paul.millet@lameduse.fr

LaMeDuSe

PYA : Python, perfectionnement

Version : 28/08/2024

LaMeDuSe

Prérequis

- Bonnes connaissances en développement Python, ou connaissances équivalentes à celles apportées par le cours PYT. Expérience requise.

Objectifs

- Implémenter de manière rigoureuse des Design Patterns reconnus
- Utiliser les techniques avancées du langage Python : Context Manager, métaclasses, closures, fonctions avancées
- Optimiser les performances de vos programmes à l'aide du monitoring et du parallélisme
- Packager et déployer ses artefacts Python
- Exploiter des librairies contribuant au succès du langage : calcul scientifique, Intelligence Artificielle, XML, réseau

Tour de table

- Présentation des membres
- Vos attentes vis-à-vis de la formation

Programme de la formation

1. Rappels importants sur le langage
2. Fonctions avancées
3. Programmation Orientée Objet avancée
4. Déploiement et qualité
5. Le parallélisme : optimiser les performances de vos programmes
6. Les librairies contribuant au succès du langage
7. Ressources des TP
8. Annexe & Ressources connexes

paul.millet@lameduse.fr

LaMeDuSe

PYA : Python, perfectionnement

Rappels importants sur le langage

1. Affectation par référence et types de données modifiables / non modifiables
2. Passage d'arguments et gestion des variables
3. Variables de classe et d'instances
4. Les slices et structures de données avancées
5. L'introspection
6. Éléments avancés des structures de contrôle
7. Travaux pratiques : Optimisation

Affectation par Référence et Types de Données

- Mutable vs Immutable
  - Mutable (modifiable) : liste, dictionnaire, ensemble (list, dict, set)
  - Immutable (non modifiable) : tuple, chaîne de caractères, nombre (tuple, str, int)
- Affectation par Référence
  - En Python, les variables pointent vers des objets en mémoire.
  - Une modification sur une référence mutable affecte toutes les variables qui la partagent.

```
a = [1, 2, 3]
b = a
b.append(4)  # a est aussi modifié !
```

Affectation par Référence et Types de Données : les tuples

- Un tuple est une séquence ordonné et ne peut donc être modifié
- Utilisé pour leur immuablilite, empêche une modification accidentelle
- meilleur performance comparé au liste
- Peuvent servir de clef pour les dictionnaires
- les tuples peuvent permettre un retour de multiples valeur sur les fonctions
- Exemple d'utilisation : des coordonnées gps, retour de plusieurs valeurs, stockage de configuration

Affectation par Référence et Types de Données : les ints

- Les ints stockques des entiers
- il sont immuable car une opération sur la donné assigne une nouvelle valeurs et ne modifie pas la valeur existante.
- Exemple : x += 1 # cette opération ne modifie pas la valeur mais assigne directement 6 (si x vaut 5 par exemple)

Affectation par Référence et Types de Données : les str

- Les str stoque les chaînes de caractères
- ils sont immuable : pour modifier une chaîne il faut la reassigner, on peut pas juste modifier un caractère sans devoir la reassigner

Passage d'Arguments et Gestion des Variables

- Passage d'Arguments dans les Fonctions
  - Par position : def ma_fonction(x, y)
  - Par nom : ma_fonction(x=1, y=2)
  - Valeurs par défaut : def ma_fonction(x=10)
- Variables Locales vs Globales
  - Variables locales : définies dans une fonction, accessibles uniquement à l'intérieur.
  - Variables globales : accessibles dans tout le module, avec global si modifiées.

```
def fonction():
    global var_globale
    var_globale = 10
```

Variables de Classe et d'Instances

- Variables de Classe
  - Partagées entre toutes les instances de la classe.
  - Déclarées directement dans la classe.
  - note : la modification doit être au niveau de la classe modifier au niveau de l'instance entraînera la création d'une variable d'instance
- Variables d'Instance
  - Spécifiques à chaque instance, définies dans __init__().

```
class Exemple:
    var_classe = 0  # Variable de classe
    def __init__(self, var_instance):
        self.var_instance = var_instance
```

Les Slices et Structures de Données Avancées

- Slices
  - Syntaxe : sequence[start:stop:step]
  - Utilisation avancée pour créer des sous-ensembles de données.
- Structures de Données
  - Listes, tuples, ensembles (set), dictionnaires (dict).
  - Collections avancées : deque, Counter, defaultdict.

```
my_list = [0, 1, 2, 3, 4, 5]
sub_list = my_list[1:5:2]  # [1, 3]
```

Les Slices et Structures de Données Avancées : deque

- Double ended queue
- Optimiser pour les fifo ou lifo (first in first out, last in fist out)

```
from collections import deque

# Création d'un deque
d = deque([1, 2, 3])
print(d)  # Affiche deque([1, 2, 3])

d.append(4)        # Ajoute 4 à la fin
d.appendleft(0)    # Ajoute 0 au début
print(d)  # Affiche deque([0, 1, 2, 3, 4])

dernier = d.pop()        # Retire 4
premier = d.popleft()    # Retire 0
print(d)  # Affiche deque([1, 2, 3])
```

Les Slices et Structures de Données Avancées : deque

- Double ended queue
- Optimiser pour les fifo ou lifo (first in first out, last in fist out)

```
from collections import deque

# Création d'un deque
d = deque([1, 2, 3])
print(d)  # Affiche deque([1, 2, 3])

d.append(4)        # Ajoute 4 à la fin
d.appendleft(0)    # Ajoute 0 au début
print(d)  # Affiche deque([0, 1, 2, 3, 4])

d = deque([1, 2, 3, 4, 5])
d.rotate(2)  # Rotation vers la droite de 2 positions
print(d)  # Affiche deque([4, 5, 1, 2, 3])
```

Les Slices et Structures de Données Avancées : deque

- Double ended queue
- Optimisé pour les fifo ou lifo (first in first out, last in fist out)
- les opération d'ajout ou de suppression en début de liste sont coûteuse en python (complexité 0(n) : taille de l'entrée double = temps d'exécution double)
- les opération sur une double ended queue sont de complexité 0(1) : même temps d'exécution peut importe la taille de l'entrée

Les Slices et Structures de Données Avancées : deque

- append(x) : Ajoute l'élément x à la fin du deque.
- appendleft(x) : Ajoute l'élément x au début du deque.
- pop() : Retire et retourne l'élément à la fin du deque.
- popleft() : Retire et retourne l'élément au début du deque.
- clear() : Vide le deque, supprimant tous ses éléments.
- extend(iterable) : Ajoute tous les éléments de l'itérable à la fin du deque.
- extendleft(iterable) : Ajoute tous les éléments de l'itérable au début du deque dans l'ordre inverse.
- rotate(n) : Fait pivoter les éléments du deque de n positions (droite pour n > 0, gauche pour n < 0).
- count(x) : Retourne le nombre d'occurrences de x dans le deque.
- remove(x) : Supprime la première occurrence de l'élément x dans le deque.

Les Slices et Structures de Données Avancées : deque

- index(x, start=0, stop=len(d)) : Retourne l'index de la première occurrence de x dans le deque entre start et stop.
- insert(i, x) : Insère l'élément x à l'indice i dans le deque.
- __getitem__(index) : Permet d'accéder à un élément du deque par son index.
- __setitem__(index, value) : Permet de modifier un élément du deque par son index.
- __delitem__(index) : Permet de supprimer un élément du deque par son index.
- maxlen : Retourne la taille maximale du deque si définie, sinon None.

Les Slices et Structures de Données Avancées : Counter

- Format similaire a un dictionnaire

```
from collections import Counter
c = Counter("aabbbc")
print(c)
# Résultat : Counter({'b': 3, 'a': 2, 'c': 1})

c = Counter("aabbbc")
print(list(c.elements()))
# Résultat : ['a', 'a', 'b', 'b', 'b', 'c']

c = Counter("aabbbc")
c.update("ccc")
print(c)
# Résultat : Counter({'b': 3, 'c': 4, 'a': 2})
```

Les Slices et Structures de Données Avancées : Counter

- Dictionnaire avec valeur par défaut (évite l'erreur keyerror)

```
from collections import defaultdict

# Créer un defaultdict avec des entiers par défaut (initialisés à 0)
d = defaultdict(int)

# Ajouter des éléments
d["a"] += 1
d["b"] += 2

print(d)  # Résultat : defaultdict(<class 'int'>, {'a': 1, 'b': 2})
```

Les Slices et Structures de Données Avancées : Sets

- Entrées unique : Pas de duplicatas
- Convetir une liste en set : set(liste)
- Déclarer un set : var = {1, 2, 3}

```
my_list = [1, 2, 2, 3, 4, 4, 5]
unique_items = set(my_list)  # Output: {1, 2, 3, 4, 5}

# Intersection (elements commun)
intersection = set1 & set2  # Output: {3, 4}
# Union (tout les élements uniques)
union = set1 | set2  # Output: {1, 2, 3, 4, 5, 6}
# Difference (différences dans le set1 mais pas dans le set2)
difference = set1 - set2  # Output: {1, 2}
# Difference Symétrique (soit dans le set1 soit dans le set2)
symmetric_difference = set1 ^ set2  # Output: {1, 2, 5, 6}
```

Les Slices et Structures de Données Avancées

- Introspection des Objets
  - Exploration des attributs et méthodes d’un objet.
  - Utilisation de dir(), type(), id(), getattr(), hasattr().

```
obj = "Hello"
print(dir(obj))  # Liste les méthodes de l'objet
```

Les Slices et Structures de Données Avancées

- dir : listes les méthodes d'un objet

```
x = [1, 2, 3]
print(dir(x))
# Résultat : ['__add__', '__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__ge__', ...]
```

Les Slices et Structures de Données Avancées

- Type : retourne le type de la variable

```
x = [1, 2, 3]
print(type(x))  # Résultat : <class 'list'>
```

Éléments Avancés des Structures de Contrôle

- La Clause else
  - Peut être utilisée avec for, while, try/except.
  - else dans for/while : exécuté si la boucle se termine sans break.
  - else dans try/except : exécuté si aucune exception ne survient.

```
for i in range(3):
    if i == 5:
        break
else:
    print("Boucle terminée sans interruption.")
```

Travaux Pratiques : Optimisation

- Objectif : Intersection de Listes
  - Comparer deux listes et en tirer les éléments communs.
  - Calculer la complexité en temps de l'algorithme.
- Calcul de Complexité
  - Discuter des méthodes pour optimiser cette tâche, par exemple via des ensembles.

```
list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7]
intersection = [value for value in list1 if value
in list2]
```

paul.millet@lameduse.fr

LaMeDuSe

PYA : Python, perfectionnement

Fonctions Avancées et Décorateurs

1. Utilisation avancée des décorateurs
2. Les décorateurs et Design Patterns
3. Fermetures (Closures)
4. Travaux pratiques : Chaînage de consommateurs de données et gestion d'événements

Utilisation avancée des décorateurs

- Qu'est-ce qu'un décorateur ?
  - Une fonction qui prend une autre fonction en argument et renvoie une nouvelle fonction.
  - Ajoute/modifie des fonctionnalités sans altérer le code de la fonction d'origine.
- Création de Décorateurs
  - Syntaxe de base : @mon_decorateur
  - Peut être appliqué sur des fonctions ou des classes.

```
def mon_decorateur(fonction):
    def nouvelle_fonction(*args, **kwargs):
        print("Fonction appelée")
        return fonction(*args, **kwargs)
    return nouvelle_fonction

@mon_decorateur
def dire_bonjour():
    print("Bonjour !")
```

Pipeline de Consommateurs avec Décorateurs

- Chaînage de Décorateurs
  - Enchaîner plusieurs décorateurs pour créer un pipeline de transformation/consommation de données.
- Pipeline de Consommateurs
  - Utiliser plusieurs décorateurs pour traiter une donnée à travers plusieurs étapes.

```
@decorateur_1
@decorateur_2
def traiter_donnees(data):
    pass
```

Les Décorateurs et Design Patterns

- Appliquer des Design Patterns avec les Décorateurs
  - Singleton : Assure qu'une classe n'ait qu'une seule instance.
  - Logger : Enregistre les appels de fonction et les paramètres.

```
def singleton(cls):
    instances = {}
    def get_instance(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]
    return get_instance

@singleton
class MaClasse:
    pass
```

Fermetures (Closures)

- Définition d'une Closure
  - Une fonction définie dans une autre fonction qui capture et conserve les variables locales de la fonction englobante.
- Utilisation des Closures
  - Permet de créer des fonctions avec des variables d’état.
  - Idéales pour la création de générateurs de fonctions.

```
def make_multiplier(n):
    def multiplier(x):
        return x * n
    return multiplier

doubler = make_multiplier(2)
print(doubler(5))  # Affiche 10
```

Travaux Pratiques

1. Chaînage de Consommateurs de Données
   - Objectif : Enchaîner plusieurs décorateurs pour traiter un flux de données.
   - Exercice : Créer une série de décorateurs modifiant des données textuelles, comme l'ajout de préfixes, majuscules, etc.
2. Abonnement à des Événements
   - Objectif : Utiliser les décorateurs pour créer un système d’abonnement et de notification d’événements.
   - Exercice : Implémenter des décorateurs qui écoutent et déclenchent des événements dans une application simple.

Chaînage de Consommateurs de Données - Exemple de Code

```
def ajouter_prefixe(f):
    def wrapper(*args, **kwargs):
        return "Préfixe : " + f(*args, **kwargs)
    return wrapper

def mettre_en_majuscule(f):
    def wrapper(*args, **kwargs):
        return f(*args, **kwargs).upper()
    return wrapper

@ajouter_prefixe
@mettre_en_majuscule
def dire_bonjour():
    return "bonjour"

print(dire_bonjour())  # Affiche "Préfixe : BONJOUR"
```

Abonnement à des Événements avec Décorateurs

```
evenements = {}
def on_event(event_type):
    def decorator(func):
        if event_type not in evenements:
            evenements[event_type] = []
        evenements[event_type].append(func)
        return func
    return decorator

@on_event("message_reçu")
def alerter():
    print("Message reçu !")

def declencher_evenement(event_type):
    for func in evenements.get(event_type, []):
        func()

declencher_evenement("message_reçu")  # Affiche "Message reçu !"
```

paul.millet@lameduse.fr

LaMeDuSe

PYA : Python, perfectionnement

Programmation Orientée Objet avancée

1. Propriétés (property)
2. Itérateurs
3. Héritage multiple et ses travers
4. Context Managers
5. Classes et méthodes abstraites (ABC)
6. Métaclasses
7. Travaux pratiques : Implémenter une métaclasse pour créer des classes de type singleton

Propriétés (property)

- Qu'est-ce qu'une Propriété ?
  - Permet de gérer l'accès aux attributs d'une classe.
  - Utilise des méthodes pour obtenir et définir la valeur d'un attribut.
- Syntaxe
- Utilisation de @property pour le getter.
- Utilisation de @nom_de_la_propriété.setter pour le setter.

```
class Exemple:
    def __init__(self, valeur):
        self._valeur = valeur

    @property
    def valeur(self):
        return self._valeur

    @valeur.setter
    def valeur(self, nouvelle_valeur):
        self._valeur = nouvelle_valeur
```

Itérateurs

- Définition d'un Itérateur
  - Un objet qui implémente les méthodes __iter__() et __next__().
  - Permet de parcourir les éléments d'une collection sans exposer son implémentation interne.
- Création d'Itérateurs
  - Implémenter une classe personnalisée pour créer un itérateur.

```
class Compteur:
    def __init__(self, maximum):
        self.maximum = maximum
        self.compteur = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.compteur < self.maximum:
            self.compteur += 1
            return self.compteur
        raise StopIteration
```

Héritage Multiple et ses Travers

- Héritage Multiple
  - Permet à une classe d'hériter de plusieurs classes.
  - Utilisé pour éviter la duplication de code, mais peut mener à des complexités.
- Problèmes Potentiels
  - Ambiguïtés dues aux méthodes avec le même nom dans les classes parente.
  - Utilisation de la méthode de résolution de méthode (MRO) pour déterminer l'ordre d'appel des classes.

```
class A:
    def methode(self):
        print("A")

class B:
    def methode(self):
        print("B")

class C(A, B):
    pass

obj = C()
obj.methode()  # Affiche "A" (premier dans l'ordre de MRO)
```

Context Managers

- Qu'est-ce qu'un Context Manager ?
  - Gère les ressources, comme les fichiers, de manière efficace.
  - Utilise les méthodes __enter__() et __exit__().
- Utilisation du mot-clé with
  - Assure la libération des ressources après utilisation, même en cas d'erreur.

```
class MonContextManager:
    def __enter__(self):
        print("Entrée dans le contexte")
        return self

    def __exit__(self, exc_type,
exc_value, traceback):
        print("Sortie du contexte")

with MonContextManager() as
manager:
    print("À l'intérieur du contexte")
```

Classes et Méthodes Abstraites (ABC)

- Définition d'une Classe Abstraite
  - Utilisée comme modèle pour d'autres classes.
  - Ne peut pas être instanciée directement.
- Utilisation du module abc
  - Définir des méthodes abstraites avec @abstractmethod.

```
from abc import ABC, abstractmethod

class Forme(ABC):
    @abstractmethod
    def aire(self):
        pass

class Cercle(Forme):
    def __init__(self, rayon):
        self.rayon = rayon

    def aire(self):
        return 3.14 * self.rayon ** 2
```

Métaclasses

- Qu'est-ce qu'une Métaclasse ?
  - Classe de classe, définit le comportement des classes.
  - Permet de créer des classes dynamiquement.
- Utilisation de Métaclasses
  - Modifier la création ou l'initialisation de classes.

```
class Meta(type):
    def __new__(cls, name, bases, attrs):
        attrs['ajout'] = "Ajouté par la métaclasse"
        return super().__new__(cls, name, bases, attrs)

class MaClasse(metaclass=Meta):
    pass

print(MaClasse.ajout)  # Affiche "Ajouté par la métaclasse"
```

Travaux Pratiques

- Implémenter une Métaclasse pour Créer des Classes de Type Singleton
  - Objectif
    - Créer une métaclasse qui assure qu'une classe n'a qu'une seule instance (Singleton).
  - Exercice
    - Écrire une métaclasse Singleton et l'appliquer à une classe d'exemple.

```
class SingletonMeta(type):
    instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls.instances:
            instance = super().__call__(*args, **kwargs)
            cls.instances[cls] = instance
        return cls.instances[cls]

class MaClasse(metaclass=SingletonMeta):
    pass
```

paul.millet@lameduse.fr

LaMeDuSe

PYA : Python, perfectionnement

Déploiement et Qualité

1. Installation de librairies tierces
2. Python Package Index (PyPI)
3. Packager ses librairies
4. Déployer un environnement autonome
5. Travaux pratiques : Packager une librairie et la déposer sur PyPI

PIP: Utilisation

- Utilisation de pip
  - Outil de gestion de paquets pour installer et gérer des bibliothèques Python.
  - Commande de base : pip install nom_du_package
- Utilisation de easy_install
  - Ancien gestionnaire de paquets, moins courant aujourd'hui.
  - Généralement remplacé par pip.

```
pip install requests
```

Python Package Index (PyPI)

- Qu'est-ce que PyPI ?
  - Dépôt centralisé pour les bibliothèques Python.
  - Permet aux développeurs de partager et de trouver des packages.
- Naviguer sur PyPI
  - Accéder à pypi.org pour rechercher des bibliothèques et leurs documentations.
- Télécharger un Package
  - Commande : pip install nom_du_package télécharge et installe depuis PyPI.

Packager ses Librairies

- Outils de Packaging :
  - distutils : Outil de base pour créer et distribuer des modules Python.
  - setuptools : Extension de distutils, offre des fonctionnalités supplémentaires pour la gestion des dépendances.
- Fichiers Essentiels :
  - setup.py : Fichier de configuration pour définir les métadonnées du package.
  - MANIFEST.in : Spécifie les fichiers supplémentaires à inclure dans le package.

Packager ses Librairies

```
from setuptools import setup

setup(
    name='mon_package',
    version='0.1',
    description='Description de mon package',
    author='Mon Nom',
    packages=['mon_package'],
    install_requires=['requests'],
)
```

```
my_project/
├─── my_package/
│    ├─── __init__.py
│    └─── module.py
├─── setup.py
└─── README.md
```

Déployer un Environnement Autonome

- Utilisation de Virtualenv
  - Crée un environnement isolé pour gérer les dépendances spécifiques à un projet.
  - Commande : virtualenv nom_environnement
- Utilisation de Buildout
  - Outil avancé pour gérer des environnements complexes, incluant les dépendances et les scripts.

```
# Créer un environnement
virtualenv mon_environnement

# Activer l'environnement
source mon_environnement/bin/activate

# Installer des dépendances
pip install -r requirements.txt
```

Travaux Pratiques

- Packager une Librairie et la Déposer sur PyPI
  - Objectif
    - Créer un package Python et le publier sur PyPI.
- Étapes à Suivre :
  - Créer un fichier setup.py et d'autres fichiers nécessaires.
  - Construire le package avec python setup.py sdist bdist_wheel.
  - Installer twine pour publier le package sur PyPI.
  - Utiliser twine upload dist/* pour déposer le package sur PyPI.

Travaux Pratiques

```
pip install twine
python setup.py sdist bdist_wheel
twine upload dist/*
```

5 - Le Parallélisme et Optimisation des Performances

paul.millet@lameduse.fr

LaMeDuSe

PYA : Python, perfectionnement

Le Parallélisme et Optimisation des Performances

1. Profilage des programmes avec timeit et cProfile
2. Parallélisation : éviter le multithreading, privilégier le multiprocessing
3. Calcul distribué avec la librairie Celery
4. Travaux pratiques : Répartition et consolidation (Map Reduce) de calculs avec Celery

Profilage des Programmes

- Qu'est-ce que le Profilage ?
  - Technique permettant d'analyser la performance d'un programme.
  - Identifie les parties du code qui consomment le plus de temps.
- Outils de Profilage :
  - timeit : Utilisé pour mesurer le temps d'exécution de petites portions de code.
  - cProfile : Fournit des statistiques détaillées sur le temps d'exécution de chaque fonction.

Profilage des Programmes

```
import timeit

code = """
total = 0
for i in range(1000):
    total += i
"""
execution_time = timeit.timeit(code, number=1000)
print(f"Temps d'exécution : {execution_time} secondes")
```

Parallélisation

- Pourquoi Privilégier le Multiprocessing ?
  - Le multithreading peut être limité par le Global Interpreter Lock (GIL) en Python.
  - Le multiprocessing permet d'exécuter des tâches en parallèle sur plusieurs cœurs de processeur.
- Implémentation de Multiprocessing :
  - Utilisation du module multiprocessing pour créer des processus distincts.

Parallélisation

```
from multiprocessing import Pool

def carre(x):
    return x * x

if __name__ == "__main__":
    with Pool(4) as p:
        result = p.map(carre, range(10))
    print(result)  # Affiche les carrés des nombres de 0 à 9
```

Calcul Distribué avec Celery

- Qu'est-ce que Celery ?
  - Outil pour la gestion de tâches asynchrones et le calcul distribué.
  - Idéal pour exécuter des tâches en arrière-plan et paralléliser le traitement.
- Principes de Fonctionnement :
  - Utilise des brokers comme RabbitMQ ou Redis pour gérer les messages entre les producteurs et les consommateurs de tâches.
  - Permet de distribuer des tâches sur plusieurs machines.

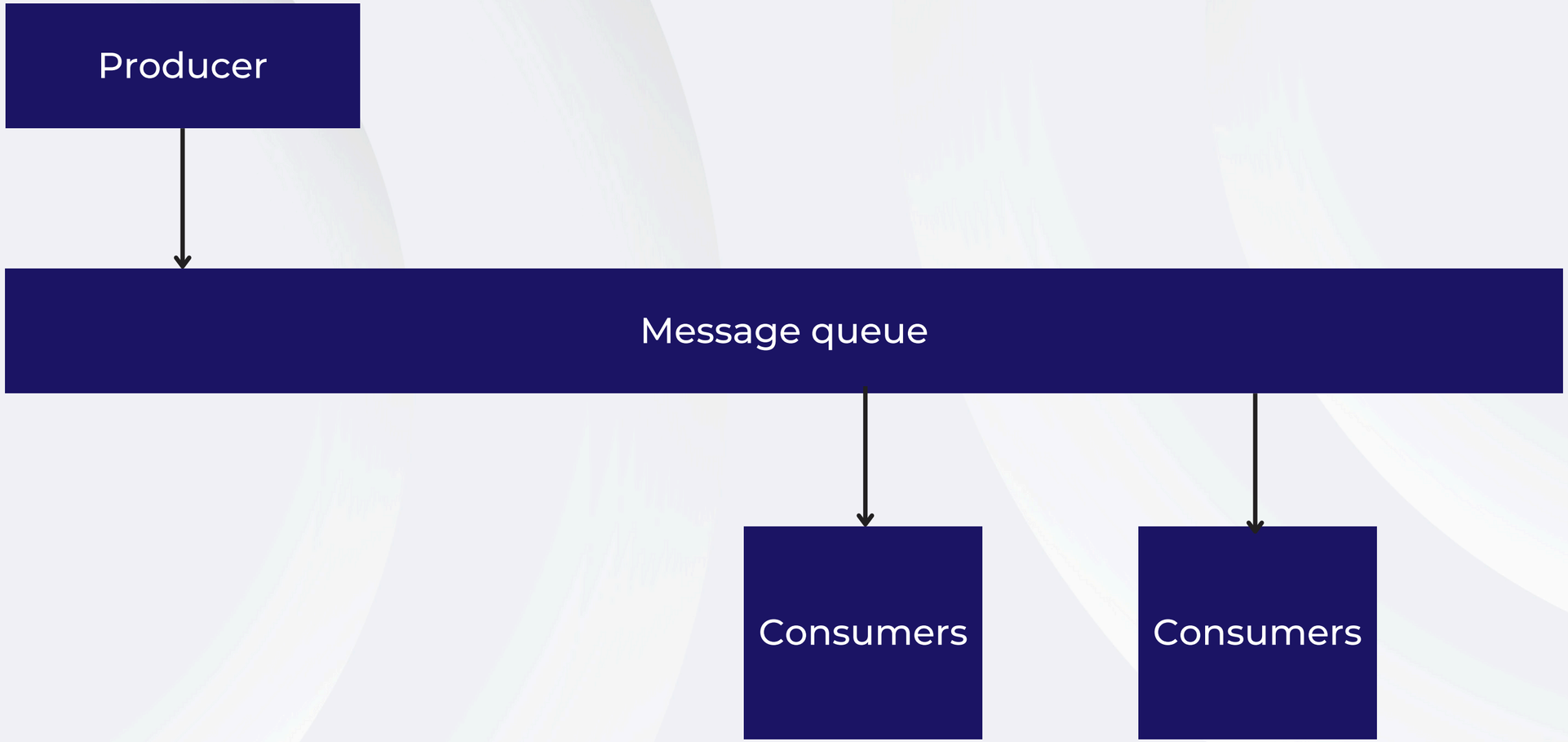
Calcul Distribué avec Celery

```
from celery import Celery

app = Celery('tasks', broker='redis://localhost:6379/0')

@app.task
def addition(x, y):
    return x + y
```

Calcul Distribué (notion)



Travaux Pratiques

- Répartition et Consolidation (Map Reduce) de Calculs avec Celery
  - Objectif
  - Implémenter un système Map Reduce pour traiter de grandes quantités de données.
- Étapes à Suivre :
  - Créer une tâche map qui divise les données en sous-ensembles.
  - Créer une tâche reduce qui agrège les résultats des tâches map.
  - Tester le système avec des données fictives pour évaluer la performance.

Travaux Pratiques

```
@app.task
def mapper(data_chunk):
    return [x * 2 for x in data_chunk]

@app.task
def reducer(results):
    return sum(results)
```

paul.millet@lameduse.fr

LaMeDuSe

PYA : Python, perfectionnement

Librairies Clés et Applications

1. Calcul scientifique et statistiques
2. Intelligence artificielle et algorithmes d'apprentissage
3. Recherche d'informations dans des fichiers XML
4. Réseau : relay TCP et supervision SNMP
5. Travaux pratiques : Extraction et visualisation de données

Calcul Scientifique et Statistiques

- Bibliothèques Clés :
  - NumPy : Fournit des structures de données et des fonctions pour le calcul numérique efficace.
  - SciPy : Étend NumPy avec des algorithmes et des fonctions mathématiques avancées.
  - Matplotlib : Bibliothèque pour créer des visualisations graphiques 2D.
  - Pandas : Outil pour la manipulation et l'analyse de données, basé sur des DataFrames.

Calcul Scientifique et Statistiques

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y)
plt.title("Courbe Sinusoïdale")
plt.xlabel("x")
plt.ylabel("sin(x)")
plt.show()
```

Intelligence Artificielle et Algorithmes d'Apprentissage

- Bibliothèque Clé :
  - Scikit-Learn : Outil pour le machine learning, incluant des fonctions pour la classification, la régression, et le clustering.
- Principales Fonctions :
  - Prétraitement des données : normalisation, encodage.
  - Modèles de classification : SVM, arbres de décision, etc.
  - Évaluation des performances : validation croisée, courbes ROC.

Intelligence Artificielle et Algorithmes d'Apprentissage

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

data = load_iris()
X_train, X_test, y_train, y_test =
train_test_split(data.data, data.target, test_size=0.3)
model = RandomForestClassifier()
model.fit(X_train, y_train)
predictions = model.predict(X_test)

print(f"Accuracy : {accuracy_score(y_test, predictions)}")
```

Recherche d'Informations dans des Fichiers XML

- Bibliothèque Clé :
  - ElementTree : Permet de parser et de créer des documents XML de manière simple et efficace.
- Principales Fonctions :
  - Chargement de fichiers XML, navigation dans la structure, extraction de données.

Recherche d'Informations dans des Fichiers XML

```
import xml.etree.ElementTree as ET

tree = ET.parse('fichier.xml')
root = tree.getroot()

for enfant in root.findall('élément'):
    print(enfant.text)
```

Réseau : Relay TCP et Supervision SNMP

- Bibliothèques Clés :
  - Twisted : Framework pour la programmation réseau asynchrone en Python.
  - PySNMP : Utilisé pour la gestion et la supervision SNMP (Simple Network Management Protocol).
- Utilisation de Twisted pour un Relay TCP :
  - Permet de créer des serveurs et clients réseau avec des protocoles personnalisés.

Réseau : Relay TCP et Supervision SNMP

```
from twisted.internet import reactor, protocol

class MonServeur(protocol.Protocol):
    def dataReceived(self, data):
        self.transport.write(b"Message reçu : " + data)

factory = protocol.Factory()
factory.protocol = MonServeur
reactor.listenTCP(8000, factory)
reactor.run()
```

Travaux Pratiques

- Extraction d'Informations dans des Fichiers de Log XML
  - Objectif :
    - Lire un fichier de log XML, appliquer des filtres, effectuer des statistiques sur les données, et représenter les résultats graphiquement.
  - Étapes à Suivre :
    - Charger et parser le fichier XML avec ElementTree.
    - Appliquer des filtres pour extraire des informations pertinentes.
    - Calculer des statistiques (par exemple, le nombre d'occurrences de chaque type de log).
    - Utiliser Matplotlib pour créer des graphiques des tendances des informations collectées.

Travaux Pratiques

```
# Charger et analyser le fichier de log
import xml.etree.ElementTree as ET
import matplotlib.pyplot as plt

tree = ET.parse('log.xml')
root = tree.getroot()
# Filtrer et analyser les données...
# Tracer les résultats...
```

paul.millet@lameduse.fr

LaMeDuSe

PYA : Python, perfectionnement

Ressources des TP

Les TP sont présent directement dans les parties de la formation.
Seul des TP additionnels sont présent ici

Web Scraping et Analyse de Données

Objectif : Créer un script en Python pour extraire des données d’un site web (exemple : des informations de produits, de météo, ou d'actualités), les analyser, et en afficher les résultats sous forme de graphiques. Le TP permet de se familiariser avec des concepts d'extraction de données, de traitement, de stockage et de visualisation.

Prérequis :

- Maîtrise des bases du Python
- Connaissance des librairies requests, BeautifulSoup, pandas, matplotlib

paul.millet@lameduse.fr

LaMeDuSe

PYA : Python, perfectionnement

Sources

Sources :

-