)),
68         (2, (0, 1)),
69         (3, (0, 2)),
70         (4, (1, 0)),
71         (5, (1, 1)),
72         (6, (1, 2)),
73         (7, (2, 0)),
74         (8, (2, 1)),
75         (9, (2, 2)),
76     ],
77 )
78 def test_ai_choose_not_random_multiples(monkeypatch, empty_cells, number, expected):
79     # Given an empty cells
80     # When ai want to choose a cell
81     # Then we should get a empty cell
82     monkeypatch.setattr("random.choice", lambda cells: number)
83     cell = morpy.ai_choose_random(empty_cells)
84     assert expected == cell

```

Dans morpy_test.py

Nous allons passer dans le décorateur plusieurs arguments. Le premier est le nom de **toutes les variables ou fixtures** que l'on va passer au test, le tout entre guillemets et séparés par des virgules si nécessaire. Le second est un tableau de tuples contenant à la fois le numéro de la cellule à tester, ainsi que le tuple résultant.

Je ne terminerai pas le code du projet ici, mais cela peut faire un excellent exercice ! Vous pouvez toutefois me le demander.