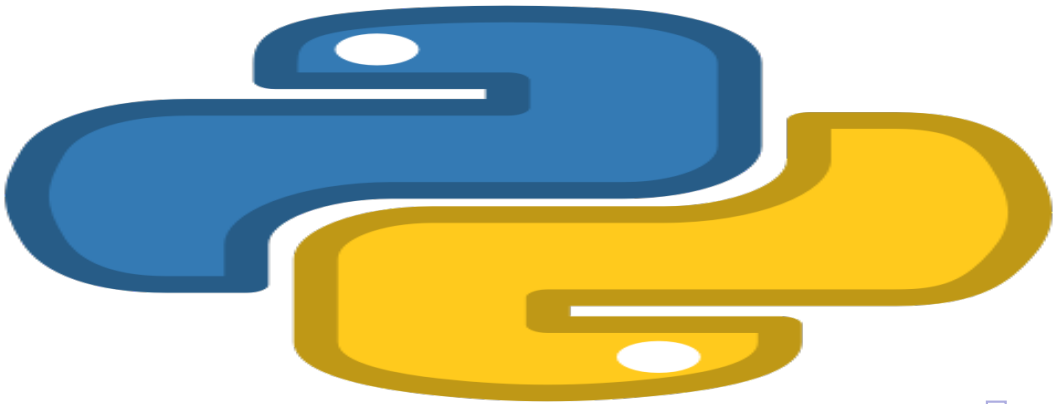
Python : modules et packages

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com



Plan

Plan

Plan

Plan

Python

Dans une application **Python**

- On peut utiliser des éléments définis dans un autre fichier : une variable, une fonction, une classe, une interface...
- Pour cela, il faut l'importer là où on a besoin de l'utiliser
- Un fichier contenant du code **Python** = `module`

Quatre types de modules

- **Built-in modules** :
  - Définis dans la librairie standard de **Python** : **STDLIB**.
  - Pour les utiliser, ils ne nécessitent aucune importation.
- **Core modules** :
  - Définis dans **STDLIB**.
  - Pour les utiliser, il faut les importer.
- **Community modules** :
  - Proposés par la communauté **Python**.
  - Pour les utiliser, il faut les installer et ensuite les importer.
- **Custom modules** :
  - Définis par le développeur.
  - Pour les utiliser, il faut les importer.

Python

Avant de commencer

- Créez un nouveau projet `cours-modules` dans votre espace de travail
- Créez un fichier `main.py`
- Validez

Python

Étant donné le fichier `fonctions.py` ayant le contenu suivant

```
somme = lambda a, b: a + b

produit = lambda a, b: a * b
```

Python

Étant donné le fichier `fonctions.py` ayant le contenu suivant

```
somme = lambda a, b: a + b

produit = lambda a, b: a * b
```

Pour importer le contenu de `fonctions.py` dans `main.py`

```
import fonctions

print(fonctions.somme (2, 3))
# affiche 5

print(fonctions.produit (2, 3))
# affiche 6
```

Python

On peut aussi utiliser des alias

```
import fonctions as f

print(f.somme (2, 3))
# affiche 5

print(f.produit (2, 3))
# affiche 6
```

Python

On peut aussi indiquer ce que l'on souhaite importer et simplifier l'utilisation

```
from fonctions import somme, produit

print(somme (2, 3))
# affiche 5

print(produit (2, 3))
# affiche 6
```

Python

Pour tout importer, on peut utiliser *

```
from fonctions import *

print(somme (2, 3))
# affiche 5

print(produit (2, 3))
# affiche 6
```

Python

Python : interprété, compilé ou les deux ?

- **Python** est un langage interprété : le code source est généralement exécuté ligne par ligne par l'interpréteur **Python**
- Cependant, lors de l'exécution d'un script contenant des importations de modules, **Python** compile le code source en bytecode, qui est ensuite exécuté par la machine virtuelle Python (PVM).
- Le bytecode est stocké dans des fichiers `.pyc` dans le dossier `__pycache__` pour optimiser les exécutions ultérieures.
- Dans certains cas, comme l'exécution directe de scripts simples, **Python** peut choisir de ne pas générer ces fichiers `.pyc` pour éviter l'encombrement du système de fichiers

Python

STDLIB

- Librairie standard de **Python**
- Collection de modules et de packages intégrés : disponibles dès l'installation de **Python**
- Conçu pour simplifier le développement de logiciels
- Offrant des outils pour des tâches courantes

Python

Quelques modules **STDLIB** : fonctions mathématiques

- `math` pour les fonctions arithmétiques
- `random` pour la génération de nombres aléatoires
- `decimal` pour une meilleure manipulation de nombres décimaux
- `statistics` pour les fonctions statistiques de base
- ...

Python

Quelques modules **STDLIB** : manipulation de données

- `datetime` : Manipulation des dates et des heures.
- `json` : Encodage et décodage de données **JSON**.
- `csv` : Lecture et écriture de fichiers au format **CSV**.
- ...

Python

Quelques modules **STDLIB** : fonctionnalités de Fichiers et d'**OS**

- `glob` pour la recherche de fichiers/répertoires
- `os` pour réaliser des opérations sur le système d'exploitation
- `os.path` pour manipuler les chemins de fichiers
- `pathlib` pour la manipulation orientée objet des chemins de fichiers
- `sys` pour effectuer des opérations d'entrée et sortie plus personnalisées
- `shutil` pour réaliser des opérations sur les fichiers
- ...

Python

Quelques modules **STDLIB** : réseau et communication

- `socket` : Interface de bas niveau pour les communications réseau
- `http` : Modules pour travailler avec **HTTP**
  - `http.client` pour le client
  - `http.server` pour le serveur
- `urllib` : Collection de modules pour travailler avec les URLs.
- ...

Python

Quelques modules **STDLIB** : tests

- `unittest`
- `doctest`

Python

Quelques modules **STDLIB** : tests

- `unittest`
- `doctest`

Quelques modules **STDLIB** : développement d'interface graphique (**GUI**)

tkinter : développement d'applications desktop.

Python

Packages

- Un répertoire contenant des fichiers et/ou répertoires = `package`
- Avant **Python 3.3**, un package contenant des modules **Python** doit contenir un fichier `__init__.py`

Python

Packages

- Un répertoire contenant des fichiers et/ou répertoires = `package`
- Avant **Python 3.3**, un package contenant des modules **Python** doit contenir un fichier `__init__.py`

Pour la suite

- créons un répertoire appelé `package` à la racine du projet
- déplaçons le fichier `fonctions.py` dans `package`

Python

Dans `main.py`, pour importer et utiliser les fonctions définies dans `fonctions.py`

```
from package.fonctions import produit, somme

print(somme (2, 3))
# affiche 5

print(produit (2, 3))
# affiche 6
```

Python

Et si `fonctions.py` **était dans** `subpackage` **qui est défini dans** `package`

```
from package.subpackage.fonctions import produit, somme

print(somme (2, 3))
# affiche 5

print(produit (2, 3))
# affiche 6
```

Python

Recommandation pour les projets **Python**

- un seul fichier d'entrée à la racine du projet (appelé souvent `main.py` ou `app.py`)
- un package (`src` par exemple) à la racine contenant des fichiers sources classées dans des sous-packages

Python

PyPI : Python Package Index

- Dépôt de packages, officiel et principal, pour le langage **Python**.
- Contenant une grande variété de packages open-source pour divers usages
- Lien vers le dépôt : `https://pypi.org/`
- Tous les packages **PyPI** sont gérés par **PIP**

Python

PIP : **P**ackage **i**nstaller for **P**ython

- Gestionnaire de packages/modules par défaut pour **Python**
- Utilisable en ligne de commandes
- Utilisé pour installer, mettre à jour et gérer les packages **Python**

Python

Pour connaître la version de PIP

```
pip --version
```

Python

Pour connaître la version de PIP

```
pip --version
```

Toutes les commandes PIP peuvent être exécutées de la manière suivante

```
python -m pip [suite]
```

Python

Pour lister les packages installés (en global avec Python)

```
pip list
```

Python

Pour lister les packages installés (en global avec Python)

```
pip list
```

Pour avoir plus de détails sur un package installé

```
pip show nom_package
```

Python

Pour installer un package

```
pip install nom_package
```

Python

Pour installer un package

```
pip install nom_package
```

Pour installer plusieurs packages

```
pip install p1 p2 p3
```

Python

Pour installer un package

```
pip install nom_package
```

Pour installer plusieurs packages

```
pip install p1 p2 p3
```

Pour désinstaller un package

```
pip uninstall nom_package
```

Python

Pour chercher un package

```
pip search nom_package
```

Python

Pour chercher un package

```
pip search nom_package
```

Pour lister les packages installés

```
pip list
```

Python

Deux types de package que l'on peut installer avec `pip`

- **Bibliothèques et Frameworks** : destinés à être importés et utilisés dans vos propres projets Python. Ils peuvent offrir des fonctionnalités allant des opérations mathématiques (comme **NumPy**) aux frameworks web complets (comme **Django**, **Fast API** ou **Flask**)
- **Outils de développement et Utilitaires** : fournissent des fonctionnalités qui aident au développement, au déploiement, ou à la maintenance de projets Python, mais ne sont généralement pas importés dans le code source.

Python

Exemple : installons `numpy`

```
pip install numpy
```

Python

Exemple : installons `numpy`

```
pip install numpy
```

Pour l'utiliser, commençons par l'importer

```
import numpy as np
```

Python

Exemple : installons `numpy`

```
pip install numpy
```

Pour l'utiliser, commençons par l'importer

```
import numpy as np
```


Utilisons le pour calculer et afficher la somme des éléments du tableau

```
arr = np.array([1, 2, 3, 4, 5])

sum_result = np.sum(arr)
print("Somme des éléments du tableau :", sum_result)
```

Python

Pour générer un exécutable de notre application

- Installer le module `pip install pyinstaller`
- Lancer la commande `pyinstaller.exe --onefile main.py`
- Un fichier `main.exe` a été généré dans le répertoire `dist`.

Python

Exemple : installons `mypy`

```
pip install mypy
```

Python

Exemple : installons `mypy`

```
pip install mypy
```

Considérons un fichier `tester_types.py` ayant le contenu suivant

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

x: str = '2'
y: int = '3'

print(somme(x, y))
```

Python

Exemple : installons `mypy`

```
pip install mypy
```

Considérons un fichier `tester_types.py` ayant le contenu suivant

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

x: str = '2'
y: int = '3'

print(somme(x, y))
```

Lancez le code et vérifiez le résultat suivant

```
23
```

Python

Lancez la commande suivante pour vérifier les incohérences de type

```
mypy tester_types.py
```

Résultat

```
tester_types.py:5: error: Incompatible types in assignment
(expression has type "str", variable has type "int")  [assignment]
tester_types.py:7: error: Argument 1 to "somme" has incompatible type "
   str";
expected "int"  [arg-type]
Found 2 errors in 1 file (checked 1 source file)
```

Python

Pour analyser tous les fichiers du répertoire courant et ses sous-répertoires avec `mypy`

```
mypy .
```

Node.js

pyenv : **P**ython **env**ironments

- Gestionnaire de versions pour **Python** : permettant de travailler avec plusieurs versions de **Python**
- Disponible pour **Windows**, **Mac OS** et **Linux**
- Conçu initialement pour **Mac OS** et **Linux** ensuite pour **Windows**
- Deux dépôts **GitHub** différents :
  - Pour **Mac OS** et **Linux** : `https://github.com/pyenv/pyenv`
  - Pour **Windows** : `https://github.com/pyenv-win/pyenv-win`

Python

Pour l'installer sous Windows depuis PowerShell

```
Invoke-WebRequest -UseBasicParsing -Uri
"https://raw.githubusercontent.com/pyenv-win/pyenv-win/master/py
-OutFile "./install-pyenv-win.ps1"; &"./install-pyenv-win.ps1"
```

Python

Pour l'installer sous Windows depuis PowerShell

```
Invoke-WebRequest -UseBasicParsing -Uri
"https://raw.githubusercontent.com/pyenv-win/pyenv-win/master/py
-OutFile "./install-pyenv-win.ps1"; &"./install-pyenv-win.ps1"
```

Depuis Mac ou Linux

```
curl https://pyenv.run | bash
```

Python

Pour l'installer sous Windows depuis PowerShell

```
Invoke-WebRequest -UseBasicParsing -Uri
"https://raw.githubusercontent.com/pyenv-win/pyenv-win/master/py
-OutFile "./install-pyenv-win.ps1"; &"./install-pyenv-win.ps1"
```

Depuis Mac ou Linux

```
curl https://pyenv.run | bash
```

Pour vérifier la version de pyenv

```
pyenv --version
```

Pour lister les versions Python disponibles pour pyenv

```
pyenv install -l
```

Pour lister les versions Python disponibles pour pyenv

```
pyenv install -l
```

Pour installer une version particulière de Python (3.8.1 par exemple)

```
pyenv install 3.8.1
```

Pour lister les versions Python disponibles pour pyenv

```
pyenv install -l
```

Pour installer une version particulière de Python (3.8.1 par exemple)

```
pyenv install 3.8.1
```

Pour lister les versions installées avec pyenv

```
pyenv versions
```

Pour lister les versions Python disponibles pour pyenv

```
pyenv install -l
```

Pour installer une version particulière de Python (3.8.1 par exemple)

```
pyenv install 3.8.1
```

Pour lister les versions installées avec pyenv

```
pyenv versions
```

Pour afficher la version courante de Python utilisée par pyenv

```
pyenv version
```

Python

Pour définir une version de Python comme étant globale (3.8.1 par exemple)

```
pyenv global 3.8.1
```

Node.js

venv : **v**irtual **env**ironments

- Outil utilisé pour créer des environnements virtuels en **Python**
- Inclus par défaut avec **Python** à partir de la version 3.3
- Fournissant les fonctionnalités de base nécessaires pour créer et gérer des environnements virtuels
- Si vous deviez utiliser une version < 3.3, alors vous devriez installer **virtualenv**

Python

Pour créer un environnement virtuel (appelé `my_env`)

```
python -m venv chemin/vers/my_env
```

Python

Pour créer un environnement virtuel (appelé `my_env`)

```
python -m venv chemin/vers/my_env
```

Pour activer l'environnement virtuel

```
my_env/Scripts/activate
```

Python

Pour créer un environnement virtuel (appelé `my_env`)

```
python -m venv chemin/vers/my_env
```

Pour activer l'environnement virtuel

```
my_env/Scripts/activate
```

Résultat

```
(my_env) Chemin/vers/répertoire/courant>
```

Node.js

Remarques

- Maintenant, toute installation de package via `pip` sera limitée à cet environnement virtuel, et vous utiliserez la version de **Python** spécifiée pour ce projet.
- L'environnement peut être utilisé dans n'importe quel répertoire sue votre disque dur
- L'environnement peut être utilisé dans plusieurs projets différents

Python

Vérifions les packages installés dans notre environnement

```
pip list
```

Python

Vérifions les packages installés dans notre environnement

```
pip list
```

Installons le package `password-maker` dans notre environnement

```
pip install password-maker
```

Python

Vérifions les packages installés dans notre environnement

```
pip list
```

Installons le package `password-maker` dans notre environnement

```
pip install password-maker
```


Désactivons l'environnement virtuel

```
my_env/Scripts/deactivate
```

Node.js

Remarque

Vérifiez que `password-maker` n'est pas disponible dans l'environnement global.

Python

Pour lister les dépendances d'un projet dans un fichier `requirements.txt`

```
pip freeze > requirements.txt
```

Python

Pour lister les dépendances d'un projet dans un fichier `requirements.txt`

```
pip freeze > requirements.txt
```

Pour installer les dépendances d'un projet spécifiées dans un fichier `requirements.txt`

```
pip install -r requirements.txt
```

Python

Constantes pour modules et packages

- `__name__` contient
  - soit la valeur `__main__` dans le fichier d'entrée
  - soit le chemin complet depuis la racine du projet vers le fichier source
- `__package__` contient le package et les sous-packages du fichier source
- `__file__` contient le chemin complet depuis la racine du disque vers le fichier source

Python

En plaçant les constantes précédentes dans `main.py`, le résultat est

```
print(__name__)
# affiche __main__

print(__package__)
# affiche None

print(__file__)
# affiche C:/Users/user/cours-modules/main.py
```

Python

En les plaçant dans `fonctions.py`, le résultat est

```
print(__name__)
# affiche package.subpackage.fonctions

print(__package__)
# affiche package.subpackage

print(__file__)
# affiche C:\Users\user\cours-modules\package\subpackage\fonctions.py
```

Python

Dans `fonctions.py`, ajoutons la fonction `somme_carre`

```
somme = lambda a, b: a + b

produit = lambda a, b: a * b

somme_carre = lambda a, b: somme(a * a, b * b)

print(somme_carre(3, 4))
```

Python

Dans `fonctions.py`, ajoutons la fonction `somme_carre`

```
somme = lambda a, b: a + b

produit = lambda a, b: a * b

somme_carre = lambda a, b: somme(a * a, b * b)

print(somme_carre(3, 4))
```


En exécutant `fonctions.py`, le résultat est :

25

Python

En exécutant `main.py`

```
from package.subpackage.fonctions import *

print(somme (2, 3))

print(produit (2, 3))
```

Python

En exécutant `main.py`

```
from package.subpackage.fonctions import *

print(somme (2, 3))

print(produit (2, 3))
```


Le résultat est :

```
25
5
6
```

Python

Modifions `fonctions.py` pour ne plus exécuter `somme_carre` que si on exécute `fonctions.py`

```
somme = lambda a, b: a + b

produit = lambda a, b: a * b

somme_carre = lambda a, b: somme(a * a, b * b)

if __name__ == '__main__':
    print(somme_carre(3, 4))
```

Python

En exécutant `fonctions.py`, le résultat est :

25

Python

En exécutant `fonctions.py`, le résultat est :

25

En exécutant `main.py`, le résultat est :

5
6

Python

Fonctions mathématiques prédéfinies (**Built-in functions**)

- `abs(x)` : retourne la valeur absolue de `x`
- `pow(x, y)` : retourne `x` puissance `y`
- `max(x, y)` : retourne le max de `x` et `y`
- `min(x, y)` : retourne le min de `x` et `y`
- `round(x)` : retourne l'arrondi de `x`

Python

Fonctions mathématiques prédéfinies (**Built-in functions**)

- `abs(x)` : retourne la valeur absolue de `x`
- `pow(x, y)` : retourne `x` puissance `y`
- `max(x, y)` : retourne le max de `x` et `y`
- `min(x, y)` : retourne le min de `x` et `y`
- `round(x)` : retourne l'arrondi de `x`

Exemple

```
print(max(1, abs(-3), 2))
# affiche 3
```

Python

Fonctions mathématiques nécessitant l'importation du module `math`

- `sqrt(x)` : retourne la racine carré de `x`
- `trunc(x)` : retourne la partie entière de `x`
- `fsum(iterable)` : retourne la somme de tous les éléments de `iterable`
- `factorial(x)` : retourne la factorielle de `x`
- `floor(x)` et `ceil(x)` : retournent l'arrondi de `x`
- ...

Python

Fonctions mathématiques nécessitant l'importation du module `math`

- `sqrt(x)` : retourne la racine carré de `x`
- `trunc(x)` : retourne la partie entière de `x`
- `fsum(iterable)` : retourne la somme de tous les éléments de `iterable`
- `factorial(x)` : retourne la factorielle de `x`
- `floor(x)` et `ceil(x)` : retournent l'arrondi de `x`
- ...

Exemple

```
import math

print(math.sqrt(25))
# affiche 5.0
```

Python

Exemple avec `math.floor(x)`, `math.ceil(x)` **et** `round(x)`

```
print(round(1.9))
# affiche 2
print(round(1.5))
# affiche 2
print(round(1.4))
# affiche 1
print(math.ceil(1.9))
# affiche 2
print(math.ceil(1.5))
# affiche 2
print(math.ceil(1.4))
# affiche 2
print(math.floor(1.9))
# affiche 1
print(math.floor(1.5))
# affiche 1
print(math.floor(1.4))
# affiche 1
```

Python

Pour extraire la partie entière et la partie décimale, on utilise la fonction `modf` (modulus and fraction)

```
frac, ent = math.modf(2.5)

print(frac)
# affiche 0.5

print(ent)
# affiche 2.0
```

Exercice

Écrire une fonction `equation_second_degre(a, b, c)` qui permet de résoudre une équation de second degré ($ax^2 + bx + c = 0$)

- $\Delta = b^2 - 4ac$
- Si $\Delta < 0$: pas de solution
- Si $\Delta = 0$: une solution $-b/2a$
- Si $\Delta > 0$: deux solutions $(-b - \sqrt{\Delta})/2a$ et $(-b + \sqrt{\Delta})/2a$

Résultat attendu

```
print(equation_second_degre(1, 1, -2))
# affiche (-2.0, 1.0)

print(equation_second_degre(1, 2, 1))
# affiche -1.0

print(equation_second_degre(2, 3, 5))
# affiche pas de solution
```

Python

Solution

```
import math

delta = lambda a, b, c : pow(b, 2) - 4 * a * c

def equation_second_degre(a, b, c):
    d = delta(a, b, c)
    if d < 0:
        return "pas de solution"
    elif d == 0:
        return -b / (2 * a)
    else:
        return (-b - math.sqrt(d)) / (2 * a), (-b + math.sqrt(d)) / (2 * a)

print(equation_second_degre(1, 1, -2))
# affiche (-2.0, 1.0)

print(equation_second_degre(1, 2, 1))
# affiche -1.0

print(equation_second_degre(2, 3, 5))
# affiche pas de solution
```

Python

Fonctions mathématiques nécessitant l'importation du module `random`

- `random()` : retourne un nombre réel aléatoire entre 0 et 1
- `randint(x, y)` : retourne un nombre entier aléatoire entre `x` et `y` (`x` et `y` inclus)
- `choice(iterable)` : retourne un élément aléatoire appartenant à `iterable`
- ...

Python

Fonctions mathématiques nécessitant l'importation du module `random`

- `random()` : retourne un nombre réel aléatoire entre 0 et 1
- `randint(x, y)` : retourne un nombre entier aléatoire entre `x` et `y` (`x` et `y` inclus)
- `choice(iterable)` : retourne un élément aléatoire appartenant à `iterable`
- ...

Exemple

```
import random

print(random.randint(5, 10))
# affiche un nombre entre 5 et 10
```

Python

decimal

- Module **Python** permettant de manipuler des nombres à virgules dont le calcul est plus exacte en comparaison avec le type `float`
- Type conçu principalement pour les utilisateurs et pas pour la machine

Python

decimal

- Module **Python** permettant de manipuler des nombres à virgules dont le calcul est plus exacte en comparaison avec le type `float`
- Type conçu principalement pour les utilisateurs et pas pour la machine

Quelques problèmes avec `float`

```
print(0.1 + 0.1 + 0.1 - 0.3)
# affiche 5.551115123125783e-17

print(1.1 + 2.2)
# affiche 3.3000000000000003
```

Python

On peut construire un décimal à partir d'un entier ou une chaîne de caractères

```
from decimal import *

print(Decimal(3))
# affiche 3

print(Decimal("3.33"))
# affiche 3.33

print(Decimal(str(2.0 ** 2)))
# affiche 4.0
```

Python

On peut construire un décimal à partir d'un entier ou une chaîne de caractères

```
from decimal import *

print(Decimal(3))
# affiche 3

print(Decimal("3.33"))
# affiche 3.33

print(Decimal(str(2.0 ** 2)))
# affiche 4.0
```


On peut aussi construire un décimal à partir d'un float (mais il est conseillé de passer par les chaînes)

```
print (Decimal(3.3))
# affiche 3.29999999999999982236431605997495353221893310546875

print (Decimal(str(3.3)))
# affiche 3.3
```

Python

On peut modifier la précision

```
x = Decimal('2.4444')
y = Decimal('3.4321')

print(x + y)
# affiche 5.8765

getcontext().prec = 3
print(x + y)
# affiche 5.88

getcontext().prec = 2
print(x + y)
# affiche 5.9

getcontext().prec = 1
print(x + y)
# affiche 6
```

Python

`fraction`

- Module **Python** permettant de manipuler des nombres rationnels
- Construction possible depuis une paire d'entiers, un autre nombre rationnel, ou une chaîne de caractères.

Python

Exemples de construction de fraction

```
from decimal import Decimal
from fractions import Fraction

print(Fraction(5, 2))
# affiche 5/2

print(Fraction(2.5))
# affiche 5/2

print(Fraction(Decimal("2.5")))
# affiche 5/2

print(Fraction(5 / 2))
# affiche 5/2

print(Fraction("5/2"))
# affiche 5/2
```

Python

Pour extraire le numérateur ou le dénominateur

```
x = Fraction(2.5)

print(x.numerator)
# affiche 5

print(x.denominator)
# affiche 2
```

Les opérateur arithmétiques peuvent être appliqués sur les fractions

```
f1 = Fraction(1, 2)
f2 = Fraction(3, 4)

# Addition
result_add = f1 + f2
print(result_add)
# affiche 5/4

# Soustraction
result_sub = f1 - f2
print(result_sub)
# affiche -1/4

# Multiplication
result_mul = f1 * f2
print(result_mul)
# affiche 3/8

# Division
result_div = f1 / f2
print(result_div)
# affiche 2/3
```

Python

Tableaux statiques : `array`

- Structure de données acceptant plusieurs valeurs de même type
- Type à spécifier à la création du tableau en utilisant le type code
- Seuls les types primitifs suivants sont acceptés : caractères, entiers et flottants

Python

Quelques types codes

- `b` : pour les caractères signés (byte : nombre compris entre -128 et 127)
- `B` : pour les caractères non-signés (byte : combre compris entre 0 et 255)
- `h` : pour les shorts signés
- `H` : pour les shorts non-signés
- `i` : pour les entiers signés
- `I` : pour les entiers non-signés
- `l` : pour les longs signés
- `L` : pour les longs non-signés
- `f` : pour les flottants
- `d` : pour les doubles

Python

Pour utiliser les tableaux statiques, il faut faire l'import suivant

```
from array import array
```

Python

Pour utiliser les tableaux statiques, il faut faire l'import suivant

```
from array import array
```

Pour déclarer un tableau

```
arr = array('i')
```

Python

Pour utiliser les tableaux statiques, il faut faire l'import suivant

```
from array import array
```

Pour déclarer un tableau

```
arr = array('i')
```

Pour déclarer un tableau avec quelques valeurs initiales

```
arr = array('i', [2, 3, 8, -5])
```

Python

Pour utiliser les tableaux statiques, il faut faire l'import suivant

```
from array import array
```

Pour déclarer un tableau

```
arr = array('i')
```

Pour déclarer un tableau avec quelques valeurs initiales

```
arr = array('i', [2, 3, 8, -5])
```

Pour afficher les caractéristiques du tableau

```
print(arr)
# affiche array('i', [2, 3, 8, -5])
```

Python

Pour connaître le nombre d'éléments d'un tableau

```
print(len(arr))
# affiche 4
```

Python

Pour connaître le nombre d'éléments d'un tableau

```
print(len(arr))
# affiche 4
```

Pour ajouter un élément (à la fin)

```
arr.append(10)

print(arr)
# affiche array('i', [2, 3, 8, -5, 10])
```

Python

Pour connaître le nombre d'éléments d'un tableau

```
print(len(arr))
# affiche 4
```

Pour ajouter un élément (à la fin)

```
arr.append(10)

print(arr)
# affiche array('i', [2, 3, 8, -5, 10])
```


Pour ajouter un élément à une position donnée (ici 2)

```
arr.insert(2, 6)

print(arr)
# affiche array('i', [2, 3, 6, 8, -5])
```

Python

Pour récupérer l'indice de la première occurrence d'un élément dans le tableau, on utilise `index`. Si l'élément n'existe pas, la méthode lève une exception.

```
print(arr.index(8))
# affiche 2
```

Python

Pour récupérer l'indice de la première occurrence d'un élément dans le tableau, on utilise `index`. Si l'élément n'existe pas, la méthode lève une exception.

```
print(arr.index(8))
# affiche 2
```

Pour supprimer un élément, on utilise `remove`. Si l'élément n'existe pas, la méthode lève une exception.

```
arr.remove(3)

print(arr)
# affiche array('i', [2, 8, -5])
```

Python

Autres opérations sur les tableaux statiques

- `count(x)` renvoi le nombre d'occurrences de `x` dans le tableau.
- `fromlist(list)` ajoute les éléments de la liste à notre tableau
- `pop([i])` supprime l'élément d'indice `i` du tableau
- `reverse()` inverse l'ordre des éléments du tableau.
- `tolist()` convertit le tableau en une liste ordinaire avec les mêmes éléments.
- ...

Python

array vs list

- list peut contenir des éléments de types différents mais pas array.
- list est généralement plus grande en termes d'utilisation de la mémoire pour stocker le même nombre d'éléments que array.
- list offre un large éventail de méthodes qui permettent des opérations telles que l'ajout, la suppression et la modification.

Python

Pour récupérer la data du jour

```
from datetime import datetime

date = datetime.now()
print(date)
# affiche 2020-06-25 21:20:02.420685
```

Python

Pour récupérer la data du jour

```
from datetime import datetime

date = datetime.now()
print(date)
# affiche 2020-06-25 21:20:02.420685
```


datetime.today() VS datetime.new()

- `datetime.now()` : retourne la date et l'heure actuelles, avec la possibilité d'ajuster le résultat au fuseau horaire spécifié. Si aucun fuseau horaire n'est fourni, elle retourne l'heure locale.
- `datetime.today()` : retourne la date et l'heure actuelles, sans la possibilité d'ajuster le résultat au fuseau horaire spécifié.

Python

Pour extraire les différentes parties de la date

```
from datetime import datetime

date = datetime.now()

annee = date.year
mois = date.month
jour = date.day
heure = date.hour
minutes = date.minute

print(f"Année: {annee}")
print(f"Mois: {mois}")
print(f"Jour: {jour}")
print(f"Heure: {heure}")
print(f"Minutes: {minutes}")
```

Python

Pour récupérer le nom du mois de la date actuelle en toute lettre

```
print(date.strftime('%B'))
# affiche June
```

Python

Pour récupérer le nom du mois de la date actuelle en toute lettre

```
print(date.strftime('%B'))
# affiche June
```

Pour récupérer le nom du mois de la date actuelle en toute lettre en français

```
import datetime
import locale

locale.setlocale(locale.LC_TIME,'fr_FR')
date = datetime.datetime.now()
print(date.strftime('%B'))
# affiche juin
```

Python

Quelques autres indices pour les dates

- `d` : le jour du mois (de 01 à 31)
- `a` : une représentation textuelle du jour (trois lettres)
- `A` : une représentation textuelle complète du jour
- `w` : une représentation numérique du jour (0 pour dimanche, 6 pour samedi)
- `j` : le jour de l'année (de 001 à 366)
- `B` : une représentation textuelle complète du mois (January à December)
- `m` : une représentation numérique du mois (de 01 à 12)
- `b` : une représentation textuelle courte du mois (trois lettres)
- `Y` : une représentation numérique de l'année (4 chiffres)
- `y` : une représentation numérique de l'année (2 chiffres)

PHP

Quelques autres indices pour les heures

- `p` : `AM` ou `PM` en majuscule
- `I` : format d'heure de 00 à 12
- `H` : format d'heure de 00 à 23
- `M` : minutes avec un zéro au début (de 00 à 59)
- `S` : secondes avec un zéro au début (de 00 à 59)
- `f` : microsecondes
- `Z` : fuseau horaire (exemples : UTC, GMT, Atlantic/Azores)
- `T` : abréviations du fuseau horaire (exemples : EST, MDT)

Python

Pour afficher une date sous un format précis, on utilise les options précédentes

```
print(date1.strftime('%Y/%m/%d  %H:%M:%S'))
```

Python

Pour ajouter ou soustraire un certain nombre de jours (ou autre d'une `datetime`)

```
from datetime import datetime, timedelta

# Créer une date initiale
date_initiale = datetime(2024, 2, 4)

# Ajouter 10 jours
date_plus_10_jours = date_initiale + timedelta(days=10)

print(f"Date initiale: {date_initiale.date()}")
# affiche Date initiale: 2024-02-04

print(f"Après ajout de 10 jours: {date_plus_10_jours.date()}")
#affiche Après ajout de 10 jours: 2024-02-14

date_moins_5_jours = date_initiale - timedelta(days=5)

print(f"Après soustraction de 5 jours: {date_moins_5_jours.date()}")
# affiche Après soustraction de 5 jours: 2024-01-30
```

Python

Pour comparer les dates, on peut utiliser les opérateurs de comparaison

```
from datetime import datetime

date1 = datetime(2023, 2, 4)
date2 = datetime(2024, 3, 1)

print(f"Date1 est avant Date2: {date1 < date2}")
# affiche Date1 est avant Date2 : True

print(f"Date1 est après Date2: {date1 > date2}")
# affiche Date1 est après Date2 : False

print(f"Date1 est la même que Date2: {date1 == date2}")
# affiche Date1 est la même que Date2 : False
```

Python

Pour calculer la différence entre deux dates en jours

```
from datetime import datetime

date1 = datetime(2023, 2, 4)
date2 = datetime(2024, 3, 1)

difference = date2 - date1
print(f"Différence en jours: {difference.days}")
# affiche Différence en jours: 391
```

Python

Commençons par importer `time`

```
from time import *
```

Python

Commençons par importer `time`

```
from time import *
```

Pour obtenir le timestamp actuel

```
timestamp = time()
print("Timestamp actuel:", timestamp)
Timestamp actuel: 1707979755.572501
```

Python

Pour obtenir le timestamp à partir d'un `datetime`

```
from datetime import datetime

now = datetime.now()
timestamp = datetime.timestamp(now)
print("Timestamp actuel:", timestamp)
# affiche Timestamp actuel: 1707979755.572501
```

Python

Pour obtenir le timestamp à partir d'un `datetime`

```
from datetime import datetime

now = datetime.now()
timestamp = datetime.timestamp(now)
print("Timestamp actuel:", timestamp)
# affiche Timestamp actuel: 1707979755.572501
```

Pour obtenir un `datetime` à partir d'un timestamp

```
from datetime import datetime

timestamp = 1644921600

date = datetime.fromtimestamp(timestamp)

print("datetime :", date)
# affiche datetime : 2022-02-15 11:40:00
```

Python

typing

- Introduit dans **Python 3.5**.
- Facilitant ainsi le typage statique.
- Fournissant des outils pour spécifier les types de variables et les retours de fonction.

Python

typing : vue d'ensemble

- Généricité : `TypeVar:` et `Generic[T]`
- Types spéciaux : `Union`, `Any`, `Optional[T]`, `NoReturn`...
- Valeurs autorisées : `Literal`
- ...

Python

Exemple avec `Literal`

```
from typing import Literal

def couleur_preferee(couleur: Literal['rouge', 'vert', 'bleu']):
    print(f"Ma couleur préférée est {couleur}")

couleur_preferee('rouge')
# affche : Ma couleur préférée est rouge

couleur_preferee('jaune')
# erreur avec MyPy
```

Python

Quelques fonctions prédéfinies sur les collections utilisant callback et Lambda

- `map(callback, iterable)` **[Built-in function]**
- `filter(callback, iterable)` **[Built-in function]**
- `reduce(callback, iterable, [initial])`

Python

Exemple avec `map`

```
tab = [2, 3, 8, 5]

def carre(x):
    return x ** 2

result = list(map(carre, tab))

print(result)
# affiche [4, 9, 64, 25]
```

Python

Exemple avec `map`

```
tab = [2, 3, 8, 5]

def carre(x):
    return x ** 2

result = list(map(carre, tab))

print(result)
# affiche [4, 9, 64, 25]
```


Ou en utilisant une fonction Lambda

```
result = list(map(lambda x: x ** 2, tab))

print(result)
# affiche [4, 9, 64, 25]
```

Python

Exemple avec `filter`

```
tab = [2, 3, 8, 5]

def est_pair(x):
    return x % 2 == 0

result = list(filter(est_pair, tab))

print(result)
# affiche [2, 8]
```

Python

Exemple avec `filter`

```
tab = [2, 3, 8, 5]

def est_pair(x):
    return x % 2 == 0

result = list(filter(est_pair, tab))

print(result)
# affiche [2, 8]
```


Ou en utilisant une fonction Lambda

```
result = list(filter(lambda x: x % 2 == 0, tab))

print(result)
# affiche [2, 8]
```

Python

`map`, `filter` **et** `reduce` **peuvent être appliqués sur les tableaux statiques**

```
from array import array

arr = array('i', [2, 3, 8, -5])

print(array('i', (map(lambda elt: elt + 2, arr))))
# affiche array('i', [4, 5, 10, -3])
```

Python

Exemple avec `reduce`

```
from functools import reduce

tab = [2, 3, 8, 5]

def somme(x, y):
    return x + y

result = reduce(somme, tab)
print(result)
# affiche 18
```

Python

Exemple avec `reduce`

```
from functools import reduce

tab = [2, 3, 8, 5]

def somme(x, y):
    return x + y

result = reduce(somme, tab)
print(result)
# affiche 18
```


Ou en utilisant une fonction Lambda

```
result = reduce(lambda x, y: x + y, tab)

print(result)
# affiche 18
```

TypeScript

Remarques

- Le premier paramètre de `reduce` correspond à une fonction callback ou Lambda
  - Le premier paramètre de la fonction callback ou Lambda correspond au résultat
  - Le deuxième correspond à l'élément du tableau de l'itération courante
  - Le premier paramètre est initialisé par la valeur du premier élément du tableau
- Le deuxième correspond à la liste
- On peut ajouter un troisième paramètre qui correspondra à la valeur initiale du premier paramètre de la fonction callback ou Lambda

Python

Exemple avec `reduce` utilisant un troisième paramètre

```
from functools import reduce

tab = [2, 3, 8, 5]

result = reduce(lambda x, y: x + y, tab, 5)

print(result)
# affiche 23
```

Python

Pour enchaîner les différentes méthodes précédentes, on fait des imbrications

```
from functools import reduce

tab = [2, 3, 8, 5]
result = (reduce(lambda x, y: x + y, filter(lambda x
    : x % 2 == 0, map(lambda x: x ** 2, tab))))

print(result)
# affiche 68
```

Python

Remarques

- Le code précédent permettait d'imbriquer les méthodes.
- Il ne permettait pas le chaînage comme en **Java**, **C#** et **JavaScript** (Approche fonctionnelle)
- Pour pouvoir enchaîner les méthodes précédentes comme en **Java**, **JavaScript** et **C#**, on peut utiliser **PyFunctional** (anciennement appelé **ScalaFunctional**)

Python

Remarques

- Le code précédent permettait d'imbriquer les méthodes.
- Il ne permettait pas le chaînage comme en **Java**, **C#** et **JavaScript** (Approche fonctionnelle)
- Pour pouvoir enchaîner les méthodes précédentes comme en **Java**, **JavaScript** et **C#**, on peut utiliser **PyFunctional** (anciennement appelé **ScalaFunctional**)

Pour installer `PyFunctional`

```
pip install PyFunctional
```

Python

Ensuite, nous pourrons enchaîner ces méthodes de la manière suivante

```
from functional import seq

tab = [2, 3, 8, 5]

result = seq(tab)\
    .map(lambda x: x ** 2)\
    .filter(lambda x: x % 2 == 0)\
    .reduce(lambda x, y: x + y)

print(result)
# affiche 68
```

Python

Ou avec une syntaxe Linq (de Microsoft)

```
from functional import seq

tab = [2, 3, 8, 5]

result = seq(tab)\
    .select(lambda x: x ** 2)\
    .where(lambda x: x % 2 == 0)\
    .reduce(lambda x, y: x + y)

print(result)
# affiche 68
```

Python

Étant donné le tuple suivant :

```
marques = ("peugeot", "ford", "toyota")
```

Exercice

En utilisant `map`, `filter` et `reduce`, calculer le nombre de caractères total des marques ayant un nombre pair de caractères.

Python

Solution

```
from functional import seq

marques = ("peugeot", "ford", "toyota")

result = seq(marques)\
    .map(lambda x: len(x))\
    .filter(lambda x: x % 2 == 0)\
    .reduce(lambda x, y: x + y)

print(result)
# affiche 10
```

Python

Autres fonctions de `functional`

- `find(func)`
- `flat_map(func)`
- `drop(n)`
- `drop_right(n)`
- `drop_while(func))`
- `for_each(func)`
- `count(func)`
- ...

Python

Quelques autres exemples

```
from functional import seq

tab = [2, 3, 8, 5]

print(seq(tab).find(lambda v: v % 2 == 0))
# affiche 2

print(seq(tab).count(lambda v: v % 2 == 0))
# affiche 2
```

Python

Fonctions de quantification

- `take(n)`
- `take_while(func)`
- `any()`
- `exists(func)`
- `all()`
- `for_all(func)`
- ...

Python

Exemples avec les opérateurs de quantification

```
from functional import seq

tab = [2, 3, 8, 5]

print(seq(tab).all())
# affiche True

print(seq(tab).any())
# affiche True

print(seq(tab).exists(lambda v: v % 2 == 0))
# affiche True

print(seq(tab).for_all(lambda v: v % 2 == 0))
# affiche False
```

Python

Fonctions d'agrégation de `functional`

- `min()`
- `min_by(func)`
- `max()`
- `max_by(func)`
- `average`, `sum`, `product`
- `distinct()`
- `distinct_by(func)`
- ...

Python

Built-in functions vs `Functional`

- L'utilisation de `Functional` peut être plus claire pour ceux qui préfèrent la programmation fonctionnelle
- Pour de petites à moyennes collections, la différence de performance entre ces deux approches est généralement négligeable.
- `Functional` pourrait ajouter un léger surcoût en raison des abstractions supplémentaires : un surcoût souvent minime et n'est significatif que dans des scénarios de performance critique ou avec de très grands ensembles de données.

Python

Pour en apprendre plus

- https://pypi.org/project/ScalaFunctional/
- https://docs.pyfunctional.pedro.ai/en/latest/functional.html#module-functional.pipeline

Python

Fonctionnalités du module `os`

- Navigation dans le système de fichiers : `getcwd`, `chdir`, `listdir`...
- Gestion de fichiers et de répertoires : `mkdir`, `rmdir`, `rename`...
- Manipulation de chemins : `path.dirname`, `path.basename`...
- Exécution de commandes système : `system(...)`
- Travail avec des variables d'environnement : `getenv`...
- Gestion des permissions et des identifiants : `chmod`, `chown`...

Python

Fonctionnalités du module `os`

- Navigation dans le système de fichiers : `getcwd`, `chdir`, `listdir`...
- Gestion de fichiers et de répertoires : `mkdir`, `rmdir`, `rename`...
- Manipulation de chemins : `path.dirname`, `path.basename`...
- Exécution de commandes système : `system(...)`
- Travail avec des variables d'environnement : `getenv`...
- Gestion des permissions et des identifiants : `chmod`, `chown`...

Commençons par importer le module

```
import os
```

Python

Pour récupérer le nom du répertoire de travail (CWD : Current Working Directory) du fichier utilisé

```
print(os.getcwd())
# affiche C:\Users\user\cours-modules
```

Python

Pour récupérer le nom du répertoire de travail (CWD : Current Working Directory) du fichier utilisé

```
print(os.getcwd())
# affiche C:\Users\user\cours-modules
```

Pour récupérer les fichiers d'un répertoire passé en paramètre (sinon le répertoire courant)

```
arr = os.listdir()
print(arr)
# affiche ['.idea', 'main.py', 'package', 'venv', '__pycache__']
```

Python

Pour récupérer le nom du répertoire de travail (CWD : Current Working Directory) du fichier utilisé

```
print(os.getcwd())
# affiche C:\Users\user\cours-modules
```

Pour récupérer les fichiers d'un répertoire passé en paramètre (sinon le répertoire courant)

```
arr = os.listdir()
print(arr)
# affiche ['.idea', 'main.py', 'package', 'venv', '__pycache__']
```

Pour changer de répertoire, on utilise `chdir()`

```
os.chdir('package')
print(os.getcwd())
# affiche C:\Users\user\cours-modules\package
```

Python

Pour créer un répertoire

```
os.mkdir('a')
```

Python

Pour créer un répertoire

```
os.mkdir('a')
```


mkdir

- lance une exception si le répertoire existe
- ne permet pas de créer une arborescence de répertoires

Python

Pour créer une arborescence de répertoires

```
os.makedirs('b/c/d')
```

Python

Pour créer une arborescence de répertoires

```
os.makedirs('b/c/d')
```


Remarque

`makedirs` lance une exception si le répertoire existe déjà

Python

Pour créer une arborescence de répertoires

```
os.makedirs('b/c/d')
```


Remarque

`makedirs` lance une exception si le répertoire existe déjà

Pour créer récursivement des répertoires et ne pas lever d'exception si le répertoire existe déjà

```
os.makedirs('b/c/d', exist_ok=True)
```

Python

Pour supprimer un répertoire vide

```
os.rmdir('a')
```

Python

Pour supprimer un répertoire vide

```
os.rmdir('a')
```


rmdir

- lance une exception si le répertoire n'existe pas
- ne permet pas de supprimer un répertoire non vide
- ne permet pas de supprimer une arborescence de répertoires

Python

Pour supprimer un répertoire et tous ses sous-répertoires vides de manière récursive

```
os.removedirs('b\c\d')
```

Python

Pour supprimer un répertoire et tous ses sous-répertoires vides de manière récursive

```
os.removedirs('b\c\d')
```

Remarques

- `removedirs` ne permet pas de supprimer un répertoire non vide
- Pour supprimer un répertoire non-vide, il faut utiliser module `shutil`

Python

Pour renommer un répertoire

```
os.rename('package', 'paquet')
```

Python

Pour renommer un répertoire

```
os.rename('package', 'paquet')
```

Pour supprimer un fichier

```
os.remove('file.py')
```

Python

os.system()

`os.system()` prend comme paramètre une commande qu'on peut exécuter dans un invite de commande (terminal).

Python

Pour vider l'écran (sous Windows)

```
os.system('cls')
```

Python

Pour vider l'écran (sous Windows)

```
os.system('cls')
```

Pour vider l'écran (sous Linux)

```
os.system('clear')
```

Python

Pour vider l'écran (sous Windows)

```
os.system('cls')
```

Pour vider l'écran (sous Linux)

```
os.system('clear')
```

Pour récupérer la date système

```
os.system('date')
```

Python

Pour construire un chemin relatif avec une gestion correcte des séparateurs.

```
chemin = os.path.join('paquet', 'subpackage', 'fonctions.py')

print (chemin)
# affiche paquet\subpackage\fonctions.py
```

Python

Pour avoir des informations sur un chemin, on utilise `os.path()`

```
print(os.path.dirname(chemin))
# affiche paquet\subpackage

print(os.path.basename(chemin))
# affiche fonctions.py

print(os.path.exists(chemin))
# affiche True

print(os.path.isfile(chemin))
#affiche True

print(os.path.isdir(chemin))
#affiche False
```

Python

Exercice

- Affichez le contenu du répertoire courant
- Pour chaque élément, affichez :
  - s'il est répertoire
  - s'il est fichier
  - son chemin absolu

Python

Pour afficher tous les dossiers du répertoire de travail, on utilise `walk` qui retourne à chaque itération le chemin vers le répertoire parcouru, ses sous-répertoires et ses fichiers

```
for root, dirs, files in os.walk(os.getcwd()):
    for file in files:
        print(os.path.join(root, file))
```

Python

glob

- fonction de recherche selon un motif
- similaire à `walk` du module `os`

Python

glob

- fonction de recherche selon un motif
- similaire à `walk` du module `os`

Commençons par importer `glob`

```
from glob import *
```

Python

Pour afficher le contenu du répertoire de travail (la racine du projet)

```
for elt in glob("*"):
    print(elt)
```

Python

Pour afficher le contenu du répertoire de travail (la racine du projet)

```
for elt in glob("*"):
    print(elt)
```

Pour afficher le contenu des sous répertoires définis dans le répertoire de travail (la racine du projet)

```
for elt in glob("**/*"):
    print(elt)
```

Python

Pour afficher le contenu du répertoire de travail (la racine du projet)

```
for elt in glob("*"):
    print(elt)
```

Pour afficher le contenu des sous répertoires définis dans le répertoire de travail (la racine du projet)

```
for elt in glob("**/*"):
    print(elt)
```

Depuis Python 3.5, tt glob permet d'effectuer une recherche récursive

```
for elt in glob("**/*", recursive=True):
    print(elt)
```

Python

Pour afficher les fichiers ayant une extension `.py` situés dans les sous répertoires définis dans le répertoire de travail (la racine du projet)

```
for elt in glob("**/*.py"):
    print(elt)
```

Python

Pour afficher les fichiers ayant une extension `.py` situés dans les sous répertoires définis dans le répertoire de travail (la racine du projet)

```
for elt in glob("**/*.py"):
    print(elt)
```

Pour une recherche en profondeur des fichiers ayant une extension `.py` dans le répertoire de travail (la racine du projet)

```
for elt in glob("**/*.py", recursive=True):
    print(elt)
```

Python

`os.walk` VS `glob`

- `os.walk` est principalement conçu pour parcourir récursivement une arborescence de répertoires.
- `os.walk` n'effectue pas de filtrage basé sur des motifs.
- `glob.glob` n'est pas récurisve par défaut mais permet d'effectuer une recherche selon un motif.

Python

`shutil` : **sh**ell **util**ities

propose des opérations de haut niveau

- `rmtree()` pour supprimer un répertoire non-vide
- `copyfile()` pour copier un fichier
- `copytree()` pour copier un répertoire non-vide
- `move()` pour déplacer un répertoire non-vide
- ...

Python

`shutil` : **sh**ell **util**ities

propose des opérations de haut niveau

- `rmtree()` pour supprimer un répertoire non-vide
- `copyfile()` pour copier un fichier
- `copytree()` pour copier un répertoire non-vide
- `move()` pour déplacer un répertoire non-vide
- ...

Commençons par importer `shutil`

```
import shutil
```

Python

Pour copier le répertoire non vide `paquet` dans `package` (qui sera créé)

```
shutil.copytree('paquet', 'package')
```

Python

Pour copier le répertoire non vide `paquet` dans `package` (qui sera créé)

```
shutil.copytree('paquet', 'package')
```

Pour supprimer le répertoire non vide et toute l'arborescence

```
shutil.rmtree('paquet')
```

Python

Pour copier le répertoire non vide `paquet` dans `package` (qui sera créé)

```
shutil.copytree('paquet', 'package')
```

Pour supprimer le répertoire non vide et toute l'arborescence

```
shutil.rmtree('paquet')
```

Pour déplacer `package` dans `paquet` (qui sera recréé, `package` sera supprimé)

```
shutil.move('package', 'paquet')
```

Exercice

Écrire un script **Python** qui permet de :

1. afficher le chemin du répertoire de travail
2. créer un répertoire `a`
3. se déplacer dans `a`
4. afficher le chemin du répertoire de travail
5. dans `a` créer trois fichiers `f1.txt`, `f2.txt` et `f3.txt`
6. dans `a` créer deux répertoires `b` et `c`
7. supprimer `f2.txt`
8. renommer `f3.txt` en `f2.txt`
9. se déplacer dans le répertoire parent (racine du projet)
10. afficher le chemin du répertoire de travail
11. lister le contenu de `a` en précisant pour chaque élément s'il est fichier ou répertoire

Python

Correction

```
import os, glob

# question 1
print(os.getcwd())
# question 2
os.system('mkdir a')
# question 3
os.chdir('a')
# question 4
print(os.getcwd())
# question 5
os.system('copy NUL f1.txt')
os.system('copy NUL f2.txt')
os.system('copy NUL f3.txt')
# question 6
os.mkdir('b')
os.system('mkdir c')
# question 7
os.remove('f2.txt')
# question 8
os.rename('f3.txt', 'f2.txt')
# question 9
os.chdir('..')
# question 10
print(os.getcwd())
# question 11
for elt in glob.glob("a/*", recursive=True):
    print(elt, 'file' if os.path.isfile(elt) else 'directory')
```

Pour supprimer le répertoire `a` s'il existe à chaque exécution du projet on utilise la méthode `rmtree()` de `shutil` (qui permet de supprimer un répertoire non vide)

```
import os, glob, shutil

if os.path.exists('a'):
    shutil.rmtree('a', ignore_errors=True)
# question 1
print(os.getcwd())
# question 2
os.system('mkdir a')
# question 3
os.chdir('a')
# question 4
print(os.getcwd())
# question 5
os.system('copy NUL f1.txt')
os.system('copy NUL f2.txt')
os.system('copy NUL f3.txt')
# question 6
os.mkdir('b')
os.system('mkdir c')
# question 7
os.remove('f2.txt')
# question 8
os.rename('f3.txt', 'f2.txt')
# question 9
os.chdir('..')
# question 10
print(os.getcwd())
# question 11
for elt in glob.glob("a/*", recursive=True):
    print(elt, 'file' if os.path.isfile(elt) else 'directory')
```

Python

sys

module fournissant l'accès à certaines variables et fonctions qui ont une forte interaction avec l'interpréteur **Python**

Python

sys

module fournissant l'accès à certaines variables et fonctions qui ont une forte interaction avec l'interpréteur **Python**

Commençons par importer sys

```
import sys
```

Python

Pour récupérer les paramètres envoyés au lancement du programme

```
print(sys.argv)
```

Python

Pour récupérer les paramètres envoyés au lancement du programme

```
print(sys.argv)
```

Pour lancer le programme

```
python main.py 2 5 8
```

Python

Pour récupérer les paramètres envoyés au lancement du programme

```
print(sys.argv)
```

Pour lancer le programme

```
python main.py 2 5 8
```

Résultat

```
['main.py', '2', '5', '8']
```

Python

`sys` **peut aussi nous permettre de quitter le programme**

```
if len(sys.argv) < 2:
    sys.exit("Erreur détectée, arrêt du programme.")

print(sys.argv)
```

Python

`sys` **peut aussi nous permettre de quitter le programme**

```
if len(sys.argv) < 2:
    sys.exit("Erreur détectée, arrêt du programme.")

print(sys.argv)
```

Pour lancer le programme

```
python main.py
```

Python

`sys` **peut aussi nous permettre de quitter le programme**

```
if len(sys.argv) < 2:
    sys.exit("Erreur détectée, arrêt du programme.")

print(sys.argv)
```

Pour lancer le programme

```
python main.py
```

Résultat

```
Erreur détectée, arrêt du programme.
```

Python

Pour récupérer la version de Python utilisée

```
print(sys.version)
# affiche 3.11.7 (tags/v3.11.7:fa7a6f2, Dec  4 2023, 19:24:49) [MSC v
   .1937 64 bit (AMD64)]
```

Python

importlib

module permettant de réaliser une importation dynamique

Python

importlib

module permettant de réaliser une importation dynamique

Considérons le fichier `fr.py` ayant le contenu suivant

```
def salutation():
    print('salut')
```

Python

importlib

module permettant de réaliser une importation dynamique

Considérons le fichier `fr.py` ayant le contenu suivant

```
def salutation():
    print('salut')
```

Et `en.py` avec le contenu suivant

```
def salutation():
    print('hello')
```

Python

Importons `importlib` et réalisons l'import dynamique suivant

```
import importlib
import locale

language = "fr" if locale.getlocale()[0] == 'fr_FR' else "en"
module = importlib.import_module(language)

module.salutation()
```

Python

Considérons la liste suivante

```
liste1 = [1, 2, 3, 4]
```

Python

Considérons la liste suivante

```
liste1 = [1, 2, 3, 4]
```

Créons `liste2` à partir de `liste1`

```
liste2 = liste1
```

Python

Considérons la liste suivante

```
liste1 = [1, 2, 3, 4]
```

Créons `liste2` **à partir de** `liste1`

```
liste2 = liste1
```


Modifier `liste1` $\Rightarrow$ **modifier** `liste2`

```
liste1[1] = 10
print(liste2)
# affiche [1, 10, 3, 4]
```

Python

Pour éviter le comportement précédent , commençons par importe `copy`

```
import copy
```

Python

Pour éviter le comportement précédent , commençons par importe `copy`

```
import copy
```

Utilisons `deepcopy` pour cloner la liste

```
liste2 = copy.deepcopy(liste1)
```

Python

Pour éviter le comportement précédent , commençons par importe `copy`

```
import copy
```

Utilisons `deepcopy` **pour cloner la liste**

```
liste2 = copy.deepcopy(liste1)
```

Modifier `liste1` $\not\Rightarrow$ **modifier** `liste2`

```
liste1[1] = 10

print(liste1)
# affiche [1, 10, 3, 4]

print(liste2)
# affiche [1, 2, 3, 4]
```

Python

Conventions

- Éviter `import *`
- Une documentation par module à placer au tout début du fichier (avant les `import`)
- Les `imports` au tout début du fichier après la documentation
- Un `import` par ligne
- Placer les `imports` dans un bloc `try ... except` pour capturer les exceptions de type `ImportError`

Python

Conventions

- Éviter `import *`
- Une documentation par module à placer au tout début du fichier (avant les `import`)
- Les `imports` au tout début du fichier après la documentation
- Un `import` par ligne
- Placer les `imports` dans un bloc `try ... except` pour capturer les exceptions de type `ImportError`

Extension **VSC**

Flake8 : permet de vérifier si les bonnes pratiques sont respectées, signaler les imports inutiles...