Avant toutes choses

Ce cours est une introduction à la programmation en Python et à la programmation orientée objet (POO), préparée par Gilles CHEHADE, et publié dans le domaine public sous license ISC.

Introduction

Python est un langage de programmation de haut niveau, interprété et polyvalent.

Créé par Guido van Rossum et lancé pour la première fois en 1991, il est conçu pour être simple à lire et à écrire, grâce à une syntaxe claire et épurée.

Voici quelques-unes de ses particularités :

- Lisible : Python privilégie une syntaxe claire et lisible, rendant le code plus compréhensible.
- Polyvalent : Il est utilisé dans de nombreux domaines, allant du développement web à la science des données, en passant par l'automatisation et le développement de jeux.
- Bibliothèques riches : Python possède une vaste bibliothèque standard, et il existe de nombreux packages tiers disponibles pour presque toutes les applications imaginables.
- Dynamiquement typé : Pas besoin de déclarer le type de variable à l'avance; le type est déterminé au moment de l'exécution.
- Interprété : Python est un langage interprété, ce qui signifie qu'il n'est pas nécessaire de le compiler avant de l'exécuter.
- Communauté active : La communauté Python est vaste et active, offrant une abondance de ressources, de tutoriels et de soutien.

Python est souvent recommandé comme premier langage de programmation pour les débutants en raison de sa simplicité… mais ne vous y trompez pas: sa puissance et sa flexibilité en font un excellent choix pour les projets professionnels de grande envergure.

Session 1

Installation de Python

Pour installer Python, suivez les étapes ci-dessous en fonction de votre système d'exploitation:

WINDOWS

Rendez-vous sur le site officiel de Python: python.org

Téléchargez l'installateur pour Windows.

Exécutez l'installateur et suivez les instructions à l'écran.

Assurez-vous de cocher la case "Add Python to PATH" lors de l'installation.

MACOS

Utilisez Homebrew: `brew install python3` Ou téléchargez l'installateur depuis python.org

LINUX / BSD

Python est généralement préinstallé sur la plupart des distributions Linux. Sinon, utilisez le gestionnaire de paquets de votre distribution pour l'installer.

Gestionnaire de paquets pour Python

`pip` est le gestionnaire de paquets standard pour Python.

Il permet aux développeurs d'installer et de gérer des bibliothèques et des dépendances supplémentaires qui ne sont pas incluses dans la bibliothèque standard de Python.

Avec `pip`, il est facile d'ajouter des fonctionnalités à vos projets en installant des paquets depuis le Python Package Index (PyPI).

Pour installer un paquet, il suffit d'utiliser la commande `pip install nom_du_paquet`.

De plus, pip offre d'autres fonctionnalités utiles comme la mise à jour des paquets (`pip install --upgrade nom_du_paquet`), la désinstallation (`pip uninstall nom_du_paquet`) et la liste des paquets installés (`pip list`).

Il est essentiel pour tout développeur Python de se familiariser avec pip afin de tirer pleinement parti de l'écosystème riche et varié des bibliothèques Python disponibles.

Mise en place d'un environnement virtuel

Un environnement virtuel est un espace isolé où vous pouvez installer des paquets sans affecter le reste de votre système.

Pour créer un environnement virtuel:

- Installez virtualenv: `pip install virtualenv`
- Créez un environnement virtuel: `virtualenv mon_env`
- Activez l'environnement:
  - Windows: `mon_env\Scripts\activate`
  - MacOS/Linux/BSD: `source mon_env/bin/activate`

Pour désactiver l'environnement, tapez simplement: `deactivate`

La documentation officielle

Lorsqu'il s'agit de maîtriser Python ou de résoudre des problèmes complexes, la documentation officielle de Python est inestimable.

Accessible sur python.org, cette documentation est exhaustive, à jour et couvre chaque aspect du langage. Elle offre des explications détaillées, des exemples pertinents et des conseils pratiques pour chaque fonction, module ou concept. Plutôt que de s'appuyer uniquement sur des forums ou des ressources tierces, tout développeur Python devrait prendre l'habitude de consulter la documentation officielle. Non seulement elle fournit des informations précises, mais elle permet également de comprendre les meilleures pratiques et les intentions originales des concepteurs du langage. En bref, la documentation officielle de Python est un outil indispensable et le meilleur ami de tout développeur Python sérieux.

Un langage interprété ?

En quelques mots:

- Interprété : Le code est exécuté ligne par ligne directement par un interpréteur au moment de l'exécution. Pas de phase de compilation séparée. Exemple : Python.
- Compilé : Le code source est transformé en code machine par un compilateur avant l'exécution. Le résultat est un fichier exécutable. Exemple : C, C++.

En bref, les langages compilés sont transformés en code machine avant l'exécution, tandis que les langages interprétés sont exécutés directement par un interpréteur au moment de l'exécution. Les premiers sont plus rapide car le code machine n'est créé qu'une fois et compris directement par le processeur, les seconds sont pus lents car le code source est réinterprété à chaque exécution et doit être traduit en instructions que le processeur comprends.

Il vaut mieux un langage compilé alors ? Pourquoi on fait du Python ?

Déjà parce que la plupart des langages compilés sont sensiblement complexe en comparaison aux langages interprétés, et le gain à l'exécution est souvent le résultat d'un cycle de développement plus long. Ensuite, parce que les performances des langages interprétés sont largement suffisantes dans une écrasante majorité de cas, tout le monde ne fait pas du trading haute-fréquence ou de l'embarqué temps-réel. Enfin, parce que la limite entre interprété et compilé était très nette il y a vingt ans, mais beaucoup moins aujourd'hui.

En pratique, la plupart des langages interprétés aujourd'hui sont compilés pour une machine virtuelle, Python n'est pas vraiment interprété ligne par ligne mais il est compilé pour la PVM (Python Virtual Machine).

- Lorsque vous exécutez un script Python, le code source .py est d'abord compilé en bytecode, Ce bytecode est une représentation de bas niveau, condensée, indépendante de la plateforme, du code source.
- Ce bytecode est ensuite écrit dans un fichier .pyc qui est stocké dans un dossier **pycache**. Cela permet d'accélérer les exécutions ultérieures du script, car le code n'a pas besoin d'être recompilé à moins qu'il n'ait été modifié.
- La machine virtuelle Python (PVM) exécute ensuite ce bytecode, ligne par ligne.

Le bytecode Python n'est pas du code machine natif (comme celui produit pour les langages compilés comme C ou C++). Au lieu de cela, il est conçu pour être exécuté par l'interpréteur Python, ce qui permet à Python de conserver sa portabilité entre différentes plateformes.

Les performances sont nécessairement en dessous d'un programme compilé en code natif, puisque le bytecode va ensuite devoir être interprété en instructions natives, mais une interprétation de bytecode est sensiblement plus rapide qu'une interprétation ligne à ligne.

Notez qu'il y a actuellement un projet en cours de développement, le projet Mojo, qui vise à produire un compilateur capable de convertir du Python en code natif: on aurai alors les performances d'un langage comme C++ mais avec la simplicité de Python.

Syntaxe

Nous n'allons pas voir tous les détails de la syntaxe d'un coup, on va pouvoir les découvrir progressivement au fur et à mesure que le cours avance, mais voici quelques spécificités pour pouvoir commencer à comprendre les premiers exemples.

INDENTATION

Contrairement à de nombreux autres langages, Python utilise l'indentation (espaces ou tabulations) pour délimiter les blocs de code. Cela rend le code Python propre et lisible.

indentation

```
if True:
    print("Ceci est vrai.")
else:
    print("Ceci est faux.")
```

COMMENTAIRES

Les commentaires en Python commencent par le symbole #. Tout ce qui suit ce symbole sur la même ligne est considéré comme un commentaire.

commentaires

```
# Ceci est un commentaire
print("Ceci n'est pas un commentaire.")  # Mais ceci en est un.
```

VARIABLES

En Python, les variables n'ont pas besoin d'être déclarées avec un type spécifique. Vous pouvez simplement les assigner à une valeur.

Variables

```
a = 10
b = "Bonjour"
c = 3.14
```

Elles peuvent aussi changer de type en cours de route:

Changement de type en cours de route

```
a = 10
a = "Bonjour"
```

INSTRUCTIONS

Les instructions sont exécutées de haut en bas. Vous pouvez utiliser des points-virgules pour séparer plusieurs instructions sur une seule ligne, bien que cela ne soit pas courant en Python.

instructions

```
x = 5
y = 10
z = 20; total = x + y + z
```

IMPORTATION DE MODULES

Python possède une riche bibliothèque standard, et vous pouvez également utiliser des bibliothèques tierces. Pour accéder aux fonctions d'un module, vous devez l'importer.

import

```
import math
racine = math.sqrt(16)  # Utilise la fonction sqrt du module math
```

Les points d'entrées

Un "point d'entrée" désigne le point de départ d'un programme ou d'un script. Il existe plusieurs façons d'exécuter du code Python, et chacune a son propre point d'entrée. Voici une vue d'ensemble des différents points d'entrée en Python :

SCRIPT VIA `__MAIN__`

Lorsque vous exécutez un fichier Python directement (par exemple, python mon_script.py), le code à l'intérieur de ce fichier est exécuté. Dans ce contexte, la variable spéciale `__name__` du fichier est définie sur `__main__`. Cela permet aux développeurs d'ajouter une condition if `__name__` == `__main__`: pour s'assurer que certaines parties du code ne sont exécutées que lorsque le fichier est lancé directement, et non lorsqu'il est importé comme un module.

script via **main**

```
#! /usr/bin/env python

if __name__ == "__main__":
    print('hello world !')
```

REPL (READ-EVAL-PRINT LOOP)

Le REPL est un environnement interactif où vous pouvez saisir et exécuter du code Python ligne par ligne. Il est souvent utilisé pour des tests rapides, des débogages ou des expérimentations. Vous pouvez accéder au REPL simplement en tapant python (ou python3 selon votre installation) dans votre terminal ou console. Dans le REPL, chaque instruction est immédiatement évaluée et le résultat est affiché.

le REPL

```
$ python
Python 3.11.4 (main, Jul  5 2023, 08:40:20) [Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('hello world !')
hello
>>> x = 42
>>> x
42
>>> quit()
$
```

MODULE

Un module est un fichier Python contenant des fonctions, des classes et des variables, ainsi que du code exécutable.

Les modules peuvent être importés dans d'autres fichiers Python ou modules à l'aide de l'instruction import. Par exemple, `import math` importe le module math standard de Python aussi bien dans un script que dans le REPL.

module importé dans un script **main**

```
#! /usr/bin/env python
```

```python
import math

if __name__ == "__main__":
    print(math.floor(1.1))
```

module importé dans le REPL

```
$ python
Python 3.11.4 (main, Jul  5 2023, 08:40:20) [Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import math
>>> math.floor(1.1)
1
>>> quit()
$
```

Lorsqu'un module est importé, le code à l'intérieur du module est accessible au script ou au REPL. Python fourni un grand nombre de modules dans sa librairie standard, encore plus de modules mis à disposition par la communauté des développeurs Pythons et accessible via des outils d'installation, et vous pouvez faire vos propres modules qui vous permetteront de réutiliser le même code dans plusieurs projets.

Commentaires

Les commentaires en programmation sont des annotations ajoutées au code qui ne sont pas exécutées lors de son fonctionnement. Ils sont essentiels pour plusieurs raisons. Tout d'abord, ils permettent au développeur d'expliquer son raisonnement, de décrire la fonction d'un segment de code ou de donner des informations sur la manière dont une partie spécifique du programme fonctionne. Cela est particulièrement utile pour les équipes de développement, car cela facilite la compréhension du code par d'autres membres. De plus, les commentaires peuvent servir à désactiver temporairement certaines parties du code sans les supprimer, ce qui est pratique lors du débogage. En Python, les commentaires sont précédés du caractère # et s'étendent jusqu'à la fin de la ligne. Bien que le code puisse fonctionner sans commentaires, un code bien commenté est toujours plus maintenable, compréhensible et collaboratif.

Hello world avec commentaire

```python
#! /usr/bin/env python

# Ce programme affiche le message "Hello, world!" à l'écran
print("Hello, world!")
```

Variables

Dans le monde de la programmation, les variables jouent un rôle central et fondamental. Une variable peut être imaginée comme une boîte dans la mémoire de l'ordinateur, où l'on peut stocker, récupérer ou modifier des informations. Chaque variable possède un nom unique qui permet d'identifier et d'accéder à son contenu. De plus, les variables ont des types, tels que entier, chaîne de caractères ou liste, qui déterminent la nature des données qu'elles peuvent contenir. Comprendre le concept de variables est essentiel, car elles servent de pont entre le code et les données, permettant aux programmes d'interagir dynamiquement avec les informations. En maîtrisant les variables, on acquiert la capacité de manipuler des données de manière flexible et puissante, ouvrant la porte à des applications plus complexes et interactives.

Hello world avec variable et f-string

```python
#! /usr/bin/env python

# Déclaration d'une variable contenant un nom
nom = "Alice"

# Utilisation de la variable pour personnaliser le message
# attention au f devant le guillemet ouvrant:
# il indique que la chaine contient une variable qui sera subtituée par son contenu
print(f"Hello, {nom}!")
```

Fonctions

Les fonctions sont des éléments fondamentaux de la programmation en Python, servant à regrouper des blocs de code pour effectuer une tâche spécifique. Elles permettent de structurer et d'organiser le code, le rendant plus lisible et réutilisable.

Une fonction est définie avec le mot-clé `def``, suivi du nom de la fonction et d'une paire de parenthèses contenant éventuellement des paramètres. Le corps de la fonction est indenté et contient les instructions à exécuter.

définition d'une fonction et appel

```python
def myprint(arg):
    print(arg)

myprint("Hello, Alice!")
```

Une fois définie, une fonction peut être appelée n'importe où dans le code en utilisant son nom suivi de parenthèses. Les fonctions peuvent retourner des valeurs à l'aide du mot-clé return.

définition d'une fonction avec valeur de retour

```
def myprint2(arg):
    print(arg)
    return 42

x = myprint("Hello, Alice!")
```

En Python, les fonctions peuvent accepter un nombre variable d'arguments, avoir des valeurs par défaut pour certains paramètres et même gérer des arguments sous forme de clés. L'utilisation de fonctions favorise les bonnes pratiques de programmation, comme la modularité et la DRY (Don't Repeat Yourself), en évitant la redondance et en facilitant la maintenance du code.

Il existe deux types de fonctions:

FONCTIONS "BUILTIN"

Les fonctions builtin sont des fonctions qui font partie du language lui-même, comme par exemple `print()`, `len()` ou encore `type()` qui sont immédiatement à disposition.

Dans les exemples précédents, `myprint()` et `myprint2()` n'existaient pas avant d'être définies, mais `print()` était accessible sans avoir à importer le moindre module.

Il y a plus d'une soixantaine de fonctions builtins en Python, nous en verrons quelques unes ensemble au fur et à mesure de l'avancée du cours, et en général on tombe assez rapidement sur celles dont on a besoin donc elles ne méritent pas qu'on s'attarde trop: elles reviendront se faire connaitre le moment venu.

quelques builtins

```
>>> x = 42
>>> type(x)
<class 'int'>
>>> len("test")
4
>>> min(2, 3)
2
>>> max(2, 3)
3
```

FONCTIONS NON BUILTIN

Les fonctions non-builtin sont toutes les fonctions qui ne sont pas fournies par le langage lui-même, mais qui sont définies par vos soins ou exposées par un module que vous importez.

Paramètres de fonctions

Les fonctions en Python peuvent accepter des arguments, appelés paramètres, qui permettent de passer des informations à la fonction. Ces paramètres peuvent être de différents types :

PARAMÈTRES POSITIONNELS

Ce sont les paramètres les plus courants. Ils sont définis par leur position dans la définition de la fonction.

Paramètres positionnels

```
def ma_fonction(a, b, c):
    return a + b + c
```

PARAMÈTRES PAR DÉFAUT

Ces paramètres ont une valeur par défaut qui est utilisée si aucune valeur n'est fournie lors de l'appel de la fonction.

Paramètres par défaut

```
def ma_fonction(a, b=5):
    return a + b
```

PARAMÈTRES MOT-CLÉ

Lors de l'appel d'une fonction, vous pouvez spécifier des arguments en utilisant le nom du paramètre, ce qui permet de passer les arguments dans n'importe quel ordre.

Paramètres mot-clé

```
def ma_fonction(a, b, c):
    return a + b + c

ma_fonction(b=2, c=3, a=1)
```

PARAMÈTRES ARBITRAIRES

Si vous ne savez pas combien d'arguments seront passés à votre fonction, vous pouvez utiliser *args pour les paramètres positionnels et **kwargs pour les paramètres mot-clé.

Paramètres arbitraires

```
>>> def ma_fonction(*args, **kwargs):
...     print(args)
```

```
...     print(kwargs)
...
>>> ma_fonction(1, 2, 3)
(1, 2, 3)
{}
>>> ma_fonction(a=1, b=2, c=3)
()
{'a': 1, 'b': 2, 'c': 3}
```

Types

En Python, chaque valeur est associée à un type de données spécifique qui détermine la nature de cette valeur. Les types de données fondamentaux en Python sont :

- Entiers (`int`): Ce sont des nombres entiers, sans partie décimale. Exemple : 5, -3, 42.
- Nombres à virgule flottante (`float`) : Ce sont des nombres qui ont une partie décimale. Exemple : 5.0, -3.14, 2.71.
- Chaînes de caractères (`str`) : Ce sont des séquences de caractères, généralement utilisées pour représenter des mots ou des textes. Exemple : "Hello", 'Python'.
- Listes (`list`) : Collections ordonnées d'éléments qui peuvent être de n'importe quel type. Exemple : [1, 2, 3], ["a", "b", "c"].
- Tuples (`tuple`) : Similaires aux listes, mais immuables, ce qui signifie que leurs éléments ne peuvent pas être modifiés une fois définis. Exemple : (1, 2, 3).
- Dictionnaires (`dict`) : Collections non ordonnées de paires clé-valeur. Exemple : {"nom": "Alice", "âge": 30}.
- Ensembles (`set`) : Collections non ordonnées d'éléments uniques. Ils sont utiles pour stocker des éléments sans doublons. Exemple : {1, 2, 3, 3} donnera {1, 2, 3}.
- Booléens (`bool`) : Représentent les valeurs de vérité, soit `True` (vrai) ou `False` (faux).
- Classes : Elles permettent de définir des objets et représentent la base de la programmation orientée objet en Python, on reviendra dessus.

Ces types de données sont les briques de base de la programmation en Python. Ils permettent aux développeurs de représenter et de manipuler une grande variété d'informations, des simples nombres aux structures de données complexes.

LISTES DE COMPRÉHENSION

Les listes de compréhension sont une manière concise de créer des listes ou des dictionnaires.

Listes: `[expression for item in iterable if condition]`

List comprehension

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

Dictionnaires: `{key: value for item in iterable if condition}`

Dict comprehension

```
>>> x = {x: x**2 for x in (2, 3, 4)}
>>> x
{2: 4, 3: 9, 4: 16}
>>>
```

TRUTHINESS

À ma connaissance, tous les langages ont une notion de "truthiness": toute valeur de tout type est implicitement vraie ou fausse.

Par exemple, en C, la valeur 0 est considérée comme `False` et toute valeur différente de 0 est considérée comme `True`, donc `if x { ... }` est considéré comme vrai et exécute le bloc à partir du moment ou `x` n'est pas égal à zéro.

Python a une approche intéressante de la truthiness: toute valeur égale à 0, ou "vide" pour les types non numériques, est considérée comme `False`: un tableau vide est faux, une chaine vide est fausse, un dictionnaire vide est faux...

Il est donc possible de tester rapidement si un type est vide en testant sa truthiness comme dans les exemples suivants.

Truthiness

```
>>> x = []
>>> x == False
False
>>> if x:
...     print("has data")
... else:
...     print("is empty")
...
is empty
>>>
```

NONE

`None` est un type spécial en Python qui représente l'absence de valeur ou la nullité. Il est souvent utilisé pour signifier qu'une variable existe, mais qu'elle n'a pas encore été assignée à une valeur spécifique ou pour indiquer qu'une fonction ne renvoie rien.

None

```
>>> def foobar():
...     pass
...
>>> x = foobar()
>>> x is None
True
>>>
```

f-string et spécificateurs

Depuis peu, Python propose une fonctionnalité nommée `f-string` que nous avons utilisé un peu plus haut: les chaines de caractères préfixées par `f` sont considérées comme contenant des variables à remplacer. Par exemple, `f"Salut, {prenom}"` va remplacer `{prenom}` par la valeur actuelle de la variable `prenom` au moment où la `f-string` est évaluée.

Avant, il fallait avoir recours à un autre mécanisme, toujours utile selon les cas d'usage: les spécificateurs.

Les spécificateurs sont des symboles utilisés principalement dans les fonctions de formatage de chaînes, comme `print()` ou la méthode `.format()` de tous les objets de types `str`, pour contrôler la manière dont les valeurs sont affichées.

Voici quelques spécificateurs courants :

- %s : Formatage pour les chaînes de caractères.
- %d : Formatage pour les entiers.
- %f : Formatage pour les nombres à virgule flottante.
- %.2f : Formatage pour les nombres à virgule flottante avec deux décimales.

Spécificateurs

```
nom = "Alice"
age = 30
taille = 1.75

print("Mon nom est %s, j'ai %d ans et je mesure %.2f mètres." % (nom, age, taille))
```

Opérateurs

OPÉRATEURS ARITHMÉTIQUES

Ils sont utilisés pour effectuer des opératins mathématiques.

ADDITION: +

```
>>> x = 5 + 3
>>> x
8
```

SOUSTRACTION: -

```
>>> x = 5 - 3
>>> x
2
```

MULTIPLICATION: *

```
>>> x = 5 * 3
>>> x
15
```

DIVISION: /

```
>>> x = 8 / 2
>>> x
4.0
```

MODULO: %

```
>>> x = 13 % 12
>>> x
1
```

EXPONENTIATION (PUISSANCE): **

```
>>> x = 2 ** 3
>>> x
8
```

DIVISION ENTIÈRE: //

```
>>> x = 5 // 3
>>> x
```

```
1
```

OPÉRATEURS DE COMPARAISON

Ils sont utilisés pour comparer deux valeurs.

ÉGAL: ==

```
>>> x = 5 == 3
>>> x
False
```

DIFFÉRENT: !=

```
>>> x = 5 != 3
>>> x
True
```

SUPÉRIEUR: >

```
>>> x = 5 > 3
>>> x
True
```

INFÉRIEUR: <

```
>>> x = 8 < 2
>>> x
False
```

SUPÉRIEUR OU ÉGAL: >=

```
>>> x = 5 >= 5
>>> x
True
```

INFÉRIEUR OU ÉGAL: <=

```
>>> x = 2 <= 3
>>> x
True
```

OPÉRATEURS LOGIQUES

Ils sont utilisés pour combiner des expressions conditionnelles.

AND

```
>>> x = 5 > 3 and 5 < 10
>>> x
True
```

OR

```
>>> x = 5 > 3 or 5 > 10
>>> x
True
```

NOT

```
>>> x = 5 > 3
>>> x
True
>>> not x
False
>>> x = not(5 > 3)
>>> x
False
```

OPÉRATEURS D'AFFECTATION

Ils sont utilisés pour assigner des valeurs aux variables.

=

Assigne une valeur.

```
>>> x = 5
>>> x
5
```

+=

Équivaut à ajouter à la valeur existante, `x += 5` équivaut à `x = x + 5`.

```
>>> x += 5
>>> x
10
```

-=

Équivaut à supprimer à la valeur existante.

```
>>> x -= 1
>>> x
9
```

***=**

Équivaut à multiplier la valeur existante.

```
>>> x *= 2
>>> x
18
```

/=

Équivaut à diviser la valeur existante.

```
>>> x /= 2
>>> x
9.0
```

%=

Équivaut à appliquer un modulo à la valeur existante.

```
>>> x = 10
>>> x %= 3
>>> x
1
```

//=

Équivaut à appliquer une division entière à la valeur existante.

```
>>> x = 10
>>> x //= 3
>>> x
3
```

****=**

Équivaut à appliquer une puissance à la valeur existante.

```
>>> x = 2
>>> x **= 10
>>> x
1024
```

OPÉRATEURS BINAIRES

Ces opérateurs sont particulièrement utiles dans des domaines tels que la programmation de bas niveau, la cryptographie, la compression de données et d'autres domaines où la manipulation directe des bits est nécessaire.

Bien que leur utilisation soit moins courante dans la programmation de haut niveau, il est toujours bon de les connaître et de comprendre comment ils fonctionnent.

ET BINAIRE `&`

Cet opérateur renvoie un nombre dont chaque bit est le résultat de l'opération "ET" bit à bit des opérandes.

ET binaire

```
>>> x = 5  # 0101 en binaire
>>> y = 3  # 0011 en binaire
```

```
>>> print(x & y)  # 0001 en binaire, soit 1 en décimal
```

OU BINAIRE `|`

Renvoie un nombre dont chaque bit est le résultat de l'opération "OU" bit à bit des opérandes.

OU binaire

```
>>> x = 5  # 0101 en binaire
>>> y = 3  # 0011 en binaire
>>> print(x | y)  # 0111 en binaire, soit 7 en décimal
```

OU EXCLUSIF BINAIRE `^`

Renvoie un nombre dont chaque bit est le résultat de l'opération "OU exclusif" bit à bit des opérandes.

OU binaire

```
>>> x = 5  # 0101 en binaire
>>> y = 3  # 0011 en binaire
>>> print(x ^ y)  # 0110 en binaire, soit 6 en décimal
```

NÉGATION BINAIRE `~`

Inverse tous les bits du nombre.

Négation binaire

```
>>> x = 5  # 0101 en binaire
>>> print(~x)  # 1010 en binaire
```

DÉCALAGE À GAUCHE `<<`

Décale les bits du premier opérande vers la gauche d'un nombre de positions spécifié par le second opérande.

Décalage à gauche

```
>>> x = 4  # 0100 en binaire
>>> print(x<<1)  # 1000 en binaire, 8
```

DÉCALAGE À DROITE `>>`

Décale les bits du premier opérande vers la gauche d'un nombre de positions spécifié par le second opérande.

Décalage à droite

```
>>> x = 4  # 0100 en binaire
>>> print(x >> 1)  # 0010 en binaire, soit 2 en décimal
```

ASSIGNATIONS

Tous les opérateurs supportent des versions "avec assignation" comme on a pu voir sur les opérateurs arithmétiques: `x &= 1`, `x |= 1`, `x ^= 1`, `x <<= 1` ou encore `x >>= 1`.

À QUOI ÇA PEUT DONC BIEN SERVIR ?

Au delà des cas où les opérations binaires s'imposent à vous, parce que vous implémentez une specification où il est explicitement écrit "mettre le 13ème bit à 1", les opérateurs binaires permettent de simplifier certains motifs de code.

Je vais donner un seul exemple pour que vous compreniez leur puissance et pourquoi ils est intéressant de les creuser:

Si je veux stocker des informations concernant un utilisateur, comme par exemple ses droits sur une application, je peux le faire en assignant à chaque permission une variable de type `bool`... mais lorsque je voudrais mettre ça en base de donnée, ma table aura elle aussi une colonne pour chaque permission. Donc si j'ai 30 permissions différentes, j'ai 30 bool et 30 colonnes... qui débouche sur 30 ensemble de fonctions pour chaque droit, des test dans tous les sens pour vérifier si des droits sont compatibles (est-ce que c'est ok si j'ai le droit X et le droit Y), etc...

Ça marche, c'est ce que font la plupart des personnes.

Ou alors...

On prends un `int`, on considère chacun de ces bits comme un `bool`: si le bit est à 0 c'est faux, s'il est à 1 c'est vrai. On a une seule valeur qui encode les permissions, une seule colonne en base de donnée, un seul set de fonction qui utilise les opérateurs binaires pour savoir si oui ou non un bit est "setté", et les tests sont plus simples car il est possible de vérifier plusieurs bits d'un coup et de fournir une liste de combinaisons interdites.

```
user.canExec = False
user.canWrite = True
user.canRead = True
if not user.canExec and user.canWrite and user.canRead:
    print("l'utilisateur peut ecrire et lire mais pas executer")

vs.

user.permissions = WRITE|READ
if user.permissions & EXEC|WRITE|READ == WRITE|READ:
    print("l'utilisateur peut ecrire et lire mais pas executer")
```

C'est un coup à prendre, un peu de travail en amont pour comprendre la logique des opérateurs &, |, ^ et ~, mais le bénéfice est immense: il existe de nombreux cas où le code est grandement simplifié, où le stockage en base de donnée est simplifié, où les performances sont améliorées, et surtout c'est une connaissance qui se transpose à tous les langages.

IF TERNAIRE

L'`if ternaire` est une manière concise d'écrire une condition if/else.

if ternaire

```
>>> x = 5
>>> y = "plus grand" if x > 3 else "plus petit"
>>> y
'plus grand'
```

Structures de contrôle et boucles

Les structures de contrôle sont essentielles en programmation car elles permettent de diriger le flux d'exécution d'un programme. En Python, comme dans la plupart des langages de programmation, il existe plusieurs structures de contrôle principales.

STRUCTURES DE CONTRÔLE CONDITIONNELLES

Elles permettent d'exécuter certains blocs de code en fonction de conditions spécifiques.

IF

Exécute un bloc de code si une condition est vraie.

```
#! /usr/bin/env python

# Déclaration d'une variable avec la valeur 42
x = 42
if x == 42:
    print("x est égal à 42")
```

ELIF

Vérifie une autre condition si la condition précédente n'est pas vraie.

```
#! /usr/bin/env python

x = 43
if x == 42:
    print("x est égal à 42")
elif x == 43:
    print("x est égal à 43")
```

ELSE

Exécute un bloc de code si aucune des conditions précédentes n'est vraie.

```
#! /usr/bin/env python

x = 44
if x == 42:
    print("x est égal à 42")
elif x == 43:
    print("x est égal à 43")
else:
    print("x n'est égal ni à 42, ni à 43)
```

PASS

Permet de remplacer un bloc de code pour... ne rien faire. La plupart du temps, il sert à ce que la syntaxe du langage soit respectée "le temps de" finir une implémentation, mais nous verrons par la suite qu'il s'imposera de lui-même dans certains cas.

pass

```
#! /usr/bin/env python

def mafonction():
    pass

x = 42
if x == 42:
    print("x est égal à 42")
else:
    pass # TODO: à implémenter plus tard
```

MATCH

Le mot-clé `match` a été introduit dans Python 3.10 comme une extension de la capacité de correspondance de motifs (ou "pattern matching") du langage. Il offre une manière plus expressive et lisible de traiter les structures de données et de prendre des décisions basées sur la forme et le contenu de ces structures.

La correspondance de motifs avec match peut être vue comme une version généralisée et améliorée de l'instruction switch/case présente dans d'autres langages, mais avec des capacités beaucoup plus puissantes.

match

```
def http_status(code):
    match code:
        case 200:
            return "OK"
        case 403:
            return "Forbidden"
        case 404:
            return "Not Found"
        case 500:
            return "Internal Server Error"
        case _:
            return "Autre"

vs

def http_status(code):
    if code == 200:
        return "OK"
    elif code == 403:
        return "Forbidden"
    elif code == 404:
        return "Not Found"
    elif code == 500:
        return "Internal Server Error"
    else:
        return "Autre"
```

La structure de contrôle match est extrêmement puissante et flexible, elle peut être utilisée avec des motifs de "matching" plus complexes. Nous ne les verrons pas tous tout de suite, certains repointeront le bout de leur nez plus tard, mais nous allons voir encore un exemple qui exploite les variables de capture.

VARIABLES DE CAPTURE

Dans l'exemple de match plus complexe, les variables `x` et `y` prennent a valeur de leur position dans la variable `point`.

match plus complexe

```
point = (2, 3)

match point:
    case (0, 0):
        print("Origine")
    case (0, y):
        print(f"Sur l'axe des Y, à la position {y}")
    case (x, 0):
        print(f"Sur l'axe des X, à la position {x}")
    case (x, y):
        print(f"Position ({x}, {y})")
    case _:
        print("Autre")
```

STRUCTURES DE CONTRÔLE DE BOUCLE

Elles permettent d'exécuter un bloc de code plusieurs fois.

FOR

Parcourt une séquence et exécute un bloc de code pour chaque élément de cette séquence.

```
#! /usr/bin/env python

for i in range(0, 10):
    print(f"tour de boucle {i}")
```

WHILE

Exécute un bloc de code tant qu'une condition est vraie.

```
#! /usr/bin/env python

i = 0
while i < 10:
    print(f"tour de boucle {i}")
    i = i + 1
```

STRUCTURES DE CONTRÔLE DE BOUCLE

Elles permettent d'altérer le comportement d'une boucle.

BREAK

Termine la boucle en cours et passe à la suite du programme.

```
#! /usr/bin/env python

for i in range(0, 10):
    if i == 3:
        break
    print(f"tour de boucle {i}")
```

CONTINUE

Termine la boucle en cours et passe à la suite du programme.

```
#! /usr/bin/env python

for i in range(0, 10):
    if i == 3:
        continue
    print(f"tour de boucle {i}")
```

Pointeurs et références

En Python, les concepts de pointeurs et de références sont gérés de manière différente par rapport à des langages comme C ou C++.

références

```
>>> a = [1, 2, 3]
>>> b = a
>>> a
[1, 2, 3]
>>> b
[1, 2, 3]
>>> a.append(4)
>>> b
[1, 2, 3, 4]
>>>
```

Dans l'exemple ci-joint, `a` et `b` font référence au même objet liste en mémoire. Si vous modifiez la liste via la variable `a`, les modifications seront également visibles via la variable `b`, car elles pointent vers le même objet.

Il faut imaginer que le tableau a été alloué quelque part en mémoire, et que `a` et `b` sont des étiquettes collées sur cet emplacement mémoire. Les deux étiquttes sont sur la même case, si le contenu change, peu importe que l'on demande ce qu'il y a derrière l'étiquette `a` ou `b`, c'est le même contenu modifié.

ABSENCE DE POINTEURS ET PASSAGE PAR RÉFÉRENCE

Contrairement à des langages comme C ou C++, Python n'a pas de pointeurs explicites.

C'est-à-dire que vous ne pouvez pas accéder directement à la mémoire ou manipuler des adresses mémoire comme vous le feriez avec des pointeurs. Cela rend Python plus sûr et plus facile à utiliser, car il évite de nombreux pièges et erreurs courants associés à la manipulation directe de la mémoire.

Lorsque vous passez un objet mutable (comme une liste ou un dictionnaire) à une fonction, vous passez en réalité une référence à cet objet. Cela signifie que si la fonction modifie l'objet, ces modifications seront reflétées à l'extérieur de la fonction.

Passage par référence

```
def ajouter_element(lst):
    lst.append(4)

a = [1, 2, 3]
ajouter_element(a)
print(a)  # Affiche [1, 2, 3, 4]
```

Dans l'exemple du passage par référence, la liste a est modifiée à l'intérieur de la fonction `ajouter_element()` car une référence à la liste est passée à la fonction.

En Python, il est essentiel de comprendre que les variables sont des références à des objets et que la manipulation de ces références peut avoir des effets sur les objets sous-jacents. Bien que Python n'ait pas de pointeurs explicites, la manière dont il gère les références offre une grande flexibilité tout en évitant les complications associées à la gestion directe de la mémoire.

Allocation et désallocation dynamique

L'allocation et la désallocation dynamiques sont des concepts essentiels en programmation, permettant de gérer la mémoire utilisée par les applications. En Python, ces concepts sont traités de manière quelque peu différente par rapport à des langages de bas niveau comme C ou C++.

ALLOCATION DYNAMIQUE

En Python, l'allocation de mémoire est gérée automatiquement. Lorsque vous créez un nouvel objet, Python alloue automatiquement la mémoire nécessaire pour cet objet. Par exemple, lorsque vous créez une nouvelle liste ou un nouveau dictionnaire, Python s'occupe de l'allocation de mémoire pour ces structures.

allocation dynamique

```
# Allocation de mémoire pour une liste
ma_liste = [1, 2, 3, 4, 5]

# Allocation de mémoire pour un dictionnaire
mon_dict = {"clé": "valeur"}
```

DÉSALLOCATION DYNAMIQUE ET GARBAGE COLLECTION

La désallocation de mémoire est également gérée automatiquement en Python grâce à un mécanisme appelé "garbage collection" (collecte des déchets). Le garbage collector de Python détecte les objets qui ne sont plus référencés par le programme et libère la mémoire qu'ils occupent.

Lorsqu'une variable n'est plus utilisée ou qu'elle sort de la portée, elle n'est pas immédiatement désallouée. Au lieu de cela, Python marque cette mémoire comme étant récupérable. Le garbage collector intervient périodiquement pour récupérer cette mémoire.

garbage collection

```
def ma_fonction():
    temp_liste = [10, 20, 30]  # Allocation de mémoire pour une liste
    # À la fin de cette fonction, temp_liste sort de la portée

# Après l'appel de cette fonction,
# la mémoire utilisée par temp_liste est marquée comme récupérable
ma_fonction()
```

RÉFÉRENCES CIRCULAIRES

Une référence circulaire se produit lorsque deux objets (ou plus) se réfèrent mutuellement, créant ainsi un cycle. Le garbage collector de Python est capable de détecter et de gérer les références circulaires, évitant ainsi les fuites de mémoire.

Références circulaires

```
>>> a = {}
>>> b = {}
>>> a['b'] = b
>>> b['a'] = a
>>> a
{'b': {'a': {...}}}
>>> b
{'a': {'b': {...}}}
>>>
```

UN DERNIER MOT SUR LA MÉMOIRE

En Python, les développeurs n'ont généralement pas à se soucier de l'allocation et de la désallocation manuelles de la mémoire, car le langage s'en occupe automatiquement.

Cela simplifie le développement et réduit le risque d'erreurs liées à la mémoire... mais, il est toujours bon de comprendre comment la gestion de la mémoire fonctionne en coulisse, en particulier pour les applications à forte consommation de ressources ou pour le débogage.

Python ne vous force pas à faire des allocations et désallocations vous-même, il les fait à votre place, c'est à double tranchant: le développement est simplifié et les bugs de gestion mémoire sont quasi inexistants... mais vous ne pouvez pas gérer la mémoire aussi finement que vous le voulez.

Paramètres passés au `main` via la ligne de commande

Lors de l'exécution d'un script Python depuis la ligne de commande, il est courant de vouloir passer des arguments ou des paramètres au programme.

Ces arguments sont accessibles dans le script via le module `sys` et la liste `sys.argv`.

Le premier élément, `sys.argv[0]`, est toujours le nom du script lui-même, tandis que les arguments suivants sont stockés aux indices suivants.

Par exemple, en exécutant `python mon_script.py arg1 arg2`, `sys.argv[1]` contiendra `"arg1"` et `sys.argv[2]` contiendra `"arg2"`.

paramètres de la ligne de commande

```
import sys

if __name__ == "__main__":
    for i, arg in enumerate(sys.argv):
        print(f"Argument {i}: {arg}")
```

Entrées/sorties

Les opérations d'entrée/sortie (I/O) sont fondamentales en informatique.

Elles permettent à un programme d'interagir avec le monde extérieur, que ce soit pour recevoir des données (entrée) ou pour envoyer des résultats (sortie).

Ces interactions peuvent se faire via divers moyens, tels que le clavier, la souris, l'écran, les fichiers, ou même les réseaux.

En Python, les fonctions de base pour les I/O incluent `input()` pour lire une chaîne de caractères depuis le clavier et `print()` pour afficher du texte à l'écran.

La manipulation de fichiers, comme l'ouverture, la lecture et l'écriture, est également une forme d'I/O.

Comprendre et maîtriser les I/O est essentiel pour tout développeur, car cela permet de créer des programmes interactifs et dynamiques qui peuvent traiter des données, les stocker et les présenter à l'utilisateur de manière efficace.

Manipulation de fichiers en Python

La manipulation de fichiers est une tâche courante en programmation.

Python offre des outils simples et efficaces pour lire et écrire des fichiers.

Pour ouvrir un fichier, utilisez la fonction `open()`, qui retourne un objet fichier. Les modes d'ouverture les plus courants sont 'r' pour la lecture et 'w' pour l'écriture, 'a' pour l'ajout.

Une fois le fichier ouvert, vous pouvez utiliser les méthodes `read()` pour lire son contenu ou `write()` pour écrire dedans. Il est crucial de toujours fermer le fichier après l'avoir utilisé avec la méthode `close()`` pour libérer les ressources.

Lecture et écriture dans un fichier

```
# Lire un fichier
fichier = open('mon_fichier.txt', 'r')
contenu = fichier.read()
print(contenu)
fichier.close()

# Écrire dans un fichier
fichier = open('mon_fichier.txt', 'w')
fichier.write("Bonjour, monde!")
fichier.close()
```

Context managers

Les context managers permettent de gérer efficacement les ressources, comme les fichiers ou les connexions réseau. Ils sont généralement utilisés avec l'instruction `with`.

Context manager

```
with open('mon_fichier.txt', 'r') as fichier:
    contenu = fichier.read()
```

Lecture de fichier ligne à ligne avec un context manager

```
with open('mon_fichier.txt', 'r') as fichier:
    while True:
        ligne = fichier.readline()
        if not ligne:
            break
        print(line.strip())
```

Lecture de l'entrée standard avec un context manager

```
import sys

with sys.stdin as fichier:
    while True:
        ligne = fichier.readline()
        if not ligne:
            break
        print(line.strip())
```

Gestion des erreurs et exceptions en Python

Dans le processus de développement, il est courant de rencontrer des erreurs.

En Python, il existe deux types principaux d'erreurs : les erreurs de syntaxe et les exceptions.

Les erreurs de syntaxe, souvent appelées erreurs de parsing, se produisent lorsque le programme contient une instruction qui n'est pas conforme à la syntaxe du langage. Par exemple, oublier de fermer une parenthèse ou d'ajouter un deux-points à la fin d'une instruction conditionnelle.

Les exceptions, en revanche, se produisent lors de l'exécution du programme, même si la syntaxe est correcte. Cela peut être dû à des opérations invalides, comme tenter de diviser par zéro ou d'accéder à un fichier qui n'existe pas.

Heureusement, Python offre des mécanismes pour gérer ces exceptions grâce aux instructions try et except. En enveloppant le code susceptible de lever une exception dans un bloc `try`, et en définissant comment traiter cette exception dans un bloc `except`, vous pouvez contrôler la manière dont votre programme réagit aux erreurs inattendues, permettant ainsi une exécution plus robuste et prévisible.

Exceptions simples

```
try:
    # Demande à l'utilisateur de saisir un nombre
    num = int(input("Entrez un nombre: "))
    print(f"Vous avez saisi {num}")

except ValueError:
    print("Erreur : Ce n'est pas un nombre valide !")
```

Exceptions simples mais avec capture d'informatiosn

```
try:
    # Demande à l'utilisateur de saisir un nombre et tente de le diviser par zéro
    num = int(input("Entrez un nombre: "))
    resultat = num / 0

except Exception as exc:
```

```
    print(f"Une erreur s'est produite : {exc}")
```

Gestion totale des exceptions

```
try:
    # Demande à l'utilisateur de saisir deux nombres
    num1 = int(input("Entrez le premier nombre: "))
    num2 = int(input("Entrez le deuxième nombre: "))

    # Tente de diviser les deux nombres
    resultat = num1 / num2
    print(f"Le résultat de {num1} divisé par {num2} est {resultat}")

except ZeroDivisionError:
    # s'il y a division par zéro
    print("Erreur : Division par zéro !")

except ValueError:
    # si num1 ou num2 ne sont pas des nombres
    print("Erreur : Veuillez saisir un nombre valide !")

except Exception as exc:
    # toute autre erreur (ici, aucune possible)
    print(f"Erreur : Je ne comprends pas ce qu'il se passe: {exc}")

finally:
    # éxecuté dans tous les cas, exception ou non
    print("Fin de l'opération.")
```

Docstrings: Documenter son code

En Python, les “docstrings” sont des chaînes de caractères utilisées pour documenter des parties spécifiques du code, telles que les modules, les classes, les méthodes ou les fonctions.

Contrairement aux commentaires classiques, les docstrings sont conservées tout au long de l’exécution du programme et peuvent être accessibles à l’aide de la fonction `help()` ou via l’attribut `__doc__` de l’objet concerné.

Pour définir une docstring, il suffit d’encadrer la description avec des triples guillemets (simples ou doubles) au début de la section de code que vous souhaitez documenter.

Docstring

```
def ma_fonction(x, y):
    """
    Cette fonction calcule la somme de deux nombres.

    Arguments:
    x -- le premier nombre
    y -- le second nombre

    Retourne:
    La somme de x et y.
    """
    return x + y
```

Linter

Un “linter” est un outil qui analyse le code source pour détecter des erreurs, des bugs, des styles non conformes et d’autres problèmes potentiels.

En Python, l’utilisation d’un linter est essentielle pour maintenir la qualité du code, assurer sa lisibilité et prévenir les erreurs avant l’exécution. Des linters populaires comme `pylint` ou `flake8` offrent une analyse approfondie du code, allant des erreurs de syntaxe aux conventions de nommage en passant par la complexité des fonctions.

En intégrant un linter dans le processus de développement, les développeurs peuvent s’assurer que leur code respecte les standards de la communauté, facilite la collaboration et réduit les risques d’erreurs. De plus, de nombreux environnements de développement intégrés (IDE) supportent l’intégration de linters, permettant ainsi une vérification en temps réel à mesure que le code est écrit.

Exemple de code avec des “problèmes”

```
def maFonction():
  return "Bonjour tout le monde!"
```

Sortie de pylint

```
************* Module exemple
exemple.py:1:0: C0103: Function name "maFonction" doesn't conform to snake_case naming style (invalid-name)
exemple.py:1:0: C0116: Missing function or method docstring (missing-function-docstring)

-----------------------------------
Your code has been rated at 5.00/10 (previous run: 5.00/10, +0.00)
```

Code corrigé

```
def ma_fonction():
    """
```

```
    Retourne un message de salutation.
    """
    return "Bonjour tout le monde!"
```

Sortie de pylint sur le code corrigé

```
--------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 5.00/10, +5.00)
```

Création d'un package avec des modules en Python

Dans Python, un package est une manière d'organiser des modules associés dans un répertoire unique.

Ce regroupement permet de structurer le code de manière plus claire, surtout lorsque votre projet commence à grandir.

Pour créer un package, commencez par créer un répertoire (dossier) qui portera le nom de votre package.

À l'intérieur de ce répertoire, placez un fichier spécial nommé `__init__.py` (qui peut être vide) pour indiquer à Python que ce répertoire doit être traité comme un package ou un module.

Ensuite, vous pouvez ajouter autant de modules (fichiers `.py`) que vous le souhaitez dans ce répertoire. Par exemple, si vous créez un package nommé `mon_package` contenant les modules `module1.py` et `module2.py`, vous pourrez ensuite importer ces modules dans votre code principal en utilisant `from mon_package import module1, module2`.

Cette organisation modulaire facilite la maintenance, le partage et la réutilisation du code.

Session 2

Dans la première session, nous avons survolé le langage et sa syntaxe de manière générale.

Dans cette session, nous allons aborder les fonctionnalités orientées objet du langage.

La Programmation Orientée Objet (POO) en très très résumé

La POO est un paradigme de programmation, une façon de se représenter les problèmes d'une certaine manière, au même titre que la programmation procédurale, ou la programmation fonctionnelle.

La POO est une façon d'écrire des programmes en pensant les différentes composantes de ce programme en termes d'"objets", un peu comme si vous organisiez une boîte à outils. Chaque objet est comme un outil spécifique ayant une fonction précise (comme un marteau ou une vis). Ces objets peuvent avoir des caractéristiques (par exemple, la couleur du marteau) et des actions qu'ils peuvent effectuer (comme clouer avec le marteau). En regroupant les caractéristiques et les actions au sein d'objets, cela rend le code plus organisé, plus clair et plus facile à gérer.

La POO repose fortement sur la notion d'objets, avec leurs caractéristiques et leurs actions, mais aussi fortement sur la relation que les objets ont les uns avec les autres comme nous allons le voir.

Python est-il un bon langage pour la programmation orientée objet ?

Lorsqu'il est question de POO, il est courant d'y associer immédiatement le Java ou le C++ tant le paradigme objet est au coeur de ces langages.

Dans la pratique, tout paradigme est problématique dans son excès et la POO n'est pas un cas à part.

Lorsque l'on suit tous les préceptes de la POO à la lettre, le code est théoriquement très élégant mais l'on peut se perdre dans des abstractions et ce que l'on appelle souvent l'over-engineering, monter une solution beaucoup plus complexe que ce qu'elle a besoin d'être: les abstractions s'empilent les unes sur les autres, on comprends à la lecture du code ce qu'il est censé faire en théorie, mais on perds le fil de ce qu'il se passe en pratique au travers des différentes couches. La POO demande un certain niveau d'expertise pour produire quelque chose de proprement élégant.

Python est un très bon compromis car il permet la POO mais sans imposer toute la théorie rigide. Il est possible de respecter strictement les préceptes de la POO, mais il est aussi possible de s'en écarter par moment, ce qui permet de trouver l'équilibre entre code proprement découpé et lasagne d'abstractions.

Personnellement, si je dois faire de la POO, j'irai naturellement vers Python *justement* parce qu'il va permettre d'être pragmatique, tout simplement.

Classes, instances et cycle de vie de l'objet

Les classes permettent de créer de nouveau types à partir de types existants qui sont eux même des classes.

En Python, `int` ou `float` sont des classes, `True` et `False` sont des instances de la classe `bool`, `None` est une instance de la classe `NoneType` et... même la fonction builtin `print()` est une instance de la classe `builtin_function_or_method`.

En Python tout est objet, donc creusons un peu ce qu'est une classe, une instance et le cycle de vie des objets.

CLASSE

Une classe est un modèle à partir duquel des objets sont créés. Elle définit des attributs et des méthodes qui caractérisent tout objet créé à partir de cette classe.

classe

```
class Chat:
    couleur = "noir"
```

INSTANCE

Une instance est un objet individuel créé à partir d'une classe. Chaque instance a ses propres attributs qui peuvent être différents des valeurs par défaut définies dans la classe.

instance

```
mon_chat = Chat()
```

CYCLE DE VIE DE L'OBJET

Le cycle de vie d'un objet commence lorsqu'il est créé (instancié) et se termine lorsqu'il est détruit. Python gère automatiquement la gestion de la mémoire, mais fournit des méthodes spéciales (comme `__init__` et `__del__`) pour initialiser et nettoyer les ressources.

Méthodes, constructeurs et déstructeurs

MÉTHODES

Ce sont des fonctions définies à l'intérieur d'une classe et elles opèrent sur des données membres de cette classe.

Méthodes

```
class Chat:
    def miauler(self):
        print("Miaou!")
```

CONSTRUCTEURS (`__INIT__`)

C'est une méthode spéciale qui est automatiquement appelée lors de la création d'une instance. Elle est généralement utilisée pour initialiser les attributs.

Constructeur

```
class Chat:
    def __init__(self, nom):
        self.nom = nom
```

DESTRUCTEURS (`__DEL__`)

Bien que rarement utilisé en Python (car Python a un ramasse-miettes), c'est une méthode qui est appelée lorsque l'objet est sur le point d'être détruit.

Destructeur

```
class Chat:
    def __del__(self):
        print("L'objet chat est détruit.")
```

GETTERS ET SETTERS

Les getters et setters sont des méthodes utilisées en programmation orientée objet pour contrôler l'accès aux attributs d'un objet.

- Getter : C'est une méthode qui permet d'obtenir la valeur d'un attribut privé. Au lieu d'accéder directement à l'attribut, vous utilisez le getter pour le récupérer. Cela permet d'encapsuler (ou cacher) la représentation interne de l'attribut.
- Setter : C'est une méthode qui permet de définir ou de modifier la valeur d'un attribut privé. Au lieu de modifier directement l'attribut, vous utilisez le setter. Cela permet d'ajouter des contrôles ou des validations lors de la modification de l'attribut.

Getter & Setter

```
class Chat:
    def __init__(self, value):
        self._nom = value

    def get_nom(self):
        return self._nom

    def set_nom(self, value):
        if not value:
            raise ValueError("Le nom ne peut pas être vide.")
        self._nom = value

>>> c = Chat('Ragazza')
>>> c.get_nom()
'Ragazza'
>>> c.set_nom('Gaufrette')
>>> c.get_nom()
'Gaufrette'
```

En Python, les getters et setters sont souvent définis à l'aide des propriétés (`property`), nous verrons les annotations plus tard, c'est juste une manière de montrer qu'il y a plusieurs approches pour cacher un attribut privé.

Getter & Setter pour les fortiches

```
class Chat:
    def __init__(self, value):
        self._nom = value

    @property
    def nom(self):
        return self._nom

    @nom.setter
    def nom(self, value):
        if not value:
            raise ValueError("Le nom ne peut pas être vide.")
        self._nom = value

>>> c = Chat('Ragazza')
>>> c.nom
'Ragazza'
>>> c.nom = 'Gaufrette'
>>> c.nom
'Gaufrette'
```

Dans le premier exemple, au lieu d'accéder directement à l'attribut _marque, on utilise les méthodes get_nom (getter) et set_nom (setter) pour obtenir et définir sa valeur, tout en ajoutant une validation dans le setter.

Dans le second, on utilise une fonctionnalité avancée, les annotations, pour exposer une méthode comme un attribut et s'en servir comme tel. L'avantage dans ce cas est que la méthode peut embarquer de la logique, puisqu'il ne sá git pas d'un vrai attribut mais d'une fonction, elle pourrait convertir tout en majuscule ou minuscule de manière transparente:

Getter & Setter pour les fortiches avec logique dans le getter

```
class Chat:
    def __init__(self, value):
        self._nom = value

    @property
    def nom(self):
        return self._nom.upper

    @nom.setter
    def nom(self, value):
        if not value:
            raise ValueError("Le nom ne peut pas être vide.")
        self._nom = value

>>> c = Chat('Ragazza')
>>> c.nom
'RAGAZZA'
>>> c.nom = 'Gaufrette'
>>> c.nom
'GAUFRETTE'
```

Encapsulation et visibilité

ENCAPSULATION

C'est le regroupement des données et des méthodes qui opèrent sur ces données en une seule unité (classe). Cela permet de cacher les détails de mise en œuvre.

VISIBILITÉ

En Python, la visibilité des membres de la classe est déterminée par des conventions de nommage. Un nom commençant par un underscore (comme _privé) est traité comme "protégé", et un nom commençant par deux underscores (comme __privé) est traité comme "privé".

Dans d'autres langages, cette notion de visibilité est stricte et implique l'utilisation de mots clefs pour indiquer si un membre est public, privé ou protégé, mais en Python c'est une convention d'écriture: on peut décider de passer outre si l'on veut vraiment.

Cette visibilité s'applique aussi bien aux variables qu'aux méthodes, de fait la méthode `__init__` peut être considérée comme "privée" puisqu'elle débute par '__'.

Visibilité

```
class Voiture:
    def __init__(self):
        self.marque = "Toyota"  # Public
        self._secret = "12345"  # Protégé
        self.__code_privé = "abcd"  # Privé
```

Membre de classe

Les membres de classe (ou variables de classe) sont des attributs qui sont définis au niveau de la classe, et non au niveau de l'instance. Ils sont partagés par toutes les instances de la classe.

Membre de Classe

```
class Voiture:
    nombre_de_roues = 4  # Membre de classe
```

```
    def __init__(self, marque):
        self.marque = marque  # Attribut d'instance
```

Session 3

Héritage

L'héritage est l'un des piliers fondamentaux de la programmation orientée objet. Il permet à une classe (appelée sous-classe ou classe dérivée) d'hériter des attributs et des méthodes d'une autre classe (appelée classe parent ou classe de base). L'héritage vise à promouvoir la réutilisation du code et à établir une relation de type "est un" entre la sous-classe et la classe parent.

En Python, l'héritage est réalisé en passant la classe parent comme un paramètre lors de la définition de la sous-classe.

HÉRITAGE SIMPLE

Python supporte l'héritage simple où une sous-classe peut hériter d'une seule classe parent.

Héritage simple

```
class Animal:
    def __init__(self, nom):
        self.nom = nom

class Chien(Animal):
    pass

class Chat(Animal):
    pass

>>> animal1 = Chien("Rex")
>>> animal2 = Chat("Gaufrette")
>>>
```

Dans cet exemple, `Chien` et `Chat` sont des sous-classes de `Animal`, ils ont tous deux un nom.

HÉRITAGE MULTIPLE

Contrairement à de nombreux autres langages, Python supporte l'héritage multiple, où une sous-classe peut hériter de plusieurs classes parent.

Héritage multiple

```
class Animal:
    def __init__(self, nom):
        self.nom = nom

class Aquatique:
    def plouf(self):
        return "plouf !"

class Chien(Animal):
    pass

class Chat(Animal):
    pass

class Poisson(Animal, Aquatique):
    pass

>>> animal1 = Chien("Rex")
>>> animal2 = Chat("Gaufrette")
>>> animal3 = Poisson("Bulle")
>>> animal3.plouf()
"plouf !"
```

Dans cet exemple, `Chien` et `Chat` sont des sous-classes de `Animal`... mais `Poisson` est une sous-classe de `Animal` ET `Aquatique`.

En réalité, ce n'est pas un très bon découpage, c'était simplement pour avoir un exemple facile.

Le GROS du travail en POO est précisément se travail de découpe: regrouper dans des classes les propriétés communes des objets que l'on manipule, trouver ceux qui sont spécifiques et méritent leurs propres classes, tout en jonglant avec un niveau d'abstractin suffisant mais pas non plus démesuré:

Peut-être qu'il aurait été intéressant d'introduire une classe `Mammifère` ici, peut-être même qu'il aurai été intéressant de différencier le mon `Animal` du monde `Végétal` avec une classe parent `Organisme`. Est-ce qu'il faut remonter jusqu'à avoir une classe `Cellule`, une classe `Atome`, ...

Vous saisissez la difficulté de la POO: le système d'héritage est très puissant et permet de décrire les choses de manière précise, mais il faut jauger correctement le niveau de détail pour ne pas s'embourber dans 50 abstractions inutiles.

FONCTION SUPER

En Python, la fonction `super()` est utilisée pour appeler une méthode de la classe parent. Elle est souvent utilisée dans le constructeur (**init**) pour initialiser la partie parent de l'objet.

```
class Animal:
    def __init__(self, nom, espece):
        self.nom = nom
        self.espece = espece

class Chien(Animal):
    def __init__(self, nom, race):
        super().__init__(nom, "chien")
        self.race = race

>>> rex = Chien("Rex", "Berger Allemand")
```

On pourrait initialiser `self.nom` et `self.espece` dans le constructeur de `Chien`, mais alors on risquerai de passer à côté d'autres initialisations si nous n'avions pas écrit `Animal` nous même et que son constructeur faisait autre chose, ou encore si `Animal` venait à évoluer à l'avenir et que nous ne répliquions pas ses changements dans toutes les sous-classe.

La bonne pratique est d'appeler `super()` pour initialiser les champs du parent correctement.

REDÉFINITION DE MÉTHODES

Une sous-classe peut redéfinir une méthode héritée de la classe parent pour fournir une implémentation spécifique.

Redéfinition de méthodes

```
class Animal:
    def __init__(self, nom):
        self.nom = nom

    def communiquer(self):
        pass

class Aquatique:
    def nager(self):
        pass

class Chien(Animal):
    def communiquer(self):
        return f"{self.nom} dit Woof!"

class Chat(Animal):
    def communiquer(self):
        return f"{self.nom} dit Miaou!"

class Poisson(Animal, Aquatique):
    def communiquer(self):
        return f"{self.nom} dit Miaou!"

    def nager(self):
        return f"{self.nom} fait des brasses"
```

Dans l'exemple qui ouvre la question de l'héritage, la classe Animal fournit une méthode `communiquer()` qui ne fait rien (`pass`). Les classes Chien et Chat fournissent héritent chacunes de la classe Animal, mais redéfinissent la méthode `communiquer` avec une implémentation qui leur est propre. C'est ce que l'on appelle le `polymorphisme`, la faculté pour des objets de se comporter de façon différente en fonction de leur forme (poly-morphisme): tous les animaux communiquent, mais selon la spécialisation de l'animal, la communication prendra une forme différente.

Classes abstraites

Les classes abstraites sont des classes qui ne peuvent pas être instanciées directement: elles servent de base pour d'autres classes.

Une classe abstraite peut contenir des méthodes abstraites, c'est-à-dire des méthodes qui sont déclarées mais qui n'ont pas d'implémentation dans la classe abstraite elle-même. Les classes qui héritent de classes abstraites sont tenues de fournir une implémentation pour ces méthodes abstraites.

En Python, le module `abc` (pour "Abstract Base Class") fournit les mécanismes nécessaires pour définir des classes abstraites et des méthodes abstraites.

classe abstraite

```
from abc import ABC, abstractmethod

class Animal(ABC):

    def __init__(self, nom):
        self.nom = nom

    @abstractmethod
    def son(self):
        pass

class Chien(Animal):

    def son(self):
        return f"{self.nom} aboie"

class Chat(Animal):

    def son(self):
        return f"{self.nom} miaule"
```

Dans cet exemple :

- Animal est une classe abstraite car elle hérite de ABC et contient une méthode abstraite son.
- Vous ne pouvez pas créer une instance directe de Animal car c'est une classe abstraite.
- Les classes Chien et Chat héritent de Animal et fournissent une implémentation concrète de la méthode son.
- Vous pouvez créer des instances de Chien et Chat car elles ne sont pas abstraites et ont implémenté toutes les méthodes abstraites de leur classe parent.

POURQUOI UTILISER DES CLASSES ABSTRAITES ?

- Forcer l'implémentation : Elles permettent de définir un ensemble de méthodes que toutes les sous-classes doivent impérativement implémenter.
- Réutilisation du code : Elles peuvent également contenir du code concret que toutes les sous-classes peuvent utiliser, évitant ainsi la duplication de code.
- Conception : Elles fournissent un modèle clair pour la structure future des sous-classes, facilitant la conception orientée objet.

En résumé, les classes abstraites en Python permettent de définir un "contrat" que toutes les sous-classes doivent respecter, tout en offrant une structure et une réutilisation du code.

Classes concrètes

Ça va être rapide...

Dans notre exemple précédent, `Chien` et `Chat` sont des classes concrètes car elles implémentent le contrat d'une classe abstraite et peuvent être instanciés.

Session 4

Design patterns

Les design patterns sont des solutions standardisées pour résoudre des problèmes courants en programmation orientée objet. Ils offrent des modèles éprouvés pour concevoir des logiciels robustes et maintenables. Il y a de très nombreux design patterns, et des ouvrages consacrés, nous allons explorer six design patterns clés.

Il existe de nombreux design patterns, l'ouvrage de référence "Design Patterns: Elements of Reusable Object-Oriented Software" en recense 23, mais d'autres ont été créés par la suite et vous êtes libres de trouver vos propres patterns.

Dans ceux de l'ouvrage de référence, on dénombre 5 patterns "créationnels" qui concernent la création d'objects, 7 patterns "structurels" qui concernent la structure des objets, et 11 patterns "comportementaux" qui concernent le comportement des objets et leurs intéractions.

SINGLETON

Le pattern Singleton garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à celle-ci. Ce modèle est souvent utilisé pour gérer des ressources partagées.

singleton

```
class Singleton:
    _instance = None

    @classmethod
    def getInstance(cls):
        if cls._instance is None:
            cls._instance = cls()
        return cls._instance
```

singleton, methode alternative

```
singleton_object = None

class Singleton:
    pass

def get_singleton_object():
    global singleton_object
    if singleton_object is None:
        singleton_object = Singleton()
    return singleton_object
```

FACTORY

La Factory fournit une interface pour créer des objets dans une super-classe, mais permet aux sous-classes de modifier le type d'objets qui seront créés. Il est utile pour décentraliser la création d'objets.

factory

```
class Animal:
    def parler(self):
        pass

class Chien(Animal):
    def parler(self):
        return "Woof!"

class Chat(Animal):
    def parler(self):
        return "Miaou!"

def get_animal(type_animal):
    animaux = {"chien": Chien(), "chat": Chat()}
    return animaux[type_animal]
```

OBSERVER

Ce pattern établit une relation entre des objets de manière à ce que, lorsqu'un objet change d'état, tous ceux qui en dépendent en sont automatiquement notifiés.

observer

```
class Observer:
    def notifier(self, message):
        pass

class Sujet:
    def __init__(self):
        self._observateurs = []

    def ajouter(self, observateur):
        self._observateurs.append(observateur)

    def notifier_tous(self, message):
        for observateur in self._observateurs:
            observateur.notifier(message)
```

DECORATOR

Le Decorator ajoute dynamiquement des responsabilités supplémentaires à un objet. Il offre une alternative flexible à l'héritage pour étendre les fonctionnalités.

decorator

```
def mon_decorator(func):
    def wrapper():
        return "<b>" + func() + "</b>"
    return wrapper

@mon_decorator
def dire_bonjour():
    return "Bonjour!"
```

STRATEGY

Le pattern Strategy définit une famille d'algorithmes, les encapsule et les rend interchangeables. Il permet à l'algorithme de varier indépendamment des clients qui l'utilisent.

strategy

```
class Operation:
    def executer(self, a, b):
        pass

class Addition(Operation):
    def executer(self, a, b):
        return a + b

class Soustraction(Operation):
    def executer(self, a, b):
        return a - b

class Contexte:
    def __init__(self, strategie):
        self._strategie = strategie

    def executer_strategie(self, a, b):
        return self._strategie.executer(a, b)
```

VISITOR

Le pattern Visitor permet de séparer les algorithmes des objets sur lesquels ils opèrent. Il est particulièrement utile dans les situations où vous avez besoin de réaliser des opérations sur une série d'objets de différentes classes sans changer leur code.

visitor

```
# Chien et Chat implémentent la méthode accept
class Animal:
    def accept(self, visitor):
        pass

class Chien(Animal):
    def __init__(self, nom):
        self.nom = nom

    def accept(self, visitor):
        visitor.visiter_chien(self)

class Chat(Animal):
    def __init__(self, nom):
        self.nom = nom

    def accept(self, visitor):
        visitor.visiter_chat(self)

# voici concrètement une interface, ce pourrait être une classe abstraite
class VisiteurAnimal:
    def visiter_chien(self, chien):
        pass

    def visiter_chat(self, chat):
        pass

# on va écrire un visiteur pour saluer les animux
class VisiteurSalutation(VisiteurAnimal):
    def visiter_chien(self, chien):
        print(f"Bonjour, chien {chien.nom}!")

    def visiter_chat(self, chat):
        print(f"Bonjour, chat {chat.nom}!")


# Création des animaux
rex = Chien("Rex")
felix = Chat("Felix")

# Création et utilisation du visiteur
visiteur = VisiteurSalutation()
rex.accept(visiteur)
felix.accept(visiteur)

# ou encore:
for animal in [rex, felix]:
    animal.accept(visiteur)
```

ADAPTER

Le pattern Adapter, également connu sous le nom d'Adaptateur, permet à des interfaces incompatibles de travailler ensemble. Il agit comme un intermédiaire entre deux classes, convertissant l'interface d'une classe en une autre interface attendue par les clients.

adapter

```
class Chien:
    def aboyer(self):
        return "Woof!"

class Chat:
    def miauler(self):
```

```
        return "Miaou!"

class Adapter:
    def __init__(self, objet, adaptations):
        self.objet = objet
        self.__dict__.update(adaptations)

    def faire_du_bruit(self):
        pass

# Utilisation de l'Adapter
chien = Chien()
chat = Chat()

adaptateur_chien = Adapter(chien, {"faire_du_bruit": chien.aboyer})
adaptateur_chat = Adapter(chat, {"faire_du_bruit": chat.miauler})

print(adaptateur_chien.faire_du_bruit())  # Woof!
print(adaptateur_chat.faire_du_bruit())  # Miaou!
```

Évaluation

Projet : système de gestion de bibliothèque simplifié

Lisez-bien les instructions avant de commencer à travailler sur le projet !

OBJECTIF

Créer un système de gestion de bibliothèque simple qui permette de gérer les livres et les utilisateurs de la bibliothèque. Le système doit permettre aux utilisateurs d'ajouter et de retirer des livres de la bibliothèque, de rechercher des livres par titre, par auteur ou catégorie, d'enregistrer de nouveaux utilisateurs de la bibliothèque, d'emprunter et de retourner des livres, et de savoir qui détient un livre.

FONCTIONNALITÉS

1. Ajout et retrait de livres : Permettre à l'utilisateur d'ajouter ou de retirer des livres de la bibliothèque.
2. Recherche de livres : Rechercher des livres par titre, par auteur ou par catégorie.
3. Enregistrement des utilisateurs : Enregistrer de nouveaux utilisateurs de la bibliothèque.
4. Emprunt et retour de livres : Permettre aux utilisateurs d'emprunter et de retourner des livres (si un livre est pris, il n'est plus disponible).
5. Sauvegarde de l'état : Sauvegarder l'état de la bibliothèque dans un fichier au format JSON.

DESIGN PATTERNS À UTILISER

Vous devez idéalement implémenter ces 4 design patterns (et/ou d'autres) et les utiliser dans votre code, là où vous pensez que cela a du sens, les utilisations décrites ici sont de simples suggestions.

1. Singleton : Pour gérer une instance unique de la base de données de la bibliothèque.
2. Factory : Pour créer des objets Livre ou Utilisateur.
3. Observer : Pour notifier les utilisateurs lorsqu'un livre recherché devient disponible.
4. Strategy : Pour différentes stratégies de recherche de livres.

ÉVALUATION

1. Implémentation des patterns : Évaluer comment les étudiants implémentent et utilisent les design patterns mentionnés.
2. Cohérence du code : Vérifier la clarté et la logique du code, ainsi que son organisation, son découpage en classes.
3. Fonctionnalité : Tester si les fonctionnalités clés sont implémentées correctement et fonctionnent comme prévu.
4. Commentaires : Si vous voulez me partager une information, ajoutez des commentaires dans votre code pour m'expliquer ce que vous avez fait et pourquoi.

BONUS

Attention: Les bonus ne sont pas obligatoires, mais ils peuvent vous aider à améliorer votre note finale. Les bonus ne sont pas évalués si les fonctionnalités principales ne sont pas couvertes.

1. Utiliser SQLite pour stocker les données de la bibliothèque.
2. Exposer l'API de la bibliothèque via une API REST avec le framework Bottle.

RESSOURCES FOURNIES

1. Vous avez le droit d'utiliser le code fourni dans les exemples de ce cours et les exercices que nous avons fait ensemble.
2. Vous pouvez utiliser internet pour rechercher des informations sur les design patterns et les implémentations en Python.
3. Vous pouvez utiliser ChatGPT *raisonnablement* pour obtenir de l'aide sur les design patterns et les implémentations en Python.

Attention: Si vous vous inspirez de code trouvé sur internet ou généré par ChatGPT, faites bien attention parce que je vais évaluer votre code et votre compréhension du code, et je reconnais assez facilement le code généré par ChatGPT parce qu'il est presque systématiquement incorrect ou de mauvaise qualité. Inspirez-vous, mais ne copiez-collez pas sans comprendre ce que vous faites.

MODALITÉ DE RENDU

1. Vous devez créer une archive (.zip, .tar ou .tgz) contenant votre code et un fichier README.md de quelques lignes qui explique comment utiliser votre projet.
2. Vous devez envoyer votre archive par email à l'adresse `gilles.chehade@mail-formateur.net` avant la fin de la session.
3. Assurez-vous que j'ai bien reçu votre e-mail (demandez-moi !) et que j'ai pu télécharger votre archive.

Je ferais l'évaluation des différents rendus assez rapidement, et je vous enverrai un e-mail avec mes commentaires.