



# Python intermédiaire



# Expression lambda

Les expressions lambda en Python permettent de créer des fonctions anonymes, c'est-à-dire des fonctions sans nom. Elles sont souvent utilisées pour des opérations simples qui sont passées en tant qu'arguments à des fonctions de plus haut niveau comme `map()` et `filter()`. Pour la définir il faut utiliser le mot clé **lambda**.

Syntaxe :

*lambda arguments : expression*

```
add_ten = lambda x: x + 10
print(add_ten(5)) # Sortie : 15

# Fonction lambda qui multiplie deux nombres
multiply = lambda x, y: x * y
print(multiply(x: 3, y: 4)) # Sortie : 12
```

# Dans les fonction map et filter



**Map** prend en paramètre un tableau, dont chaque élément sera traité par une fonction, un nouveau tableau de longueur équivalent à celui traité sera retourné.

```
# Utilisation de map() avec une expression lambda pour ajouter 10 à chaque élément d'une liste
numbers = [1, 2, 3, 4]
result = map(lambda x: x + 10, numbers)
print(list(result)) # Sortie : [11, 12, 13, 14]
```

**Filter** prend en paramètre un tableau, dont chaque élément sera évalué par une fonction afin d'être filtré, un nouveau tableau avec les éléments retenus sera retourné.

```
# Utilisation de filter() avec une expression lambda pour filtrer les nombres pairs d'une liste
numbers = [1, 2, 3, 4, 5, 6]
result = filter(lambda x: x % 2 == 0, numbers)
print(list(result)) # Sortie : [2, 4, 6]
```

# Fonction avec lambda en argument



```
def apply_function(f, x):  
    return f(x)  
  
# Utilisation d'une lambda pour doubler un nombre  
result = apply_function(lambda x: x * 2, x: 5)  
print(result) # Sortie : 10
```

# Décorateurs en Python

---

- Les décorateurs permettent d'étendre et de modifier le comportement des fonctions et méthodes. Ils "décorent" une fonction existante avec une autre fonction.
- Ils sont souvent utilisés pour ajouter des fonctionnalités transversales (logging, mesure de performance, vérification des droits d'accès, etc.) sans modifier le corps de la fonction décorée.

```
def mon_decorateur(fonction):  
    def enveloppe():  
        print("Exécuté avant la fonction principale.")  
        fonction()  
        print("Exécuté après la fonction principale.")  
    return enveloppe  
  
@mon_decorateur  
def dit_bonjour():  
    print("Bonjour !")  
  
dit_bonjour()
```

# Les contexte avec 'with'

Les context managers en Python sont utilisés pour gérer des ressources de manière sûre et efficace. Ils permettent d'allouer et de libérer des ressources de manière automatique. L'instruction `with` est couramment utilisée pour encapsuler les opérations nécessitant une initialisation et une finalisation.

En utilisant `with` nous sommes certain que le fichier sera bien fermé après sa lecture.

```
with open('example.txt', 'r') as file:  
    content = file.read()  
    print(content)
```

# Création d'un contexte manager

Utilisation du décorateur context manager qui sera préalablement importé

Le mot-clé `yield` renvoie le curseur au bloc `with` qui utilise ce context manager. Le code à l'intérieur du bloc `with` peut maintenant utiliser ce curseur pour exécuter des commandes SQL.

```
import sqlite3
from contextlib import contextmanager

@contextmanager
def database_connection(db_name):
    conn = sqlite3.connect(db_name)
    cursor = conn.cursor()
    try:
        yield cursor
    finally:
        conn.commit()
        conn.close()

# Utilisation du context manager pour la base de données
with database_connection('example.db') as cursor:
    cursor.execute('SELECT * FROM users')
    print(cursor.fetchall())
```

Une fois que le bloc `with` est terminé (que ce soit par une exécution normale ou en raison d'une exception), le contrôle revient à la fonction `database_connection`. Le code après `yield` est alors exécuté : `conn.commit()` et `conn.close()`

# Itérable et itérateur

Exemples d'itérables : listes, tuples, dictionnaires, ensembles, chaînes de caractères

`iter()` permet de créer un itérateur.

```
numbers = [1, 2, 3, 4, 5]
it = iter(numbers)
print(next(it)) # Sortie : 1
print(next(it)) # Sortie : 2
```

A chaque appel de `next()`, l'élément suivant est appelé



# Fonction Génératrice



Une fonction génératrice ressemble à une fonction normale mais utilise **yield** pour renvoyer des valeurs une par une. Avantages :

- Les générateurs ne chargent pas tout le contenu en mémoire, ils génèrent les éléments à la volée.
- Les valeurs sont calculées seulement quand elles sont nécessaires.

```
def generate_numbers():  
    yield 1  
    yield 2  
    yield 3  
  
gen = generate_numbers()  
print(next(gen)) # Sortie : 1  
print(next(gen)) # Sortie : 2  
print(next(gen)) # Sortie : 3
```

# Exemple pratique, lecture d'un fichier ligne après ligne



```
def read_large_file(file_path):  
    with open(file_path) as file:  
        for line in file:  
            yield line  
  
line = read_large_file('large_file.txt')  
print(next(line))  
print(next(line))  
print(next(line))
```

# Metaclasse



Les métaclasses sont une fonctionnalité avancée de Python permettant de contrôler la création et le comportement des classes. Une métaclasse est une classe de classe, c'est-à-dire qu'elle définit comment les classes se comportent.

```
class AttributeValidationMeta(type):
    def __new__(cls, name, bases, dct):
        if 'required_attr' not in dct:
            raise TypeError(f"La classe {name} doit définir required_attr")
        return super().__new__(cls, name, bases, dct)

class MyClass(metaclass=AttributeValidationMeta):
    required_attr = "Ce champ est requis"

class MyInvalidClass(metaclass=AttributeValidationMeta):
    pass # Cette classe lève une exception car required_attr n'est pas défini
```

# Métaclasse

```
class MyMeta(type):
    def __new__(cls, name, bases, dct):
        print(f"Création de la classe {name}")
        return super().__new__(cls, name, bases, dct)
```

1 usage

```
class MyClass(metaclass=MyMeta):
    pass
```

```
# Création d'une instance de MyClass
instance = MyClass()
```

**MyMeta** : Cette classe hérite de `type` et redéfinit la méthode `__new__`.

`__new__` : Cette méthode est appelée avant `__init__`. Elle prend comme arguments :

- `cls` : La classe actuelle.
- `name` : Le nom de la classe à créer.
- `bases` : Les classes de base de la classe à créer.
- `dct` : Le dictionnaire contenant les attributs et méthodes de la classe.

`super().__new__` : Appelle la méthode `__new__` de la superclasse pour créer la classe.

# Les modules



Les modules permettent de diviser le code en plusieurs parties, facilitant ainsi son organisation, sa réutilisabilité et sa maintenabilité. Il existe 3 catégories de module :

- **Modules standards** : Ils sont inclus dans la bibliothèque standard de Python et offrent un ensemble de fonctionnalités prêtes à l'emploi.
- **Modules externes** : Développés par d'autres, ces modules sont disponibles via des gestionnaires de paquets comme PyPI.
- **Modules personnalisés** : Ceux que l'on développe soi-même pour répondre à des besoins spécifiques.

# Installer des bibliothèques externes avec PIP

- `pip` est l'outil de gestion de paquets standard pour Python, permettant d'installer, mettre à jour et supprimer des bibliothèques et des outils Python. Savoir utiliser `pip` est essentiel pour gérer efficacement les dépendances de vos projets Python.
- Principales commandes de `pip` :
  - Utilisez la commande `pip install nom_de_la_bibliotheque` pour installer une nouvelle bibliothèque.
  - `pip list` affiche la liste de toutes les bibliothèques Python installées dans l'environnement actuel.
  - Pour mettre à jour une bibliothèque existante : `pip install --upgrade nom_de_la_bibliotheque`.
  - `pip uninstall nom_de_la_bibliotheque` pour supprimer une bibliothèque installée.

Installation de la bibliothèque request :

```
mathias@mathias-XPS-13-9370 > ~/Documents/Dawan/Cours_python/exemple_code > pip install requests
```

Liste les bibliothèques installées :

```
(base) mathias@mathias-XPS-13-9370 > ~/Documents/Dawan/Cours_python/exemple_code > pip list
```

Package	Version
alabaster	0.7.12
anaconda-client	1.7.2
anaconda-navigator	2.0.3
anaconda-project	0.9.1

# Construire une Bibliothèque Python

- Une bibliothèque Python peut être un simple fichier `.py` contenant des fonctions et des classes.
- Exemple : Créez un fichier `mathutils.py` contenant des fonctions mathématiques personnalisées.

```
# mathutils.py
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y
```

- Importez et utilisez votre bibliothèque dans d'autres scripts Python.

```
# main.py
from mathutils import add

print(add(5, 3))
```

# Fonctions Utilitaires et Bibliothèques en Python



- Python intègre des fonctions directement accessibles, nous en avons déjà utilisées : `print()`, `type()`, `len()`, `int()`, `float()`, `input()`, `sorted()`, etc.
- D'autres fonctions utilitaires spécialisées nécessitent l'import de bibliothèques :
  - Les bibliothèques en Python sont des ensembles de fonctions et de modules qui vous permettent d'effectuer de nombreuses tâches sans avoir à réinventer la roue. Elles étendent les fonctionnalités de Python, vous permettant d'effectuer des opérations complexes, souvent en une seule ligne de code.
  - Elles contribuent à la standardisation du code et améliorent sa lisibilité et sa maintenance.



# Importer des Bibliothèques :

- L'utilisation du mot-clé `import` permet d'accéder à toutes les fonctions et classes disponibles dans une bibliothèque.
- Vous pouvez importer une bibliothèque entière ou des éléments spécifiques en utilisant la syntaxe `from [bibliothèque] import [fonction]`.
- Exemples :

Import des bibliothèques de l'ensemble des fonctionnalités  
math et datetime :

```
import math
import datetime

# Utiliser la constante PI de la bibliothèque math
print(math.pi) # Affiche la valeur de PI

# Utiliser la fonction today de la classe date du module datetime
print(datetime.date.today()) # Affiche la date d'aujourd'hui
```

Import de la méthode sqrt et de la propriété pi  
uniquement de la bibliothèque math :

```
from math import sqrt, pi

print(sqrt(25)) # Affiche 5.0
print(pi)       # Affiche la valeur de PI
```

Import de la méthode datetime en la renommant dt :

```
from datetime import datetime as dt

print(dt.now()) # Affiche la date et l'heure actuelles
```

# Construire une Bibliothèque Python

- Une bibliothèque Python peut être un simple fichier `.py` contenant des fonctions et des classes.
- Exemple : Créez un fichier `mathutils.py` contenant des fonctions mathématiques personnalisées.

```
# mathutils.py
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y
```

- Importez et utilisez votre bibliothèque dans d'autres scripts Python.

```
# main.py
from mathutils import add

print(add(5, 3))
```

# Création d'un Package Python

- Un package est un dossier contenant plusieurs modules (fichiers `.py`) et un fichier spécial `__init__.py`.
- Structure :

```
monpackage/  
├─ __init__.py  
├─ module1.py  
└─ module2.py
```

- `__init__.py` peut être vide ou contenir du code d'initialisation pour le package.

# Zoom sur `__init__.py`

- Le fichier `__init__.py` est un script Python qui est exécuté lorsqu'un package est importé.
- Il peut être utilisé pour exécuter du code d'initialisation pour un package ou pour définir ce qui sera importé lorsque vous utilisez `from package import *`.
- Sans `__init__.py`, Python ne reconnaîtra pas le dossier comme un package valide.
- `__init__.py` peut initialiser des variables, importer des modules spécifiques ou définir des variables `__all__` pour contrôler ce qui est importé avec `import *`. Exemple :

```
# __init__.py
__all__ = ['module1', 'module2']
print("Initialisation du package monpackage")
```

- **Bonnes Pratiques :**
  - Gardez `__init__.py` léger pour des importations rapides.
  - Utilisez-le judicieusement pour initialiser votre package, mais évitez d'y mettre trop de logique ou de code lourd.

# Importer des Éléments d'un Package

- Importez des fonctions spécifiques à partir de modules dans un package.

```
# main.py
from monpackage import module1

module1.ma_fonction()
```

L'appel de la fonction se avec le nom du module "point" le nom de la fonction

# Les environnements virtuels



## Définition d'un Environnement Virtuel en Python

Un **environnement virtuel en Python** est un espace isolé dans lequel il est possible d'installer des paquets et des dépendances nécessaires pour un projet spécifique sans interférer avec les autres projets ou avec les paquets installés au niveau du système.

## Objectifs d'un Environnement Virtuel

1. **Isolation des Dépendances** : Permet d'éviter les conflits entre différentes versions de paquets nécessaires à différents projets.
2. **Gestion Facile des Paquets** : Facilite l'installation, la mise à jour et la suppression des paquets sans affecter d'autres projets ou l'environnement global Python.
3. **Reproductibilité** : Assure que d'autres développeurs peuvent reproduire l'environnement de développement exact, garantissant que les projets fonctionnent de la même manière sur différentes machines.

# venv (module standard)



Inclus dans Python 3.3 et les versions ultérieures.

Création : `python -m venv env`

Activation :

- Windows : `myenv\Scripts\activate`
- Unix ou MacOS : `source myenv/bin/activate`

Désactivation : `deactivate`

# virtualenv



**virtualenv** permet de créer des environnements isolés pour les projets Python. Chaque environnement peut avoir ses propres bibliothèques et versions de bibliothèques, indépendamment des autres projets.

Installation de virtualenv :

```
pip install virtualenv
```

Création d'un environnement virtuel:

```
virtualenv cours_virtualenv
```

Activation de cet environnement virtuel:

```
source cours_virtualenv/bin/activate
```

avec Linux

```
myenv\Scripts\activate
```

avec Windows



# Conserver les dépendances liées à un environnements

Installation de dépendances :

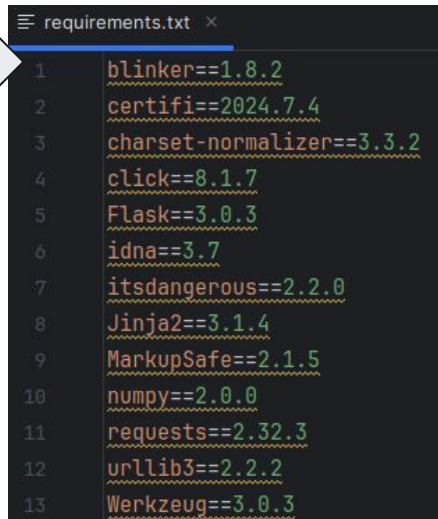
```
pip install requests flask numpy
```

Création de requirements.txt :

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

Création d'un nouvel environnement virtuels avec les dépendances définies dans requirements.txt:

```
virtualenv myenv
myenv\Scripts\activate # Windows
source myenv/bin/activate # Unix ou MacOS
pip install -r requirements.txt
```



```
requirements.txt
1 blinker==1.8.2
2 certifi==2024.7.4
3 charset-normalizer==3.3.2
4 click==8.1.7
5 Flask==3.0.3
6 idna==3.7
7 itsdangerous==2.2.0
8 Jinja2==3.1.4
9 MarkupSafe==2.1.5
10 numpy==2.0.0
11 requests==2.32.3
12 urllib3==2.2.2
13 Werkzeug==3.0.3
```

# Désactiver un environnement virtuel virtualenv

Commande :

```
$ deactivate
```



Après le passage de la commande `deactivate`, (nom\_env) n'est plus indiqué en début de prompt : nous sommes sorti de l'environnement virtuel.

```
(cours_virtualenv) mathias-tranzer@mathias-t  
mathias-tranzer@mathias-tranzer-Dell-G15-552
```

# buildout : Gestion Avancée des Environnements et des Dépendances

**buildout** est un outil de gestion de projet qui permet de créer des environnements isolés et de gérer les dépendances de manière avancée.

Installation de buildout :

```
pip install zc.buildout
```

Créer un fichier *buildout.cfg* :  
indiquant les dépendances à  
installer (et autres tâches) lors  
de la création de  
l'environnement virtuel.

```
buildout.cfg x
1 [buildout]
2
3 parts = python
4
5 [python]
6 recipe = zc.recipe.egg
7 eggs = request flask numpy
8
```

# buildout.cfg

La section buildout indique les différentes étapes et configuration à réaliser lors de du lancement de buildout

```
buildout.cfg x
1 [buildout]
2
3 parts = python
4
5 [python]
6 recipe= zc.recipe.egg
7 eggs = request flask numpy
8
```

Etapes à réaliser dans le bloc "python"

- **recipe = zc.recipe.egg** : Utilise la recette `zc.recipe.egg` pour installer des paquets Python. Les "recipes" dans `buildout` sont des plugins qui définissent comment installer ou configurer des parties spécifiques d'un projet. Chaque recipe offre une fonctionnalité spécifique, permettant de gérer différents types de dépendances, configurations et environnements.
- **eggs** : Liste des paquets Python à installer dans cette partie. Ici, `requests`, `flask`, et `numpy` seront installés.

# Exemple de *buildout.cfg* complexe

```
buildout.cfg
1 [buildout]
2 parts = python flake8 node download
3 extends = base.cfg
4 allow-picked-versions = false
5 show-picked-versions = true
6 versions = versions
7
8 [python]
9 recipe = zc.recipe.egg
10 eggs =
11     requests
12     flask
13     numpy
14
15 [flake8]
16 recipe = zc.recipe.egg
17 eggs = flake8
18
19
20 [node]
21 recipe = gp.recipe.node
22 node-version = 12.16.1
23 npm-version = 6.14.4
24
25 [download]
26 recipe = hexagonit.recipe.download
27 url = http://example.com/file.tar.gz
28 download-only = true
29
30 [versions]
31 requests = 2.25.1
32 flask = 1.1.2
33 numpy = 1.19.5
34 flake8 = 3.9.2
```

## Section [buildout]

- `parts = python flake8, node, download` : Indique que `buildout` doit configurer les parties `python`, `flake8`, et `node` et `downloads`.
- `extends = base.cfg` : Inclut un fichier de configuration supplémentaire nommé `base.cfg`.
- `allow-picked-versions = false` : Empêche `buildout` de choisir des versions de paquets non spécifiées.
- `show-picked-versions = true` : Affiche les versions des paquets choisis par `buildout`.
- `versions = versions` : Spécifie que les versions des paquets se trouvent dans la section `[versions]`.

## Autres Sections

- `[python]`, `[flake8]`, `[node]`, `[downloads]` : Configurent les parties respectives en utilisant la recette `zc.recipe.egg`, `zc.recipe.node`, `hexagonit.recipe.download` pour installer des paquets et dépendances.
- `[versions]` : Spécifie les versions exactes des paquets à utiliser pour garantir la reproductibilité.

# Exécution du buildout

Taper la commande :

```
▶ buildout
```



Vous pouvez rencontrer une erreur de ce type : “`ImportError: cannot import name 'packaging' from 'pkg_resources'`” lié à un problème de compatibilité avec la dernière version du package *setuptools*. Pour régler le problème vous pouvez installer une version antérieure de *setuptools* :

```
$ pip install setuptools==69.5.1
```