Programmation en Python

Dimitri Merejkowsky

oct. 10, 2020

Table des matières:

1 Chapitre 1 - Introduction 3

2 Chapitre 2 - Premiers pas 5

3 Chapitre 3 - Instructions, expressions, variables et types 11

4 Chapitre 4 - Contrôle de flux 19

5 Chapitre 5 - Introduction aux fonctions 25

6 Chapitre 6 - Listes 31

7 Chapitre 7 - None et pass 37

8 Chapitre 8 - Dictionnaires 41

9 Chapitre 9 - Tuples 45

10 Chapitre 10 - Introduction aux classes 49

11 Chapitre 11 - Introduction aux modules 55

12 Chapitre 12 - Couplage et composition 59

13 Chapitre 13 - Bibliothèques 63

14 Chapitre 14 - Données binaires et fichiers 73

15 Chapitre 15 - L'interpréteur interactif 83

16 Chapitre 16 - Héritage 87

17 Chapitre 17 - Décorateurs 93

18 Chapitre 18 - Exceptions 99

19 Chapitre 19 - Objets 105

Ce cours est principalement destiné aux élève de l'École du Logiciel Libre, où je donne des cours sur Python depuis 2018. mais il peut être utile à toute personne qui voudrait découvrir la programmation.

Il est aussi disponible en pdf à l'adresse : https://dmerej.info/python/cours-python.pdf

J'espère qu'il vous plaira ! Sinon, vous pouvez vous rendre sur ma page de contact et me faire part de vos remarques.

Enfin, notez que ce cours est placé sous licence CC BY 4.0. Bonne lecture !

CHAPITRE 1

Chapitre 1 - Introduction

1.1 Objet de ce cours

Apprendre la programmation en partant de rien, en utilisant Python et la ligne de commande

1.1.1 Pourquoi Python ?

- Excellent langage pour débuter - il a d'ailleurs été créé avec cet objectif en tête
- Mon langage préféré
- Facile à installer quelque soit le système d'exploitation (Linux, macOS, Windows)

1.1.2 Pourquoi la ligne de commande ?

C'est un interface universelle. Pour faire des programmes en lignes de commande, on n'a juste besoin de manipuler du *texte*, et c'est le plus simple au début.

1.2 Présentation du langage Python

1.2.1 Utilisation de Python

- Aussi appelé « langage de script », *glue language*
- Bon partout, excellent nulle part
- Exemples d'utilisation :
  - Sciences (physique, chimie, linguistique ...)
  - Animation (Pixar, Disney ...)
  - Sites web (journaux, youtube, ...)
  - Ligne de commande
  - ...

1.2.2 Petit détour : version d'un programme

- Comme les versions d'un document
- Si le nombre est plus grand, c'est plus récent
- Souvent en plusieurs morceaux : *1.3, 1.4, 3.2.5*. etc
- **Plus l'écart est grand, plus le programme a changé.**
    - *3.2.5* -> *3.2.6* : pas grand-chose
    - *1.5.1* -> *4.3* : beaucoup de changements
- On omet souvent le reste des numéros quand c'est pas nécessaire

1.2.3 Historique

- Créé par Guido van Rossum. Conçu à la base pour l'enseignement.
- Le nom vient des Monty Python (si, si)
- Python 1 : Sortie en 1991
- Python 2 : en 2000
- Python 3 : en 2008

1.2.4 Le grand schisme

La plupart des langages continuent à être compatibles d'une version à l'autre.

Ce n'est pas le cas pour Python3, et ça a causé beaucoup de confusion et de débats.

Heureusement, 10 ans plus tard, la situation s'est arrangée, et Python2 cessera d'être maintenu le premier janvier 2020.

1.2.5 Python3

Ce cours fonctionne donc uniquement avec Python3.

N'utilisez *pas* Python2, sinon certaines choses expliquées ici ne marcheront pas :/

CHAPITRE 2

Chapitre 2 - Premiers pas

2.1 La ligne de commande

2.1.1 Pourquoi la ligne de commande ?

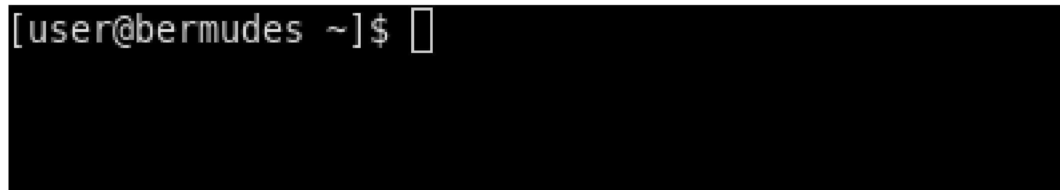
- Très puissant
- Ancien, mais toujours d'actualité
- Indispensable dans de nombreux cas
- Écrire des programmes qui marche dans la ligne de commande est (relativement) simple
- Possibilités infines, même si on ne fait que manipuler du texte

2.1.2 Les bases

Les lignes de commandes sont à taper dans un *terminal*.

- Sur linux, vous en avez probablement un insallé par défaut.
- Sur macOS, il se trouve dans *Applications/Utilitaires*
- Sur Windows, il se lance en tapant `cmd.exe`

Dans tous les cas, cela ressemble à ceci :



Notez le petit bout de texte avant le curseur : on appelle ça une *invite* de commande (`prompt` en anglais).

Pour lancer une commande, on tape son nom, suivi parfois d'un certain nombre de mots séparés par des espaces, puis on appuie sur entrée.

Par exemple, on peut utiliser `ls` (ou `dir`) sous Windows pour lister le contenu du répertoire courant :

```
[user@bermudes ~]$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
[user@bermudes ~]$
```

Et on peut utiliser `cd` suivi du nom d'un répertoire pour changer de répertoire courrant :

```
[user@bermudes ~]$ cd Documents/
[user@bermudes Documents]$
```

Notez que l'invite de commande a changé.

2.2 Installation

Pour suivre ce cours, il vous faudra installer deux composants essentiels :
- L'interpréteur Python
- Un éditeur de texte

2.2.1 Installation de Python

Linux

Il y a toutes les chances que Python soit déjà installé sur votre distribution. Pour vous en assurer, tapez :

```
python3 --version
```

Sinon, installez le paquet correspondant à Python3 (quelque chose comme `sudo apt install python3``

macOS

Python3 est disponible dans homebrew

```
brew install python
```

Windows

Python3 est disponible dans Microsoft store.

Sinon, vous pouvez aussi télécharger l'installeur depuis https://python.org

Veillez à cocher la case « Ajouter Python au PATH »

2.2.2 Installation d'un éditeur de texte

Je vous conseille pour commencer d'utiliser un éditeur de texte basique. Vous n'avez pas besoin d'un IDE, surtout si vous débutez

- Linux ; `gedit`, `kate`, ...
- macOS : `CotEditor`
- Windows : `Notepad++`

J'insiste sur **simple**. Inutile d'installer un IDE pour le moment.

2.2.3 Configuration de l'éditeur de texte

Prenez le tempe de configurer votre éditeur de texte de la façon suivante :

- Police de caractères à chasse fixe
- Indentation de *4 espaces*
- Remplacer les tabulations par des espaces
- Activer la coloration syntaxique

Cela vous évitera des soucis plus tard.

2.3 Code source

2.3.1 Définition

Aussi appelé : « code source », ou « source ».

L'essence du logiciel libre :)

2.3.2 Notre premier fichier source

Insérez le code suivant dans votre éditeur de texte

```
print("Bonjour, monde")
# affiche: Bnojour, monde
```

Oui, juste ces deux lignes.

Sauvegardez dans un fichier *bonjour.py* dans *Documents/e2l/python* par exemple

2.3.3 Lancer du code en ligne de commande

Lancez une invite de commandes et tapez quelque chose comme :

```
cd Documents/e2l/python/
python3 bonjour.py
```

(Utilisez *python* sous Windows)

Si tout se passe bien, vous devrez voir s'afficher ceci :

```
Bonjour, monde
```

Vous savez maintenant comment exécuter du code Python dans n'importe quel fichier :

1. Écrire le code dans un fichier

2. Se rendre dans le répertoire contenant le fichier et lancer *python3* (ou *python*) suivi du nom du fichier.

2.3.4 print()

Revenons sur ce qu'il s'est passé : nous avions le mot *print* avec des parenthèses et quelque chose à l'intérieur des parenthèses, et ceci a provoqué l'affichage du contenu des parenthèses dans le terminal.

2.3.5 Commentaires

La deuxième ligne, quant à elle, a été complètement ignorée par l'interpréteur parce qu'elle commençait par un `#`. Il s'agit d'un *commentaire*, et il sert principalement aux humains qui lisent le code.

2.3.6 Note à propos des examples

La plupart des examples de ce cours contiendront un ou plusieurs appels à *print* afin d'afficher les opérations que l'interpréteur a effectué.

Pour lire ce cours de manière efficace, il est conseillé de lancer les examples de code sur votre machine, et de vérifier si ce qui est affiché sur votre machine correspond à ce qui est écrit dans le cours.

Il est aussi recommandé de **ne pas** copier/coller le code.

À la place, prenez le temps de retaper le code dans votre éditeur.

Plusieurs raisons à cela :
- Recopier le code vous aidera à vous souvenir de la syntaxe
- Si vous faites des erreurs, Python vous préviendra et vous découvrirer les erreurs courantes
- Il est possible que des erreurs subsistent dans ce cours, et procéder ainsi nous permettra de les corriger.

2.4 Maths simples

2.4.1 Opérations avec des entiers

On peut utiliser `+, *, -` avec des entiers :

```
print(1 + 2)
# affiche: 3

print(6 - 3)
# affiche: 3

print(2 * 4)    # une étoile pour la multiplication
# affiche: 8

print(3 ** 2)   # deux étoiles pour l'opération 'puissance'
# affiche: 9
```

2.4.2 Opérations avec des flottants

C'est le `.` qui fait le flottant

```
print(0.5 + 0.2)
# affiche: 0.7
print(10 / 3)
3.3333333333333335
```

Note : Les flottants sont imprécis, ce qui explique le *5* à la fin de l'affichage de la division de 10 par 3

2.4.3 Division entières et modulo

14 divisé par 3 font 4 avec un reste de 2.

On peut récupérer le quotient avec *//* et le reste avec `%`.

```
print(14 // 3)
# affiche: 4

print(14 % 3)
# affiche: 2
```

Avertissement : Le `%` n'a rien à voir avec un pourcentage !

2.4.4 Priorité des opérations

Comme en maths, la multiplication est prioritaire sur les autres opérations :

```
print(1 + 2 * 3)
# affiche: 7
```

et on peut utiliser des parenthèses pour grouper les opérations :

```
print((1 + 2) * 3)
# affiche: 9
```

CHAPITRE 3

Chapitre 3 - Instructions, expressions, variables et types

3.1 Expressions, instructions, et variables

3.1.1 Instructions

Pour l'instant, dans tous les exemples de code, chaque ligne qu'on a écrit contenait une *instruction*.

Un instruction a un effet sur le programe dans lequel elle est présented

Par exemple, l'instruction `print("bonjour")` affiche « bonjour » dans le terminal. On dit que l'instruction est *exécutée*.

En règle générale, les instructions sont éxécutées une par une, de haut en bas.

3.1.2 Expressions

Les instructions peuvent contenir des *expressions*.

Un expression est toujours *évaluée* pour retourner une *valeur*

Par exemple, `1 + 2` es une expression qui renvoie la valeur `3` une fois évaluée.

Elle est constituée de 3 éléments :

- Le *littéral* 1
- L'"*opérateur* +
- Le *littéral* 2

Pour évaluer une expression, Python remplace les littéraux par leur valeur, puis calcule la valeur finale en utilisant les opérateurs.

Notez que les expressions peuvent être imbriquées

```
1 + (2 * 3)
```

À droite du plus, on a une expression `2 + 3`. Quand Python évaluera l'expression, il verra d'abord le littéral `1` et le `+`, puis il évaluera l'expression à droite (`2*3 = 6`), et finalement l'expression en entier (`1 + 6 = 7`).

Notez que si vous écrivez une ligne contenant une *expression*, elle est évaluée mais rien n'est éxécuté :

```
print("Bonjour")
40 + 2
print("Au revoir")
# Affiche: 'Bonjour' puis 'Au revoir'
```

Ici l'expression `40 + 2` a été évaluée, mais Python n'a rien fait avec le résulat, il est simplement passé à l'instruction suivante.

3.1.3 Variables et valeurs

On peut associer des *variables* à des *valeurs* en les plaçant de part et d'autre du signe = : on appelle cette opération une *assignation* :

```
a = 2
```

Ici on assigne l'entier 2 à la variable `a`.

Notez qu'une assignation *n'est pas* une expression, c'est une *instruction*.

Si plus tard dans le code, on utilise le nom de la variable, tout se passera comme si nom de la variable avait été remplacé par sa valeur :

```
a = 2
print(a)
# affiche: 2
```

3.1.4 Variables et expressions

En fait, on peut assigner n'importe qu'elle *expression* à une variable, et pas simplement des littéraux :

```
a = 1 + 2
print(a)
# affiche: 3
```

3.1.5 Variables et expressions contenant d'autres variables

Les expressions peuvent également contenir des variables.

Quand Python évalue une expression qui contient des noms de variables, il remplace celles-ci par leur valeur :

```
a = 1
print(a + 2)    # ici a a été remplacé par 1
# affiche: 3
```

Notez que la valeur de variable *a* n'a pas changé :

```
a = 1
print(a + 2)
# affiche: 3
```

(suite sur la page suivante)

(suite de la page précédente)

```
print(a)
# affiche: 1
```

Autres exemples :

```
x = 1
y = 2
print(x+y)  # ici x a été remplacé par 1 et y par 2
# affiche: 3
```

3.1.6 Changer la valeur d'une variable

On peut aussi *changer* la valeur d'une variable en assignant une nouvelle valeur à celle-ci :

```
a = 2
print(a)
a = 3
print(a)
# affiche: 2, puis 3
```

3.1.7 Combiner opération et assignation

La notation += permet de combiner addition et assignation : les deux exemples ci-dessous sont équivalents :

```
x = 3
x = x + 1
```

```
x = 3
x += 1
```

Cela fonctionne aussi pour `-=`, `/=` etc.

3.1.8 Nom des variables

Ci-dessus j'ai utilisé des noms de variables à une lettre, mais il est préférable d'avoir des noms longs et descriptifs

Aussi, la convention est de :

— Les écrire en minuscules

— De séparer les mots par des tirets bas (*underscore*) :

```
score = 42
âge_moyen = 22
```

Notez que certains noms ne peuvent être utilisés comme nom de variables. On les appelle des *mots-clés*. La liste est disponible ici : https://docs.python.org/fr/3/reference/lexical_analysis.html#keywords

3.2 Chaînes de caractères

Les chaînes de caractères, aussi appelées « string », permettent de représenter du texte. On a utilisé une string pour afficher « bonjour monde » dans le chapitre précédent.

On écrit toujours les strings entre guillemets.

soit avec des doubles guillemets :

```
print("Bonjour monde!")
# affiche: Bonjour monde!
```

soit avec des guillemets simples :

```
print("Bonjour monde!")
# affiche:  Bonjour monde!
```

3.2.1 Double et simple quotes

On peut mettre des simples quotes dans des double quotes et vice-versa :

```
print("Il a dit: 'bonjour' ce matin.")
# affiche: Il a dit: 'bonjour' ce matin.

print('Il a dit: "bonjour" ce matin')
# affiche: Il a dit: "bonjour" ce matin
```

3.2.2 Échappement

On peut aussi *échapper* des caractères avec la barre oblique inversée \\ - backslash.

```
print('Il a dit: "bonjour". C\'est sympa!')
# affiche: Il a dit: "bonjour". C'est sympa!
```

3.2.3 Concaténation

On peut construire de longues chaînes de caractères en en concaténatant de plus petites, avec l'opérateur + :

```
name = "John"
message = "Bonjour " + name + " !"
print(message)
# affiche: Bonjour John !
```

3.2.4 Répétition

On peut construire une longue string en répétant la même petite string plusieurs fois avec l'opérateur * :

```
message = "Na" * 3
print(message)
# affiche: NaNaNa
```

3.3 Types

On a vu qu'on pouvait utiliser + **à la fois** pour additionner des nombres ou concaténer des strings. Mais on ne peut pas utiliser + avec une string d'un côté et un entier de l'autre :

```
a = 42
b = 4
c = a + b  # ok

salutation = "bonjour, "
prénom = "Bob"
salutation + prénom  # ok

résultat = a + prénom
# affiche:
# TypeError: can only concatenate str (not "int") to str
```

Ceci est notre premier message d'erreur : si l'interpréteur est incapable d'éxécuter une instruction, il affiche un message d'erreur et s'interrompt immédiatement.

3.3.1 Conversions

Pour résoudre le problème ci-dessus, il faut effectuer une *conversion de types :*

Entier vers string

On peut convertir un entier en string en utilisant le mot `str` et des parenthèses autour de l'expression :

```
x = 40
y = 2
message = "La réponse est: " + str(x + y)
print(message)
# affiche: La réponse est 42
```

String vers nombres

Inversement, on peut convertir un string en entrier en utilisant le mot `int` et des parenthèses :

```
quarante_en_chiffres = "40"
réponse = int(quarante_en_chiffres) + 2
print(réponse)
# affiche: 42
```

Pour convertir une string en flottant, on peut utiliser `float()` :

```
taille_sur_le_passeport = "1.62"
taille_en_mètres = float(taille_sur_le_passeport)
```

3.4 Booléens

On appelle *booléenne* une valeur qui peut être soit vraie, soit fausse.

En Python, les littéraux `True` et `False` représentent respectivement les valeurs vraies et fausses.

(Notez qu'ils commencent par une majuscule)

3.4.1 Comparaisons

Certaines expressions renvoient des booléens, c'est à dire soit la valeur `True`, soit la valeur `False`

==	égal
!=	différent
>	strictement supérieur
>=	supérieur ou égal
<	strictement inférieur
<=	inférieur ou égal

Par exemple :

```
a = 2
b = 3
print(a > b)
# affiche: False

print(2 + 2 == 4)
# affiche: True
```

Avertissement : Ne pas confondre : == pour la comparaison et = pour l'assignation

3.4.2 Autres opérations booléennes

not	négation
and	et
or	ou

Exemples :

```
a = not True
print(a)
# affiche `False`

il_pleut = True
j_ai_un_parapluie = False
print(il_pleut and j_ai_un_parapluie)
# afiche: False

je_suis_mouillé = il_pleut and not j_ai_un_parapluie
print(je_suis_mouillé)
# afiche: True
```

CHAPITRE 4

Chapitre 4 - Contrôle de flux

4.1 Contrôle de flux

Pour l'instant, toutes les instructions que nous avons écrites ont été exécutée une par une et dans l'ordre d'apparition dans le code source.

De plus, chaque ligne était constituée d'une unique expression.

Modifier l'ordre d'exécution de ces instructions s'appelle le *contrôle de flux*, et c'est l'essence de la programmation !

4.1.1 if

On peut utiliser le mot-clé `if` pour autoriser ou empécher l'éxcution des instructions suivantes :

```
a = 3
b = 4
if a == b:
    print("a et b sont égaux")
# n'affiche rien
```

La 4ème ligne n'est pas éxécutée parce la condition est fausse.

Notes :

- il y a le caractère `:` (deux points) à la fin de la ligne
- le code en-dessous du `if` commence par 4 espaces : on appelle cela une *indentation*

Notez qu'on a utilisé une **expression** après le if. On ne peut pas utiliser d'instruction après un `if` en Python.

Autrement dit, ce code ne fonctionne pas :

```
if a = 3:
    print("a égale 3")
# affiche: SyntaxError
```

On parle aussi de « bloc » si plusieurs lignes sont identées :

```
a = 3
b = 4
if a == b:
    # début du bloc
    print("a et b sont égaux")
    c = 2 * a
    # fin du bloc
```

Notez qu'on reprend l'ordre habituel d'éxécution des instructions s'il y a un bloc identé dans l'autre sens après le `if` :

```
a = 3
b = 4
if a == b:
    # début du bloc
    print("a et b sont égaux")
    c = 2 * a
    # fin du bloc
# on est après le bloc if
print("on continue")
```

Ce code n'affiche pas « a et b sont égaux », mais affiche bien « on continue ».

if / else

On peut utiliser le mot-clé `else` après un condition en `if` pour éxécutér un bloc si la condition est fausse :

```
a = 3
b = 4
if a == b:
    print("a et b sont égaux")
else:
    print("a et b sont différent")
# affiche: a et b sont différents
```

if / elif

On peut utiliser `if`, `elif` et `else` pour enchaîner plusieurs conditions :

```
age = 23
if age < 10:
    print("âge inférieur à dix")
elif 10 <= age < 20:
    print("âge entre 10 et 20")
elif 20 <= age < 40:
    print("âge entre 20 et 40")
else:
    print("âge supérieur à 40")
# affiche: âge entre 20 et 40
```

while

On peut utiliser le mot-clé `while` pour répéter un bloc tant qu'une condition est vraie :

```
i = 0
while i < 3:
    print(i)
    i = i + 1
```

```
0
1
2
```

Notez que la variable `i` passe par plusieurs valeurs différentes.

Boucle infinie

On parle de *boucle infinie* si la condition ne devient jamais fausse :

```
while True:
    print("spam!")
```

Dans ce cas, il faut appuyer sur `CTRL-C` pour interrompre le programme.

Combiner while, if, et break

On peut « sortir » de la boucle `while` avec le mot-clé `break` :

```
i = 0
while True:
    i = i + 1
    print(i)
    if i > 3:
        break
```

```
1
2
3
4
```

4.2 Falsy et truthy

4.2.1 Expressions après un if

Jusqu'ici les expressions qu'on a utilisées donnait un booléens une fois évaluées, mais un expression apès un `if` peut être d'un autre type.

Par exemple, un entier :

```
x = 0
if x:
    print("x n'est pas nul")
```

(suite sur la page suivante)

(suite de la page précédente)

```
else:
    print("x est nul")

# affiche: x est nul
```

On dit que `0` est `Falsy`, parce qu'après un `if`, il se comporte comme une expression qui vaudrait False.

Réciproquement, tous les entiers sauf 0 sont `Truthy`, parce qu'ils se comportent comme une expression qui vaudrait True :

```
y = 6
if y:
    print("y n'est pas nul")
else:
    print("y est nul")

# affiche: y n'est pas nul
```

On retrouve ce principe avec les chaînes de caractères :

```
message = ""
if message:
    print("le message n'est pas vide")
else:
    print("le message est vide")
# affiche: le message est vide
```

Le chaînes vides sont falsy, les autres sont truthy.

4.2.2 Expressions quelconques

En fait, on peut utiliser tous les opérateurs booléens avec des expressions quelconques :

```
message = ""
if not message:
    print("le message est vide")
# affiche : le message est vide


score = 42
if message and score:
    print("le message et le score sont truthy")
# affiche : le message et le score sont truthy
```

4.3 Exercice

Ceci étant le premier exercice du cours, il mérite quelques explications.

Chaque exercice comporte une suite de consignes, et quelques indices.

À vous ensuite d'écrire le code qui correspond aux consignes.

4.3.1 Consignes

Il vous faut implémenter le programme suivant :

1. Tirer un nombre au hasard entre 1 et 100 (appelons-le le `nombre_secret`)
2. Démarrer une boucle
3. À chaque étape :
   - Afficher « Devine le nombre secret »
   - Bloquer le programme jusqu'à ce que l'utilisateur entre un nombre et appuie sur entrée (appelons-le `entrée_utilisateur`)
   - Si l'entrée utilisateur est plus grande que le nombre secret, afficher « trop grand ».
   - Si l'entrée utilisateur est plus petite que le nombre secret, afficher « trop petit »
   - Si l'entrée utilisateur est égale au nombre secret, afficher « gagné ! » et quitter la boucle.

4.3.2 Indices

Lire une entrée utilisateur

Pour bloquer le programme et lire une entrée utilisateur, vous pouvez utiliser la ligne suivante :

```
entrée_utilisateur = input()
```

Cette instruction va :

- interrompre le script
- lire ce que l'utilisateur tape jusqu'à ce qu'il tape « entrée ».
- et assigner la valeur correspondante à la variable `entrée_utilisateur`.

Tirer un nombre au hasard

Pour tirer un nombre au hasard entre 1 et 100, vous pouvez utiliser les deux lignes suivantes :

```
import random
nombre_secret = random.randint(0, 100)
```

À la fin de ces deux instructions, une valeur entre 1 et 100 tirée au hasard sera assignée à la variable *nombre_secret*.

4.3.3 Squelette

Pour commencer, copier le code suivant dans un fichier (par exemple, *devine-nombre.py*)

```
import random

nombre_secret = random.randint(0, 100)

print("devine le nombre auquel je pense entre 0 et 100")
entrée_utilisateur = int(input())

while True:
    if entrée_utilisateur == nombre_secret:
        print("bravo")
        break
    else:
```

(suite sur la page suivante)

(suite de la page précédente)

```
        print("mauvaise réponse")
        entrée_utilisateur = int(input())
```

Si vous lancez *python3 devine-nombre.py* vous constaterez que le jeu est très difficile parce que le programme affiche simplement « mauvaise réponse » en boucle jusqu'à ce que l'utilisateur devine le nombre secret.

Le but est de modifier le code pour implémenter correctement le programme, et ainsi rendre le jeu jouable :)

4.3.4 Obtenir de l'aide

Si vous lisez ceci en dehors d'un cours et que vous êtes coincés, n'hésitez pas à me contacter via mon mail e2l.

Bon courage !

CHAPITRE 5

Chapitre 5 - Introduction aux fonctions

5.1 Fonctions

5.1.1 Fonctions sans argument

Définition :

```
def dire_bonjour():
    print("Bonjour")
```

- avec le mot-clé *def*
- avec un *:* à la fin et un *bloc indenté* (appelé *le corps de la fonction*).

Appel :

```
dire_bonjour()
```

- avec le nom de la fonction et des parenthèses

Exemple complet :

```
def dire_bonjour():
    print("Bonjour")

dire_bonjour()
# Affiche: bonjour'
```

- Le *nom* de la fonction est le mot utilisé pour la définir et l'appeler
- Le *corps* de la fonction est le bloc après le *def()*
- Quand on *définit* une fonction, on associe un nom avec un corps.
- Quand on *appelle* une fonction, on exécute le corps.

5.1.2 Le pouvoir des fonctions

Ici on vient de créer une nouvelle fonctionnalité à Python. Avant qu'on définisse la fonction *dire_bonjour()*, il ne savait pas dire bonjour, il savait uniquement afficher des messages à l'écran.

On dit qu'on a *créé une abstraction*. Et c'est une technique extrêmement utile en programmation.

5.1.3 Fonction avec un argument

Définition : avec l'argument à l'intérieur des parenthèses :

```
def dire_bonjour(prénom):
    print("Bonjour " + prénom)
```

Appel : en passant une variable ou une valeur dans les parenthèses :

```
dire_bonjour("Germaine")
```

Pour évaluer une expression qui contient l'appel a une fonction, on :
- assigne le contenu des parenthèses aux arguments de la fonction
- puis on éxécute les instructions dans le corps de la fonction

```
# Ceci:
dire_bonjour("Dimitri")

# Est équivalent à cela:
prénom_de_dimitri = "Dimitri"
print("Bonjour " + prénom_de_dimitri)

# Lui-même équivalent à:
print("Bonjour " + "Dimitri")
```

Exemple complet :

```
def dire_bonjour(prénom):
    print("Bonjour " + prénom)
dire_bonjour("Germaine")
# affiche: Bonjour Germaine

prénom_de_charlotte = "Charlotte"
dire_bonjour(prénom_de_charlotte)
# affiche: Bonjour Charlotte
```

5.2 Portée des variables

Les arguments d'une fonction n'existent que dans le corps de celle-ci :

```
def dire_bonjour(prénom):
    print("Bonjour " + prénom)

dire_bonjour("Dimitri") # Ok
print(prénom)   # Erreur
```

Les variables en dehors des fonctions sont disponibles partout :

```
salutation = "Bonjour "

def dire_bonjour(prénom):
    print(salutation + prénom)

dire_bonjour("Dimitri")
```

Une variable peut avoir en « cacher » une autre si elle a une portée différente :

```
prénom = "Dimitri"   # portée: dans tout le programme

def dire_bonjour():
    prénom = Max  # portée: uniquement dans
                  # le corps dire_bonjour

    print("Bonjour " + prénom)
    # affiche: Bonjour Max

print("Bonjour " + prénom) j
# affiche: Bonjour Dimitri
```

5.3 Valeur de retour d'une fonction

On peut également appeler une fonction dans une expression à droite d'une assignation de variable.

Dans ce cas, la valeur de l'expression est obtenue en exécutant le corps de la fonction jusqu'à rencontrer l'instruction *return* et en évaluant l'expression à droite du return.

Par exemple :

```
def retourne_42():
    return 42

x = retourne_42()
print(x)
# Affiche: 42
```

Ici, on peut dire que *42* est le *résultat* de l'appel de la fonction *retourne_42()*.

On peut utiliser *if* avec plusieurs *return* pour changer le résultat d'une fonction :

```
def peut_conduire(âge):
    if âge < 18:
        return False
    else:
        return True

x = peut_conduire(16)
print(x)
# Affiche: False
```

5.4 Fonctions à plusieurs arguments

On peut mettre autant d'arguments qu'on veut, séparés par des virgules :

```
def soustraction(x, y):
    résultat = x - y
    return résultat

résultat = soustraction(5, 4)
print(résultat)
# affiche: 1
```

5.4.1 Arguments nommés

En Python, on peut aussi utiliser le *nom* des arguments au lieu de leur position :

```
def dire_bonjour(prénom):
    print("Bonjour " + prénom)

dire_bonjour(prénom="Gertrude")
# Affiche: Bonjour Gertrude

résultat = soustraction(y=4, x=5)
print(résultat)
# affiche: 1
```

5.5 Arguments par défaut

On peut aussi mettre des valeurs par défaut :

```
def dire_bonjour(prénom, enthousiaste=False):
    message = "Bonjour " + prénom
    if enthousiaste:
        message += "!"
    print(message)
```

Appel :

```
dire_bonjour("Thomas")
# affiche: Bonjour Thomas

dire_bonjour("Thomas", enthousiaste=True)
# affiche: Bonjour Thomas!

dire_bonjour("Thomas", enthousiaste=False)
# affiche: Bonjour Thomas
```

5.6 Fonctions natives

Fonctions qui sont toujours présentes dans l'interpréteur. On en a déjà vu quelques unes :
- `print`, `input` : écrire et lire sur la ligne de commande
- `str`, `int` : convertir des entiers en strings et vice-versa

Il y en a tout un tas !

La liste ici : https://docs.python.org/fr/3/library/functions.html

5.6.1 Retour sur print

On peut passer autant d'arguments qu'on veut à `print` et :
- Il les sépare par des espaces
- Ajoute un retour à la ligne à la fin :

```
prénom = "Charlotte"
print("Bonjour", prénom)
print("Ça va ?")
```

```
Bonjour Charlotte
Ça va ?
```

On peut demander à *print* de changer son séparateur :

```
a = "chauve"
b = "souris"
print(a, b, sep="-")
```

```
chauve-souris
```

Ou de changer le caractère de fin :

```
print("Ceci tient", end="")
print("sur une seule ligne")
```

```
Ceci tient sur une seule ligne
```

5.7 Exercice

Le but de l'exercice est d'afficher un sapin de largeur arbitraire dans la console, comme ceci :

```
    #
   ###
  #####
 #######
#########
    #
    #
```

Le sapin est composé d'une suite de lignes, chacune des lignes étant constituée uniquement de dièses.

Il y a deux parties au sapin : les feuilles qui forment un triangle de largeur 1 tout en haut jusqu'à une ligne de largeur 9 tout en bas, et un pied constitué de deux dièses superposés

5.7.1 Indices

Pour construire une chaîne de caractères constituée uniquement de dièses vous pouvez utiliser l'expression suivante :

```
cinq_diéses = "#" * 5
print(cinq_diéses)
```

```
#####
```

5.7.2 Consignes

Partir du code suivant :

```
largeur = 9


def affiche_ligne(début, fin):
    blancs_au_début = " " * début
    largeur_ligne = fin - début
    dièses = "#" * largeur_ligne
    print(blancs_au_début + dièses)


def affiche_feuilles():
    affiche_ligne(5, 6)
    affiche_ligne(4, 7)
    # à compléter


def affiche_pied():
    affiche_ligne(5, 6)
    # à compléter


def affiche_sapin():
    affiche_feuilles()
    affiche_pied()


affiche_sapin()
```

- Compléter le code pour afficher le sapin en entier
- Remplacer tous les litéraux (5, 6, 4, 7. . .) par des expressions utilisant la variable *largeur*
- Demander à l'utilisateur la largeur du sapin en début de programme au lieu d'utiliser la valeur litérale *9* - vous pouvez faire l'hypothèse que la largeur est toujours un nombre impair.

CHAPITRE 6

Chapitre 6 - Listes

6.1 Définition

Une liste est une *suite ordonnée* d'éléments.

6.2 Créer une liste

Avec des crochets : `[`, `]`, et les éléments séparés par des virgules :

```
liste_vide = []
trois_entiers = [1, 2, 3]
```

6.3 Listes hétérogènes

On peut mettre des types différents dans la même liste :

```
ma_liste = [True, 2, "trois"]
```

On peut aussi mettre des listes dans des listes :

```
liste_de_listes = [[1, 2], ["Germaine", "Gertrude"]]
```

6.4 Connaître la taille d'une liste

Avec `len()` - encore une fonction native :

```
liste_vide = []
taille = len(liste_vide)
print(taille)
# affiche:  0

trois_entiers = [1, 2, 3]
taille = len(trois_entiers)
print(taille)
# affiche:  3
```

6.5 Concaténation de listes

Avec + :

```
prénoms = ["Alice", "Bob"]
prénoms += ["Charlie", "Eve"]
print(prénoms)
# affiche: ['Alice', 'Bob', "Charlie", 'Eve']
```

On ne peut concaténer des listes que avec d'autres listes :

```
scores = [1, 2, 3]
scores += 4
# erreur

scores += [4]
print(scores)
# affiche: [1,2,3,4]
```

6.6 Test d'appartenance

Avec `in` :

```
prénoms = ["Alice", "Bob"]
print("Alice" in prénoms)
# affiche: True

prénoms = ["Alice", "Bob"]
print("Charlie" in prénoms)
# affiche: False
```

6.7 Itérer sur les éléments d'une liste

Avec les mots-clés `for` et `` in`` :

```
prénoms = ["Alice", "Bob", "Charlie"]
for prénom in prénoms:
        # Chaque élément de la liste est assigné tour à tour
    # à la variable 'prénom"
    print("Bonjour", prénom)
```

```
Bonjour Alice
Bonjour Bob
Bonjour Charlie
```

6.8 Continue

On peut interrompre l'exécution du bloc courant avec le mot-clé `continue` :

```
prénoms = ["Alice", "Bob", "Charlie"]
for prénom in prénoms:
    if prénom == "Bob":
        continue
    print("Bonjour", prénom)
```

```
Bonjour Alice
Bonjour Charlie
```

6.9 Indéxer une liste

- Avec `[]` et un entier
- Les index valides vont de 0 à `n-1` où `n` est la taille de la liste :

```
fruits = ["pomme", "orange", "poire"]

print(fruits[0])
# affiche: "pomme"

print(fruits[1])
# affiche: "orange"

print(list[2])
# affiche: "poire"

fruits[3]
# erreur: IndexError
```

6.10 Modifier une liste

Encore une assignation :

```
fruits = ["pomme", "orange", "poire"]
fruits[0] = "abricot"
print(fruits)
# affiche: ["abricot", "orange", "poire"]
```

6.11 Les strings sont aussi des listes (presque)

On peut itérer sur les caractères d'une string :

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

On peut tester si un caractère est présent :

```
print("e" in "vache")
# affiche: True

print(x" in "vache")
# affiche: False
```

Mais on ne peut pas modifier une string :

```
prénom = "Charlotte"
l = prénom[0]
print(l)
# affiche: "C"

l = prénom[3]
print(l)
# affiche: "r"

prénom[0] = "X"
# erreur: TypeError
```

6.12 Falsy et truthy

En réalité on peut mettre autre chose qu'une comparaison ou une variable booléenne après le if.

Si on met une liste vide, `if` se comportera comme si on avait mis une valeur fausse, et si la liste n'est pas vide , `if` se comportera comme si on avait mis une valeur vraie. :

```
ma_liste = [1, 2, 3]
if ma_liste:
    print("ma_liste n'est pas vide")
# affiche: ma_liste n'est pas vide

mon_autre_liste = []
if not mon_autre_liste:
```

(suite sur la page suivante)

(suite de la page précédente)

```
    print("mon_autre_liste est vide")
# affiche: mon_autre_liste est vide
```

On dit que les listes vides sont *Falsy* et les listes non-vides *Truthy*

CHAPITRE 7

Chapitre 7 - None et pass

7.1 None

7.1.1 Définition

`None` est un mot-clé qui sert à représenter l'absence.

Un peu comme `True` et `False` qui sonts des mot-clés qui réprésentent des booléens.

7.1.2 Retourner None

En réalité, *toutes* les fonctions pythons retournent *quelque chose*, même quand elle ne contiennent pas le mot-clé `return`. :

```
def ne_renvoie_rien():
    x = 2

resultat = ne_renvoie_rien()

print(resultat)
# affiche: None
```

7.1.3 Opérations avec None

La plupart des fonctions que nous avons vues échouent si on leur passe `None` en argument :

```
x = len(None)
# erreur: TypeError

x = None < 3
# erreur: TypeError

x = int(None)
# erreur: TypeError
```

Mais `str` fonctionne :

```
x = str(None)
print(x)
# affiche: 'None'
```

On peut vérifier si une variable vaut `None` avec `is None` :

```
x = ne_renvoie_rien()
print(x is None)
# affiche: True
```

7.1.4 Example d'utilisation

```
def trouve_dans_liste(valeur, liste):
    for element in liste:
        if element == valeur:
            return element
    return None

x = trouve_dans_liste(2, [1, 2, 3])
print(x)
# affiche: 2

x = trouve_dans_liste(1, [3, 4])
print(x)
# affiche: None
```

7.2 pass

À cause de l'organisation du flot de contrôle en blocs indentés, on ne peut pas vraiment avoir de blocs vides. Mais parfois, on a besoin d'un bloc qui ne fasse rien - c'est là que le mot-clé *pass* rentre en jeu.

pass représente une instruction qui ne fait rien.

Un exemple :

```
une_condition = False
if une_condition:
    pass
```

(suite sur la page suivante)

(suite de la page précédente)

```
else:
    print("une_condition n'est pas vraie")
```

On peut aussi - et c'est l'usage le plus courant - utiliser *pass* pour définir une fonction qui ne fait rien :

```
def ne_fait_rien():
    pass
```

CHAPITRE 8

Chapitre 8 - Dictionnaires

8.1 Définition

Un dictionaire est une _association_ entre des clés et des valeurs.

- Les clés sont uniques
- Les valeurs sont arbitraires

8.2 Création de dictionnaires

Avec des accolades : `{`, `}`

```
dictionaire_vide =  {}

une_clé_une_valeur = {"a": 42}

deux_clés_deux_valeurs = {"a": 42, "b": 53}
```

Les clés sont uniques :

```
{"a": 42, "a": 53} == {"a": 53}
```

Note : tous les dictionnaires sont truthy, sauf les dictionnaires vides.

8.3 Accès aux valeurs

Avec `[]`, comme pour les listes, mais avec une *clé* à la place d'un *index* :

```
scores = {"john": 10, "bob": 42}

print(scores["john"])
# affiche: 10

print(scores["bob"])
# affiche: 42

print(scores["charlie"])
# erreur: KeyError
```

8.4 Test d'appartenance

Avec `in`, comme le listes :

```
scores = {"john": 10, "bob": 42}
print("charlie" in scores)
# affiche: False
```

8.5 Modifier la valeur d'une clé

Comme pour les listes, avec une assignation :

```
scores = {"john": 10, "bob": 42}
scores["john"] = 20
print(scores)
# affiche: {"john": 20, "bob": 42}
```

8.6 Créer une nouvelle clé

Même mécanisme que pour la modification des clés existantes :

```
scores = {"john": 10, "bob": 42}
scores["charlie"] = 30
print(scores)
# affiche: {"john": 20, "bob": 42, "charlie": 30}
```

rappel : ceci ne fonctionne pas avec les listes : on ne peut pas « créer » de nouveaux éléments dans la liste juste avec un index :

```
ma_liste = ["a", "b"]
ma_liste[1] = "c"
print(ma_liste)
# affiche: ["a", "c"]
```

(suite sur la page suivante)

(suite de la page précédente)

```
ma_liste[3] = "d"
# erreur: IndexError
```

8.7 Itérer sur les clés

Avec `for ... in ...`, comme pour les listes :

```
scores = {"john": 10, "bob": 42}
for nom in scores:
    # on assigne la valeur "john" à la variable `nom`, puis "bob"
    score_associé_au_nom = scores[nom]
    # on assigne la valeur 10 puis la valeur 42 à la variable
    # score_associé_au_nom
    print(nom, score_associé_au_nom)
```

```
john 10
bob 42
```

8.8 del

8.8.1 Détruire une clé

Avec `del` - un nouveau mot-clé :

```
scores = {"john": 10, "bob": 42}
del scores["bob"]
print(scores)
# affiche: {"john": 10}
```

8.8.2 Détruire un élément d'une liste

Aussi avec `del` :

```
fruits = ["pomme", "banane", "poire"]
del fruits[1]
print(fruits)
# affiche: ["pomme", "poire"]
```

8.8.3 Détruire une variable

Encore et toujours `del` :

```
mon_entier = 42
mon_entier += 3
print(mon_entier)
# affiche: 45
```

(suite sur la page suivante)

(suite de la page précédente)

```
del mon_entier
mon_entier += 1
# erreur: NameError
```

CHAPITRE 9

Chapitre 9 - Tuples

9.1 Définition

Un tuple est un ensemble *ordonné* et *immuable* d'éléments. Le nombre, l'ordre et la valeur des éléments sont fixes.

9.2 Création de tuples

Avec des parenthèses :

```
tuple_vide = ()
tuple_à_un_élement = (1,)    # notez la virgule
tupble_à_deux_éléments = (1, 2)    # on dit aussi: "couple"
```

Sauf pour le tuple vide, c'est la *virgule* qui fait le tuple

Note : tous les tuples sont truthy, sauf les tuples vides.

9.3 Tuples hétérogènes

Comme les listes, les tuples peuvent contenir des éléments de types différents :

```
# Un entier et une string
mon_tuple = (42, "bonjour")

# Un entier et un autre tuple
mon_tuple = (21, (True, "au revoir"))
```

9.4 Accès

Avec `[]` et l'index de l'élément dans le tuple :

```
mon_tuple = (42, "bonjour")
print(mon_tuple[0])
# affiche: 42

print(mon_tuple[1])
# affiche: "bonjour"
```

9.5 Modification

Interdit :

```
mon_tuple = (42, "bonjour")
mon_tuple[0] = 44
# erreur: TypeError: 'tuple' object does not support item assignment
```

9.6 Test d'appartenance

Avec `in` :

mon_tuple = (42, 14) print(42 in mon_tuple) # affiche : True
print(14 in mon_tuple) # affiche : True
print(13 in mon_tuple) # affiche : False

9.7 Déstructuration

Créer plusieurs variables en une seule ligne :

```
couple = ("Batman", "Robin")
héros, side_kick = couple
print(héros)
# affiche: Batman

print(side_kick)
# affiche: Robin
```

9.8 Quelques erreurs classiques

```
héros, side_kick, ennemi = couple
# erreur: ValueError (3 != 2)

(héros,) = couple
# erreur: ValueError (1 != 2)
```

(suite sur la page suivante)

(suite de la page précédente)

```
# Gare à la virgule:
héros, = couple
# erreur: ValueError (1 != 2)
```

9.9 Pièges

```
f(a, b, c)   # appelle f() avec trois arguments

f((a, b, c)) # appelle f() avec un seul argument
             # (qui est lui-même un tuple à 3 valeurs)

f(())        # appelle f() avec un tuple vide


(a)      # juste la valeur de a entre parenthèses
(a,)     # un tuple à un élément, qui vaut la valeur de a
```

On peut aussi déstructurer des listes :

```
fruits = ["pomme", "banane", "orange"]
premier, deuxième, troisième = fruits

print(premier)
# affiche: pomme

print(deuxième)
# affiche: banane

print(troisième)
# affiche: orange
```

On dit aussi : *unpacking*

9.10 Utilisations des tuples

Pour simplifier des conditions :

```
# Avant:
if (
   ma_valeur == "nord" or
   ma_valeur == "sud" or
   ma_valeur == "ouest" or
   ma_valeur == "est"):
            print("direction", ma_valeur)

# Après:
if ma_valeur in ("nord", "sud", "est", "ouest"):
            print("direction", ma_valeur)
```

Pour retourner plusieurs valeurs :

```python
def tire_carte():
    valeur = "10"
    couleur = "trèfle"
    return (valeur, couleur)

v, c = tire_carte()
print(v, "de", c)
# affiche: 10 de trèfle
```

Ce n'est pas une nouvelle syntaxe, juste de la manipulation de tuples !

CHAPITRE 10

Chapitre 10 - Introduction aux classes

Ce qu'on a vu jusqu'ici :
- Des types simples (entiers, booléens, ...)
- Des structures de données (listes, dictionnaires, ...)
- Des fonctions qui manipulent ces types ou ces types
- Des fonctions qui s'appellent les unes les autres

On appelle cet ensemble de concepts, cette façon d'écrire du code, un *paradigme* - et c'est un paradigme *procédural*.

On va passer à un autre paradigme : l'"*orienté objet*.

10.1 Orienté objet - une première définition

Un « objet » informatique *représente* un véritable « objet » physique dans le vrai monde véritable.

Ce n'est pas une très bonne définition :

1. Ce n'est pas nécessaire
2. Ce n'est même pas forcément souhaitable !

Je le mentionne juste parce que c'est une idée reçue très répandue.

10.2 Orienté objet - 2ème définition

Une meilleure définition, c'est de dire que la programmation orientée objet permet de mettre au même endroit :
- des données
- des fonctions qui opèrent sur ces données

L'important c'est que les deux aillent ensemble !

*Note : ce n'est pas **la* meilleure définition de l'orienté objet, mais on s'en contentera pour le moment ...*

10.3 Les classes

On va parler *d'une* façon de faire de l'orienté objet : avec des classes.

Mais notez bien qu'on peut faire de l'orienté objet *sans* classes !

10.4 Le plan de construction

On dit souvent qu'en Python, « tout est objet ».

Pour bien comprendre cela, il faut d'abord parler des *classes* et des *instances de classes*.

Une classe est un *plan de construction*, et est définie ainsi :

```
class MaClasse:
    # du code ici
```

Comme les fonctions, les classes contienent un *corps*, qui est le bloc *identé* en dessous du mot-clé *class*, de nom de la classe et du *:* en fin de ligne.

Les classes sont utilisées pour construire des *instances*.

10.5 Créons des instances

On peut faire un plan de construction vide avec le mot-clé pass :

```
class MaClasse:
    pass
```

Dans ce cas, on crée une instance en mettant le nom de la classe suivi d'une paire de parenthèses - un peu comme pour appeler une fonction :

```
mon_instance = MaClasse()
```

Ici, `mon_instance` est une *instance* de la classe `MaClasse`.

10.6 Attributs

Les attributs sont des éléments **nommés** à *l'intérieur* d'une instance.

On peut y accéder avec la syntaxe `<instance>.<attribut>` :

```
y = a.x
```

Ici, `y` est l'attribut `x` de l'instance `a`.

Les attributs peuvent être des fonctions :

```
func = a.x
func(10)
```

Ici, on crée une variable `func` qui prend la valeur de l'attribut `x` dans l'instance `a`, puis on l'appelle avec l'argument `10` à la ligne suivante.

Le code suivant fait exactement la même chose, mais avec une ligne de moins :

```
a.x(10)
```

On peut *créer* des attributs dans *n'importe quel instance*, en utilisant l'"*assignation* :

```
mon_instance = MaClasse()

# Création de l'attribut `x` dans `mon_instance`
mon_instance.x = 42

# Accés à l'attribut `x` dans `mon_instance`
print(mon_instance.x)
# affiche: 42
```

Ici on assigne la valeur `42` à l'attribut `x` de l'instance `mon_instance`

10.7 Méthodes - définition

On peut aussi mettre des *méthodes* dans des classes.

On utilise *def*, comme pour les fonctions, mais les méthodes *doivent* avoir au moins un argument appelé *self*, et être à l'intérieur du bloc de la classe :

```
class MaClasse:
    def ma_méthode(self):
        return 42
```

Notez que les méthodes *sont aussi des attributs*. Leur valeur est une *fonction* qui se comporte légèrement différemment des fonctions qu'on a vu jusqu'ici.

10.8 Méthodes - appel

Une méthode ne peut être appelée que depuis une *instance* de la classe :

```
class MaClasse:
    def ma_méthode(self):
            return 42

ma_méthode()
# erreur: NameError

mon_instance = MaClasse()
résultat = mon_instance.ma_méthode()
print(résultat)
# affiche: 42
```

Notez qu'on ne passe *pas* d'argument quand on apelle *ma_méthode* depuis l'instance.

10.9 Méthodes et attributs

`self` *prend la valeur de l'instance courante* quand la méthode est appelée.

On peut le voir en utilisant des attributs :

```
class MaClasse:
    def affiche_attribut_x(self):
        # Accès à l'attribut `x` dans `self`
        print(self.x)


mon_instance = MaClasse()
mon_instance.x = 42
mon_instance.affiche_attribut_x()
# Affiche: 42
```

On peut aussi *créer* des attributs dans une méthode :

```
class MaClasse:
    def crée_attribut_x(self):
        self.x = 42
    def affiche_attribut_x(self):
        print(self.x)

mon_instance = MaClasse()
mon_instance.affiche_attribut_x()
# erreur: `mon_instance` n'a pas d'attribut `x`

mon_instance.crée_attribut_x()
mon_instance.affiche_attribut_x()
# affiche: 42
```

Les méthodes peuveunt aussi prendre plusieurs arguments, en plus de `self` - mais `self` doit toujours être le premier argument.

Par example, pour créer un attribut avec une certaine valeur :

```
class MaClasse
    def crée_attribut_x(self, valeur_de_x):
        self.x = valeur_de_x

    def affiche_attribut_x(self);
        print(self.x)

mon_instance = MaClasse()
mon_instance.crée_attribut_x(42)
mon_instance.affiche_attribut_x()
# affiche: 42
```

10.10 Méthodes appelant d'autres méthodes

Comme les méthodes sont *aussi* des attributs, les méthodes d'une instance peuvent s'appeler les unes les autres :

```
class MaClasse:
    def méthode_1(self):
        print("démarrage de la méthode 1")
        print("la méthode 1 affiche bonjour")
        print("bonjour")
        print("fin de la méthode 1")


    def méthode_2(self):
        print("la méthode 2 appelle la méthode 1")
        self.méthode_1()
        print("fin de la méthode 2")


mon_instance = MaClasse()
mon_instance.méthode_2()
```

```
la méthode 2 appelle la méthode 1
démarrage de la méthode 1
la méthode 1 affiche bonjour
bonjour
fin de la méthode 1
fin de la méthode 2
```

10.11 Une méthode spéciale

Si vous définissez une méthode nomée `__init__`, celle-ci est appelée *automatiquement* quand l'instance est construite.

On dit que c'est une méthode « magique » parce qu'elle fait quelque chose _sans_ qu'on l'appelle explicitement.

On utilise souvent `__init__` pour créer des attributs :

```
class MaClasse:
    def __init__(self):
        self.x = 1
        self.y = 2

mon_instance = MaClasse()

# __init__ est appelée automatiquement!
print(mon_instance.x)
# affiche: 1
print(mon_instance.y)
# affiche: 2
```

On prend souvent les *valeurs* des attributs à créer en arguments de la méthode `__init__` :

```
class MaClasse:
    def __init__(self, x, y):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        self.x = x
        self.y = y
```

Dans ce cas, les arguments de la méthode `__init__` apparaissent à l'intérieur des parenthèses après le nom de la classe :

```
mon_instance = MaClasse(3, 4)
print(mon_instance.x)
# affiche: 3
print(mon_instance.y)
# affiche: 4
```

Note : Pour cette raison, __init__ est souvent appelé le **constructeur** de la classe.

10.12 Récapitulatif

- Classe : plan de construction
- Instance : valeur issue d'une classe
- Attribut : variable dans une instance
- Méthode : fonction dans une instance (qui prend *self* en premier argument)
- `__init__` : méthode magique appelée automatiquement pendant l'instanciation

10.13 Classes et programmation orienté objet

Ainsi, on peut ranger au même endroit des données et des fonctions opérant sur ces données.

Les données sont les attributs, et les fonctions opérant sur ces attributs sont les méthodes.

On peut ainsi séparer les *responsabilités* à l'intérieur d'un code en les répartissant entres plusieurs classes.

CHAPITRE 11

Chapitre 11 - Introduction aux modules

11.1 Un fichier = un module

Et oui, vous faites des modules sans le savoir depuis le début :)

Un fichier *foo.py* correspond *toujours* module *foo*

Attention : Ce n'est pas tout à fait réciproque. Le module `foo` peut venir d'autre chose qu'un fichier foo.py.

11.2 Importer un module

Ou : accéder à du code provenant d'un *autre* fichier source.

Créons un fichier *bonjour.py* contenant seulement une assignation de l'entier 42 à la variable *a* :

```
# Dans bonjour.py
a = 42
```

Comme un fichier = un module, on vient de crée un module `bonjour` contenant une variable `a`.

Si maintenant on crée un fichier `salutations.py` dans le même répertoire, on peut accéder à cette variable en *important* le module avec le mot-clé `import` :

```
# Dans salutations.py
import bonjour
print(bonjour.a)
# affiche: 42
```

Note : Le nom du module est écrit directement, ce n'est *pas* une chaîne de caractères.

On voit que la variable *a* dans *bonjour.py* est devenue un *attribut* du module *bonjour* lorsque *bonjour* a été importé

Si maintenant on rajoute une fonction `dire_bonjour` dans `bonjour.py` :

```
# Dans bonjour.py
a = 42
def dire_bonjour():
    print("Bonjour!")
```

On peut appeler la fonction `dire_bonjour` depuis `salutations.py` en utilisant l'attribut `dire_bonjour` du module `bonjour` :

```
# Dans salutations.py
import bonjour
bonjour.dire_bonjour()
# affiche: Bonjour!
```

11.3 Les imports ne sont faits qu'une seule fois

Il est important de noter que :
- **tout** le code à l'intérieur d'un module est éxécuté lors du premier import
- mais il n'est *pas* ré-éxécuté si le module a déjà été importé auparavant.

On peut le voir en mettant du code dans *bonjour.py*, en plus des simples définitions de fonctions et de créations de variables :

```
# Dans bonjour.py
print("Je suis le module bonjour et tu viens de m'importer")
def dire_bonjour():
    ....
```

```
# Dans salutation.py
import bonjour
# affiche: Je suis le module bonjour et tu viens de m'importer

import bojour
# n'affiche rien
```

11.4 La bibliothèque standard

La bibliothèque standard est une collection de modules directement utilisables fournis à l'installation de Python.

Exemple : `sys`, `random`, ...

Toute la bibliothèque standard est documentée - et la traduction en Français est en cours :

https://docs.python.org/fr/3/library/index.html

Mettez ce lien dans vos favoris - il vous sera très utile.

11.5 Quelques exemples de modules de la bibliothèque standard

11.5.1 Easter eggs

(Ou fonctionnalités cachées)

- `import antigravity`
- `import this`

Je vous laisse découvrir ce que fait le premier. Quant au deuxième, il contient une liste de préceptes que la plupart des développeurs Python s'efforcent de respecter. On en reparlera . . .

CHAPITRE 12

Chapitre 12 - Couplage et composition

12.1 Couplage

12.1.1 Définition

Un couplage décrit une relation entre deux classes.

12.1.2 Exemple

Ici on veut représenter des chats et des humains qui adoptent (on non) des chats. Tous les chats ont un nom, et tous les humains ont un prénom.

On peut utiliser pour cela deux classes : *Chat* et *Humain* :

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

chat = Chat("Monsieur Moustaches")
prin(chat.nom)
# affiche: Monsieur Moustaches

class Humain:
    def __init__(self,  prénom):
        self.prénom = prénom

alice = Humain(prénom="Alice")
print(alice.prénom)
# affiche: Alice
```

Maintenant on veut que les humains puissent adopter des chats. Pour cela, on peut rajouter la méthode `adopte` dans la classe `Humain`.

Cette méthode va prendre un argument - une instance de la classe `Chat` :

```
class Humain:
    def __init__(self,  prénom):
        self.prénom = prénom

    def adopte(self, chat):
        print(self.prénom, "adopte un chat")

boule_de_poils = Chat("Boule de Poils")
alice = Humain("Alice")
alice.adopte(boule_de_poils)
# affiche: "Alice adopte un chat"
```

On peut accéder au nom du chat depuis la méthode `adopte`, en utilisant la syntaxe `nom.attribut` vue précédemment :

```
class Humain:
    def __init__(self,  prénom):
        self.prénom = prénom

    def adopte(self, chat):
        print(self.prénom, "adopte", chat.nom)

boule_de_poils = Chat("Boule de Poils")
alice = Humain("Alice")
alice.adopte(boule_de_poils)
# affiche: Alice adopte Boule de Poils
```

12.1.3 Couplage

```
class Humain:
    ...
    def adopte(self, chat):
        print(self.prénom, "adopte", chat.nom)
```

Notez également que nous avons écrit `chat.nom`. ainsi, la méthode `adopte()` ne peut être appelée que part une instance qui a un attribut `nom` - sinon on aura une erreur.

Donc si on modifie la classe `Chat` et qu'on renomme l'attribut `nom` en `surnom` par exemple, la méthode `adopte()` de la classe `Humain` cessera de fonctionner : on dit qu'on a un *couplage* entre les classes `Chat` et `Humain`.

12.2 Composition

12.2.1 Définition

Une classe à l'intérieur d'une autre classe.

12.2.2 Dépendances entre fonctions

Exemple

```
def calcule_x():
    ...
    # du code ici

def fait_un_truc_avec_x(x):
    ...
    # du code ici

x = calcule_x()
fait_un_truc_avec_x(x)
```

On voit que la fonction `fait_un_truc_avec_x()` prend un argument `x`, qui est retourné par la fonction `calcule_x()`.

`fait_un_truc_avec_x()` doit donc être appelée *après* `calcule_x()`. On dit que `fait_un_truc_avec_x()` *dépend* de `calcule_x()`.

12.2.3 Dépendances entre classes

Un bon moyen d'introduire une dépendance entre deux classes est d'utiliser les constructeurs.

Revoyons la classe Chat :

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

Comme le constructeur de la classe Chat prend un nom en argument, il est impossible de construire des chats sans nom :

```
chat = Chat()
# erreur: TypeError: __init__() missing 1 required positional argument: 'nom'
```

De la même façon, si on veut que tous les enfants aient un chat (pourquoi pas, après tout), on peut avoir une classe Enfant, dont le constructeur prend une instance de chat en plus du prénom :

```
class Enfant:
    def __init__(self, prénom, chat):
        self.prénom = prénom
        self.chat = chat

 alice = Enfant("Alice")
 # erreur: TypeError: __init__() missing 1 required positional argument: 'chat'

 boule_de_poils = Chat("Boule de Poils")
 alice = Enfant("Alice", boule_de_poils)
 # OK!
```

12.2.4 Utilisation de la composition

Maintenant qu'on vit dans un monde où tous les enfants ont chacun un chat, on peut par exemple consoler tous les enfants en leur demandant de caresser leur chat, chat qui va ronronner et faire plaisir à son propriétaire.

voici comment on peut coder cela : d'abord, on rajoute les méthodes `caresse()` et `ronronne()` dans la classe chat :

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

    def ronronne(self):
        print(self.nom, 'fait: "prrrrr"')

    def caresse(self):
        self.ronronne()


boule_de_poils = Chat("Boule de Poils")
boule_de_poils.caresse()
# affiche: Boule de Poils fait "prrrrr"
```

Ensuite, on peut rajouter la méthode `console()` dans la classe Enfant, qui va :

— récupérer l'instance de la classe Chat dans `self` - comme n'importe quel attribut

— puis appeler la méthode `caresse()` de cette instance :

```
class Enfant:
    def __init__(self, prénom, chat):
        self.prénom = prénom
        self.chat = chat

    def console(self):
        self.chat.caresse()
```

Si on combine ces deux classes dans le même morceau de code et qu'on exécute les instructions suivantes :

```
boule_de_poils = Chat("Boule de Poils")
alice = Enfant("Alice", boule_de_poils)
alice.console()
```

On afffichera `` `Boule de Poils fait "prrr" `` et alice sera consolée.

On dit parfois qu'on a *délégué* l'implémentation de la méthode `console()` de la classe Enfant à la méthode `caresse()` de la classe Chat.

CHAPITRE 13

Chapitre 13 - Bibliothèques

13.1 Introduction

13.1.1 Importer un module

Souvenez-vous, dans le chapitre 12 nous avons vu que le code suivant Ce code fonctionne s'il y a un ficher *foo.py* quelque part qui contient la fonction `bar` : :

```
import foo
foo.bar()
```

Ce fichier peut être présent soit dans le répertoire courant, soit dans la bibliothèque standard Python.

13.1.2 La variable PATH

Vous connaissez peut-être le rôle de la variable d'environnement `PATH`. Celle-ci contient une liste de chemins, séparés par le caractère `:` et est utilisée par votre shell pour trouver le chemin complet des commandes que vous lancez.

Par exemple :

```
PATH="/bin:/usr/bin:/usr/sbin"
$ ifconfig
# lance le binaire /usr/sbin/ifconfig
$ ls
# lance le binaire /bin/ls
```

Le chemin est « résolu » par le shell en parcourant la liste de tout les chemins de la variable *PATH*, et en regardant si le chemin complet existe. La résolution s'arrête dès le premier chemin trouvé.

Par exemple, si vous avez `PATH="/home/user/bin:/usr/bin"` et un fichier `ls` dans `/home/user/bin/ls`, c'est ce fichier-là (et non `/bin/ls`) qui sera utilisé quand vous taperez `ls`.

13.2 sys.path

En Python, il existe une variable `path` prédéfinie dans le module `sys` qui fonctionne de manière similaire à la variable d'environnement `PATH`.

Si j'essaye de l'afficher sur ma machine, voici ce que j'obtiens :

```
import sys
print(sys.path)
```

```
[
 "",
 "/usr/lib/python3.8",
 "/usr/lib/python3.8/lib-dynload",
 "/home/dmerej/.local/lib/python3.8/",
 "/usr/lib/python3.8/site-packages",
]
```

Le résultat dépend :
- du système d'exploitation
- de la façon dont Python a été installé
- et de la présence ou non de certains réportoires.

En fait, `sys.path` est construit dynamiquement par l'interpréteur Python au démarrage.

Notez également que `sys.path` commence par une chaîne vide. En pratique, cela signifie que le répertoire courant a la priorité sur tout le reste.

13.2.1 Priorité du répertoire courant

Prenons un exemple. Si vous ouvrez un explorateur de fichiers dans le deuxième élément de la liste de `sys.path` (`/usr/lib/python3.8/` sur ma machine), vous trouverez un grand nombre de fichiers Python.

notamment, vous devriez trouver un fichier `random.py` dans ce répertoire.

En fait, vous trouverez la plupart des modules de la bibliothèque standard dans ce répertoire.

Maintenant, imaginons que vous avez un deuxième fichier `random.py` dans votre répertoire courant. Finalement, imaginez que vous lancez un fichier `foo.py` contentant `import random` dans ce même réportoire.

Et bien, c'est le fichier `random.py` de votre répertoire qui sera utilisé, et non celui de la bibliothèque standard !

13.2.2 Permissions des répertoires de sys.path

Un autre aspect notable de `sys.path` est qu'il ne contient que deux répertoires dans lesquels l'utilisateur courant peut potentiellement écrire : le chemin courant et le chemin dans `~/.local/lib`. Tous les autres (`/usr/lib/python3.8/`, etc.) sont des chemins « système » et ne peuvent être modifiés que par un compte administrateur (avec `root` ou `sudo`, donc).

La situation est semblable sur macOS et Windows.

13.3 Bibliothèques tierces

Prenons un exemple :

```
# dans foo.py
import tabulate

scores = [
  ["John", 345],
  ["Mary-Jane", 2],
  ["Bob", 543],
]
table = tabulate.tabulate(scores)
print(table)
```

```
$ python3 foo.py
---------  ---
John       345
Mary-Jane    2
Bob        543
---------  ---
```

Ici, le module `tabulate` n'est ni dans la bibliothèque standard, ni écrit par l'auteur du script `foo.py`. On dit que c'est une bibliothèque tierce.

On peut trouver le code source de tabulate facilement. La question qui se pose alors est : comment faire en sorte que *sys.path* contienne le module `tabulate` ?

Eh bien, plusieurs solutions s'offrent à vous.

13.3.1 Le gestionnaire de paquets

Si vous utilisez une distribution Linux, peut-être pourrez-vous utiliser votre gestionnaire de paquets :

```
$ sudo apt install python3-tabulate
```

Comme vous lancez votre gestionnaire de paquets avec `sudo`, celui-ci sera capable d'écrire dans les chemins système de `sys.path`.

13.3.2 À la main

Une autre méthode consiste à partir des sources - par exemple, si le paquet de votre distribution n'est pas assez récent, ou si vous avez besoin de modifier le code de la bibliothèque en question.

Voici une marche à suivre possible :

1. Récupérer les sources de la version qui vous intéresse dans la section téléchargement de bitbucket. 1. Extraire l'archive, par exemple dans `src/tabulate` 1. Se rendre dans `src/tabulate` et lancer `python3 setup.py install --user`

13.3.3 Anatomie du fichier setup.py

La plupart des bibliothèques Python contiennent un `setup.py` à la racine de leurs sources. Il sert à plein de choses, la commande `install` n'étant qu'une parmi d'autres.

Le fichier `setup.py` contient en général simplement un `import` de `setuptools`, et un appel à la fonction `setup()`, avec de nombreux arguments :

```
# tabulate/setup.py
from setuptools import setup

setup(
  name='tabulate',
  version='0.8.1',
  description='Pretty-print tabular data',
  py_modules=["tabulate"],
  scripts=["bin/tabulate"],
  ...
)
```

13.3.4 Résultat de l'invocation de setup.py

Par défaut, `setup.py` essaiera d'écrire dans un des chemins système de `sys.path`, d'où l'utilisation de l'option `--user`.

Voici à quoi ressemble la sortie de la commande :

```
$ cd src/tabulate
$ python3 setup.py install --user
running install
...
Copying tabulate-0.8.4-py3.7.egg to /home/dmerej/.local/lib/python3.7/site-packages
...
Installing tabulate script to /home/dmerej/.local/bin
```

Notez que module a été copié dans `~/.local/lib/python3.7/site-packages/` et le script dans `~/.local/bin`. Cela signifie que *tous* les scripts Python lancés par l'utilisateur courant auront accès au module `tabulate`.

Notez également qu'un script a été installé dans `~/.local/bin` - Une bibliothèque Python peut contenir aussi bien des modules que des scripts.

Un point important est que vous n'avez en général pas besoin de lancer le script directement. Vous pouvez utiliser `python3 -m tabulate`. Procéder de cette façon est intéressant puisque vous n'avez pas à vous soucier de rajouter le chemin d'installation des scripts dans la variable d'environnement PATH.

13.4 Dépendances

Prenons une autre bibliothèque : cli-ui.

Elle permet d'afficher du texte en couleur dans un terminal :

```
import cli_ui

cli_ui.info("Ceci est en", cli_ui.red, "rouge")
```

Elle permet également d'afficher des tableaux en couleur :

```
headers=["name", "score"]
data = [
  [(bold, "John"), (green, 10.0)],
  [(bold, "Jane"), (green, 5.0)],
]
cli_ui.info_table(data, headers=headers)
```

Pour ce faire, elle repose sur la bibliothèque tabulate vue précédemment. On dit que cli-ui *dépend* de tabulate.

13.4.1 Déclaration des dépendances

La déclaration de la dépendance de cli-ui vers tabulate s'effectue également dans le fichier setup.py :

```
setup(
  name="cli-ui",
  version="0.9.1",
  install_requires=[
     "tabulate",
     ...
  ],
  ...
)
```

13.4.2 pypi.org

On comprend dès lors qu'il doit nécessairement exister un *annuaire* permettant de relier les noms de dépendances à leur code source.

Cet annuaire, c'est le site pypi.org. Vous y trouverez les pages correspondant à tabulate et cli-ui.

13.4.3 pip

pip est un outil qui vient par défaut avec Python3[^4]. Vous pouvez également l'installer grâce au script get-pip.py, en lançant python3 get-pip.py --user.

Il est conseillé de *toujours* lancer pip avec python3 -m pip. De cette façon, vous êtes certains d'utiliser le module pip correspondant à votre binaire python3, et vous ne dépendez pas de ce qu'il y a dans votre PATH.

pip est capable d'interroger le site pypi.org pour retrouver les dépendances, et également de lancer les différents scripts setup.py.

Comme de nombreux outils, il s'utilise à l'aide de *commandes*. Voici comment installer `cli-ui` à l'aide de la commande "install" de `pip` :

```
$ python3 -m pip install cli-ui --user
Collecting cli-ui
...
Requirement already satisfied: unidecode in /usr/lib/python3.7/site-packages (from cli-ui) (1.0.23)
Requirement already satisfied: colorama in /usr/lib/python3.7/site-packages (from cli-ui) (0.4.1)
Requirement already satisfied: tabulate in /mnt/data/dmerej/src/python-tabulate (from cli-ui) (0.8.4)
Installing collected packages: cli-ui
Successfully installed cli-ui-0.9.1
```

On constate ici quelques limitations de `pip` :

— Il faut penser à utiliser `--user` (de la même façon que lorsqu'on lance `setup.py` à la main)

— Si le paquet est déjà installé dans le système, pip ne saura pas le mettre à jour - il faudra passer par le gestionnaire de paquet de la distribution

En revanche, *pip* contient de nombreuses fonctionnalités intéressantes :

— Il est capable de désinstaller des bibliothèques (à condition toutefois qu'elles ne soient pas dans un répertoire système)

— Il est aussi capable d'afficher la liste complète des bibliothèques Python accessibles par l'utilisateur courant avec *freeze*.

Voici un extrait de la commande `python3 -m pip freeze` au moment de la rédaction de cet article sur ma machine :

```
$ python3 -m pip freeze
apipkg==1.5
cli-ui==0.9.1
gaupol==1.5
tabulate==0.8.4
```

On y retrouve les bibliothèques `cli-ui` et `tabulate`, bien sûr, mais aussi la bibliothèque `gaupol`, qui correspond au [programme d'édition de sous-titres](https://otsaloma.io/gaupol/) que j'ai installé à l'aide du gestionnaire de paquets de ma distribution. Précisons que les modules de la bibliothèque standard et ceux utilisés directement par pip sont omis de la liste.

On constate également que chaque bibliothèque possède un *numéro de version*.

13.4.4 Numéros de version

Les numéros de version remplissent plusieurs rôles, mais l'un des principaux est de spécifier des changements incompatibles.

Par exemple, pour `cli-ui`, la façon d'appeler la fonction `ask_choice` a changé entre les versions 0.7 et 0.8, comme le montre cet extrait du changelog :

> *The list of choices used by ask_choice is now a named keyword argument :*

```
# Old (<= 0.7)
ask_choice("select a fruit", ["apple", "banana"])
# New (>= 0.8)
ask_choice("select a fruit", choices=["apple", "banana"])
```

Ceci s'appelle un *changement d'API*.

13.4.5 Réagir aux changements d'API

Plusieurs possibilités :
- On peut bien sûr adapter le code pour utiliser la nouvelle API, mais cela n'est pas toujours possible ni souhaitable.
- Une autre solution est de spécifier des *contraintes* sur le numéro de version dans la déclaration des dépendances. Par exemple :

```
setup(
  install_requires=[
    "cli-ui < 0.8",
    ...
  ]
)
```

13.4.6 Aparté : pourquoi éviter sudo pip

Souvenez-vous que les fichiers systèmes sont contrôlés par votre gestionnaire de paquets.

Les mainteneurs de votre distribution font en sorte qu'ils fonctionnent bien les uns avec les autres. Par exemple, le paquet `python3-cli-ui` ne sera mis à jour que lorsque tous les paquets qui en dépendent seront prêts à utiliser la nouvelle API.

En revanche, si vous lancez `sudo pip` (où `pip` avec un compte root), vous allez écrire dans ces mêmes répertoire et vous risquez de « casser » certains programmes de votre système.

Mais il y a un autre problème encore pire.

13.4.7 Conflit de dépendances

Supposons deux projets A et B dans votre répertoire personnel. Ils dépendent tous les deux de `cli-ui`, mais l'un des deux utilise `cli-ui 0.7` et l'autre `cli-ui 0.9`. Que faire ?

13.5 Environnements virtuels

La solution est d'utiliser un environnement virtuel (*virtualenv* en abrégé). C'est un répertoire *isolé* du reste du système.

13.5.1 Aparté : python3 -m venv sur Debian

La commande *python3 -m venv* fonctionne en général partout, dès l'installation de Python3 (*out of the box*, en Anglais), *sauf* sur Debian et ses dérivées.

Si vous utilisez Debian, la commande pourrait ne pas fonctionner. En fonction des messages d'erreur que vous obtenez, il est possible de résoudre le problème en :
- installant le paquet `python3-venv`,
- ou en utilisant d'abord `pip` pour installer `virtualenv`, avec `python3 -m pip install virtualenv --user` puis en lançant `python3 -m virtualenv foo-venv`.

13.5.2 Comportement de python dans le virtualenv

Ce répertoire contient de nombreux fichiers et dossiers, et notamment un binaire dans `foo-venv/bin/python3`.

Voyons comment il se comporte en le comparant au binaire `/usr/bin/python3` habituel :

```
$ /usr/bin/python3 -c 'import sys; print(sys.path)'
['',
  ...
 '/usr/lib/python3.7',
 '/usr/lib/python3.7.zip',
 '/usr/lib/python3.7/lib-dynload',
 '/home/dmerej/.local/lib/python3.7/site-packages',
 '/usr/lib/python3.7/site-packages'
]
```

```
$ /home/dmerej/foo-venv/bin/python -c 'import sys; print(sys.path)'
['',
 '/usr/lib/python3.7',
 '/usr/lib/python3.7.zip',
 '/usr/lib/python3.7/lib-dynload',
 '/home/dmerej/foo-venv/lib/python3.7/site-packages,
]
```

À noter :
- Le répertoire « global » dans `~/.local/lib` a disparu
- Seuls quelques répertoires systèmes sont présents (ils correspondent plus ou moins à l'emplacement des modules de la bibliothèque standard)
- Un répertoire *au sein* du virtualenv a été rajouté

Ainsi, l'isolation du virtualenv est reflété dans la différence de la valeur de `sys.path`.

Il faut aussi préciser que le virtualenv n'est pas complètement isolé du reste du système. En particulier, il dépend encore du binaire Python utilisé pour le créer.

Par exemple, si vous utilisez `/usr/local/bin/python3.7 -m venv foo-37`, le virtualenv dans `foo-37` utilisera Python 3.7 et fonctionnera tant que le binaire `/usr/local/bin/python3.7` existe.

Cela signifie également qu'il est possible qu'en mettant à jour le paquet `python3` sur votre distribution, vous rendiez inutilisables les virtualenvs créés avec l'ancienne version du paquet.

13.5.3 Comportement de pip dans le virtualenv

D'après ce qui précède, le virtualenv ne devrait contenir aucun module en dehors de la bibliothèque standard et de `pip` lui-même.

On peut s'en assurer en lançant `python3 -m pip freeze` depuis le virtualenv et en vérifiant que rien ne s'affiche :

```
$ python3 -m pip freeze
# de nombreuses bibliothèques en dehors du virtualenv
apipkg==1.5
cli-ui==0.9.1
gaupol==1.5
tabulate==0.8.4
```

```
$ /home/dmerej/foo-venv/bin/python3 -m pip freeze
# rien :)
```

On peut alors utiliser le module `pip` *du virtualenv* pour installer des bibliothèques dans celui-ci :

```
$ /home/dmerej/foo-venv/bin/python3 -m pip install cli-ui
Collecting cli-ui
  Using cached https://pythonhosted.org/..cli_ui-0.9.1-py3-none-any.whl
Collecting colorama (from cli-ui)
  Using cached https://pythonhosted.org/..colorama-0.4.1-py2.py3-none-any.whl
Collecting unidecode (from cli-ui)
  Using cached https://pythonhosted.org/..Unidecode-1.0.23-py2.py3-none-any.whl
Collecting tabulate (from cli-ui)
Installing collected packages: colorama, unidecode, tabulate, cli-ui
Successfully installed cli-ui-0.9.1 colorama-0.4.1 tabulate-0.8.3
  unidecode-1.0.23
```

Cette fois, aucune bibliothèque n'est marquée comme déjà installée, et on récupère donc `cli-ui` et toutes ses dépendances.

On a enfin notre solution pour résoudre notre conflit de dépendances : on peut simplement créer un virtualenv par projet. Ceci nous permettra d'avoir effectivement deux versions différentes de `cli-ui`, isolées les unes des autres.

13.5.4 Activer un virtualenv

Devoir préciser le chemin du virtualenv en entier pour chaque commande peut devenir fastidieux ; heureusement, il est possible *d'activer* un virtualenv, en lançant une des commandes suivantes :

- `source foo-venv/bin/activate` - si vous utilisez un shell POSIX
- `source foo-venv/bin/activate.fish` - si vous utilisez Fish
- `foo-venv\bin\activate.bat` - sous Windows

Une fois le virtualenv activé, taper `python`, `python3` ou `pip` utilisera les binaires correspondants dans le virtualenv automatiquement, et ce, tant que la session du shell sera ouverte.

Le script d'activation ne fait en réalité pas grand-chose à part modifier la variable `PATH` et rajouter le nom du virtualenv au début de l'invite de commandes :

```
# Avant
user@host:~/src $ source foo-env/bin/activate
# Après
(foo-env) user@host:~/src $
```

Pour sortir du virtualenv, entrez la commande `deactivate`.

13.5.5 Les environnements virtuels en pratique

Le système de gestions des dépendances de Python peut paraître compliqué et bizarre, surtout venant d'autres langages.

Mon conseil est de toujours suivre ces deux règles :

- Un virtualenv par projet
- Toujours utiliser `pip` *depuis* un virtualenv

Certes, cela peut paraître fastidieux, mais c'est une méthode qui vous évitera probablement de vous arracher les cheveux (croyez-en mon expérience).

Voici un exemple d'arborescence qui montre deux projets, `projet-1` et `projet-2`, chacun avec son virtualenv dédié :

```
projets
├── projet-1
```

(suite sur la page suivante)

(suite de la page précédente)

```
│    ├── projet_1.py
│    └── .venv
│         ├── bin
│         ├── include
│         ├── lib
│         └── ...
└── projet-2
     ├── projet_2.py
     └── .venv
          ├── bin
          ├── include
          ├── lib
          └── ...
```

Notez bien que les sources de chaque projet `projet_1.py` et `projet_2.py` sont au même niveau que le répertoire `.venv` et non à l'intérieur de celui-ci.

Vous pouvez aussi retenir la règle suivante : étant donné un réportoire « X » contenant les sources d'un projet, les commandes à lancer pour créer le virtualenv seront :

```
cd <X>
python3 -m venv .venv
```

CHAPITRE 14

Chapitre 14 - Données binaires et fichiers

On vous a peut-être déjà dit que l'informatique consiste à manipuler des suites de 0 et et de 1s, mais qu'en est-t-il exactement ?

De manière surprenante, la réponse à cette question va nous emmener sur un chemin qui va parler de maths, puis de comment le langage Python peut interagir avec « l'extérieur ».

14.1 Données binaires

14.1.1 Introduction : chiffres et nombres

Si je vous parle de ce que représente le texte : `342` vous pouvez le voir de deux façons :

1. C'est une **suite de chiffres** : `3`, puis `4`, puis `2`. 1. C'est un **nombre** (quelque part entre 300 et 350)

On se sert des *chiffres* de 0 à 9 pour *représenter* des *nombres*

14.1.2 La base 10

Plus exactement, pour passer de la suite de chiffres `342` au nombre, on part de la fin, puis on ajoute chaque chiffre, multiplié par la puissance de 10 adéquate :

```
   2 * 1
+  4 * 10
+  2 * 10 * 10
```

soit :

```
    2
+  40
+ 300
```

ce qui fait bien 342.

14.1.3 La base 16

En informatique, on se sert souvent de la base 16. C'est le même principe : on se sert des « chiffres » de 0 à F (A vaut 10, B vaut 11, jusqu'à F qui vaut 15)

Ainsi, la suite `DA2` peut être interprétée comme suit

```
    2 * 1
+  10 * 16
+  13 * 16 * 16
```

soit

```
     2
+   160
+  3328
```

soit 3746

On appelle aussi la base 16 la base *hexadécimale*, ou *hexa* en abrrégé.

14.1.4 La base 2

La base 2 c'est pareil, mais avec deux « chiffres » - 0 et 1.

Ainsi, la suite *110* peut être interprétée comme suit

```
    0 * 1
+   1 * 2
+   1 * 2 * 2
```

soit

```
     0
+    2
+    4
```

soit 6.

14.1.5 Bits et octets

- Un bit (*bit* en anglais) c'est la valeur 1 ou 0
- Un octet (*byte* en anglais) c'est une suite de 8 bits

14.1.6 À retenir

Ces paquets de 8 ne veulent rien dire en eux-mêmes. Ils n'ont de sens que dans le cadre d'une *convention*.

Par exemple, l'octet "10100100" peut être un nombre écrit en binaire (164 en l'occurrence), mais peut avoir une toute autre signification

Le nombre de valeurs possible augmente *très* rapidement avec le nombre d'octets :
- 1 octet, c'est 255 valeurs possibles (`2 ** 8`)
- 2 octets, c'est 65.536 valeurs possibles (`2 ** 16`)
- 4 octets, c'est 4.294.967.296 valeurs possibles (`2 ** 32`)

14.1.7 Bases en Python

On se sert des préfixes *0b* et *0x* pour noter les nombres en base binaire ou hexadécimale respectivement :

```
print(0b1110)
# affiche: 6

print(0xDA2)
# affiche: 3490
```

14.1.8 Poids des bits

Regardez l'example suivant :

```
x = 0b0010010 # 18
y = 0b0010011 # 19
z = 0b1010010 # 82
```

Notez que le premier bit est plus « fort » que le dernier on dit qu'on est en « little endian ».

14.1.9 Manipuler des octets en Python

On peut construrie des listes d'octets en utilsant `bytearray` et une liste de nombres :

```
data = bytearray(
  [0b1100001,
   0b1100010,
   0b1100011
  ]
)

# equivalent:
data = bytearray([97,98,99])

# equivalent aussi:
data = bytearray([0x61, 0x62, 0x63]
```

14.1.10 Texte

On peut aussi interpréter des octets comme du texte - c'est la table ASCII

USASCII code chart

Bits b7 b6 b5					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b4	b3	b2	b1	Row ↓ Column →	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

La table se lit ainsi : si on veut connaître la suite de 0 et de 1 qui correspond à *B* : on lit les 3 premiers bits de haut en bas sur la colonne : *100*, puis les 4 bits sur la ligne : *0010*. Du coup "B" en s'écrit en 7 bits : *1000010*, soit 66 en décimal, et 42 en héxadécimal

14.1.11 ASCII - remarques

- C'est *vieux* - 1960
- Le A est pour American
- Ça sert à *envoyer* du texte sur des terminaux d'où les « caractères » non-imprimables dans la liste
- Mais c'est une convention *très* utilisée
- Un message de 4 lettres ASCII sera souvent envoyé comme 4 octets (même si seulement 7 bits sont nécessaires)

14.1.12 Utiliser ASCII en Python

Avec `chr` et `ord` :

```
x = chr(0x42)
print(x)
# affiche: B

x = ord('B')
print(x)
# affiche: 66
```

14.1.13 Affichage des bytearrays en Python

Python utilise ASCII pour afficher les bytearrays si les caractères sont « imprimables » :

```
data = bytearray([97,98,99])
print(data)
# affiche: bytearray(b"abc")
```

Note : Notez que Python rajoute quelque chose qui ressemble à un appel de fonction lorsqu'il affiche le bytearray : ce n'est pas un *vrai* appel de fonction.

Et `\x` et le code hexa sinon :

```
data = bytearray([7, 69,  76, 70])
print(data)
# affiche: bytearray(b"\x07ELF")
```

Notez bien que ce qu'affiche Python n'est qu'une *interpétation* d'une séquence d'octets.

14.1.14 Types

La variable *b »abc »* est une « chaîne d'octets », de même que *« abc »* est une « chaîne de caractères ».

Python apelle ces types *bytes* et *str* :

```
print(type("abc"))
# affiche: str

print(type(b"abc"))
# affiche: bytes
```

14.1.15 bytes et bytearray

De la même manière qu'on ne peut pas modifier un caractère à l'intérieur une string, on ne peut pas modifier un bit - ou un octet dans une variable de type *bytes* :

```
a = "foo"
# a[0] = "f" => TypeError: 'str' object does not support item assignment

b = b"foo"
# b[0] = 1 => TypeError: 'bytes' object does not support item assignment
```

Par contre on peut modifier un bytearray :

```
b = bytearray(b"foo")
b[0] = 103
print(b)
# affiche: bytearray(b"goo")
```

14.1.16 Conversion bytes - texte

Avec `encode()` et `decode()` :

```
text = "chaise"
encodé = text.encode("ascii")
print(encodé)
# affiche: b"chaise"

bytes = b"table"
décodé = bytes.decode("ascii")
print(décodé)
# affiche: b"table"
```

Notez que dans le deuxième exemple, on est bien en train de « décoder » un paquet de 0 et de 1. Il peut s'écrire ainsi :

bytes = b »x74x61x62x6cx65 » décodé = bytes.decode(« ascii ») print(décodé) # affiche : table

14.1.17 Plus loin que l'ASCII

Vous avez sûrement remarquer qu'il n'y a pas de caractères accentués dans ASCII. Du coup, il existe d'autres *conventions* qu'on appelle « encodage ».

On peut spécifier l'encodage quand on appelle la méthode `decode()` :

```
# latin-1: utilisé sur certains vieux sites
data = bytearray([233])
lettre = data.decode('latin-1')
print(lettre)
# affiche: 'é'

# cp850: dans l'invite de commande Windows
data = bytearray([233])
lettre = data.decode('cp850')
print(lettre)
# affiche: 'Ú'
```

Notez que la même suite d'octets a donné des résultats différents en fonction de l'encodage !

14.1.18 Unicode

L'Unicode c'est deux choses :

1. Une **table** qui associe un un « codepoint » à chaque caractère
2. Un encodage particulier, l'UTF-8, qui permet de convertir une suite d'octets en suite de codepoint et donc de caractères

14.1.19 UTF-8 en pratique

D'abord, UTF-8 est compatible avec ASCII :

```
encodé = "abc".encode("utf-8")
print(encodé)
# Affiche: b'abc'
```

Ensuite, certains caractères (comme `é`) sont représentés par 2 octets :

```
encodé = "café".encode("utf-8")
print(encodé)
# Affiche: b'caf\xc3\xa9"
```

Enfin, certains caractères (comme les emojis) sont représentés par 3 voire plus octets.

> **Avertissement :** Toutes les séquences d'octets ne sont pas forcément valides quand on veut les décoder en UTF-8

14.1.20 Conséquences

— On peut représenter *tout* type de texte avec UTF-8 (latin, chinois, coréen, langues disparues, ...)
— On ne peut pas accéder à la n-ème lettre directement dans une chaîne encodée en UTF-8, il faut parcourir lettre par lettre (ce qui en pratique est rarement un problème).

14.2 Fichiers

14.2.1 Ouvrir un fichier en lecture

Fichiers « textes » et fichiers « binaires »

Cela vous est peut-être déjà arrivé : Imaginons que vous ayez dans votre répertoire courant un code source python danse `mon_script.py` et un pdf dans `cours.pdf`,

Vous pourrez ouvrir `mon_script.py` dans un éditeur de texte, mais pas `cours.pdf` - ou alors ça affichera n'importe quoi.

En Pyton, on utilise la fonction native `open()`, en passant en argument le chemin du fichier.

Selon que l'on veuille accéder au *texte* dans le fichiers ou aux données binaires qui sont à l'intérieur, on utilise l'argument `"r"` ou `"rb"` ("r" comme "read", et "b" comme "binary")

Enfin, `open()` renvoie un « file-objet », qu'on note souvent "f", qui contient une méthode `read()` pour lire le contenu du fichier.

En pratique, voilà ce que cela donne :

```
f = open("mon_script.py", "r")
code = f.read()
print(code)
# affiche le code dans le fichier foo.py

f = open("cours.pdf", "rb")
données = f.read()
# données est mainteant une grosse suite
# d'octets

f = open("cours.pdf", "r")
f.read()
# Erreur: UnicodeDecodeError: 'utf-8' codec can't
# decode byte 0xd0 in position 10
```

Comme on doit utilisé l'option `rb` pour lire le pdf, on dit parfois que le fichier `pdf` est un fichier « binaire », par opposition avec `mon_script.py` qui est un fichier « texte ».

Ça n'a pas vraiment de sens : les deux fichiers sont stockés sur votre ordinateur comme des suites d'octets, indépendamment de leur contenu.

Il se trouve que l'un des deux contient une suite d'octets qui est décodable en tant que string. En fait, sous le capot, la suite d'octets renvoyée dans le premier example a été décodée avec l'encodage par défaut de votre système. On peut d'ailleurs passer l'encodage en argument à `open()` :

```
f = open("vieux_texte_en_latin_1.txt", "r", encoding="latin-1")
texte = f.read()
```

14.2.2 Ouvrir un fichier en écriture

On peut aussi *écrire* dans un fichier, toujours avec `open()`, mais cette fois avec la méthode `write()` du file-objet.

On peut écrire du texte avec l'option `"w"` et une chaîne de caractères

```
f = open("mon_script.py", "w")
f.write("Nouveau contenu!")
```

Et écrire directement des données binaires avec `"wb"` et une suite d'octets

```
f = open("cours.pdf", "wb")
f.write(b"\x0c\x1f...")
```

Encore une fois, sous le capot, la chaîne de caractères sera encodée par Python avant d'être écrite dans le fichier texte

14.2.3 Fermeture des file-objets

Notez qu'il est impératif de fermer les fichiers que vous ouvrez - que ce soit en lecture ou en écriture, en appelant la méthode `close()` :

```
f = open("mon_poéme.py", "w")
f.write(premier_vers)
f.write(deuxième_vers)
f.close()
```

14.2.4 Conseils

— On utilise souvent le binaire pour échanger entre Python et le monde extérieur
— Tout texte a un *encodage*, et il vous faut connaître cet encodage pour travailler avec
— Si vous avez le choix, utilisez UTF-8

CHAPITRE 15

Chapitre 15 - L'interpréteur interactif

15.1 Démarrage

Jusqu'ici, on a toujours lancé la commande Python avec `python` suivi du nom d'un fichier source.

Il est égalemnet possible de lancer la commande `python3` sans argument.

Dans ce cas, on se retrouve avec une **autre** invite de commandes :

```
$ python3
Python 3.7.1 (default, Oct 22 2018, 10:41:28)
[GCC 8.2.1 20180831] on linux
Type "help", "credits" or "license" for more information.
>>>
```

15.2 Deux invites de commandes

Notez les trois chevrons : `>>>`. Cela vous permet de différencier l'invite de commandes du système d'exploitation de celle de Python.

— Système d'exploitation -> Python : taper `python3` (sans arguments)
— Python -> Système d'exploitation : taper `quit()`

15.3 Fonctionnement de l'interpréteur

L'interpréteur fonctionne dans une boucle :

1. Lire le code qui a été tapé (soit une ligne, soit une succession de blocs)
2. Évaluation du code qui a été entré
3. Affichage de la valeur le cas échéant
4. Retour au début

(En anglais, on dit « REPL » - read/eval/print/loop)

Example d'une session interactive :

```
>>> a = 42
>>> a
42
>>> b = 4
>>> a + b
46
```

Notez que si la variable est None, l'interpréteur n'affiche rien :

```
>>> def ne_fait_rien():
        pass
>>> résultat = ne_fait_rien()
>>> résultat
>>> résultat is None
True
```

15.4 Interpréteur interactif et imports

Recréons un fichier `bonjour.py` contenant :

```
# dans bonjour.py
salutation = "Bonjour,"
def dire_bonjour(nom):
     print(salutation, nom)
```

On peut démarrer un interpréteur interactif dans le répertoire contenant le fichier `bonjour`.py`, et « jouer » avec les attributs du module bonjour :

```
>>> import bonjour
>>> bonjour.dire_bonjour("Bob")
Bonjour, Bob
```

```
>>> bonjour.salutation = "Hello,"
>>> bonjour.dire_bonjour("Bob")
Hello, Bob
```

Avertissement : Si le contenu de bonjour.py change, il faut *relancer* l'Interpréteur intéractif et refaire l'import.

CHAPITRE 16

Chapitre 16 - Héritage

16.1 Rappel - composition

Dans un chapitre précédent on a parlé de *composition* qui décrit une classe à l'intérieur d'une autre classe.

Pour rappel :

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

    def ronronne(self):
        print(self.nom, 'fait: "prrrrr"')

    def caresse(self):
        self.ronronne()


class Enfant:
    def __init__(self, prénom, chat):
        self.chat = chat

    def console(self):
        self.chat.caresse()
```

16.2 Vocabulaire

Ici on va parler d'héritage, qui décrit une autre relation entre classes, appelée parfois un peu abusivement « partage de code ».

Pour indiquer qu'une classe `B` hérite d'une classe `A`, on écrit `A` dans des parenthèses au moment de déclarer la classe `B` :

```
class A:
    ...

class B(A):
     ...
```

Les trois formulations suivantes sont souvent employées :

— A est la classe *parente* de B.
— B *hérite* de A.
— B est une classe *fille* de A.

16.3 Utilisation

Si une méthode n'est pas trouvée dans la classe courante, Python ira la chercher dans la classe parente :

```
class A:
   def méthode_dans_a(self):
       print("dans A")

class B(A):
   def méthode_dans_b(self):
       print("dans B")


b = B()
b.méthode_dans_b()
# Affiche: 'dans B', comme d'habitude

b.méthode_dans_a()
# Affiche: 'dans A'
```

16.4 Ordre de résolution

S'il y a plusieurs classes parentes, Python les remonte toutes une à une. On dit aussi qu'il y a une *hiérarchie* de classes :

```
class A:
    def méthode_dans_a(self):
        print("dans A")

class B(A):
    def méthode_dans_b(self):
        print("dans B")

class C(B):
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def méthode_dans_c(self):
         print("dans C")

c = C()
c.méthode_dans_a()
# affiche: 'dans A'
```

16.5 Avec __init__

La résolution fonctionne pour toutes les méthodes, y compris `__init__` :

```
class A:
    def __init__(self):
         print("initialisation de A")

class B(A):
    ...

b = B()
# affiche: "initialisation de A"
```

16.6 Attributs

Même mécanisme pour les attributs :

```
class A:
    def __init__(self):
        self.attribut_de_a = 42

class B(A):
    ...

b = B()
print(b.attribut_de_a)
# affiche: 42
```

16.7 Surcharge

On peut aussi *surcharger* la méthode de la classe parente dans la classe fille :

```
class A:
   def une_méthode(self):
       print("je viens de la classe A")

class B(A):
    def une_méthode(self):
        print("je viens de la classe B")
```

(suite sur la page suivante)

(suite de la page précédente)

```
b = B()
b.une_méthode()
# affiche: "je viens de la classe B'
```

16.8 super()

On peut utiliser `super()` pour chercher *explicitement* une méthode dans la classe parente :

```
class A:
   def une_méthode(self):
       print("je viens de la classe A")

class B(A):
    def une_méthode(self):
        super().une_méthode()
        print("je viens de la classe B")

b = B()
b.une_méthode()
# affiche:
# je viens de la classe A
# je viens de la classe B
```

16.9 super() et __init__

Erreur très courante :

```
class A:
   def __init__(self):
       self.attribut_de_a = "bonjour"

class B(A):
    def __init__(self):
       self.attribut_de_b = 42

 b = B()
 print(b.attribut_de_b)
 # affiche: 42
 print(b.attribut_de_a)
 # erreur:  AttributeError
```

On a surchargé `A.__init__()`, du coup l'initialisation de A n'a jamais été faite.

La plupart du temps, si `A` et `B` ont de constructeurs, on appellera `super().__init__()` dans le constructeur de la classe fille :

```
class A:
   def __init__(self):
       self.attribut_de_a = "bonjour"

class B(A):
    def __init__(self):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        super().__init__()
        self.attribut_de_b = 42

b = B()
print(b.attribut_de_b)
# affiche: 42
print(b.attribut_de_a)
# affiche:  "bonjour"
```

CHAPITRE 17

Chapitre 17 - Décorateurs

17.1 Jouons avec les fonctions

17.1.1 Introduction

Reprenons ce qu'on a vu jusqu'ici.

D'une part, on peut créer des variables en assignant des valeurs à celles-ci :

```
# Création d'une variable `x` avec la valeur 4
x = 4
```

D'autre part, on peut définir et appeler des fonctions :

```
# Définition de la fonction:
def dire_bonjour(nom):
    print("Bonjour " + nom)

# Appel
dire_bonjour("Max")

# Affiche: "Bonjour Max"
```

17.1.2 Fonctions en tant que variables

Il se trouve qu'en Python, on peut assigner des fonctions à des variables. C'est différent d'assigner le résultat de l'appel à une fonction à une variable, et ça permet de retarder l'appel :

```
# Définition d'une fonction `dire_bonjour_en_français`
def dire_bonjour_en_français(nom):
    print("Bonjour " + nom)

# Définition d'une fonction `dire_bonjour_en_anglais`
def dire_bonjour_en_anglais(nom):
    print("Hello " + nom)

# Assigne une fonction à la variable - aucune fonction
# n'est appelée à ce stade.
ma_fonction_qui_dit_bonjour = dire_bonjour_en_français

# Appel de la fonction (retardé)
ma_fonction_qui_dit_bonjour("Max")

# Affiche: Bonjour Max
```

De façon cruciale, notez que l'on n'a *pas* mis de parenthèses à droite lorsqu'on a créé la variable *ma_fonction_qui_dit_bonjour*.

On peut donc dire que lorsqu'on définit une fonction avec `def ma_fonction()` et un corps : il y a en réalité deux étapes :

1. Python stocke le corps de la fonction quelque part
2. Il assigne le corps de celle-ci à une variable dont le nom est `ma_fonction`.

En Python, il est assez fréquent d'utiliser de code tel que celui-ci, souvent avec un dictionnaire :

```
fonctions_connues = {
   "français": dire_bonjour_en_français,
   "anglais": dire_bonjour_en_anglais,
}

# Ici on stocke la langue parlée par l'utilisateur
# et son prénom
langue_parlée = ...
prénom = ....

if langue_parlée in fonctions_connues:
    fonction = fonctions_connues[langue_parlée]
    fonction(prénom)
```

17.1.3 Fonctions en tant qu'argement d'autres fonctions

On a vu en début de chapitre qu'on peut assigner des fonctions à des variables.

Du coup, rien n'empêche de passer des fonctions en *argument* d'autres fonctions.

Par exemple :

```
def appelle_deux_fois(f):
    f()
    f()
```

(suite sur la page suivante)

(suite de la page précédente)

```
def crier():
    print("Aline !")

appelle_deux_fois(crier)

# Affiche:
# Aline !
# Aline !
```

17.1.4 Fonctions imbriquées

On peut aussi définir une fonction dans une autre fonction :

```
def affiche_message(message):
    def affiche():
        print(message)
    affiche()

affiche_message("Bonjour")
# affiche: Bonjour
```

Deux notes importantes :

Premièrement, la fonction *affiche()* qui est imbriquées dans *affiche_message()* n'est pas accessible à l'éxtérieur de la fonction qui la contient. En d'autres termes, ce code ne fonctionne pas :

```
def affiche_message(message):
    def affiche():
        print(message)

affiche()
# NameError: 'affiche' is not defined
```

C'est un mécanisme similaire aux *portées des variables* vu précédemment.

Deuxièment, la fonction *affiche()* à l'intérieur de *affiche_message()* a accès à l'argument *message* de la fonction *affiche_message*. On appelle ça une « closure ».

17.1.5 Fonctions retournant des fonctions

En réalité, on combine souvent les closures avec des fonctions qui retournent d'autres fonctions :

```
def fabrique_fonction_qui_additionne(n):
    def fonction_résultat(x):
        return x + n
    return fonction_résultat


additionne_2 = fabrique_fonction_qui_additionne(2)
y = additionne_2(5)
print(y)
# Affiche: 7
```

17.1.6 Un autre paradigme

Le fait qu'on puisse traiter les fonctions comme n'importe quelle autre valeur (c'est-à-dire les assigner à des variables, les passer en argument et les retourner), est caractéristique des langages dits « fonctionnels ». Python est donc **à la fois** un langages *impératif*, *objet* et *fonctionnel*. On dit que c'est un langage *multi-paradigme*.

17.2 Décorateurs

17.2.1 Définition

Un décorateur est une fonction qui *enveloppe* une autre fonction.

On place le nom du décorateur avec une arobase (`@`) au-dessus de la fonction décorée :

```
def mon_décorateur(fonction):
    def fonction_retournée():
        # fait quelque chose avec l'argument `fonction`, par exemple
        # l'appeler avec un argument:
        fonction(42)
    return fonction_retournée

@mon_décorateur
def ma_fonction_décorée(un_argument):
    fais_un_truc_avec(un_argument)
```

Les deux dernières lignes sont équivalentes à :

```
def ma_fonction_décorée(un_argument):
    fais_un_truc_avec(un_argument)

ma_fonction_décorée = mon_décorateur(ma_fonction_décorée)
```

17.2.2 Exemples de décorateurs

On peut faire un décorateur qui nous empêche d'appeler une fonction sous certaines conditions :

```
def pas_pendant_la_nuit(fonction):
    def résultat():
        if il_fait_nuit:
            print("chut")
        else:
            fonction()
    return résultat

@pas_pendant_la_nuit
def dire_bonjour():
    print("bonjour")

il_fait_nuit = True
dire_bonjour()
# afiche: "chut"
```

17.2.3 Décorateur prenant des arguments

On peut aussi avoir des arguments passés aux décorateurs. Dans ce cas, on a besoin de *trois* fonctions imbriquées. En effet, il nous faut une fonction pour traiter l'argument `message` et une autre pour traiter l` argument `fonction` :

```
def affiche_message_avant_appel(message):
    def décorateur(fonction):
        def résultat():
            print(message)
            fonction()
        return résultat
    return décorateur

@affiche_message_avant_appel("dire_bonjour est appelée")
def dire_bonjour():
    print("bonjour")

dire_bonjour()
# affiche:
# dire_bonjour est appelée
# bonjour
```

CHAPITRE 18

Chapitre 18 - Exceptions

On a souvent montré des exemples de code qui provoquaient des erreurs au cours des chapitres précédents. Il est temps maintenant de se pencher davantage sur celles-ci.

18.1 Pile d'appels

Reprenons un exemple de code qui provoque une erreur, par exemple en essayant de diviser par zéro :

```
def mauvaise_fonction():
      return 1 / 0

def fonction_intermédiaire():
    mauvaise_fonction()

def fonction_principale():
    fonction_intermédiaire()

fonction_principale()
```

Si on lance ce code, voilà ce qu'on obtient :

```
Traceback (most recent call last):
  File "mauvaises_maths.py", line 13, in <module>
    fonction_principale()
  File "mauvaises_maths.py", line 10, in fonction_principale
    fonction_intermédiaire()
  File "mauvaises_maths.py", line 6, in fonction_intermédiaire
    mauvaise_fonction()
  File "mauvaises_maths.py", line 2, in mauvaise_fonction
    return 1 / 0
ZeroDivisionError: division by zero
```

Ceci s'appelle une *pile d'appels*. Elle permet de voir exactement par quelles fonction on est passé et dans quel ordre. Elle se lit de haut en bas :

— On appelé *fonction_principale()*
— Cette fonction a à son tour appelé *fonction_intermédiaire()*
— *fonction_intermédiaire()* à appelé *mauvaise_fonction()*
— *mauvaise_fonction()* a levé une exception

Notez que chaque élément de la pile comprend :
— le nom de la fonction
— le chemin du module la contetant
— le numéro et la ligne précise du code qui a été appelé

Il est important de bien lire les piles d'appels quand on cherche à comprendre d'où vient une exception.

Après la pile d'appels, on a le *nom* de l'exception et sa *description*.

18.2 Exceptions natives

Les exceptions sont toujours des instances de classes, et les classes d'exceptions héritent toujours de la class `BaseException`.

Le nom de l'exception est en réalité le nom de la classe, ici l'exception levée par la ligne `return 1 / 0` est une instance de la classe `ZeroDivisionError`.

Cette exception fait partie des nombreuses exceptions préféfinies en Python. Ensemble, elles forment une *hiérarchie* dont voici un extrait :

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- Exception
      +-- ArithmeticError
      |    +-- ZeroDivisionError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- OSError
      |    +-- FileNotFoundError
      +-- TypeError
      +-- ValueError
```

18.2.1 IndexError et KeyError

`IndexError` est levée quand on essaye d'accéder à un index trop grand dans une liste :

```
ma_liste = ["pomme"]
ma_liste[2] = "abricot"

# IndexError: list assignment index out of range
```

`KeyError` est levée quand on essaye d'accéder à une clé qui n'existe pas dans un dictionnaire :

```
scores = { "Alice" : 10 }
score_de_bob = scores["Bob"]

# KeyError: 'Bob'
```

Notez que la description de `KeyError` est la valeur de la clé manquante.

18.2.2 ValueError

`ValueError` est levée (entre autres) quand on tente une maunvaise conversions :

```
entrée_utilisateur = "pas un nombre"
valeur = int(entrée_utilisateur)
```

18.2.3 KeyboardInterrupt

`KeyboardInterrupt` est levée quand on fait `ctrl-c`.

18.2.4 FileNotFoundError

`FileNotFoundError` est levée quand on essaye d'ouvrir en lecture un fichier qui n'exsiste pas :

```
with open("fichier-inexistant.txt", "r") as f:
    contenu = f.read()
```

18.3 Gestion des exceptions

18.3.1 Bloc try/except

On peut *gérer* (ou *attraper*) une exception en utilisant un bloc `try/except` et le nom d'une classe d'exception :

```
try:
    a = 1 / 0
except ZeroDivisionError:
    print("Quelqu'un a essayé de diviser par zéro!")

# Affiche: Quelqu'un a essayé de diviser par zéro!
```

À noter : le bloc dans `try` s'interrompt dès que l'exception est levée, et on ne passe dans le bloc `except` que si une exception a effectivement été levée.

```
x = 14
y = 0
try:
    z = x / y
    print("z vaut", z)
except ZeroDivisionError:
    print("Ouelqu'un a essayé de diviser par zéro!")

# Affiche: Ouelqu'un a essayé de diviser par zéro!
```

Notez que la ligne `print("z vaut", z)` n'as pas été exécutée.

Autr exemple :

```
x = 14
y = 2
try:
    z = x / y
```

(suite sur la page suivante)

(suite de la page précédente)

```
    print("z vaut", z)
except ZeroDivisionError:
    print("Ouelqu'un a essayé de diviser par zéro!")

# Affiche: 'z vaut 7.0'
```

Notez que la ligne `print("Ouelqu'un a essayé de diviser par zéro!")` n'as pas été exécutée.

18.3.2 Gestion de plusieurs exceptions

Le mot après `except` doit être celui d'une classe, et l'exception n'est gérée que si sa classe est **égale ou une fille** de celle ci.

Par exemple, ceci fonctionne car `ZeroDivisionError` est bien une fille de la classe `ArithmeticError` :

```
x = 14
y = 0
try:
    z = x / y
    print("z vaut", z)
except ArithmeticError:
    print("Ouelqu'un a essayé une opération impossible")
```

On peut aussi mettre plusieurs blocs de `except` :

```
try:
    tente_un_truc_risqué()
except ZeroDivisionError:
    print("raté : division par zéro!")
except FileNotFoundError:
    print("raté : fichier non trouvé")
```

Ou gérer des exception de classes différentes avec le même bloc :

```
try:
    tente_un_truc_risqué()
except (ZeroDivisionError, FileNotFoundError)
    print("raté!")
```

18.3.3 Accéder à la valeur de l'exception

On peut récupérer l'instance de l'exception levée avec le mot-clé `as` :

```
try:
    ouvrir_fichier()
except FileNotFoundError as e:
    print("le fichier: ", e.filename, "n'existe pa")
```

Ici on utilise l'attribut `filename` de la classe `FileNotFoundError` pour afficher un message d'erreur

18.4 Levée d'exceptions

18.4.1 raise

On peut lever explicitement un exception en appelant le mot-clé `raise` suivi d'une **instance** d'une classe.

Par exemple en utilisant une exception native :

```
def dire_bonjour(prénom):
    if not prénom:
        raise ValueError("prénom vide")
```

18.4.2 Définition d'exceptions à la carte

On peut ré-utiliser les exceptions natives, ou définir sa propre classe :

```
class OpérationImpossible(Exception):
    pass


def ma_fonction():
    if cas_impossible:
        raise OpérationImpossible()
```

18.4.3 Gérer puis re-lever l'exception géré

Parfois il est utile de re-lever l'exception qu'on vient de géner.

Dans ce cas, on utilise `raise` sans argument :

```
try:
    tente_un_truc_risqué()
except ArithmeticError:
    ...
    raise
```

18.5 Else et finally

18.5.1 else

Si on rajoute un bloc `else` après le `except`, le bloc n'est éxécuté que si *aucune* exception n'a été levée :

```
try:
    tente_un_truc_risqué()
except (ZeroDivisionError, FileNotFoundError):
    print("raté")
else:
    print("ouf - ça a marché")
```

18.5.2 finally

Si on rajoute un bloc `finally` après le `except`, le bloc est éxécuté *dans tous les cas*, qu'une exception ait été levée ou non. On s'en sert souvent pour « annuler » du code qui aurait été utilisé dans le bloc `try` :

```
personnage = Personnage()
try:
    personnage.entre_en_scène()
    personnage.tente_un_truc_risqué()
except ZeroDivisionError:
    print("raté")
finally:
    personnage.quitte_la_scène()
```

Si dans le bloc `try` une exception **différente** de `ZeroDivisionError` est levée, on passera *quand même* dans le bloc `finally`, *puis* l'exception sera levée à nouveau.

TODO : exemple plus complet

CHAPITRE 19

Chapitre 19 - Objets

On a déjà parlé de *programmation* orientée objet en Python, mais pour l'instant on a vu que des classes et des instances.

Or le concept d'objet existe bel et bien en Python, et il est plus que temps de vous le présenter.

Mais avant, il faut faire un petit détour par les attributs et méthodes de classes.

19.1 Attributs et instances de classe

19.1.1 Rappels

Voici un exemple de classe qui contient une méthode :

```
class MaClasse
    def ma_méthode(self):
        print(self.mon_attribut)
```

Le bloc indenté en-dessous du mot-clé `class` s'appelle le *corps* de la classe. Et les méthodes sont définies avec le mot-clé `def` dans le corps de la classe.

On dit que ce sont des *méthodes d'instance* par ce qu'il faut créer une instance pour pouvoir les appeler :

```
mon_instance = MaClasse()
mon_instance.ma_méthode()
```

19.1.2 Attributs de classes

On peut également déclarer des variables dans le corps d'une classe.

On crée ainsi des *attributs de classe* :

```
class MaClasse:
    mon_attribut_de_classe = 42
```

Ici `mon_attribut_de_classe` existe *à la fois* dans les instances de `MaClasse` et dans la classe elle-même :

```
print(MaClasse.mon_attribut_de_classe)
# affiche 42
mon_instance = MaClasse()
print(mon_instance.mon_attribut_de_classe)
# affiche 42
```

Un point important est que les attributs de classe sont *partagés* entre toutes les instances. Voici un exemple d'utilisation possible :

```
class Voiture:
    nombre_total_de_voitures_fabriquées = 0

    def __init__(self, marque, couleur):
        print("Construction d'une", marque, couleur)
        Voiture.nombre_total_de_voitures_fabriquées += 1


ferrari_1 = Voiture("Ferrari", "rouge")
mercedes_1  = Voiture("Mercedes, "noire")
ferrari_2 = Voiture("Ferrari", "rouge")
print("total:", Voiture.nombre_total_de_voitures_fabriquées)
# Affiche:
# Construction d'une Ferrari rouge
# Construction d'une Mercedes noire
# Construction d'une Ferrari rouge
# total: 3
```

Notez que pour changer l'attribut de classe depuis une méthode, (comme dans le méthode `__init__` ci-dessus) on utilise le nom de la classe directement, et non pas `self`.

19.1.3 Méthodes de classes

On peut aussi définir des méthodes de classes avec le décorateur *classmethod*

Dans ce cas, le permier argument s'appelle `cls` et prend la valeur de la *classe* elle-même. Pour poursuivre sur notre exemple :

```
class Voiture:
    nombre_total_de_voitures_fabriquées = 0

    def __init__(self, marque, couleur):
        print("Construction d'une", marque, couleur)
        Voiture.nombre_total_de_voitures_fabriquées += 1

    @classmethod
    def fabrique_ferrari(cls):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return cls("ferrari", "rouge")


ferrari = Voiture.fabrique_ferrari()
```

Détaillons ce qu'il se passe sur la dernière ligne : à gauche du égal il y a une variable et à droite une expression(`Voiture.fabrique_ferrari()`)

L'expression est constitué d'une classe à gauche du point (`Voiture`) et d'un attribut à droite du point `fabrique_ferrari` suivi de parenthèses.

Comme `fabrique_ferrari` est une méthode de classe, on va appeler la méthode de classe `fabrique_ferrari` en lui passant la classe Courante en argument.

On arrive ainsi dans le corps de la méthode de classe `fabrique_ferrari`, et `cls` vaut la classe *Voiture*.

Finalement, on évalue l'expression `cls("ferrari", rouge")` en remplaçant `cls` par sa valeur, ce qui donne `Voiture("ferrari", "rouge")` qui correspond bien à ce qu'on obtient : une instance de la classe Voiture.

19.2 Objets

En fait, *tout ce qu'on manipule en Python* est un objet. Et tous les objets sont toujours des instances d'une classe - on peut accéder à la classe qui a servi à instancier un objet avec la fonction `type`, par exemple :

```
class MaClasse:
    pass

mon_instance = MaClasse()
print(type(mon_instance))
# Affiche:
# <class '__main__.MaClasse'>
```

Mais aussi :

```
print(type(2))
# affiche: int

print(type("bonjour"))
# affiche: str
```

Donc en Python, les entiers sont des instances de la classe `int`, et les strings des instances de la classe `str`.

Ainsi, vous pouvez voir l'expression `x = str(y)` de deux façons :

- Soit on appelle la fonction native `str` pour convertir `y` en string
- Soit on crée une nouvelle instance de la classe `str` en appelant le constructeur de la classe `str` avec `y` en argument.

Notez que ça ne s'arrête pas là :

```
def ma_fonction():
    pass

print(type(ma_fonction))
# affiche: function

class MaClasse:
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def ma_méthode(self):
        pass

mon_instance = MaClasse()
print(type(mon_instance.ma_méthode))
# affiche: method

import sys
print(type(sys))
# affiche: module
```

Et même :

```
print(type(MaClasse))
# affiche: type

print(type(type))
# affiche: type
```

Et oui, les classes elles-mêmes sont des instances de classe ! (la classe `type`)

Du coup en Python, le terme "objet" désigne *toujours* une instance de classe - même `None` est une instance d'une classe (elle s'appelle `NoneType`).

19.2.1 Ordre de résolution

Il est temps de revenir sur l'évaluation des expressions impliquant des attributs.

On a vu trois systèmes différents :

Appeler une fonction définie dans un module :

```
import mon_module
mon_module.ma_fonction()
```

Appeler une méthode d'instance définie dans une classe :

```
mon_instance = MaClasse()
mon_instance.ma_méthode()
```

Appeler une méthode de classe définie dans une classe :

```
MaClasse.ma_méthod_de_classe()
```

D'après ce qu'on a vu dans la section précédente, on sait maintenant que dans tous les cas, à gauche du point se situe un objet, et que tous les objets sont des instances d'une classe (appelé le « type » de l'objet).

Pour évaluer l'expression `mon_objet.mon_attribut`, où *mon_objet* ` est une instance de `mon_type`, Python cherche toujours l'attribut dans deux endroits :

- D'abord en tant qu'attribut de l'instance `mon_objet`
- Ensuite, en tant qu'attribut de la classe `mon_type`

La recherche se poursuit ainsi en suivant toutes les classe parentes de `mon_type`.

On peut voir ce mécanisme en action dans le code suivant :

```
class A:
    def f1(self):
        print("f1 dans A")

    def f2(self):
        print("f2")


class B(A):
    @classmethod
    def f3(cls):
        print("f3")

    def f1(self):
        print("f1 dans B")


b = B()
b.f1()
b.f3()
b.f2()
# affiche:
# f1 dans B
# f3
# f2
```

19.2.2 Conclusion

Maintenant vous devriez comprendre pourquoi on dit parfois qu'en Python, **tout est objet**.

Dans un prochain chapitre, on expliquera pourquoi en plus de cela on peut dire qu'en Python, **tout est dictionnaire**.