Python base

m2iformation.fr

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Python base

Présentation de Python et ses versions

- Le langage de programmation Python a été créé en 1989 par Guido Van Rossum
- La dernière version de Python est la version 3. Plus précisément, la version 3.12 qui a été publiée en Octobre 2023.
- La version 2 de Python est obsolète et n'est plus maintenue depuis le 1er Janvier 2020
- La Python Software Foundation est l'association qui organise le développement de Python et anime la communauté de développeurs et d'utilisateur.

Présentation de Python et ses versions

- Python est :
  - Multiplateforme
  - Un langage de haut niveau
  - Un langage interprété
  - Orienté objet
- Usage de Python :
  - Scripts pour automatiser des tâches
  - Analyses de données, Développement web ect...

Environnement de développement

- Par défaut, Python est déjà installé sur Linux et MacOS
- Pour Windows, il faudra le télécharger à cette adresse :
  - Download Python | Python.org
  - /!\ Attention, penser à cocher les bonnes cases à l'installation (notamment l'ajout au PATH sur windows)
- Pip est le gestionnaire de paquets de Python et qui est systématiquement présent depuis la version 3.4.

Environnement de développement

- Un script python est un fichier avec l'extension .py
- Pour démarrer un script python, on utilise la commande py (Windows) ou python/python3 (MacOS et Linux) et le nom du script.
- Exemple :
  - python mon_script.py

Environnement de développement

- L'interpréteur de Python est l'application qui permet de convertir les instructions python en un langage compréhensible par l'ordinateur.
- Il peut être utilisé pour exécuter une instruction.
- Pour ouvrir l'interpréteur, il suffit de taper dans un terminal (powershell, bash,...) py ou python3.
- Pour quitter l'interpréteur, Ctrl + D ou la fonction exit() de python.

Environnement de développement (IDE)

- Un IDE (Integrated Development Environment) est un environnement de développement intégré, il offre un ensemble d'outils tel que :
  - Un éditeur de code
  - Un compilateur
  - Un débogueur ect...

Environnement de développement (IDE)

- Il convient d'installer l'un des IDE suivant :
  - Pycharm (JetBrains, conçut spécialement pour python, avec une version payante améliorée).
  - Visual Studio Code (Microsoft, utile dans de nombreux domaines et avec de nombreuses extensions disponibles)
- Lors de l'installation, il peut être utile pour les 2 logiciels de cocher les cases permettant de faire "Ouvrir le projet avec ..." et l'ajout au PATH

Les normes et conventions en Python

- La syntaxe python est soumise à des conventions.
- Un bon développeur s'assurera qu'il les suit en écrivant ses scripts python.
- Elles permettent de normaliser le langage et de faciliter la lecture.
- Python suit la norme PEP8
- Les IDE ont souvent un auto-formatage qui respectera une grande partie de ces règles. Les raccourcis pour formater automatiquement changent selon le logiciel.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Variables

Les identificateurs

Les identificateurs sont des noms que l'on donne à des éléments de notre algorithme (variables, fonctions, classe, ...)

En Python, ils doivent respecter certaines règles :

- Ils doivent commencer par une lettre ou un underscore `_`
- Ils sont sensibles à la casse (`Nom` ≠ `nom`)
- Ils peuvent contenir des lettres, des chiffres et des `_`
- Les caractères spéciaux (`@`, `$`, `-`, etc) sont interdit.
- Pas de mot-clé réservé (`class`, `def`, `if`, etc)

Déclarer une variable

Pour déclarer une variable en Python, il suffit de donner un identificateur valide suivi de la valeur à lui affecter.

- L'opérateur **=** en python est l'opérateur **d'assignation/affectation**, il permet d'affecter à une variable une valeur donnée.
  - Ex: **ma_variable = 3** permet d'assigner la valeur entière 3 à la variable 'ma_variable'.
- Ainsi, lorsque l'on mettra **ma_variable** dans une instruction Python, on récupérera la valeur contenue dans cette variable
  - Ex: **ma_variable2 = ma_variable + 1** permet d'assigner la valeur entière 4 à la variable 'ma_variable2'.

Les variables numériques

- Les variables de type **numériques** servant à stocker des nombres. Ces variables peuvent être de plusieurs sous-types :
  - les integers **int**, qui servent à stocker des nombres **entiers**
  - les **float**s, servant à stocker des **nombres à virgule** flottante (décimaux)

```
mon_int = 514 # Variable de type integer
mon_float = 3.14 # Variable de type float
```

Les variables

- Les chaines de caractères strings **str** qui permettent de stocker du **texte**.
- Les **bool**éens, permettant de stocker des valeurs binaires (Vraie=**True** ou Fausse=**False**)
- Le vide, **None** en python, est un type à part qui ne représente 'Rien', il n'est **ni un 0, ni un False**

```
mon_bool = True # Variable de type booléens
ma_string = "Je suis une chaîne de caractère" # Variable de type string.
```

Le mot clé del

- Le mot clé **del** permet de **supprimer de la donnée** en Python (supprime une référence).
- Il peut être utile lorsque l'on cherche à **libérer de la mémoire** ou à se débarrasser de certaines variables/fonctions/classes/...
- Si le nombre de référence associer à une variable tombe à 0 alors celle-ci sera libéré de la mémoire (via garbage collector)

Un peu de lexique

- Une **fonction** est un morceau de code déjà écrit, il prend des **paramètres** et retourne une **valeur**.
- On pourra **exécuter** ce morceau de code en ajoutant des parenthèses avec les **valeurs passées en paramètres** après le nom de la fonction.

```
# Exemple de fonction
print("Un message à afficher")
```

Un peu de lexique

- Une **méthode** est similaire à une fonction, la différence est qu'elle **s'applique** à une valeur/un objet donné, on ajoutera un "." après celui-ci suivi du **nom de la méthode** et des **parenthèses**.
- Lorsque l'on utilise ces 2 concepts, les **valeurs** que l'on met **entre les parenthèses** seront appelées **paramètres** ou **arguments**.

```
# Exemple de méthode
test_maj = "test".upper()
```

Un peu de lexique

- Les mots **console** et **terminal** reviennent à peu près à la même chose, il s'agit d'une **fenêtre** permettant la **communication** entre l'utilisateur et le programme par du **texte**.
- Un **script python** est un fichier contenant des instructions.
- Un **programme** ou une **application** correspond au processus, résultat de l'interprétation du script par l'ordinateur.
- un **module python** est le contenu de l'interprétation d'un script python à l'exécution, nous y reviendrons plus tard...

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Console

Récupérer et afficher des valeurs

Pour qu'il y ait un **dialogue** possible entre l'utilisateur et l'ordinateur (application console), on a recours à ce qui s'appelle des affichages et des **récupérations de valeurs**.

- **L'affichage** se fait par la fonction **print()**, qui affichera les valeurs passées en paramètre sur le terminal.
- **La récupération** se fait par la fonction **input()**, elle récupère une saisie de l'utilisateur sous forme de **str**. il est possible d'y ajouter une chaîne à afficher en paramètre.

```
ma_recuperation = input("Veuillez entrer une valeur : ")
print("Vous avez entrer comme valeur : ", ma_recuperation)
```

Ces deux processus amènent rapidement à **deux composantes essentielles** de la programmation console :

- **L'affichage** : le **formatage** des str.
- **la récupération** : le **casting** des variables.

Le formatage

- Si l'on veut **présenter** de façon claire des **informations** à l'utilisateur, il faut souvent se servir du **formatage**. Il existe plusieurs méthodes.
- Le %-**formating**, maintenant déprécié et utilisé de nos jours dans de rares cas (formatage de dates, requêtes SQL)
- les **f-strings**, ce sont des chaînes de caractères pour **formater directement le texte** en y incluant entre accolades les variables :

```
print(f"La valeur de nombre_a vaut {nombre_A:0.2f}") # nombre_a = 3.14
```

Fonctionnement des f-strings

- Lorsque l'on utilise un f-string, on retrouve souvent ce genre de syntaxe :

```
variable = 55.2091
f"{variable:^7.2f}" # 55.21
```

- Ici la valeur sera :
  - 7: Dans un espace de 7 caractères minimum au total (virgule, décimales, ...) on ajoutera le nombre d'espaces nécessaire si pas assez de caractères.
  - 2: avec toujours 2 chiffres après la virgule (arrondi si besoin)

Les raw-strings

- Similaire au f-strings, il existe aussi en python les **raw-strings**.
- Ce sont des chaînes dans lesquelles les **caractères spéciaux** comme le **backslash** ne sont pas interprétés.
- Ils facilitent la saisie des chemins de fichier ou de regex par exemple.

```
print("\n{1}")
print(f"\n{1}")
print(r"\n{1}")
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Cast

Cast des variables

Le casting d'une variable consiste à la **convertir le type d'une variable en au autre type**.

Pour ce faire, on se sert de ce qui s'appelle le "**cast**" (En français **transtypage**) en utilisant **la fonction de cast** qui porte **le nom du type vers lequel on souhaite passer**.

```
ma_string = "599.98"
mon_prix = float(ma_string)
print(f"Ma string vaut {ma_string} et est un type {type(ma_string)}")
# Ma string vaut 599.98 et est de type <class 'str'>

print(f"Mon prix vaut {mon_prix} et est un type {type(mon_prix)}")
# Mon prix vaut 599.98 et est de type <class 'float'>
```

Cast des variables

De plus, lorsque l'on cherche à obtenir un nombre de l'utilisateur, on peut également directement caster l'input de la sorte :

```
mon_nombre = int(input("Veuillez entrer un nombre : ")) # 25

print(f"Mon nombre vaut {mon_nombre} et est un type {type(mon_nombre)}")
# Mon prix vaut 25 et est de type <class 'int'>
```

- Attention ! Le casting peut être la source de nombreux problèmes générant ce que l'on appelle des **exceptions** ! Nous verrons comment traiter les exceptions plus tard.

Cast en booléens

- Lorsque l'on fait un **cast** en **bool**, python applique certaines règles:
  - Les valeurs **None, False, 0, 0.0** et "" donnent forcément **False**
  - **Toutes les autres valeurs donnent True**
- En réalité toute valeur correspondant au vide pour son type sera considérée comme False dans une condition (cf : **partie block conditionnels**)

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Opérateurs

Les opérateurs arithmétiques et d'affectation

Opérateur	Fonction
`+`	Addition
`-`	Soustraction
`/`	Division
`//`	Division entière
`*`	Multiplication
`**`	Puissance (exponentiation)
`%`	Modulo (reste de la division Euclidienne)

Opérateur	Fonction
`=`	Affectation classique
`+=`	Addition (à une variable)
`-=`	Soustraction (à une variable)
`/=`	Division (à une variable)
`//=`	Division entière (à une variable)
`*=`	Multiplication (à une variable)
`**=`	Puissance (à une variable)

Concaténation

La concaténation consiste à **assembler plusieurs chaînes de caractères** pour en former une seule. En Python, cela peut se faire avec l'opérateur `+`.

Exemple :

```
prenom = "Tom"
nom = "Cruise"
nom_complet = prenom + " " + nom
```

La réplication

- La **réplication** consiste à répéter une même chaine de caractère. En associant l'opérateur de **multiplication** * entre un **string** et un **int**, nous pourrons donc répéter celle-ci.

Exemple :

```
variable_a_repeter = "Hello "
presentation = variable_a_repeter * 3
print(presentation) # Hello Hello Hello
```

Les méthodes de la classe math

Python fournit de nombreuses fonctions mathématiques via le package [`math`] ou directement en natif. Les fonctions de de ce package sont disponibles en l'important via le mot-clé `import`

Quelques exemples :

```
x = math.sqrt(16.0)
y = math.cbrt(27.0)
z = pow(3)
a = abs(-42)
```

```
r = round(2.7)
c = math.ceil(2.1)
f = math.floor(2.9)
c = math.cos(angle)
```

```
minimum = min(a, b)
maximum = max(a, b)
```

Opérateurs de comparaison

Les opérateurs de comparaison s'utilisent entre **deux valeurs comparables** (nombres, chaînes, etc.) et renvoient toujours un **booléen** (`true` ou `false`).

Opérateur	Description	Exemple	Résultat
`==`	Égal à	`1 == 1`	`true`
`!=`	Différent de	`1 != 0`	`true`
`>`	Supérieur à	`4 > 2`	`true`
`>=`	Supérieur ou égal à	`4 >= 4`	`true`
`<`	Inférieur à	`2 < 5`	`true`
`<=`	Inférieur ou égal à	`4 <= 4`	`true`

Opérateurs logiques

Les opérateurs logiques s'utilisent entre **deux expressions booléennes** et permettent de combiner ou inverser des conditions.

Opérateur	Description	Exemple	Résultat
`and`	**ET logique**	`true` and `false`	`false`
`or`	**OU logique**	`true` or `false`	`true`
`^`	**OU exclusif**	`true` or `true`	`false`
`not`	**NON / négation**	not `true`	`false`

Table de vérité

A	B	A and B
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

A	B	A or B
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

A	B	A xor B
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Structures conditionnelles

Les structures conditionnelles

- Si l'on veut permettre à nos programme de prendre **des chemins différents** de manière **conditionnelle** (saisies, calculs, ...) on a recours aux **structures conditionnelles**
- Pour réaliser une structure conditionnelle, on se sert des **clauses/mots clés** suivants :
  - **if** : initialiser la structure de contrôle, on donne une **condition** et on **exécute** le bloc d'instruction si elle est **vraie**.
  - **elif** : ajouter une **autre condition** dans le cas où la précédente **n'aura pas été validée**. On peut enchaîner ainsi **autant de structures elif que l'on souhaite**.
  - **else** : toujours la **dernière partie** d'une structure de contrôle, son bloc est exécuté dans le cas où **aucune des conditions précédentes n'a été validée**.

Exemple

```
mon_age = int(input("Veuillez donner votre âge : "))

if mon_age >= 21 :
  print("Vous êtes majeur aux USA")
elif mon_age >= 18 :
  print("Vous êtes majeur en France")
else:
  print("Vous êtes mineur")
```

- Les clauses **elif et else** sont facultatives dans la structure conditionnelle. Nous n'en mettons que si l'on en a réellement le besoin.

Instruction pass

- L'instruction **pass** est une instruction à part de python.
- Elle ne fait **absolument rien !**
- Il est régulier que l'on s'en serve de manière provisoire dans un bloc (if, else, for, fonction...) où l'on ne compte pas mettre d'instructions pour le moment.

match case

- Lorsque l'on travaille avec **la structure conditionnelle**, il est fréquent d'avoir beaucoup de **elif** utilisant **la même variable**.
- Depuis la version **3.10** de python, il existe une nouvelle structure, le **pattern matching** ou structure **match...case**.

```
# Sans match case
if var == 1:
  print("une")
elif var == 2:
  print("deux")
else:
  print("ni une, ni deux")
```

```
# Avec match case
match var:
  case 1:
    print("une")
  case 2:
    print("deux")
  case _:
    print("ni une, ni deux")
```

Les ternaires

En python, il est possible d'utiliser ce qu'on appelle les **ternaire**, il s'agit d'une **expression** comportant une **condition**. On peut le comparer à une **structure conditionnelle** if.

- Il se structure comme suit :

```
# var = resultat1 if condition else resultat2
age = int(input("Saisir l'âge : "))
statut = "majeur" if age >= 18 else "mineur"
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Structures itératives

Les structures itératives

- Il existe en Python deux façons de faire des **boucles/structures d'itération**. Les instructions du bloc seront exécutées à chaque **itération** de celle-ci.
  - La boucle "**while**" (Tant que ...) qui sera exécutée **tant que la condition spécifiée est vraie**
  - La boucle "**for...in...**" (Pour chaque... dans...) qui sera exécutée **pour chaque élément** d'un ensemble de type **conteneur ou interval** (fonctionne aussi avec les str). Elle met chaque élément un à un dans une **variable**.

Exemple boucle while

```
compteur = 0
while(compteur < 5)
  print("On incrémente notre compteur")
  compteur += 1
  print(f"Notre compteur est à {compteur}")

while(True)
  print("Boucle Infinie !!")
```

- Attention au boucle infinie lors de l'utilisation des structures itératives.

Exemple boucle for

```
for _ in range(0, 10):
  print("Je me répète !")

for element in [0, 1, 2, 3, 4, 5]:
  print(element)

for item in range(1, 11)
  print("Je suis l'itération n° : ", item)
```

- Lorsque l'on ne se sert jamais de la variable en question, le norme est de la nommer par "_"

Les structures itératives

- Lorsque l'on utilise une structure itérative, on peut également utiliser des **mots clés** durant l'itération, tels que :
  - "**continue**" : On passe à **l'itération suivante** en se replaçant au **début de la boucle**. On **ignore** alors tout ce qui aurait dù se dérouler **après le mot-clé**.
  - "**break**" : On **sort immédiatement de la boucle** sans effectuer les instructions après le mot-clé et dans les itérations suivantes.

```
while True:
  valeur = input("Saisir STOP pour arrêter le programme :")
  if valeur == "STOP":
    break
  elif valeur.upper() == "STOP":
    print("EN UPPERCASE !")
    continue
  else:
    pass # Ce bloc est inutile, pass ne fait rien.
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Fonctions

Les Fonctions

- En programmation, les **fonctions** sont très utiles pour réaliser plusieurs fois la même opération au sein d'un programme.
- Une fonction effectue une tâche. Pour cela, elle **reçoit** éventuellement des **arguments** et **renvoie** une **valeur** ou **None** (Rien).
- Chaque fonction effectue en général une tâche **unique et précise**. Si cela se complique, il est plus judicieux d'écrire **plusieurs fonctions** (qui peuvent éventuellement s'appeler les unes les autres).
- Les valeurs passées à l'exécution de la fonction s'appellent des **arguments**.
- Les variables entre parenthèses qui **contiendront** ces valeurs sont les **paramètres**.

Les Fonctions

- Pour **définir** une fonction, Python utilise le mot-clé **def**.
- Si on souhaite que la fonction **renvoie** quelque chose, il faut utiliser le mot-clé **return**.
- Le nombre **d'arguments** que l'on peut passer à une fonction est **variable** et dépends du **nombre de paramètres**.
- Il est possible de passer un ou plusieurs argument(s) de manière **facultatives** et de leur attribuer une valeur par **défaut**. Pour cela, on utilise le **=**.

```python
def return_number(nombre: int):
  return nombre

res = return_number(2) # retourne 2

def carre(nombre=3):
  return nombre ** 2

res = carre() # retourne 9
```

Les Fonctions

- Lorsqu'on manipule des fonctions, il est essentiel de bien comprendre comment se comportent les variables. Une variable est dite **locale** lorsqu'elle est créée **dans une fonction**. Elle n'existera et ne sera visible que lors de l'exécution de la dite fonction.
- Une variable dite **globale** lorsqu'elle est créée dans le **programme principal**. Elle sera visible partout dans le programme.
- Lorsque l'on essaie de modifier une variable globale à l'intérieur d'une fonction, il sera obligatoire d'utiliser le mot clé **global**

```
a = 10
def fonction():
  global a
  a += 1
```

les générateurs

Avancé

- Les **générateurs** sont des **fonctions particulières** qui utilisent le mot clé **yield** dans leur corps.
- Elles sont capables de "**mettre en pause**" leur exécution et de retourner **plusieurs valeurs une à une** grâce à **yield**.
- Il existe une **syntaxe courte** nommée **generator expression** similaire à la **list comprehension**.

```
def gen_int(n):
    for i in range(n):
        yield i

gen_int5 = gen_int(5)

print(next(gen_int5))
print(gen_int5.__next__())

gen_int10 = (i for i in range(10))
print(next(gen_int10))
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Modules

Modules

- Un **module** est un ensemble d'instructions provenant d'un **script** et qui peut être (ré)utilisé par d'**autres scripts**.
- Intérêts : faciliter la réutilisation, la lisibilité, le débogage, le travail d'équipe
- Python vient avec un ensemble de modules natifs : scrypt, csv, datetime, math, ...
- Pour avoir la liste complète des modules fonction **help('modules')**.
- Python nous donne la possibilité de créer nos propres modules.

Modules

Un **module** contient donc **l'ensemble des variables et des fonctions**, définies par les instructions du **script**. Il est entièrement exécuté au moment de l'instruction **import**.

```
from math import pi

def circonference_cercle(rayon):
  return 2 * pi * rayon
```

- Ici le module nommé **circle** contiendra 2 éléments :
  - La variable pi (suite à l'import)
  - La fonction circonference_cercle

Modules

- **L'importation** permet à un script d'utiliser le code d'un **autre module**.
- Syntaxes de l'importation :
  - import **nom_module**
  - from **nom_module** import **fonction1** ect...
  - from **nom_module** import *

```
import circle
result = circle.circonference(10)

from circle import circonference
result = circonference(10)
```

- Syntaxe d'accès à un membre d'un module importé
  - **module.fonction** avec la méthode 1
  - **nom_membre** avec la méthode 2 et 3
  - Il est possible d'ajouter un alias à un module ou un membre
    - import **nom_module** as mod_1
    - from **nom_module** import **fn1** as **f1**
- On peut utiliser la fonction **dir(module)** pour en connaître le contenu d'un module.

La variable __name__ (variable Dunder)

- On retrouve souvent cette structure pour les scripts python, elle comporte beaucoup d'avantages quand on travaille avec des **imports**
- **__name__** est une **variable** prédéfinie dans chaque module, elle contient :
  - La chaîne "__main__" si on est dans le module principal, lancé directement depuis le script
  - Le **nom du module** quand on est dans un module importé **import**
- De ce fait, le bloc **if __name__ == "__main__"** n'est **exécuté** que dans le cas où l'on est dans le **module principal**. Les modules importés ne l'exécuteront pas.
- On peut voir ça comme du code verrouillé qui ne s'exécute que si on lance directement le script.

```
import math

def addition(a,b):
  return a + b

def main():
  print(addition(40, 3))

if __name__ == "__main__"
  main()
```

Packages

- un **package** est comme un dossier contenant des **sous-packages** et/ou des **modules**.
- Nous pouvons également en utiliser, un package du **Python Package Index (PyPI)**, installable facilement avec **pip**.
- pour importer un package, nous utilisons l'**import (package.module)**.
- Un package doit avoir le fichier **init.py**, même si vous le laisser vide.
- Mais lorsque nous importons un package, seuls ses modules immédiats sont importés, pas les sous-package. Si vous essayez d'y accéder, cela déclenchera un AttributeError.

Modules natifs

Comme vu précédemment, Python vient avec un ensemble de module de base

- Exemples d'utilisation avec le module csv :

```
import csv
fichier = open("noms.csv", "rt") # la fonction open pour ouvrir un fichier et r pour lecture seul t pour text brut
lecteurCSV = csv.reader(fichier, delimiter=";") # Ouverture du lecteur CSV en lui fournissant le caractère séparateur
for ligne in lecteurCSV:
    print(ligne) # Exemple avec la 1e ligne du fichier d'exemple : ['Nom', 'Prénom', 'Age']
fichier.close()

# Ecriture
fichier = open("annuaire.csv", "wt", newline=";") # On peut changer le newline
ecrivainCSV = csv.writer(fichier, delimiter="|") # On peut changer le délimiter
ecrivainCSV.writerow(["Nom","Prénom","Téléphone"]) # On écrit la ligne d'en-tête avec le titre des colonnes
ecrivainCSV.writerow(["Martin","Julie;Clara","0399731590"]) # attention au caractères spéciaux (;,:« ’.)
fichier.close()
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Fichiers

Manipuler les fichiers

- On peut manipuler les fichiers via la fonction open()
- Elle prend en premier paramètre un chemin de fichier (**path**) et en second paramètre un **mode**, composé de 2 parties:
  - Le mode d'ouverture :
    - r : Lecture
    - w : Écriture
    - a : Ajout
  - Le type d'ouverture:
    - t : Ouverture sous format texte (par défaut)
    - b : Ouverture en mode binaire.
  - Il est important de penser à fermer notre fichier en fin d'utilisation sous peine d'avoir des problèmes d'accès. Pour ce faire, on utilise .close() sur notre variable résultat de la fonction open().

Écrire et lire dans un fichier

- Une fois notre fichier ouvert, on peut le lire ou le modifier. Pour ce faire, il existe en Python plusieurs méthodes.
  - read() : Pour lire **l'ensemble du fichier** tel qu'il est écrit.
  - readline() : Pour lire **ligne par ligne** le fichier (le curseur de fichier passera à la ligne suivante après la méthode). Cette méthode prendra en compte les caractères. spéciaux tels que le caractère de retour à la ligne **\n**.
  - readlines() : Pour obtenir une **liste contenant les lignes du fichier**. Cette méthode prendra en compte les caractères spéciaux tels que le caractère de retour à la ligne **\n**.
  - write() : Permet **d'écrire du texte**.
  - writelines() : Permet **d'écrire une liste de lignes**.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Conteneurs

Qu'est-ce qu'un conteneur ?

Un **conteneur** est un **objet** permettant de stocker d'autres **objets**.

- Les **conteneurs** sont **dynamiques**, et peuvent contenir **plusieurs types de données** (int, str, float, list, object...)
- Un conteneur fournit un **ensemble de méthodes** qui permettent :
  - **Ajouter** un nouveau objet
  - **Supprimer** un objet
  - **Rechercher** des objets selon des critères
  - **Trier**
  - **Filtrer** les objets du tableau

Les différents conteneurs

- `List` : regrouper des éléments en ordre (le plus courant).
- `Tuple` : regrouper des éléments non-modifiable (taille et valeurs fixes)
- `Set` : éviter les doublons
- `Dict` : associations clé-valeur

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

List

Les listes

- La **liste** est le type de **conteneur** le plus utilisé.
- Elle permet de **manipuler** facilement ses données via l'utilisation de ses **méthodes**.
- Les **méthodes** des listes les plus utilisées sont les suivantes :
  - **sort()** : trie les éléments de la liste.
  - **append(element)** : ajouter un élément à la fin de la liste.
  - **extend(list)** : ajouter une liste à la fin de la liste.
  - **pop(index)** : retirer un élément de la liste à l'index donné.
  - **remove(element)** : retirer le premier élément de la liste qui correspond.
  - **count(element)** : nombre d'occurrences d'un élément.

```
ma_liste = []
print(ma_liste) # []

ma_liste = [1, 2, 3]
print(ma_liste) # [1, 2, 3]

ma_liste = [2, 1, 3]
print(ma_liste) # [2, 1, 3]

ma_list.sort()
print(ma_liste) # [1, 2, 3]

ma_list.append(4)
print(ma_liste) # [1, 2, 3, 4]

ma_list.extend([5, 6])
print(ma_liste) # [1, 2, 3, 4, 5, 6]

ma_list.remove(4)
print(ma_liste) # [1, 2, 3, 5, 6]

ma_list.pop(2)
print(ma_liste) # [1, 2, 5, 6]
```

l'itération sur une liste

- l'**itération** est la capacité de **parcourir** (via généralement une boucle) une **série de valeurs** contenues dans un conteneur afin de les afficher ou d'en modifier les valeurs de façon séquentielle.
- Pour parcourir une liste, on utilise généralement une boucle **for**, telle que :

```
ma_liste = [1, 2, 3, 4, 5]
print(ma_liste) # [1, 2, 3, 4, 5]

for item in ma_liste:
print(item)

'''
1
2
3
4
5
'''
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Lambdas

Les lambdas

Avancé

- Les lambdas sont des fonctions simplifiées à l'extrême et anonymes.
- Elles n'ont pas de nom et s'utilisent en général comme arguments d'autres fonctions (cf diapo filter, map, reduce).
- Elles doivent rester très simples (généralement 1 instruction).

```
fct = lambda x : x**2

def fct2(x):
    return x**2

print(fct(2))
print(fct2(2))
```

Sorted, Filtred, Map et Reduce

Avancé

- Pour aller plus loin dans l'usage des listes, il existe certaines fonctions utiles.
- Ces 4 fonctions utilisent les **fonctions ou lambdas** et nous simplifient beaucoup le travail avec les listes.
  - **sorted** : trier la liste selon certains critères.
  - **filter** : filtrer les éléments de la liste.
  - **map** : créer une nouvelle liste avec tous les éléments transformés par une fonction.
  - **reduce** : réduire la liste à une seule valeur.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Tuples

Les tuples

- Le tuple permet de **regrouper** des données, on appelle ça du **packing/construction**.
- Les données sont **non-modifiables** et identifiées par leurs **indices/index**.
- Syntaxes de définition :

```
mon_tuple = ()
mon_tuples = tuple ()
mon_tuple= (1,2,3)
mon_tuple= 1,2,3
```

- Avec les mêmes méthodes que pour les listes, on pourra itérer sur les tuples et récupérer les valeurs à des index précis.
- Les opérations sur un tuple :
  - **len(tuple)**: nombre d'éléments d'un tuple.
  - **tuple.count(element)**: nombre d'occurrences d'un élément dans le tuple.
  - **tuple.index(element)**: index de la première occurrence de l'élément.
- Avec une assignation à plusieurs variables, python propose **l'unpacking**. Exemple : var1, var2 = (1, 2)

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Sets

les sets

- Un **set** est un ensemble d'éléments **uniques** et **ordonnés**, les doublons sont **impossibles**, les valeurs doivent donc être **immutable**.
- Lors de **l'ajout** ou du **retrait** d'un élément d'un set, le conteneur se voit ainsi automatiquement **réordonné**. L'ordre définit par python n'est pas toujours très sensé...
- Lorsque l'on **cast** une list en set, on obtient une série d'éléments **sans doublons** qui ne peuvent plus être modifiés via leur **index** (les sets ne permettant pas la modification des éléments via cette méthode).
- Les sets contiennent des méthodes semblables aux listes mais n'en disposent pas de beaucoup.

```
mon_set = {1, 2, 3, 5, 5, 6}
print(mon_set)  # {1, 2, 3, 5, 6}
mon_set.add(4)
print(mon_set)  # {1, 2, 3, 4, 5, 6}
mon_set.pop()
print(mon_set)  # {2, 3, 4, 5, 6}
# mon_set[2] = 5 n'est pas possible pour un set
```

```
ma_list = [1, 1, 24, 3, 10, 3, 4, 5, 5, 54, 5, 6]
print(ma_list)  # [1, 1, 24, 3, 10, 3, 4, 5, 5, 54, 5, 6]
mon_set_2 = set(ma_list)
print(mon_set_2)  # {1, 3, 4, 5, 6, 10, 54, 24}
```

Méthodes des sets

- **add(element)** : ajout d'élément.
- **update(set)** : fusion de 2 sets.
- **remove(element)** : supprime l'élément s'il est présent, sinon erreur.
- **discard(element)** : supprime l'élément s'il est présent, sinon ne fait rien.
- **isdisjoint(set)** : si aucun élément n'est commun entre les deux.
- **issubset(set2)** : si le set est compris dans le set2 .
- **issuperset(set2)** : si les set2 est compris dans le set.
- On retrouve aussi les méthodes d'**union |**, **d'insertion &**, de **différence -** et de **différence symétrique ^**.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Dictionnaires

Les dictionnaires

- un **dictionnaire** est un conteneur se servant d'une association de **clés** et de **valeur**.
- Il est possible d'accéder aux valeurs qui le constituent via l'utilisation de la clé associée entre crochets. Avec le mot clé **del**, on peut supprimer une entrée.
- Certaines méthodes du dictionnaire produisent des types spéciaux qu'il faudra **cast en list**.
  - **.values()** : récupère les valeurs.
  - **.keys()** : récupère les clés.
  - **.items()** : récupère des tuples (clés, valeurs).
- Via **L'unpacking** des tuples et la méthode **.items()**, il est possible d'afficher les informations du dictionnaire plus facilement.

```
mon_dict = {"k1": 'valeur un', "k2": 25443, "k3": 3.14, "k4": {1: 'blabla'}}
print(mon_dict)
print(mon_dict['k3']) # 3.14
print(mon_dict['k4'][1]) # blabla
print(mon_dict.values()) # ['valeur un', 25443, 3.14, {&: "blabla"}]
print(mon_dict.keys()) # ["k1", "k2", "k3", "k4"]
print(mon_dict.items()) # {"k1": 'valeur un', "k2": 25443, "k3": 3.14, "k4": {1: 'blabla'}}

for key, value in mon_dict.items():
  print(f"key : {key}, value : {value}") # key : k1, value : valeur un ect...
```

Les dictionnaires

- Pour parcourir un dictionnaire, on utilise généralement une boucle for, on peut également accéder en complément de la valeur. Les clés des dictionnaires sont forcément de types immutables

```
mon_dict = {'key1': 123, 'key2': '456', 'key3': [7, 8, 9]}
print(mon_dict)  # {'key1': 123, 'key2': '456', 'key3': [7, 8, 9]}
for key, value in mon_dict.items():
    print(f"{key}: {value}")
```

```
{'key1': 123, 'key2': '456', 'key3': [7, 8, 9]}
key1: 123
key2: 456
key3: [7, 8, 9]
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Comprehension

list/set/dict comprehension

Avancé

- Il est possible de **générer et d'itérer** sur des conteneurs à l'aide de la **comprehension**
- Pour la **list comprehension**, la syntaxe est la suivante, un **itérable** est un objet sur lequel on peut itérer:
  - var = [expression for element in iterable]
- Il est aussi possible d'ajouter un filtre avec un if après l'itérable (équivalent à la fonction filter)

```
liste_d = [x for x in range(1, 11) if x % 2 == 0]
print(liste_d)

# équivalent
liste_a = []
for x in range(1, 11):
    if x % 2 == 0:
        liste_a.append(x)
print(liste_a)
```

```
# list comprehension avec les carrés de 0 à 9
ls = [x**2 for x in range(10)]

# dict comprehension avec lettre et leur valeur ascii
dic = {chr(n): n for n in range(65, 91)}
print(dic)

# tuple comprehension avec reduction d'une chaine
chaine = "abracadabra"
s = {char for char in chaine}
print(s)
```

Mutabilité

- La **mutabilité** est la capacité d'une variable à être modifiée.
- Il ne faut pas la confondre avec la **réassignation**, qui stocke simplement une autre variable à un autre emplacement mémoire.
- Les types non-mutables/immutable sont les **bool, str, int, bytes, range, tuple et frozenset**.
- A contrario les list, dict et set sont par exemple mutables, il est possible de les modifier.
- Il faut cependant faire attention à leur utilisation au sein d'une fonction visant à les altérer, car leur valeur pourrait changer sans qu'on ne veuille.
- L'emplacement mémoire d'une variable mutable ne change pas après sa modification.

```python
mon_nombre = 5
print(id(mon_nombre))  # 2358276981104
print(mon_nombre)  # 5

mon_nombre += 2
print(id(mon_nombre))  # 2358276981168
print(mon_nombre)  # 7
```

```python
ma_liste = [1, 2, 3]
print(id(ma_liste))  # 2408503638912
print(ma_liste)  # [1, 2, 3]

ma_liste.append(4)
print(id(ma_liste))  # 2408503638912
print(ma_liste)  # [1, 2, 3, 4]
```

Hash

- Le **condensat (hash)** est la valeur obtenue lorsque l'on passe une variable dans un **algorithme de hachage**.
- Plusieurs valeurs différentes peuvent produire le **même condensat** : c'est une **collision**.
- Un objet est **hashable** s'il est **immutable** (et que tous ses éléments le sont).
- Exemples hashables : **int, str, tuple**.
- Non hashables : **list, dict, set** (car mutables).
- Par exemple, il est possible de hasher un **int, un string et un tuple** :

```
print(hash(10))  # 10
print(hash(2305843009213693961))  # 10, donc collision avec 10
print(hash('toto'))  # 1693940491935614836
print(hash((1, 2, 3)))  # 5293440672954974 51
```

Paramètres spéciaux

Les *args et **kwargs

- En Python, il est possible d'ajouter des **paramètres spéciaux** précédés avant leurs noms par une ou deux étoiles.
- Leurs **noms sont conventionnés**, il est important de les nommer **arg(arguments)** et **kwargs(keyword arguments)**
- Ils permettent d'avoir des fonctions au **nombre d'argument variable**.

```
def ma_fonction(argument_classique, argument_par_défaut="valeur par défaut", *args, **kwargs):
    print(argument_classique)
    print(argument_par_défaut)
    print(args)
    print(kwargs)
```

*args

- Le paramètre *arg se **transformera en tuple** qui aura **tous les arguments supplémentaires** non nommés en son sein, il sera possible d'y **accéder par leur index[]**

```
def ma_fonction_avec_args(*args):
    for arguments in args:
        print(arguments)

ma_fonction_avec_args(1, "5", True, "Salut", "\na\nb\nc", "Hello World !")
ma_fonction_avec_args()
ma_fonction_avec_args("aaa")
```

**kwargs

- Le paramètre **kwargs se **transformera en un dictionnaire** qui contiendra un **ensemble clé-valeur** qui aura **tous les arguments ayant un nom associé à une valeur** via la syntaxe nom = valeur

```
def ma_fonction_avec_kwargs(**kwargs):
    print(kwargs)
    for kwarg_key, kwarg_value in kwargs.items():
        print(kwarg_key, kwarg_value)

ma_fonction_avec_kwargs(agument1="test", argument2=True, arg3=300)
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Décorateurs

Le principe des décorateurs

Les décorateurs sont des **fonctions particulières** que l'on peut **appliquer à d'autres fonctions**.

- Ils permettent de réaliser des tâches **avant** et **après l'exécution de la fonction** et d'en contrôler le comportement.
- On peut s'en servir aussi pour factoriser du code commun à 2 fonctions.

Définir un décorateur

Avancé

- Lorsque l'on créé des fonctions, il est fréquent que l'on souhaite avoir une fonction similaire à une autre, mais ayant un comportement légèrement différent. Dans ce genre de cas, il faut en général surcharger les fonctions / méthodes, ou rajouter des expressions dans ces fonctions le temps nécessaire.
- En python, il existe ce que s'appelle des décorateurs de fonction, qui permettent d'altérer le fonctionnement des fonctions ou des méthodes de sorte que l'on peut appeler les versions modifiées à la volée sans avoir à retirer les ajouts si l'on veut utiliser de nouveau la version de base des fonctions.

```python
def mon_decorateur(fonction):

    def wrap_func():
        print("Code avant la fonction")
        fonction()
        print("Code après la fonction")
        pass

    return wrap_func


# En commentant simplement ce décorateur, on repasse à la fonction de base
@mon_decorateur
def fonction_de_base():
    print("Code de la fonction")
    pass


fonction_de_base()
```

Décorateur avec paramètres

Avancé

- Il est également possible d'utiliser des paramètres sans un décorateur. Le plus souvent, on se sert ainsi des paramètres de type *arg et **kwargs pour permettre plus de flexibilité.

```python
def decorator(*args, **kwargs):
    print("Dans le décorateur")

    def inner(func):
        # code functionality here
        print("Dans la fonction interne")
        print("J'aime ce fruit : ", kwargs['fruit'])

        func()

    return inner
```

- Par exemple, ici, nous avons une fonction qui permet d'ajouter des variantes à la décoration.

```python
@decorator(fruit="Banane")  # J'aime ce fruit :  Banane
def my_func():
    print("Dans la fonction de base")
```

Décorateurs multiples

Avancé

- De plus, en Python, on peut décorer une fonction déjà décorée, via l'utilisation de plusieurs décorateurs. De ce fait, on a par exemple ici une fonction décorée puis qui se voit être elle-même décorée.

```python
@mon_second_decorateur
@mon_decorateur
def fonction_de_base():
    print("Code de la fonction")
    pass
```

```
Code avant la fonction 2
Code avant la fonction
Code de la fonction
Code après la fonction
Code après la fonction 2
```

```python
def mon_second_decorateur(fonction):

    def wrap_func():
        print("Code avant la fonction 2")
        fonction()
        print("Code après la fonction 2")
        pass

    return wrap_func

def mon_decorateur(fonction):

    def wrap_func():
        print("Code avant la fonction")
        fonction()
        print("Code après la fonction")
        pass

    return wrap_func
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Fichiers JSON

Les fichiers JSON

- Pour manipuler des fichiers JSON, il va nous falloir faire appel au module **json**:
  - Ce module est présent de base dans python
- Via ce module, nous disposons ensuite de 4 méthodes principales :
  - **json.dump()** : Qui va sauvegarder des données dans un flux données
  - **json.load()** : Qui va chercher les données dans le flux et les retourner avec typage pour correspondre à python
  - **json.dumps()** : Pour récupérer une chaîne de caractère correspondant au JSON dans le but de l'afficher ou de l'envoyer
  - **json.loads()** : Pour récupérer des données correspondantes à un JSON sous la forme d'une chaîne de caractère.

Exercice

Avancé

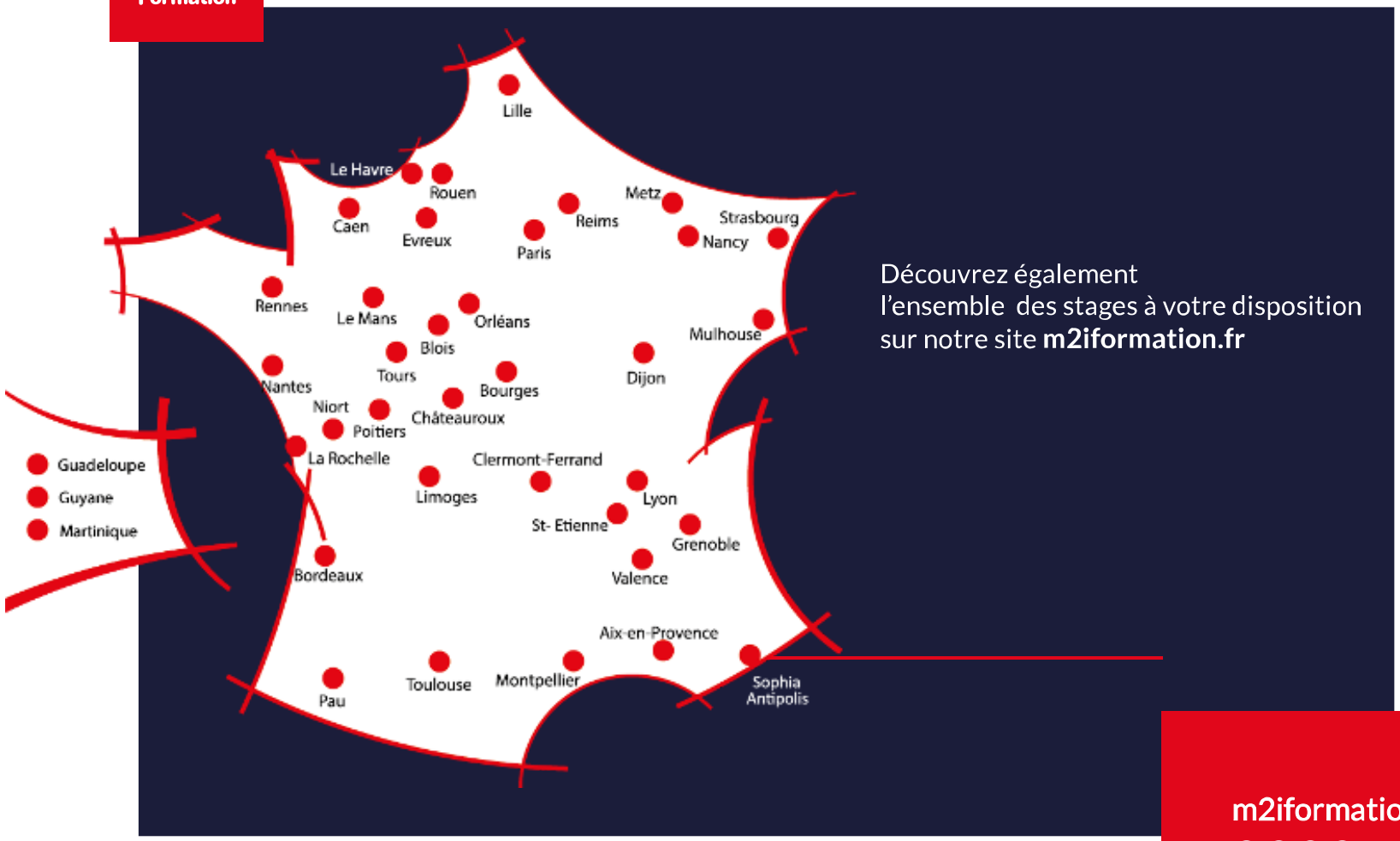
- Par l'utilisation d'un fichier JSON qui sera ouvert, lu et écrit, vous devrez réaliser un logiciel servant à un utilisateur pour stocker des informations sur des chansons. Ces chansons devront posséder comme informations un titre, un artiste, une catégorie et un score (sur 5). Lors de l'ouverture, le programme ouvrira automatiquement le fichier music.json (ou le créera dans le cas où il n'existerait pas) dans le but de l'alimenter de la liste des chansons pour l'utilisateur. La localisation du fichier devra être à la racine du programme dans le dossier nommé datas.

```
=== MENU PRINCIPAL ===
1. Ajouter une chanson
2. Voir les chansons
3. Editer une chanson
4. Supprimer une chanson
0. Quitter le programme
Faites votre choix : 1

 === AJOUTER UNE CHANSON ===
Titre de la chanson : Titre
Artiste de la chanson : Artiste
Catégorie de la chanson : Catégorie
Score de la chanson (sur 5) : 4
```

Merci pour votre attention

Des questions ?



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr