BDW - Programmation web - Python avancé

Fabien Duchateau

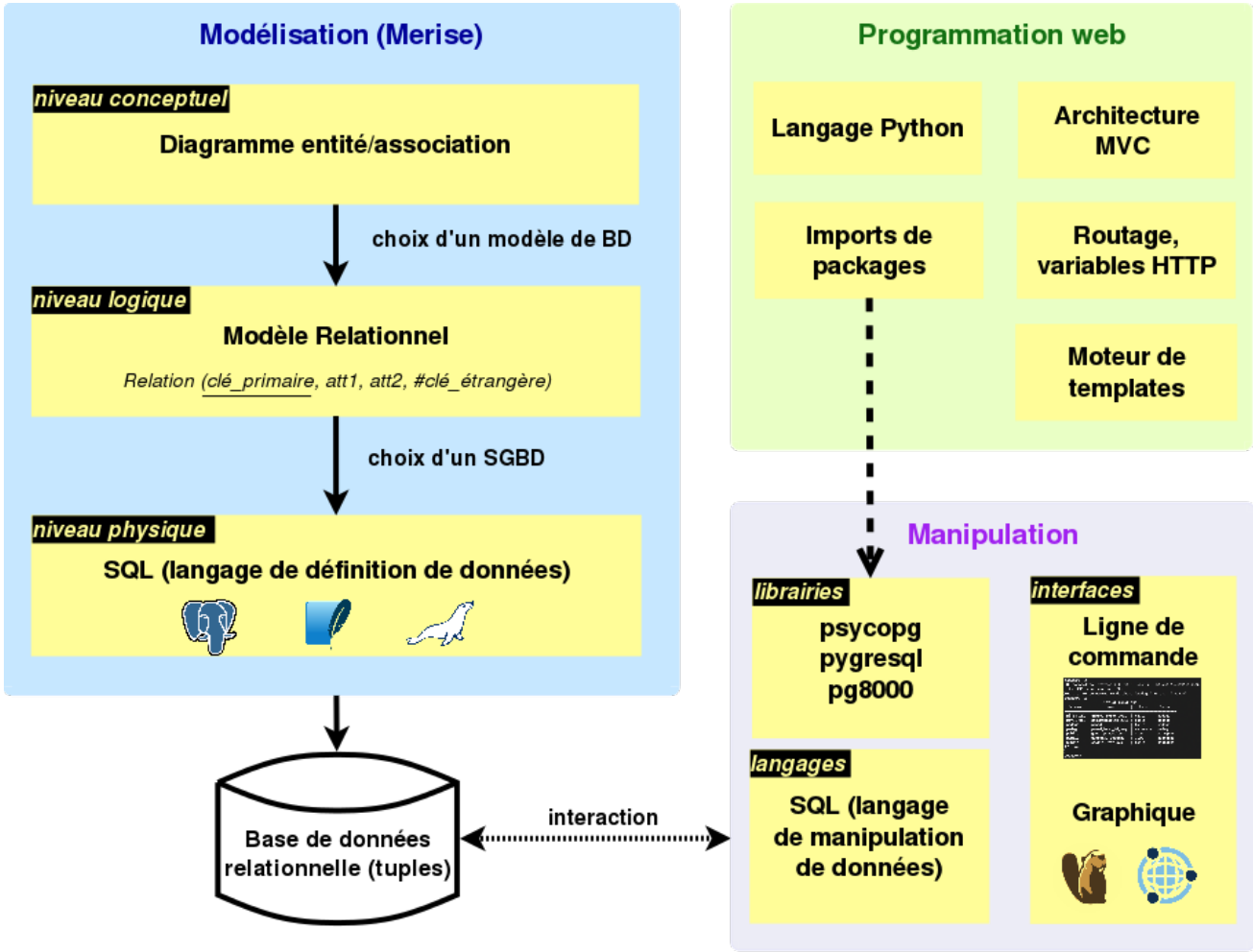
fabien.duchateau [at] univ-lyon1.fr

Université Claude Bernard Lyon 1

2024 - 2025

https://perso.liris.cnrs.fr/fabien.duchateau/BDW/

Positionnement dans BDW

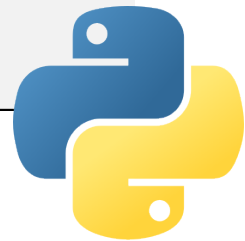


Ces diapositives utilisent le genre féminin (e.g., chercheuse, développeuses) plutôt que l'écriture inclusive (moins accessible, moins concise, et pas totalement inclusive)

Rappels

- Syntaxe de Python
- Opérations et structures de contrôle (conditionnelles et boucles)
- Définition de fonctions

```python
def ma_fonction(x):
    # Retourne la somme de 0 à x
    somme = 0
    if x > 0:  # x nombre
    ↪  positif
        for i in range(0, x):
            print(i)
            somme += i
    return somme
```



Dans ce cours, système d'import et interaction entre Python et une base de données PostgreSQL

Plan

Motivation

En programmation, deux manières de gérer les erreurs :

- *Look Before You Leap* (LBYL) = vérifier des conditions avant d'exécuter une action qui peut échouer
- *Easier to Ask Forgiveness than Permission* (EAFP) = exécuter l'action, et gérer en cas d'échec (généralement avec des exceptions)

En Python, la second option (EAFP) est conseillée (si pertinente)

Voir aussi *duck typing* et polymorphisme (encouragés par Python)
https://realpython.com/python-lbyl-vs-eafp/

Style LBYL

```python
def division(a, b):
    if b == 0:  # cas exceptionnel
        print("Erreur : division par zéro")
        return None
    return a / b  # situation fréquente
```

Exemple de fonction division (style LBYL)

- Code mettant le focus sur le cas exceptionnel
- Erreur possible malgré une vérification (e.g., existence d'un fichier vérifié, mais supprimé par un autre programme)
- Problèmes de performance (e.g., une fonction de conversion en entier vérifie qu'une variable contienne uniquement des chiffres, puis la transforme en entier, donc re-vérification)

Style EAFP

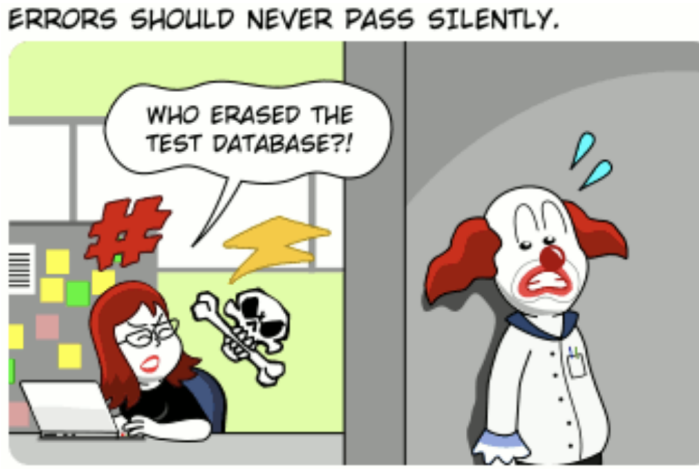
```python
def division(a, b):
    try:
        return a / b  # situation fréquente
    except ZeroDivisionError:  # cas exceptionnel
        print("Erreur : division par zéro")
        return None
```

Exemple de fonction division (style EAFP)

- Code mettant le focus sur le cas normal (meilleure lisibilité)
- Gestion des cas exceptionnels avec un bloc `try … except` et des exceptions typées
- Rares cas où les performances sont moins bonnes qu'en LBYL (e.g., initialisation d'une clé inexistante dans un dictionnaire)

Syntaxe des exceptions

```python
try:
    # instruction(s) pouvant échouer
except TypeError:  # type spécifique
    # gestion de ce type d'échec
except KeyError:  # type spécifique
    # gestion de ce type d'échec
# ...
except Exception:  # type général
    # gestion de ce type d'échec
```



- Types d'exception prédéfinis (e.g., `SyntaxError`, `TypeError`, `FileNotFoundError`)
- Possible de définir de nouveaux types
- Levée d'une exception avec `raise`

https://docs.python.org/3/library/exceptions.html#exception-hierarchy

https://docs.python.org/3/reference/simple_stmts.html#the-raise-statement

Plan

Définitions

Un module est un fichier Python contenant des définitions (variable, fonction)

- Structuration du code
- Réutilisation de code
- Évite les conflits de nommage

Un *package* est une collection de modules

- Installation facilitée
- Partage

Import de module/package

C'est l'instruction `import` qui permet d'importer un module ou package

- Généralement les imports sont regroupés en début de fichier
- Deux syntaxes pour importer tout ou partie d'un module
- Pour un module dans un sous-répertoire : `sous_rep.module`

```
import module
```

Import de toutes les définitions du module

```
from module import définition1, définition2, ...
```

Import de définitions spécifiques du module

Aussi des options de renommage du (nom de) module, etc. (voir modules)

Où trouver des modules/packages ?

Développement de ses propres modules

Python Standard Library (PSL) :

- Les modules de base (e.g., avec la fonction `print`)
- Des modules optionnels (généralement installés avec la distribution Python, besoin de les importer pour les utiliser)

Python Package Index (PyPi) :

- Nombreux modules supplémentaires (500k+ en 2024)
- Bien vérifier l'état du module (date de dernière mise à jour, communauté, etc.)

---

https://docs.python.org/3/tutorial/modules.html

Import de ses propres modules

somme.py

```python
x = 1

def sommer(a, b):
    return a+b
```

Définition d'une variable et d'une fonction dans un fichier

main.py

```python
import somme

somme.sommer(somme.x, 2)
```

Import du module, et utilisation des définitions en précisant le nom du module

main2.py

```python
from somme import sommer, x

sommer(x, 2)
```

Import de définitions spécifiques, utilisées directement

La notation alternative `from somme import *` est déconseillée

Import de packages de la PSL

Module `random` :

- Générateur de nombres aléatoires
- Sélection de valeurs aléatoires dans une séquence

```python
import random

x = random.randint(0, 10)
print(x)  # ex, 7
random.choice([1, 2, 3, 4]) # ex, 2
```

Module `datetime` :

- Manipulation de date et heure

```python
from datetime import datetime

d = datetime.now()  # objet avec la
↪  date/heure courante
d.isoformat()  # ex, 2024-09-01
↪  10:27:48.971564
```

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

Import de packages de la PSL (2)

Module os :

- Opérations en lien avec le système d'exploitation
- Répertoires (création, etc.), droits d'accès
- Gestion des processus

Module os.path :

- Manipulation du système de fichiers (chemins)

```python
import os
from os import path

chemin = '/tmp'

# test si chemin est un rép.
if path.isdir(chemin):
    rep = 'test'  # rép. à créer
    chemin_rep = path.join(chemin,
    ↪ rep)  # /tmp/test/
    os.mkdir(chemin_rep)  #
    ↪ création de /tmp/test/
    fichiers = os.listdir(chemin)
    print(f"Liste des fichiers de
    ↪ {chemin} : {fichiers}")
```

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

Import de packages de la PSL (3)

Module `sys` :

- Opérations en lien avec l'interpréteur (terminal)
- Arguments de la ligne de commande
- Chemin de recherche des modules

```python
import sys

args = sys.argv  # arguments
if '-h' in args:
    print("Affiche les arguments du
    ↪ script")
    sys.exit(0)  # quitte (code 0)
else:
    for arg in args:
        print(arg)
```

Module `re` :

- Expressions régulières

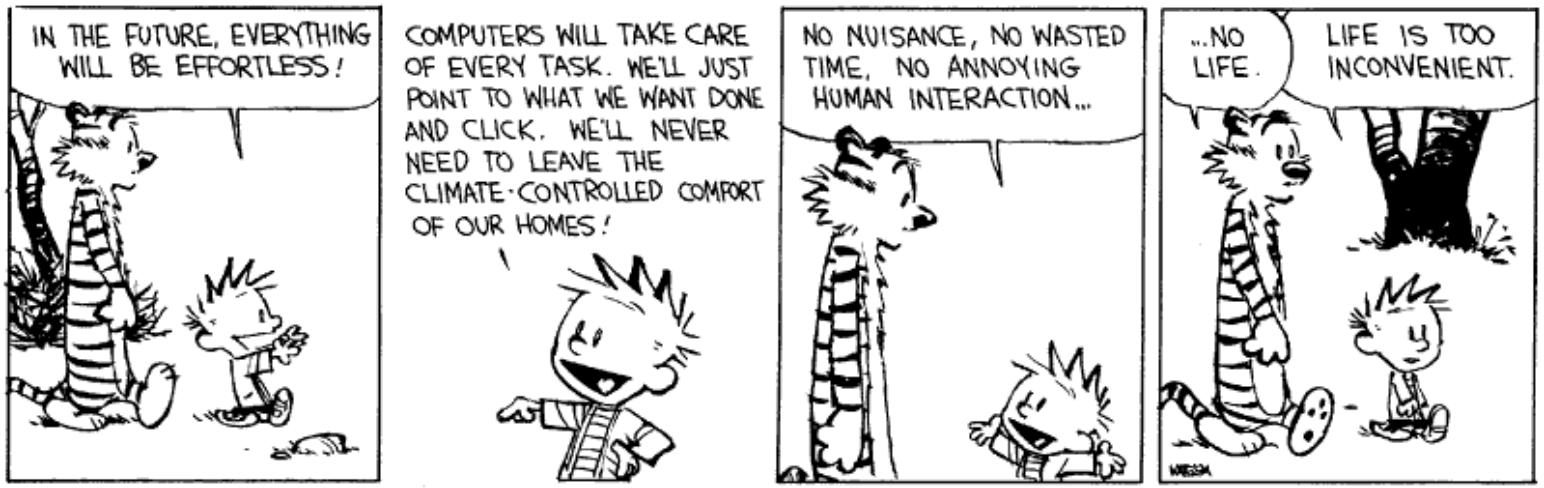
```python
import re

re.findall(r'\s?(BD.*?)\s', 'BDW LC
↪ BDA IHM BDBIO ')  # ['BDW',
↪ 'BDA', 'BDBIO']
```

Import de packages de la PSL (4)

Et beaucoup d'autres...

- `shutil`, `glob`, `tempfile`, `request`
- `math`, `statistics`
- `argparse`, `doctest`, `unittest`, `logging`
- `sqlite3`, `json`, `csv`, `xml`



Calvin et Hobbes (Bill Watterson)

Import de packages PyPI

Pour installer un package de Pypi :

- Activer un environnement virtuel !
- Utiliser `pip install`


```
# activer un env. virtuel
source .venv/bin/activate
# installer mocodo
python -m pip install mocodo
# lancer mocodo
python -m mocodo
```

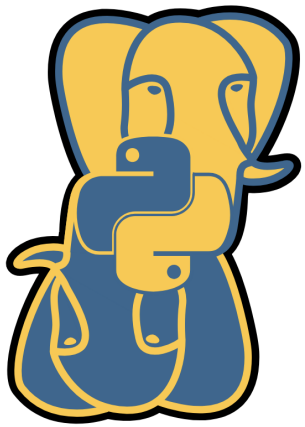
Exemple avec Mocodo (outil de modélisation en base de données)

https://pypi.org/
https://pypi.org/project/mocodo/

Plan

Généralités

Pour interagir avec les bases de données, Python possède des modules pour chaque SGBD (e.g., `sqlite3` pour SQLite, `pymongo` pour MongoDB)



Pour le SGBD PostgreSQL :

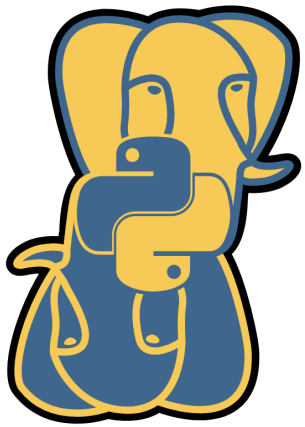
- pyscopg : librairie la plus populaire
- `pygresql`, `pyodbc`, `pg8000`, etc.
- Utilisation d'un ORM (Object Relational Mapping)

---

https://wiki.python.org/moin/PostgreSQL
https://realpython.com/python-sql-libraries/
https://www.psycopg.org/
https://www.pygresql.org/
https://github.com/mkleehammer/pyodbc

Généralités

Pour interagir avec les bases de données, Python possède des modules pour chaque SGBD (e.g., `sqlite3` pour SQLite, `pymongo` pour MongoDB)



Pour le SGBD PostgreSQL :

- **psycopg** : librairie la plus populaire
- `pygresql`, `pyodbc`, `pg8000`, etc.
- Utilisation d'un ORM (Object Relational Mapping)

---

https://wiki.python.org/moin/PostgreSQL
https://realpython.com/python-sql-libraries/
https://www.psycopg.org/
https://www.pygresql.org/
https://github.com/mkleehammer/pyodbc

Interaction avec une base de données

Les modules de base de données en Python se conforment à la PEP 0249 (DB-API), qui spécifie un accès standardisé aux SGBD

Quatre étapes pour interagir avec une base de données :

1. Connexion au SGBD
2. Exécution d'une requête (préparée ou non)
3. Récupération du résultat
4. Fermeture de la connexion

On peut itérer les étapes 2 à 3 autant de fois que l'on veut avant de fermer la connexion à l'étape 4

---

https://peps.python.org/pep-0249/

Étape de connexion

Informations de connexion :

- Paramètres : serveur hébergeant le SGBD, nom d'utilisatrice, mot de passe, nom de la base de données et nom du (répertoire) schéma
- Ces informations sensibles sont généralement stockées dans un fichier à part (avec des droits restreints)

```
SERVER = "bd-pedago.univ-lyon1.fr"  # adresse serveur
USER = "p1234567"  # login
PASSWORD = "remplace42moi"  # mot de passe
DATABASE = "p1234567"  # nom de la BD
SCHEMA = "tp1"  # nom du (répertoire) schéma
```

https://www.psycopg.org/psycopg3/docs/basic/usage.html

Étape de connexion (2)

Connexion avec `psycopg.connect()` :

- Retourne un objet `connexion` qui représente le lien de connexion vers la base de données choisie
- Une exception est levée si la connexion n'aboutit pas

```python
import psycopg

def get_connexion(host, username, password, db, schema):
    try:
        connexion = psycopg.connect(host=host, user=username,
        ↪  password=password, dbname=db, autocommit=True)
        # on sélectionne le schéma par une requête SET
        cursor = psycopg.ClientCursor(connexion)
        cursor.execute("SET search_path TO %s", [schema])
    except Exception as e:
        print(e)
        return None
    return connexion
```

Étape d'exécution

Curseur : objet qui permet d'exécuter une (ou plusieurs) requête et de récupérer le résultat sous forme d'un objet Python (liste de tuples ou de dictionnaires)

```
curseur = connexion.cursor()
```

Création d'un curseur à partir d'un objet connexion et de la méthode `cursor()`

```
curseur.execute("SELECT nomSerie FROM Series")
```

Exécution d'une requête SQL (simple) à partir du curseur et de la méthode `execute()`

https://www.psycopg.org/psycopg3/docs/api/cursors.html

Étape d'exécution - exemple de requête simple

```python
def get_series(connexion):
try:
    cursor = connexion.cursor()
    cursor.execute("SELECT * FROM Actrices")
    return cursor  # liste de tuples
except psycopg.Error as e:
    print(f"Error : {e}")
return None
```

Résultat (sous forme d'une liste de tuples) :

```
[   (111, 'Bean', 'Sean'),
    (222, 'Fairley', 'Michelle'),
    (333, 'Cuoco', 'Kaley'),
    (444, 'Parsons', 'Jims'),
    (555, 'Avgeropoulos', 'Marie'), ]
```

Étape d'exécution - exemple de requête simple (2)

```
1  from psycopg.rows import dict_row
2
3  def get_series_as_dict(connexion):
4  try:
5      cursor = connexion.cursor()
6      cursor.row_factory = dict_row  # ligne importante
7      cursor.execute("SELECT * FROM Actrices")
8      return cursor  # liste de dict
9  except psycopg.Error as e:
10     print(f"Error : {e}")
11 return None
```

Résultat (sous forme d'une liste de dictionnaires) :

```
[   {'numINSEE': 111, 'nom': 'Bean', 'prenom': 'Sean'},
    {'numINSEE': 222, 'nom': 'Fairley', 'prenom': 'Michelle'},
    {'numINSEE': 333, 'nom': 'Cuoco', 'prenom': 'Kaley'},
    ... ]
```

Étape d'exécution - exemple de requête avec variable

```python
def get_episodes(connexion, num_episode):
    try:
        cursor = connexion.cursor()
        cursor.execute("SELECT * FROM Episodes WHERE
            ↪ numéro=%s", [num_episode])
        return cursor
    except psycopg.Error as e:
        print(f"Error : {e}")
    return None
```

*Cette fonction nécessite un paramètre `num_episode` afin d'afficher tous les épisodes avec ce numéro. Pour construire la requête, on **ne doit pas concaténer** la variable au SQL, mais utiliser le second paramètre de la méthode `execute()` : une liste de variables dont les valeurs remplacent (respectivement) les %s du SQL*

Précisions sur le curseur

- Contient différentes informations (état, description, nombre d'instances récupérées ou modifiées, etc.) et des méthodes (`execute()`, `executemany()`, `fetchone()`, etc.)
- Est un objet itérable (accès aux instances résultat)
- Quand on lit les instances (résultat d'une requête) d'un curseur, pas possible de relire les résultats précédents
- Les requêtes dynamiques (i.e., avec un nom d'identifiant SQL représenté par une variable, e.g., un nom de table ou d'attribut) ont une syntaxe spéciale (sécurité)

---

https://www.psycopg.org/psycopg3/docs/api/sql.html
http://kunststube.net/escapism/

Étape d'exécution - exemple de requête dynamique

```python
from psycopg import sql

def get_instances(connexion, nom_table):
try:
    cursor = connexion.cursor()
    query = sql.SQL('SELECT * FROM
    ↪  {table}').format(table=sql.Identifier(nom_table), )
    cursor.execute(query)
    return cursor
except psycopg.Error as e:
    print(f"Error : {e}")
return None
```

Pour afficher les instances d'une table quelconque (paramètre `nom_table`), on construit la requête avec `sql.SQL()` et `sql.Identifier` permet d' associer le nom de la table dans la requête SQL au paramètre `nom_table`

Étape de récupération du résultat

Comment récupérer le résultat selon le type de requête ?

- Requête de type SELECT : le résultat est sous forme d'instances ⇒ méthodes `fetchone()`, `fetchmany()`, `fetchall()` du curseur

- Autres types de requête : le résultat est un nombre d'instances (affectées, e.g., nombre d'instances créées, mises à jour ou supprimées) ⇒ attribut `rowcount` du curseur

---

Les méthodes `fetch...()` retournent une liste contenant soit des tuples, soit des dictionnaires (selon `cursor.row_factory`)

Étape de récupération du résultat - exemples

- Lecture du nombre d'instances affectées (attribut `rowcount` du curseur) dans le cas d'une requête autre que SELECT

```
cursor.execute("UPDATE Actrices SET salaire=1000")
nb_maj = cursor.rowcount  # integer
```

- Lecture d'une seule instance en utilisant la méthode `fetchone()` du curseur

```
cursor.execute("SELECT count(*) FROM Actrices")
nb_actrices = cursor.fetchone()[0]  # 1 tuple avec 1 élement
```

Étape de récupération du résultat - exemples (2)

- Lecture de chaque instance en itérant sur le curseur

```
cursor.execute("SELECT * FROM Actrices")
for instance in cursor:
    print(instance)  # tuple avec zéro ou plusieurs éléments
```

- Lecture de chaque instance en utilisant la méthode `fetchall()` du curseur

```
cursor.execute("SELECT * FROM Actrices")
actrices = cursor.fetchall()
for instance in actrices:
    print(instance)  # tuple avec zéro ou plusieurs éléments
```

Afin de séparer les différentes parties du code, il est conseillé que les fonctions d'accès aux données retournent une liste Python (avec `fetchall()`).

Requêtes préparées

Une requête préparée (ou requête paramétrable) est une requête récurrente, que l'on compile avec des variables, et donc réutilisable en fournissant les valeurs manquantes (visibilité limitée à la session du SGBD)

Avantages d'une requête préparée :

- Performances (la requête est déjà compilée, cf optimisation)
- Économiser de la bande passante
- Éviter les risques d'injection SQL pour certains systèmes (paramètres transmis sous forme binaire)

Éxécution d'une requête préparée

```
def disconnect(connection):
    connection.close()
    return True
```

Le paramètre `prepare=True` de la méthode `execute()` indique de préparer la requête (plan calculé et stocké par PostgreSQL).

Étape de déconnexion

- Conseillé de fermer la connexion en fin de session (si non fait automatiquement, e.g., par un *context manager*)
- Méthode `close` sur l'objet `connexion`

```
    print(instance)  # tuple avec zéro ou plusieurs
    ↪ éléments
# utilisation directe du curseur
cursor.execute("SELECT * FROM Actrices")
```

En résumé

- Différents packages pour PostgreSQL, dont `psycopg`
- Quatre étapes pour se connecter et exécuter des requêtes

Exemple de page dynamique utilisant l'extension psycopg

Plan

Les templates

Un *template* (gabarit) est un modèle dans un langage (e.g., HTML, Latex) qui inclut des instructions particulières (e.g., conditionnelle, affichage de variable) interprétées par le moteur de templates

- Factorisation et réutilisation du code (DRY)
- Code lisible par les humains
- Séparation des différentes parties du code

En programmation web, un template est un fichier HTML avec des variables et des expressions

---

https://wiki.python.org/moin/Templating

https://en.wikipedia.org/wiki/Don't_repeat_yourself

Moteur de templates

Des dizaines de moteurs de templates pour Python...

- Exemple avec Mustache/Chevron (limité à de la substitution)

```python
import chevron

chevron.render('<h1>{{ titre }}</h1>',
    {'titre': 'Hello world !'})  # <h1>Hello world !</h1>
```

- Exemple avec Jinja

```python
from jinja2.nativetypes import NativeEnvironment

env = NativeEnvironment()  # environnement jinja basique
template = env.from_string('<h1>{{ titre }}</h1>')
result = template.render(titre='Hello world !')
print(result)  # <h1>Hello world !</h1>
```

Package Jinja

Un moteur de templates couplé à Python :

- Substitution de variables et traitements plus complexes
- Système d'héritage entre templates
- ...

Avec le serveur web fourni, fonctionnement simplifié (transparent) :

- Pas besoin de gérer les environnements
- Pas besoin de passer des variables aux templates
- Concentration sur l'écriture des templates

---

https://jinja.palletsprojects.com/

Syntaxe générale

Un template HTML Jinja peut contenir :

- Du code HTML (!)
- Des instructions de substitution de variables

```
<div class="error">{{ message }}</div>
```

- Des instructions avancées (structures de contrôle)

```
{% if True %}
    <span>vrai</span>
{% endif %}
```

- Des commentaires Jinja

```
{# voici un commentaire #}
```

Substitution de variables

- Délimiteurs {{…}} pour remplacer une variable par sa valeur
- Notation avec crochets ou notation pointée pour accéder aux éléments d'une structure

```python
hello = 'Hello world'  # variable définie en python
mon_dict = {'a': 1, 'b': 2}  # variable définie en python
```

Variables déclarées dans un fichier Python et passées au template

```
<div>{{ hello }}</div>
<div>{{ mon_dict['a'] }}</div>
<div>{{ mon_dict.b }}</div>
```

Fichier template avec du code HTML et des variables substituées

Filtres

Jinja fournit un certain nombre de filtres

- Symbole | pour appliquer un filtre à une variable

```python
ma_liste = list(1, 2, 3, 4, 5)  # variable définie en python
```

Variable déclarée dans un fichier Python et passée au template

```
{# nombre d'éléments dans une séquence #}
<div>{{ ma_liste | length }}</div>

{# remplacement dans une chaine de caractères #}
<div>{{ "Hello World" | replace("Hello", "Bye Bye") }}</div>
```

Fichier template avec du code HTML et des filtres

Dans la majorité des cas, mieux vaut transformer les données soit en SQL soit en Python au lieu d'utiliser ces filtres

Test conditionnel

- Permet l'affichage d'un bloc de code HTML selon une condition
- Syntaxe du `if` similaire à Python (`elif` et `else` optionnels)
- Expressions (e.g., `==`, `!=`, `and`, `or`, `not`, `in`)

```
1 {% if a_gagne %}
2     <span class="success">Bravo !</span>
3 {% else %}
4     <span class="info">Prochain tour.</span>
5 {% endif %}
```

Affichage d'un message selon la valeur de la variable `a_gagne` : des félicitations (avec classe `success`) ou le prochain tour

https://jinja.palletsprojects.com/en/3.1.x/templates/#list-of-control-structures

https://jinja.palletsprojects.com/en/3.1.x/templates/#expressions

Boucle

- Permet de répéter un bloc de code HTML (e.g., pour construire dynamiquement le contenu d'un tableau, d'une liste déroulante ou d'une liste ordonnée)
- Syntaxe du `for` similaire à Python

```
1 <ul>
2 {% for valeur in ma_liste %}
3     <li>{{ valeur }}</li>
4 {% endfor %}
5 </ul>
```

Template avec boucle pour construire une liste non ordonnée (<ul>) dont les éléments (<li>) proviennent d'une liste

```
<ul>
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
</ul>
```

Code HTML généré

Boucle (2)

```python
mon_dict = {'BDW': 'Base de données et programmation web',
            'BDA': 'Base de données avancées'}
```

Variable déclarée dans un fichier Python et passée au template

```
1 <dl>
2 {% for cle, val in mon_dict %}
3     <dt>{{ cle }}</dt>
4     <dd>{{ val }}</dd>
5 {% endfor %}
6 </dl>
```

Template avec boucle pour construire une liste de description (<dl>) dont le contenu (<dt> et <dd>) provient d'un dictionnaire

```
<dl>
    <dt>BDW</dt>
    <dd>Base de données et
    ↪   programmation web</dd>
    <dt>BDA</dt>
    <dd>Base de données
    ↪   avancées</dd>
</dl>
```

Code HTML généré

BDW
Base de données et programmation web
BDA
Base de données avancées

Includes

- Jinja permet d'inclure un fichier template depuis un autre (équivalent à un copier-coller du premier dans le second)

index.html

```
1 {% include 'header.html' %}
2 <main>
3     <h1>Ma page</h1>
4     <p>blah blah</p>
5 </main>
6 {% include 'footer.html' %}
```

Template principal qui inclut l'entête et le pied de page

header.html

```
<header>
    <img src="logo.jpeg">
    <h1>Nom du site</h1>
</header>
```

Fichier contenant le code du <header>

Utilisé pour factoriser les parties statiques d'un site (code identique quelle que soit la page, e.g., entête, pied de page)

Héritage

Dans Jinja, un template peut hériter d'un autre template :

- Le **template parent** définit des blocs nommés (éventuellement sans contenu)
- Les **templates enfants** étendent un template parent
- Les templates enfants peuvent redéfinir un contenu pour les blocs nommés

Utilisé pour avoir un agencement et un design cohérents sur toutes les pages du site, sans redondance de code :

- Le template parent définit l'agencement général (commun)
- Un template enfant définit le contenu spécifique d'une page

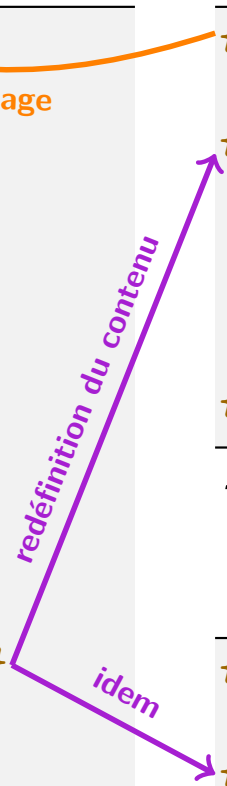
https://jinja.palletsprojects.com/en/3.1.x/templates/#template-inheritance

Héritage (2)

base.html

```
1  <!doctype html>
2  <html>
3  <head>
4    <meta charset="utf-8" />
5    <link type="text/css"
     ↪   href="style.css">
6    <title>Mon super site</title>
7  </head>
8
9  <body>
10     {% include 'header.html' %}
11     <main>
12     {% block main %}{% endblock %}
13     </main>
14   {% include 'footer.html' %}
15 </body>
16 </html>
```

Template de base, avec un bloc nommé `main` vide et qui sera redéfini par les templates enfants



afficher.html

```
{% extends "base.html" %}

{% block main %}
    <h2>Affichage</h2>
    <table>
        ...
    </table>
{% endblock %}
```

Template enfant afficher

ajouter.html

```
{% extends "base.html" %}

{% block main %}
    <h2>Ajout</h2>
    <form>
        ...
    </form>
{% endblock %}
```

Template enfant ajouter

En résumé

- Filtres et méthodes de test (e.g., `is false`, `is string`)
- Assignation de variables (`set`), contrôle fin sur les structure de contrôle, etc.
- Définition de macros (`macro`), héritage avancé
- Gestion de l'échappement, gestionnaire de contexte, internationalisation, etc.

Comment gérer et répartir le code entre les modules Python, les accès base de données et les templates HTML ?

Bilan

- Gestion des erreurs (`try ... except`)
- Utilisation de packages (`import`)
- Interaction avec une BD PostgreSQL (`psycopg`)
- Système de templates pour HTML (`jinja`)

Prochain cours :
structuration d'un site web avec l'architecture MVC