UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Python Perfectionnement

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Notions avancées du langage

Rappels globaux sur python

Global :

- Version actuelle : **3.12**
- Langage dit "**de script**"
- **Facile à apprendre** par sa syntaxe **permissive**
- Beaucoup utilisé dans les métiers : **Data**, **IA**, **Maths**, **Enseignement**, ...
- Syntaxe **très peu verbeuse**

Technique :

- Syntaxe sensible à l'**indentation**
- Typage **Dynamique/Faible**
- **Interprété**
- **Héritage multiple** en POO
- Pas d'**interfaces** en POO
- Principe de variables et méthodes "**magiques**" (ou **dunderscore**)

Rappels sur les fonctions

- Sert à **exécuter plusieurs fois un même bloc d'instructions**
- **Reçoit** éventuellement des **arguments** et **renvoie** une **valeur** ou **None** (Rien).
- Devrait effectuer **une tâche unique et précise** sinon devrait être découpée en **plusieurs fonctions**
- Les **valeurs** passées à l'**exécution** de la fonction s'appellent des **arguments**.
- Les **variables** entre parenthèses qui **contiendront ces valeurs** sont les **paramètres**.
- Il est possible de **retourner plusieurs valeurs** en les regroupant dans un **tuple** (**packing/unpacking**)

Rappels sur les fonctions

- **définir** une fonction => `def`
- **retourner** le résultat => `return`
- indications sur le **type** des **paramètres** => `param1: int`
- indications sur le **type** du **retour** => `def f() -> int:`
- arguments **facultatif** avec **valeur par défaut** => `param1=3`
- attention aux **variables globales**, utiliser `global`

```python
def carre(nombre: int = 3):
    return nombre**2
res = carre(2) # retourne 2
res = carre() # retourne 9

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

Les *args et **kwargs

- En python, il est possible d'ajouter des **paramètres spéciaux** précédés avant leurs noms par une ou deux étoiles.
- Leurs noms sont conventionnés, il est important de les nommer **arg** (arguments) et **kwargs** (keyword arguments)
- Ils permettent d'avoir des fonctions au **nombre d'argument variable** qui seront stockés dans une **liste** et un **dictionnaire**.
- Ordre de définition :

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

Exemple *args et **kwargs

```python
def ma_fonction(*args, **kwargs):
    print("args :", args) # tuple
    for arg in args:
        print(arg)

    print("kwargs :", kwargs) # dict
    for kw, value in kwargs.items():
        print(kw, ":", value)

ma_fonction("abc", True, 1, kw1="kw1valeur", test=1234, a=["A",1])
```

Fonction anonymes "lambdas"

- Les lambdas sont des **fonctions simplifiées à l'extrême** et **anonymes** (pas d'identificateur sauf si stockées dans une variable).
- Elles n'ont qu'**une seule instruction**/ligne qui donnera le résultat (`return` implicite)
- Elles s'utilisent en général **comme arguments d'autres fonctions** (ex : filter, map, reduce, sorted).

```
fct1 = lambda x : x**2
def fct2(x) :
    return x**2
print(fct(2))
print(fct2(2))
```

Sorted, filter, map et reduce

Lorsque l'on travaille avec des **conteneurs** (list, dict, ...) il existe certaines **fonctions** utiles.
Voici les 4 principales, elles utilisent des **fonctions/lambdas** et nous simplifient beaucoup le travail avec les listes.

- **sorted** : trier la liste selon certains critères
- **filter** : filtrer les éléments de la liste
- **map** : créer une nouvelle liste avec tous les éléments transformés par une fonction
- **reduce** : réduire la liste à une seule valeur

Ces fonctions ont des alternatives avec les techno Data (Pandas, Spark, ...)

Exemples Sorted, filter, map et reduce

```
t = ["bonjour","dix","cailloux", "aaa"]
```

sorted

```
# alphabétique inverse
print(sorted(t, reverse=True))
# taille
print(sorted(t, key=lambda s : s.__len__()))
```

map

```
# chaines capitalisés
print(list(map(lambda t: t.capitalize(), t)))
# longueurs
print(list(map(lambda t: len(t), t)))
```

filter

```
# commencent par 'a'
print(list(filter(lambda t: t[0] == 'a', t)))
# font 3 caractères
print(list(filter(lambda t: len(t) == 3, t)))
```

reduce

```
# besoin d'un import
from functools import reduce
# concaténation avec '/'
print(reduce(lambda c1, c2: c1 + '/' + c2, t))
```

Rappels collections

- **list** `[,]`:
  - Éléments **numérotés** de 0 à n, **accessibles** avec l'index `list[i]`
  - **Méthodes** :
    sort() , append(element), extend(list), pop(index), remove(element), count(element), index(element), mot clé `del`
- **tuple** `(,)`:
  - En **lecture seule**, **regrouper** des valeurs (**packing / unpacking**), utilise l'**index**
  - Packing : `mon_tuple=(1,2,3)`
  - Unpacking : `var1, var2 = (1, 2)`

Rappels collections

- **set** `{,}`:
  - Éléments **uniques** et **immutables**, **réordonné automatiquement**
  - **Méthodes** :
    add(element), update(set), remove(element), discard(element), isdisjoint(set), issubset(set2), issuperset(set2)
    union `|`, insertion `&`, différence `-` et différence symétrique `^`
- **dict** `{:,}`:
  - **Couples clé-valeur**, les clé **remplacent les index** d'une list, elles sont **unique** et **immutable**
  - Possibilité d'itérer sur les clés (par défaut ou `.keys()`), les valeurs (`.values()`) ou des tuples `(clés, valeurs)` (`.items()`)

Pile/Stack et File/Queue

Les principes de **pile** et **file** sont courants en programmation, souvent les langages proposent des types dédiés, en python il faudra utiliser `list` ou importer `collections.deque` (censé être plus optimisée pour cet objectif). Documentation

	Pile	File
Mode d'ajout	Last-In/First-Out (**LIFO**)	First-In/First-Out (**FIFO**)
Type	**list**	**list**
Méthodes à utiliser	-.append() -.pop() (**dernier** ajouté)	-.append() -.pop(0) (**premier** ajouté)

list/set/dict comprehension

- Pour **générer et d'itérer** sur des conteneurs
- Pour la **list comprehension** (**itérable** = objet sur lequel on peut itérer) :
  `var = [expression for element in iterable]`
- Possible d'ajouter un **filtre** avec un `if` après l'itérable (équivalent de `filter`)

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

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

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

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

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

Rappels POO

- **Classe** = concept représentant un potentiel objet ou concept dans un programme informatique
- **Objet/Instance** = un objet en particulier qui dérive du concept (classe)
- **Attributs**/variables d'instance = variables liées à un objet en particulier, commune à tout les objets d'une classe
- **Méthode** = fonction liée à un objet et ses attributs, commune à tout les objets d'une classe
- **Constructeur** = méthode qui permet de créer une instance à partir de la classe
- **Attribut et Méthode de Classe** = membres liés à la classe et non à l'instance, partagés par toutes les instances

Exemple Rappels POO

```
class Chien: # déclaration de la classe
    """ Représentation d'un chien """ # documentation
    nombre_chiens = 0 # attribut de classe
    def __init__(self,nom,age,race): # constructeur
        nombre_chiens += 1 # modification de l'attribut de classe
        self.nom = nom # définition d'un attribut, self = l'instance
        self.age = age
        self.race = race
    def aboyer(self): # méthode
        print(f"Wouf Wouf {self.nom}")
    @classmethod #annotation
    def afficher_nombre_chiens(cls): # méthode de classe
      print(f"Il y a {cls.nombre_chiens} chiens instanciés")

chien_1 = Chien("REX", 12, "Berger Allemand")
chien_1.aboyer() # Wouf Wouf REX
Chien.afficher_nombre_chiens() # Il y a 1 chiens instanciés
```

Attributs implicites/Dunderscore

- Attributs créés par défaut dans les classes
- Dits "Magiques" ou "Dunderscore"/"Dunder" (double underscore)

Pour la **classe** on a:

- __name__: le nom de la classe
- __doc__: commentaire associé à la classe
- __dict__: le dictionnaire des attributs statiques
- __bases__: un tuple des classes dont celle-ci hérite
- __module__: contient le nom du module dans lequel la classe a été définie

Pour l'**instance** on a:

- __class__: la classe de l'objet.
- __dict__: la liste des attributs d'instance

Exemple attributs implicites/Dunderscore

```
class MaClasse:
    """ une classe """
    test = 0
    def __init__(self):
      self.test1 = 1
cl = MaClasse()
# Classe
print(MaClasse.__name__)       # MaClasse
print(MaClasse.__doc__)        # une classe
print(MaClasse.__dict__)       # {"test": 0, ...}
print(MaClasse.__bases__)      # (<class 'object'>,)
print(MaClasse.__module__)     # __main__
# Instance
print(cl.__class__)            # <class '__main__.MaClasse'>
print(cl.__class__.__name__)   # MaClasse
print(cl.__dict__)             # {'test1': 1}
print(cl.__doc__)              # une classe
```

Rappel Héritage

- L'**héritage** est un mécanisme fortement utilisé dans la **programmation orienté objet**.
- Une classe peut **hériter** d'une **autre classe** et possédera les **méthodes** et les **attributs** de celle-ci.
- On parle alors de **classe fille/enfant** et de **classe mère/parent**.
- Pour **réaliser un héritage** en Python il suffit d'**ajouter des parenthèses** après le nom de la classe que l'on créé et d'y **ajouter la classe dont l'on souhaite hériter**.
- L'exemple suivant est correct sémantiquement car on peut dire qu'**un Chien est un Mammifère**.
- Toutes les classes héritent de la classe **object**

Exemple Rappel Héritage

```
class Mammifere:
    nom_latin = "Mamma"
    nombre_mammifere = 0
    def __init__(self):
        Mammifere.nombre_mammifere += 1
class Chien(Mammifere):
    nom_latin = "Canis Lupus Familiaris"
    def __init__(self, nom, age, race)
        super().__init__() # rappel du constructeur parent
        self.nom = nom
        self.age = age
        self.race = race
mon_chien = Chien("Rex", 4, "Berger Allemand")
print(Mammifere.nombre_mammifere) # 1
```

Méthodes magiques/Dunderscore

Comme pour les attributs, il existe des **méthodes dunder** qui permettent de **redéfinir un comportement spécifique relatif à un objet**.
En voici une liste non-exhaustive :

Liées à la POO:

- `__new__` : pre-constructeur (classe)
- `__init__` : constructeur (instance)
- `__repr__` : représentation textuelle
- `__getattr__`: récupération d'attribut
- `__setattr__`: définition d'attribut
- `__del__`: suppression/destructeur

Conversion :

`__str__` : str(objet)
`__bytes__` : bytes(objet)
`__bool__` : bool(objet)
`__int__` : int(objet)
`__float__` : float(objet)
`__complex__` : complex(objet)
`__dict__` : dict(objet)

Méthodes magiques/Dunderscore

Arithmétiques:

- `__add__(self, other)`: +
- `__sub__(self, other)`: -
- `__mul__(self, other)`: *
- `__matmul__(self, other)`: @
- `__truediv__(self, other)`: /
- `__floordiv__(self, other)`: //
- `__mod__(self, other)`: %
- `__divmod__ (self, other)`: divmod()
- `__pow__(self, other)`: ** ou pow()

Comparaison:

- `__eq__(self, other)`: ==
- `__lt__(self, other)`: <
- `__le__(self, other)`: <=
- `__gt__(self, other)`: >
- `__ge__(self, other)`: >=
- `__ne__(self, other)`: !=

Méthodes magiques/Dunderscore

Conteneurs :

- `__len__(self)`: len(o)
- `__getitem__(self, key)`: o[key]
- `__setitem__(self, key, value)`: o[key] = value
- `__delitem__(self, key)`: del o[key]
- `__contains__(self, key)`: key in o

Binaires/bitwise (contextuels):

- `__lshift__(self, other)`: <<
- `__rshift__(self, other)`: >>
- `__and__(self, other)`: &
- `__or__(self, other)`: |
- `__xor__(self, other)`: ^

Tp Intervals

1. Créer une classe **Interval** possédant une méthode `__init__` permettant d'initialiser une **borne inférieure** et une **borne supérieure** pour un objet de type Interval.
   Vérifier que les bornes sont numériques, positives, non nulles et placées dans le bon ordre, sinon générer une exception de type « IntervalError » affichant le message d'erreur « **Erreur : Bornes invalides !** ».
   Le type « IntervalError » est une Exception à définir.

2. Écrire une méthode dunder `__str__(self)` permettant de retourner une chaine indiquant les valeurs des deux bornes de l'intervalle.
3. Écrire une méthode dunder `__contains__(self, val)` qui teste si une valeur val appartient ou non à l'intervalle (utilisé par l'opérateur in).
4. Écrire une méthode dunder `__add__(self, autre)` qui retourne un nouvel Intervalle addition des deux intervalles. Exemple : [2,5] + [3,4] = [5,9].
5. Écrire une méthode dunder `__sub__(self, autre)` qui retourne un nouvel Intervalle soustraction des deux intervalles.
6. Écrire une méthode dunder `__mul__(self, autre)` qui retourne un nouvel Intervalle multiplication des deux intervalles. Exemple : [2,5] * [3,4] = [6,20]
7. Écrire une méthode dunder `__and__(self, autre)` (**&**) qui retourne l'intersection des deux intervalles et « None » si leur intersection est vide. Exemple: [2,5]∩[3,6]=[3,5]

La variable `__name__` (variable Dunder)

On retrouve souvent cette structure pour les scripts python, surtout quand on travaille avec des **imports**.

`__name__` est une variable prédéfinie dans chaque module, elle contient :

- `"__main__"` si on est dans le module principal (point d'entrée)
- Le nom du module dans un module importé

Le bloc `if __name__ == "__main__"` si on est dans le module principal.

On peut voir ça comme le Main dans d'autres langages.

```python
import math

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

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

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

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Multithreading et Asynchrone

Notion de thread

- Les **threads en Python** sont des **unités d'exécution légères** qui permettent à un **programme** d'**effectuer plusieurs tâches simultanément**.
- Ils partagent **le même espace mémoire** et peuvent s'**exécuter en parallèle** sur un processeur multicœur ou être répartis sur plusieurs processeurs.
- Les threads sont créés à l'aide du module `threading` de Python et offrent un moyen efficace de **réaliser des opérations concurrentes** sans avoir à recourir à des processus distincts.
- Ils sont utiles pour des tâches telles que le traitement des entrées/sorties, les opérations réseau ou le traitement parallèle de données.

Création des threads en Python

- Python fournit la classe `Thread` du module de `threading` pour créer les threads.
- Il existe différentes méthodes de création des threads :
  - Méthode-1 : Sans utiliser la classe
  - Méthode-2 : En créant une sous-classe à la classe Thread
  - Méthode-3 : Sans créer de sous-classe à la classe Thread

Création de thread en Python – Méthode 1

- Etape 1:
  - Création d'un objet de type thread.
  - Passer la fonction à exécuter par le thread comme premier argument du constructeur.
  - Passer les arguments de la fonction comme deuxième argument du constructeur.
- Etape 2:
  - Démarrer le thread

```
import random
from threading import Thread
import time

def print_vide():
    print("Bonjour !")

def print_avec_texte(text):
    time.sleep(random.random()*3)
    print("Mon texte:", text)

for i in range(3):
    t = Thread(target= print_vide)

    t.start()

for i in range(3):
    t = Thread(target= print_avec_texte, args= (i,))

    t.start()
```

Création de thread en Python – Méthode 2

- Etape 1:
  - Création d'une classe qui hérite de classe Thread.
  - Surcharger la méthode run.
  - Création d'un objet à partir de la nouvelle classe.
- Etape 2:
  - Démarrer le thread

```python
from threading import Thread

class MonThread(Thread):
    def __init__(self, str):
        super().__init__()
        self.str = str
    def run(self):
        for i in range(6):
            print(self.str, i)

t1 = MonThread("hello")
t2 = MonThread("bonjour")

t1.start() # lancer
t2.start()

t1.join() # attendre la fin
t2.join()
```

Création de thread en Python – Méthode 3

- Etape 1:
  - Création d'une classe qui implémente notre logique métier.
  - Créer un objet à partir de notre classe .
  - Créer un objet thread avec comme paramètre la méthode de notre premier objet et les arguments nécessaires.
- Etape 2:
  - Démarrer le thread

```python
from threading import Thread

class UnObjet:
    def __init__(self, str):
        self.str = str

    def multi_print(self):
        for i in range(6):
            print(self.str, i)


obj = UnObjet("hello")
obj2 = UnObjet("deux")


t1 = Thread(target= obj.multi_print)
t2 = Thread(target= obj2.multi_print)

t1.start()
t2.start()
```

Thread et Task

- Un thread peut être utiliser pour exécuter une task à la fois.
- Exemple : Préparation du thé
  - Task 1 => faire bouillir l'eau.
  - Task 2 => Ajouter le thé et laisser pendant 3 min.
  - Task 3 => Ajouter le sucre et laisser pendant 2 min.
  - Task 4 => filtrer et servir.

```python
from threading import Thread
from time import sleep

class MyThread:
    def prepare_tea(self):
        self.task1()
        self.task2()
        self.task3()
        self.task4()

    def task1(self):
        print("Faire bouillir l'eau", end = ' ')
        sleep(5)
        print("Done")

    def task2(self):
        print("Ajouter le thé et laisser pendant 3 min",end = ' ')
        sleep(3)
        print("Done")

    def task3(self):
        print("Ajouter le sucre et laisser pendant 2 min",end = ' ')
        sleep(3)
        print("Done")

    def task4(self):
        print("filtrer et servir",end = ' ')
        sleep(3)
        print("Done")

obj = MyThread()

t = Thread(target=obj.prepare_tea)
t.start()
```

Exercice

- Obtenir une liste de fichiers (du répertoire courant).
- Traitez chaque fichier :
  1. obtenir la taille du fichier
  2. compter combien de fois chaque caractère apparaît dans le fichier.
- Le script doit exécuter la fonction dans un thread.

Race-condition et thread synchronization

- Pour exécuter plusieurs tasks d'une façon concurrente, nous pouvons utiliser le multi-threading.
- Le multi-threading génère du **Race-condition**.
- Pour résoudre les race-conditions nous pouvons utiliser le mécanisme de « **thread synchronization** ».
- Le mécanisme de « **thread synchronization** » peut se faire par **Mutex** ou **Semaphores**

```
from threading import *
from time import *
class Theatre:
    #Constructor that accepts a string
    def __init__(self, str):
        self.str = str
    #A method that repeats for 5 tickets
    def movieShow(self):
        for i in range(1, 6):
            print(self.str, ":", i)
            sleep(1)

obj1 = Theatre("Cut Ticket")
obj2 = Theatre("Show chair")

t1 = Thread(target = obj1.movieShow)
t2 = Thread(target = obj2.movieShow)

#Run the threads
t1.start()
t2.start()
```

Multi-Threads et Multi-tasking - GIL

- **GIL** (global interpreter lock) est un mécanisme qui permet de garantir qu'**un seul thread** a le contrôle de l'interpréteur de Python.
- **Un seul thread** peut être en état d'**exécution** à un moment donné.

- En effet nous avons pu voir par exemple que les prints sur la sortie standard s'affichaient les uns à la suite des autres et pas en même temps, le GIL gérait leur exécution.

Mutex

- Le principe consiste à utiliser un objet de type `Lock`
- Procéder au verrouillage de l'objet à l'aide de la méthode `acquire`.
- Procéder au déverrouillage de l'objet à l'aide de la méthode `release`

```
from threading import *
from time import sleep

class Railway:
    #Constructor that accepts no. of available berths
    def __init__(self, available):
        self.available = available
        #Create a lock Object
        self.lock = Lock()

    def reserve(self, wanted): #A method that reserves berth
        self.lock.acquire() #lock the current object
        print("Available no. of berths = ", self.available)
        tName = current_thread().name #Find the thread name
        if (self.available >= wanted):
            print(f"{wanted} berths are alloted for {tName}")
            sleep(1.5) #Make time delay so that ticket is printed
            self.available -= wanted #Decrease the number of available berths
        else:
            #If available < wanted, then say sorry
            print(tName, ": Sorry, no berths to allot")
        self.lock.release() #Task is completed, release the lock


obj = Railway(2)
t1 = Thread(target = obj.reserve, args = (1, ))
t2 = Thread(target = obj.reserve, args = (3, ))
t3 = Thread(target = obj.reserve, args = (1, ))

#Give names to the threads
t1.name = "First Person"
t2.name = "Second Person"
t3.name = "Third Person"

#Start running the threads
t1.start()
t2.start()
t3.start()
```

Semaphore

- Le principe consiste à utiliser un objet de type `Semaphore`
- Semaphore permet de **limiter** l'accès aux ressources à **un nombre donné de thread** a définir à la création de l'objet.
- Procéder au **verrouillage** de l'objet à l'aide de la méthode `acquire`.
- Procéder au **déverrouillage** de l'objet à l'aide de la méthode `release`.

```python
import threading
import time

# Nombre total de ressources disponibles
nombre_ressources = 3

# Création du sémaphore
semaphore = threading.Semaphore(nombre_ressources)

# Fonction exécutée par chaque thread
def utiliser_ressource(thread_id):
    print(f"Thread {thread_id} en attente d'accès à la ressource.")
    # Acquérir le sémaphore
    semaphore.acquire()
    print(f"Thread {thread_id} accède à la ressource.")
    time.sleep(1)  # Simulation d'utilisation de la ressource
    print(f"Thread {thread_id} libère la ressource.")
    # Libérer le sémaphore
    semaphore.release()

def main():
    # Créer et démarrer plusieurs threads
    threads = []
    for i in range(6):
        thread = threading.Thread(target=utiliser_ressource, args=(i,))
        threads.append(thread)
        thread.start()

    # Attendre que tous les threads aient terminé leur travail
    for thread in threads:
        thread.join()

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

Dead Locks

Lorsqu'**un thread** a **verrouillé un objet** et **attend** qu'un **autre objet** soit **libéré** par un **autre thread**, et que l'**autre thread attend également** que **le premier thread libère le premier objet**, les deux threads continueront à attendre indéfiniment.

- Cette condition s'appelle **Deadlock**.

```python
from threading import *

l1 = Lock()
l2 = Lock()

def bookTicket():
    l1.acquire()
    print("bookTicket locked train")
    print("bookTicket wants to lock on compartment")
    l2.acquire()
    print("bookTicket locked compartment")
    l2.release()
    l1.release()
    print("bookTicket done")

def cancelTicket():
    l2.acquire()
    ## pour résoudre le problème on met d'abord
    # l1.acquire()
    print("cancelTicket locked compartment")
    print("cancelTicket wants to lock on train")

    l1.acquire()
    # l2.acquire()
    print("cancelTicket locked train")
    l1.release()
    # l2.release()
    l2.release()
    # l1.release()
    print("cancelTicket done...")

t1 = Thread(target = bookTicket)
t2 = Thread(target = cancelTicket)

t1.start()
t2.start()

# les deux threads seront bloqué,
# il faut changer l'ordre de verrouillage et déverrouillage.
```

Communication entre threads

- La communication entre deux threads consiste à :
  - Avoir un thread qui **produit**.
  - Avoir un thread qui **consomme**.
- En Python, l'écriture dans des ressources tel que dictionnaire, list, queue, sont protégé contre le **race condition** par le GIL.
- Nous pouvons utiliser ces **ressources** pour faire **communiquer** deux threads.

Exercice

En utilisant les queues du module queue:

- Écrire une classe qui s'exécute dans son propre thread et qui permet de **produire des données** (**producer**). Exemple: entiers aléatoires
- Écrire une classe qui s'exécute dans son propre thread et qui permet de **récupérer les données de la première classe** (**consumer**).

Exercice

- Écrivez une application qui gère une file d'attente de jobs dans N=5 threads.
- Chaque job contient un nombre compris entre 0 et 5.
- Chaque thread prend l'élément suivant de la file d'attente et dort pendant le nombre de secondes donné (comme une imitation du travail réel qu'il devrait faire).
- Une fois terminé, il recherche un autre job.
- S'il n'y a plus de jobs dans la file d'attente, le thread peut se fermer.

Daemon Threads

- Dans le cadre des threads qui nécessite une exécution continue en arrière-plan, nous pouvons utiliser des « Daemon Threads », ils seront arrêtés automatiquement à la fin du programme principal là où les threads non-daemon seront attendus.
- Ils sont souvent utilisés pour effectuer des tâches telles que la surveillance, la gestion de la mémoire ou la mise à jour des données en arrière-plan, sans bloquer l'exécution du programme principal.
- Pour créer un « Daemon Thread », il suffit de passer l'attribut daemon à True

```
thread = Thread(daemon=True, ...)
```

Module concurrent.future et Pools

- Supposons que nous devions **créer un grand nombre de threads** pour nos tâches multithread, nous pouvons créer un **pool de threads**.
- Un pool de threads peut être défini comme **un groupe de threads pré-instanciés et inactifs**, qui sont **prêts à recevoir des tasks**.
- La création de celui-ci est préférable à l'instanciation de nouveaux threads
- Un pool de threads peut gérer l'exécution simultanée d'un grand nombre de threads comme suit :
  - Si un thread dans un pool de threads **termine son exécution**, ce thread **peut être réutilisé**.
  - Si un thread est **terminé**, un **autre thread** sera créé pour **remplacer ce thread**.

Le module concurrent.futures

- La bibliothèque standard Python inclut le module `concurrent.futures`.
  Il s'agit d'une couche d'abstraction au-dessus des modules de `threading` Python pour fournir l'interface permettant d'exécuter les tâches à l'aide d'un pool de threads ou de processus.
- Executor est une classe abstraite du module Python `concurrent.futures`.
- Il ne peut pas être utilisé directement et nous devons utiliser l'une des sous-classes concrètes suivantes
  - `ThreadPoolExecutor`
  - `ProcessPoolExecutor`

Le module concurrent.futures

- ThreadPoolExecutor permet de définir un objet avec le nombre de threads que nous voulons dans le pool.
- Par défaut, le nombre est 5.
- Ensuite, nous pouvons soumettre une tâche au pool de threads et récupérer une future.

```python
from concurrent.futures import ThreadPoolExecutor
import time

# Fonction exécutée par chaque thread
def task(num):
    print(f"Task {num} started.")
    time.sleep(2)  # Simulation d'une tâche prenant du temps
    print(f"Task {num} finished.")
    return f"Task {num} result"

def main():
    # Création d'un ThreadPoolExecutor avec 3 threads
    with ThreadPoolExecutor(max_workers=3) as executor:
        # Soumettre des tâches au pool d'exécution
        futures = [executor.submit(task, i) for i in range(5)]

        time.sleep(2)
        for future in futures:
            print(future.done(), future.running())

        # Récupérer les résultats des tâches
        for future in futures:
            result = future.result()
            print(result)

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

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Réseau, Appels API et Sockets

TCP/IP et concepts de base de l'API socket

- TCP/IP : Ensemble de protocoles de communication utilisés sur Internet.
- API socket : Interface de programmation pour les communications réseau. Permet aux programmes de communiquer sur un réseau en utilisant les protocoles TCP/IP.

Utilisation du module socket

server

```
import socket

mon_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mon_socket.bind(('', 15555))

while True:
    mon_socket.listen(5)
    client, address = mon_socket.accept()
    print(f"{address} connected")

    response = client.recv(255) # 255 octets max
    if response != "":
        print(response)
        if response.endswith(b"CLOSE"):
            break

print("Close")
client.close()
mon_socket.close()
```

client

```
import socket

host = "localhost"
port = 15555

mon_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mon_socket.connect((host, port))
print(f"Connection on {port}")

mon_socket.send(b"Hey my name is Guillaume!")
mon_socket.send(bytes("Hey my name is Guillaume!", encoding="UTF-8"))
mon_socket.send(bytes("Nice to meet you !", encoding="UTF-8"))
mon_socket.send(bytes("CLOSE", encoding="UTF-8"))

print("Close")
mon_socket.close()
```

Socket en mode connecté : TCP ou stream

```
# Création d'un socket TCP
tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

- TCP (Transmission Control Protocol) :
  Protocole de communication fiable et orienté connexion
  - garantit la livraison des données sans perte ni erreur, grâce à des mécanismes de contrôle intégrés
  - établit d'abord une connexion avant de transférer des données entre les parties communicantes
- Utilisé pour des flux de données continus, comme la transmission de fichiers.

Socket en mode non connecté : UDP ou datagram

```
# Création d'un socket UDP
udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

- UDP (User Datagram Protocol) :
  Protocole de communication non fiable et sans connexion.
  - ne garantit pas la livraison des données ni leur ordre
  - n'établit pas de connexion préalable avant d'envoyer des données
- Utilisé pour des transmissions rapides mais non garanties, comme la diffusion de messages.

Combinaison des sockets et des threads

- Utilisation de threads pour gérer plusieurs connexions simultanées.
- Permet de traiter les connexions entrantes de manière asynchrone.

Exemple Combinaison sockets et threads

```python
import socket
import threading

class ClientThread(threading.Thread):
    def __init__(self, ip, port, clientsocket):
        super().__init__()
        self.ip = ip
        self.port = port
        self.clientsocket = clientsocket
        print(f"[+] Nouveau thread pour {self.ip} {self.port}")

    def run(self):
        print(f"Connexion de {self.ip} {self.port}")

        try:
            r = self.clientsocket.recv(2048).decode()
            print("Ouverture du fichier:", r, "...")
            with open(r, 'rb') as fp:
                self.clientsocket.send(fp.read())
        except FileNotFoundError:
            print("Fichier introuvable.")
        except Exception as e:
            print("Une erreur s'est produite:", e)

        print("Client déconnecté...")

def main():
    tcpsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    tcpsock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    tcpsock.bind(("", 1111))

    while True:
        tcpsock.listen(10)
        print("En écoute...")
        (clientsocket, (ip, port)) = tcpsock.accept()
        newthread = ClientThread(ip, port, clientsocket)
        newthread.start()

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

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

XML et XSL

Introduction à XML

- **XML** (eXtensible Markup Language) est un langage de balisage conçu pour stocker et transporter des données.

```
<livre>
    <titre>Le Seigneur des Anneaux</titre>
    <auteur>J.R.R. Tolkien</auteur>
    <genre>Fiction/Fantaisie</genre>
    <annee_parution>1954</annee_parution>
</livre>
```

- Un Parser XML est utilisé pour analyser un document XML et le convertir en une représentation utilisable en programmation.

DOM et SAX

- **DOM (Document Object Model) :**
  - Représente le document XML sous forme d'une structure d'arbre en mémoire.
  - Permet de manipuler le document en accédant et en modifiant les nœuds de l'arbre.
- **SAX (Simple API for XML) :**
  - Fournit une interface basée sur les événements pour parcourir le document XML.
  - Convient aux documents XML de grande taille car il ne nécessite pas de charger tout le document en mémoire.

Utilisation de SAX `xml.sax`

- SAX permet de parcourir un document XML de manière séquentielle.
- On définit des gestionnaires d'événements pour réagir aux balises, attributs et contenu du document.

```python
import xml.sax

# Créer une classe de gestionnaire d'événements SAX
class XMLHandler(xml.sax.ContentHandler):
    def __init__(self):
        xml.sax.ContentHandler.__init__(self)

    # Méthode appelée lorsque le parser rencontre un nouvel élément
    def startElement(self, name, attrs):
        print("Élément trouvé :", name)
        if attrs:
            print("Attributs :")
            for attr_name, attr_value in attrs.items():
                print(f"{attr_name} = {attr_value}")

    # Méthode appelée lorsque le parser rencontre une balise de fin
    def endElement(self, name):
        print("Fin de l'élément :", name)

    # Méthode appelée lorsque le parser rencontre du texte
    def characters(self, content):
        if content not in " \n":
            print("Contenu :", content)


# Créer un objet XMLHandler
handler = XMLHandler()

# Créer un parser SAX
parser = xml.sax.make_parser()

# Associer le gestionnaire d'événements au parser
parser.setContentHandler(handler)

# Lire le document XML
with open("exemple.xml", "r") as xml_file:
    parser.parse(xml_file)
```

Utilisation de DOM avec `xml.dom.minidom`

- DOM charge tout le document XML en mémoire sous forme d'un arbre.
- Permet une manipulation plus aisée du document grâce à des méthodes d'accès aux nœuds.

```python
import xml.dom.minidom

# Parse le fichier XML
dom_tree = xml.dom.minidom.parse("exemple.xml")

# Obtient le document root
root_element = dom_tree.documentElement
print("Element racine :", root_element.tagName)

# Parcours les éléments enfants du root
for node in root_element.childNodes:
    if node.nodeType == node.ELEMENT_NODE:
        print("\nÉlément :", node.tagName)

        # Parcours les attributs de l'élément
        if node.hasAttributes():
            print("Attributs :")
            for attr_name, attr_value in node.attributes.items():
                print(f"{attr_name} = {attr_value}")

        # Parcours les nœuds enfants
        for child_node in node.childNodes:
            if child_node.nodeType == child_node.TEXT_NODE:
                print("Contenu :", child_node.data)

# Exemple de modification du contenu
first_child = root_element.firstChild
first_child.data = "Nouveau contenu"

# Enregistrer les modifications dans un nouveau fichier XML
with open("nouveau_exemple.xml", "w") as new_xml_file:
    dom_tree.writexml(new_xml_file)
```

ElementTree, alternative à SAX et DOM

ElementTree (`etree`) est un module Python intégré qui fournit une API simple et flexible pour travailler avec des arbres XML.
Il permet de manipuler des documents XML de manière efficace en fournissant des méthodes pour parser, construire et manipuler la structure arborescente des documents XML.
Il est plus performant que DOM mais reste moins performant que SAX pour les très gros fichiers XML.

Demo Etree

```xml
<utilisateurs>
    <utilisateur>
        <nom>John Doe</nom>
        <age>30</age>
        <ville>New York</ville>
    </utilisateur>
    <utilisateur>
        <nom>Jane Smith</nom>
        <age>25</age>
        <ville>Los Angeles</ville>
    </utilisateur>
</utilisateurs>
```

```python
import xml.etree.ElementTree as ET

# Charger le fichier XML
tree = ET.parse('exemplo.xml')
root = tree.getroot()

# Parcourir chaque utilisateur dans le document XML
for utilisateur in root.findall('utilisateur'):
    # Extraire les informations sur l'utilisateur
    nom = utilisateur.find('nom').text
    age = utilisateur.find('age').text
    ville = utilisateur.find('ville').text

    # Afficher les informations
    print(f"Nom: {nom}, Âge: {age}, Ville: {ville}")
```

Introduction à XSL

XSL (eXtensible Stylesheet Language) :

Famille de langages utilisés pour transformer et faire le rendu de fichier XML.

- **XSLT (XSL Transformations) :**
  Un sous-ensemble de XSL utilisé pour transformer les documents XML en d'autres formats, principalement en XML (ou en HTML par exemple).
- **XPath :**
  Langage de requête utilisé pour naviguer et extraire des informations d'un document XML en spécifiant des chemins vers des éléments ou des attributs.
- **XSL-FO (XSL Formatting Objects):**
  Language pour spécifier la transformation visuelle d'un document XML (rendu).

Demo XPath en Python

Cheat Sheet XPath

```
from lxml import etree

# Charger le document XML
tree = etree.parse("exemple.xml")

# Exécuter une requête XPath
result = tree.xpath("//tag[@attribut='valeur']/text()")

# Afficher le résultat
print(result)
```

XSLT - Transformations XSL

- Un langage de transformation basé sur des modèles(templates) pour convertir un document source XML en un autre document.
- Utilise des règles de correspondance et des modèles pour décrire la transformation.
- Le module `lxml` offre une prise en charge complète de XSLT en Python
  `pip install lxml`
- Permet de charger des feuilles de style XSLT et d'appliquer des transformations sur des documents XML.

Exemple XML-XSLT-XML

Original

```xml
<?xml version="1.0" ?>
<persons>
  <person username="JS1">
    <name>John</name>
    <family-name>Smith</family-name>
  </person>
  <person username="MI1">
    <name>Morka</name>
    <family-name>Ismincius</family-name>
  </person>
</persons>
```

XSLT

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="/persons">
    <root>
      <xsl:apply-templates select="person"/>
    </root>
  </xsl:template>

  <xsl:template match="person">
    <name username="{@username}">
      <xsl:value-of select="name" />
    </name>
  </xsl:template>

</xsl:stylesheet>
```

Résultat

```xml
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <name username="JS1">John</name>
  <name username="MI1">Morka</name>
</root>
```

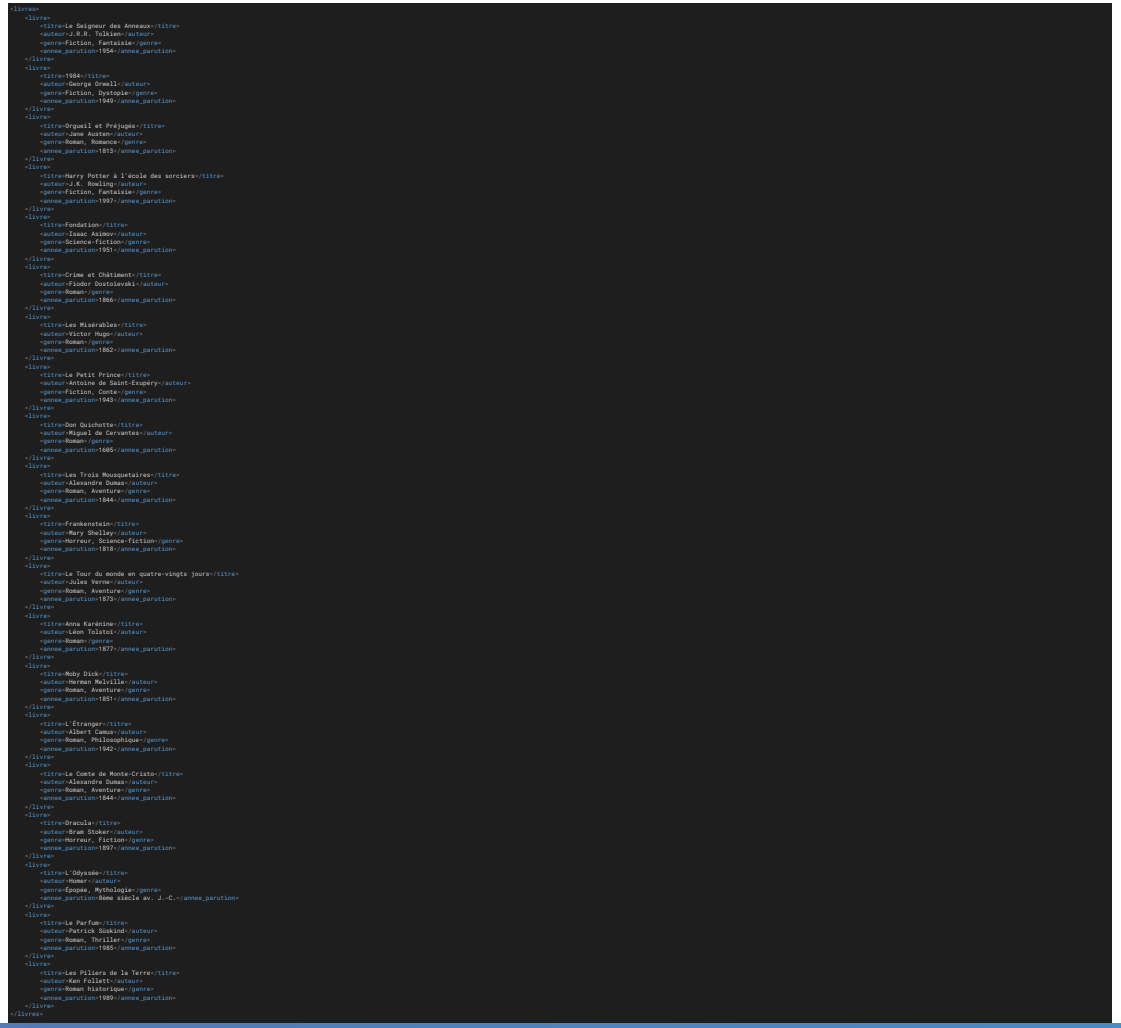
Exemple de Transformation XSLT en Python

```python
from lxml import etree

# Charger le document XML source
source_doc = etree.parse("source.xml")

# Charger la feuille de style XSLT
xslt_doc = etree.parse("style.xsl")

# Créer un transformateur XSLT
transformer = etree.XSLT(xslt_doc)

# Appliquer la transformation
result = transformer(source_doc)

# Afficher le résultat
print(result)
```

Exercice

Partir de ce fichier xml et le transformer avec un fichier xslt et un script python selon la syntaxe suivante

```xml
<livres>
  <livre annee="1954" genre="Fiction, Fantaisie">Le Seigneur des Anneaux - J.R.R. Tolkien</livre>
</livres>
```



UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

JSON

JSON

- Pour manipuler des fichiers JSON, il va nous falloir faire appel au module `json`
- Via ce module, nous disposons ensuite de 4 méthodes principales :
  - `json.dump()` : sauvegarder des données dans un flux données (ex: fichier)
  - `json.load()` : chercher les données dans le flux et les retourner avec typage pour correspondre à python (`dict`)
  - `json.dumps()` : récupérer une chaîne de caractère correspondant au JSON dans le but de l'afficher ou de l'envoyer
  - `json.loads()` : récupérer des données correspondantes à un JSON sous la forme d'une chaîne de caractère.
- Elles permettent la **sérialisation** (data => texte json) et la **désérialisation** (texte json => data) du json comme pour les modules XML vu précédemment

Exemple Json

```
import os, json  # Pour manipuler le JSON, il nous faut le module JSON

file_path = './file.json'
mon_dict = {'people': ['Albert', 'Martin', 'Louis'], 'mes Chiens': [1, 2, 4, 5]}


# Pour manipuler les JSON fichiers, il nous faut accéder aux deux méthodes ci-dessous
if os.path.exists(file_path):
    file = open(file_path, 'r')
    # Une fois le chargement du JSON, on obtient ici une liste de dictionnaire car le JSON contient plusieurs éléments dans un tableau

    # Pour charger un fichier dans un dictionnaire, il nous faut la méthode .load()
    data = json.load(file)
    file.close()
    print(data)
else:
    file = open(file_path, 'w')

    # Pour sauvegarder un objet dans un JSON, il nous faut la méthode .dump() (indent sert à avoir une présentation plus esthétique)
    json.dump(mon_dict, file, indent=4)
    file.close()

# Pour obtenir la variable string qui va être la représentation textuelle d'un objet, on peut se servir de la méthode .dumps() (Avec indent=XX pour l'esthétique ) qui va retourner un string
json_str = json.dumps(mon_dict, indent=4)
print(json_str)
print(type(json_str))

# Pour transformer une chaine de caractère au format JSON en un dictionnaire, il existe la méthode .loads() qui va retourner un dictionnaire
data = json.loads(json_str)
print(data)
print(type(data))
print(data['people'])
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Interfaces Graphiques : Tkinter, wxPython et Qt

Différentes API pour interfaces graphiques

- Tkinter (utilise Tk)
- wxPython (utilise wxWidgets)
- Qt (`PyQt` ou `PySide`)

Tkinter

- Bibliothèque standard de Python pour créer des interfaces graphiques.
- Simple et facile à apprendre.
- Convient aux applications simples et aux débutants en GUI.

```
from tkinter import *

root = Tk()
root.geometry("200x150")
frame = Frame(root)
frame.pack()

leftframe = Frame(root)
leftframe.pack(side=LEFT)

rightframe = Frame(root)
rightframe.pack(side=RIGHT)

label = Label(frame, text = "Hello world")
label.pack()

button1 = Button(leftframe, text = "Button1")
button1.pack(padx = 3, pady = 3)
button2 = Button(rightframe, text = "Button2")
button2.pack(padx = 3, pady = 3)
button3 = Button(leftframe, text = "Button3")
button3.pack(padx = 3, pady = 3)

root.title("Test")
root.mainloop()
```

wxPython

- Basée sur la bibliothèque wxWidgets, qui est écrite en C++.
- Offre une grande variété de widgets et de fonctionnalités.
- Aspect natif sur chaque plateforme.
- `pip install wxpython`

```python
import wx

# Création d'une application
app = wx.App(False)

# Création d'une fenêtre
frame = wx.Frame(None, wx.ID_ANY, "Ma première application wxPython", size=(300, 200))

# Ajout de widgets
panel = wx.Panel(frame, wx.ID_ANY)
text_ctrl = wx.TextCtrl(panel, wx.ID_ANY, "Bonjour, wxPython!", style=wx.TE_READONLY)

# Agencement des widgets
sizer = wx.BoxSizer(wx.VERTICAL)
sizer.Add(text_ctrl, 1, wx.EXPAND | wx.ALL, 10)
panel.SetSizer(sizer)

# Affichage de la fenêtre
frame.Show(True)

# Lancement de la boucle principale
app.MainLoop()
```

Qt (via PyQt)

- Basée sur la bibliothèque Qt, qui est écrite en C++.
- Offre une grande puissance et flexibilité.
- Aspect natif sur chaque plateforme.
- Écrit en C++
- `pip install pyqt6`

```python
from PyQt6.QtWidgets import QApplication, QMainWindow, QLabel

# Création d'une application
app = QApplication([])

# Création d'une fenêtre principale
fenetre = QMainWindow()
fenetre.setWindowTitle("Ma première application PyQt6")

# Ajout d'un label
label = QLabel("Bonjour, PyQt6!")
fenetre.setCentralWidget(label)

# Affichage de la fenêtre
fenetre.show()

# Lancement de la boucle principale
app.exec()
```

Composants de base de PyQt6

- **Widgets :** Interfaces graphiques tels que les boutons, les champs de texte, les fenêtres, etc.
  Il est possible de **créer et réutiliser nos propre widgets**
- **Layouts :** Organisateurs pour placer les widgets dans une fenêtre.
- **Events :** Gestion des événements tels que les clics de souris, les frappes de clavier, etc.
- **Stylesheets :** Personnalisation de l'apparence des widgets en utilisant CSS.

Demo

```
import sys
from PyQt6.QtCore import *
from PyQt6.QtGui import *
from PyQt6.QtWidgets import *
class MaFenetre(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('PyQt')
        button = QPushButton('Hello !')
        self.setCentralWidget(button)
def main():
    app = QApplication(sys.argv)
    fenetre = MaFenetre()
    fenetre.show()
    app.exec()
if __name__ == '__main__':
    main()
```

Layout de QMainWindow

- Central Widget = un objet dérivant de QWidget
  - Soit un widget **prédéfini**
  - Soit un widget **personnalisé**, défini par une classe dérivant de QWidget

Widget personnalisé

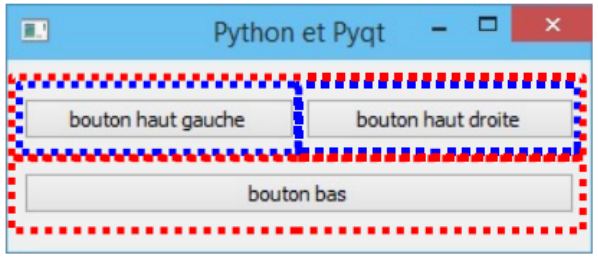
```
import sys
from PyQt6.QtCore import *
from PyQt6.QtGui import *
from PyQt6.QtWidgets import *

class MonWidget(QWidget):
    def __init__(self, parent):
        super(). __init__(parent)
        label = QLabel('Mon Titre', self)
        label.setAlignment(Qt.AlignmentFlag.AlignCenter)
        label.setGeometry(10, 10, 200, 20)
        image = QLabel(self)
        image.setPixmap(QPixmap('logo.svg').scaledToWidth(200))
        image.setGeometry(10, 30, 200, 100)
        bouton = QPushButton('OK', self)
        bouton.setGeometry(10, 150, 200, 20)
class MaFenetre(QMainWindow):
    def __init__(self):
        super(). __init__()
        self.setWindowTitle('Pyqt')
        central_widget = MonWidget(self)
        self.setCentralWidget(central_widget)
        self.resize(250, 250)
def main():
    app = QApplication(sys.argv)
    fenetre = MaFenetre()
    fenetre.show()
    app.exec()
if __name__ == '__main__':
    main()
```

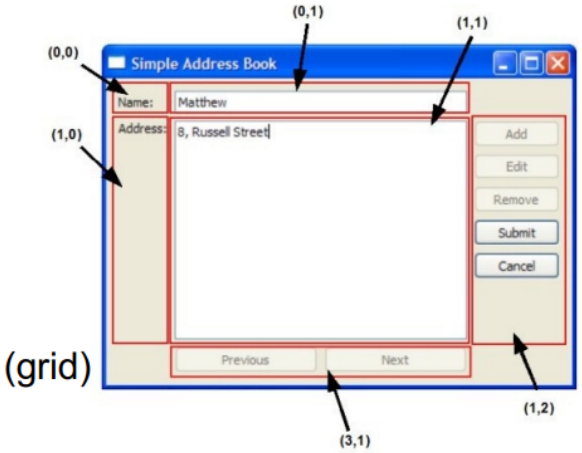
Avec Layout

```
import sys
from PyQt6.QtCore import *
from PyQt6.QtGui import *
from PyQt6.QtWidgets import *

class MonWidget(QWidget):
    def __init__(self, parent):
        super(). __init__(parent)
        layout = QVBoxLayout()
        label = QLabel('Mon Titre', self)
        label.setAlignment(Qt.AlignmentFlag.AlignCenter)
        label.setGeometry(10, 10, 200, 20)
        imagelbl = QLabel(self)
        imagelbl.setPixmap(QPixmap('logo.svg').scaledToWidth(200))
        imagelbl.setGeometry(10, 30, 200, 100)
        imagelbl.setAlignment(Qt.AlignmentFlag.AlignCenter)
        imagelbl.setStyleSheet('border: 10px solid green;background-color: #5f68ad;')
        bouton = QPushButton('OK', self)
        bouton.setGeometry(10, 150, 200, 20)
        layout.addWidget(label)
        layout.addWidget(imagelbl)
        layout.addWidget(bouton)
        self.setLayout(layout)
class MaFenetre(QMainWindow):
    def __init__(self):
        super(). __init__()
        self.setWindowTitle('Pyqt')
        central_widget = MonWidget(self)
        self.setCentralWidget(central_widget)
        self.resize(250, 250)
def main():
    app = QApplication(sys.argv)
    fenetre = MaFenetre()
    fenetre.show()
    app.exec()
if __name__ == '__main__':
    main()
```

Types de Layouts

- **QHBoxLayout** : Layout Horizontal, les éléments s'alignent **de gauche à droite** (en bleu)
- **QVBoxLayout** : Layout Vertical les éléments s'alignent de **haut en bas** (en rouge)
- **QGridLayout** : Layout sous forme de **grille**, avec lignes et colones
- **QFormLayout** : Layout pour les **formulaire**, il aligne les labels et les entrées





Gestion des évènements : Signal et Slot

- le signal permet de définir un déclencheur d'évènement
- on devra le connecter à un slot/callback, c'est à dire la méthode qui sera appelée au déclenchement de l'évènement

```
class MonWidget(QWidget):
    def __init__(self, parent):
        super().__init__(parent)
        self.bouton = QPushButton('Cliquez', self)
        self.bouton.setGeometry(10, 20, 200, 20)
        self.texte = QLineEdit(self)

        self.bouton.clicked.connect(self.on_bouton)
        #           signal                  slot/callback
    def on_bouton(self): # callback/méthode appelée au click sur le bouton
        self.texte.setText('clic sur le bouton !')
```

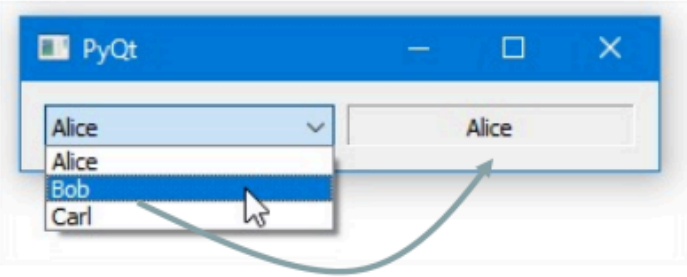
Gestion des évènements : Signal et Slot

- Dans certain cas le signal comporte des arguments qui pourront être passés au callback

```
class MonWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.label = QLabel()
        combobox = QComboBox()
        combobox.addItems(["Alice", "Bob", "Carl"])
        combobox.currentTextChanged.connect(self.afficher_nom)

    def afficher_nom(self, nom ):
        self.label.setText(nom)
```

émet un str

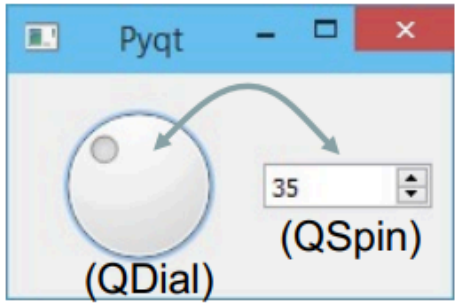
reçoit un str




```
dial = QDial(self)
spin = QSpinBox(self)
spin.valueChanged.connect(dial.setValue)
dial.valueChanged.connect(spin.setValue)
```

émet un int

reçoit un int



CSS

- QT supporte l'utilisation du CSS (Cascading StyleSheets) pour la customisation du style des widgets
  On pourra utiliser `setStyleSheet`
  - pour définir le style du widget
    ```
    imagelbl.setStyleSheet('border: 10px solid green;background-color: #5f68ad;')
    ```
  - pour définir une feuille de style au widget, qui s'appliquera à ses enfants
    ```
    self.setStyleSheet('QPushButton { font-weight: bold; font-size: 16px; }')
    ```
- Tutoriel

Widgets utiles pour aller plus loin

Boîtes de dialogue :

- **QMessageBox :** Popup message
- **QInputDialog :** Popup saisie
- **QFileDialog :** Popup fichiers

Widgets conteneurs :

- **QGroupBox :** Groupe de widgets
- **QTabWidget :** Onglets pour organiser les widgets

Barre d'outils et barre de statut :

- **QToolBar :** Raccourcis
- **QStatusBar :** État de l'application

Widgets complexes :

- **QListView :** Liste d'éléments
- **QTreeView :** Vue arborescente
- **QTableView :** Tableau

Exercice

Faire un application Qt basique qui :

- initialise un nombre mystère aléatoire entre 1 et 20,
- demande à l'utilisateur un nombre avec une entrée texte et un bouton
- affiche "plus petit" ou "plus grand" jusqu'à ce que l'utilisateur trouve le nombre mystère, dans ce cas on affichera "gagné" et le nombre mystère sera re-généré
- bonus : gérer un nombre d'essais restants à la fin duquel on affichera "perdu" et le nombre mystère sera re-généré

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Bases de données

Types de Bases de données (via les SGBD)

- **Relationnelles / SQL** : données sous formes de **tables**, utilisation de **clés primaires et secondaire** pour l'**identification** et les **relations**
- **Non-Relationnelles / NoSQL** : données **structurées différemment** ou **non-structurées**, différentes formes possibles :
  - Key-value pair (Redis)
  - Document-oriented (MongoDb)
  - Column-oriented (Cassandra)
  - Graph-based (Neo4j)
  - Time series (InfluxDB)
  - Object Based (ObjectDB)

Rappels SQL

- Le **langage de consultation/requêtage** le plus utilisé par les **SGBDR** modernes est le **SQL** (Structured Query Langage).
- Le langage a été standardisé en 1986 (ANSI) et a subit une révision majeure en 1992 et a été re-normalisé (ISO 9075).
- En 2011, SQL subit sa 7e révision majeure – technologie de l'information incluant principalement la notion de base de données temporelle.
- Malgré une norme bien établie, **son implémentation varie de façon plus ou moins importante d'un SGBD à un autre**.
- Ainsi, les SGBD existants tels que Oracle, MySQL, MS Access, SQL Server et tous les autres respectent généralement la normes mais présentent plusieurs particularités qui sont souvent ennuyeuses.

Rappels SQL

- Le langage SQL permet une **manipulation** efficace de toutes les **opérations** liées à la base de données (**CRUD**).
- SQL est un langage non procédural qui spécifie **ce qui doit être fait au lieu de comment le faire** (!= langage de programmation).
- En plus des manipulations conventionnelles, le langage SQL possède une multitude d'**outils** permettant de gérer les vues, les clés, les fonctions, les déclencheurs, les transactions, les usagers, ...
- SQL est si répandu que tous les environnements de développement possèdent une implémentation d'outils supportant son usage (C, C++, Pascal, Python, Ruby, Excel, Matlab, web, ...).

Rappels SQL

Le langage SQL utilise cette nomenclature spécifique :

- **objets** : Entités des SGBD telles que les tables, indexes, vues, usagers, ...
- **table** : Structure de données organisée en lignes et colonnes.
- **colonne** : Attribut spécifique des données dans une table.
- **ligne** : Entrée individuelle dans une table.
- **déclaration** : Instruction ou commande pour agir sur la base de données ou ses objets.

Rappels SQL

On divise le langage SQL en 4 parties :

- Langage de définition des données **DDL**
  `CREATE/DROP/ALTER/RENAME DATABASE/TABLE...`, Types, Contraintes
- Langage de manipulation des données **DML**
  `SELECT/INSERT/DELETE/UPDATE`, Jointures, Requêtes imbriquées/corrélées, ...
- Langage de contrôle des données **DCL**
  `GRANT/REVOKE ... TO user`, `CREATE/DROP/ALTER/SET ROLE`
- Langage de contrôle des transactions **TCL**
  `BEGIN TRANSACTION/COMMIT/ROLLBACK/SAVEPOINT/ROLLBACK TO SAVEPOINT`

Curseurs

- Un curseur est un mécanisme permettant de **parcourir** et **manipuler** les **résultats d'une requête SQL**, généralement ligne par ligne. Ils se classent dans la sous partie du langage nommée SQL Procédural.
- Ils sont beaucoup **utilisés** dans les **langages de programmation** (Notamment Python !)
- Ils utilisent une syntaxe SQL particulière :

```
DECLARE c CURSOR FOR/OPEN c/FETCH NEXT FROM c INTO/CLOSE c/DEALLOCATE c
```

DB-API (SQL-API) pour Python

- **Python DB-API** est **indépendante** de tout moteur de base de données, elle permet d'écrire des scripts Python pour **accéder à n'importe quel moteur de base de données**.
- La DB-API de Python se compose d'objets de connexion, d'objets de curseur, d'exceptions standard et de certains autres contenus de module
- Implémentations populaires :
  - MySQL : `mysql-connector-python`
  - SQLite3 : `sqlite3`
  - PostgreSQL : `psycopg2`
  - Oracle : `oracledb`
  - SQL Server : `pyodbc`

DB-API utilisation standard

- **Importation** du module API de la DB correspondante.
- Acquisition d'une **connexion** avec la base de données.
- Émission d'**instructions SQL** (généralement via un **curseur**).
- **Fermeture de la connexion**.

SQLite

- Système de gestion de base de données relationnelle (SGBDR) **léger, rapide et autonome**
- Largement utilisé dans les applications embarquées, les applications mobiles et les petits projets
- SQLite n'utilise pas de serveur distinct, ce qui signifie que **toute la base de données est contenue dans un seul fichier**
- Prend en charge la plupart des fonctionnalités SQL standard, mais avec **une empreinte mémoire et une configuration minimales**
- Forte **compatibilité** sur toutes les plateformes

Demo SQLite Python

```python
import sqlite3

# Connexion à la base de données (ou création si elle n'existe pas)
# => un Fichier avec l'extension sqlite, db ou sqlite3
conn = sqlite3.connect('example.sqlite3')

# Création d'un curseur
cursor = conn.cursor()

# Création d'une table
cursor.execute('''CREATE TABLE IF NOT EXISTS users
                  (id INTEGER PRIMARY KEY, name TEXT, age INTEGER)''')

# Insertion de quelques données
users_data = [
    ('Alice', 30),
    ('Bob', 25),
    ('Charlie', 35)
]
cursor.executemany('INSERT INTO users (name, age) VALUES (?, ?)', users_data)

# Commit des modifications et fermeture du curseur
conn.commit()

# Utilisation d'un curseur pour récupérer les données et les afficher
cursor.execute('SELECT * FROM users')
print("Données dans la table users :")
for row in cursor.fetchall():
    print(row)

# Fermeture de la connexion
conn.close()
```

MySQL

- Système de gestion de bases de données relationnelles (SGBDR) **open source et largement utilisé**
- Il permet de stocker, organiser et gérer des données de manière efficace.
- MySQL utilise le **langage SQL**
- Il est populaire pour sa **fiabilité**, sa **performance** et sa **compatibilité** avec de nombreuses plates-formes et langages de programmation.
- MySQL est utilisé dans de nombreux contextes, notamment pour le développement web, les applications d'entreprise et les solutions embarquées.

MySQL: installation rapide via Docker

- **Docker est une technologie de conteneurisation**, il permet pour résumer de créer des **conteneurs**, des machines virtuelles légères et simple d'utilisation
- Ainsi, via **Docker Desktop**, il est simple de supprimer l'image et le conteneur et consulter les logs de mysql
- Installation de Docker

MySQL: installation rapide via Docker

- Création du conteneur et de la database mysql (via un terminal powershell/bash) :

```
docker run --name mysql-python -e MYSQL_ALLOW_EMPTY_PASSWORD=true -e MYSQL_DATABASE="<nomDB>" -p 3306:3306 -d mysql:latest
```

- Entrer dans mysql pour taper des requêtes sql :

```
docker exec -it mysql-python mysql
```

Exemple : Lister les DB (depuis mysql dans le conteneur)

```
SHOW DATABASES;
```

- La Base MySQL créée est accessible ainsi :

```
server=localhost; port=3306(par défaut);
user=root; password=(aucun password nécessaire);
database=<nomDB>;
```

Demo MySQL Python

```python
import mysql.connector # pip install mysql-connector-python

# Connexion à la base de données MySQL
conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="",
    database="demo-python"
)

# Création d'un curseur
cursor = conn.cursor()

# Création d'une table
cursor.execute('''CREATE TABLE IF NOT EXISTS users
                  (id INT AUTO_INCREMENT PRIMARY KEY,
                   name VARCHAR(255),
                   age INT)''')

# Insertion de quelques données
users_data = [
    ('Alice', 30),
    ('Bob', 25),
    ('Charlie', 35)
]
cursor.executemany('INSERT INTO users (name, age) VALUES (%s, %s)', users_data)

# Commit des modifications
conn.commit()

# Sélection des données et affichage
cursor.execute('SELECT * FROM users')
print("Données dans la table users :")
for row in cursor.fetchall():
    print(row)

# Fermeture du curseur et de la connexion
cursor.close()
conn.close()
```

Exercice

Créer un programme python console permettant la gestion d'un chenil où les entités seront stockées dans une base MySQL.
Nous auront 2 entités :

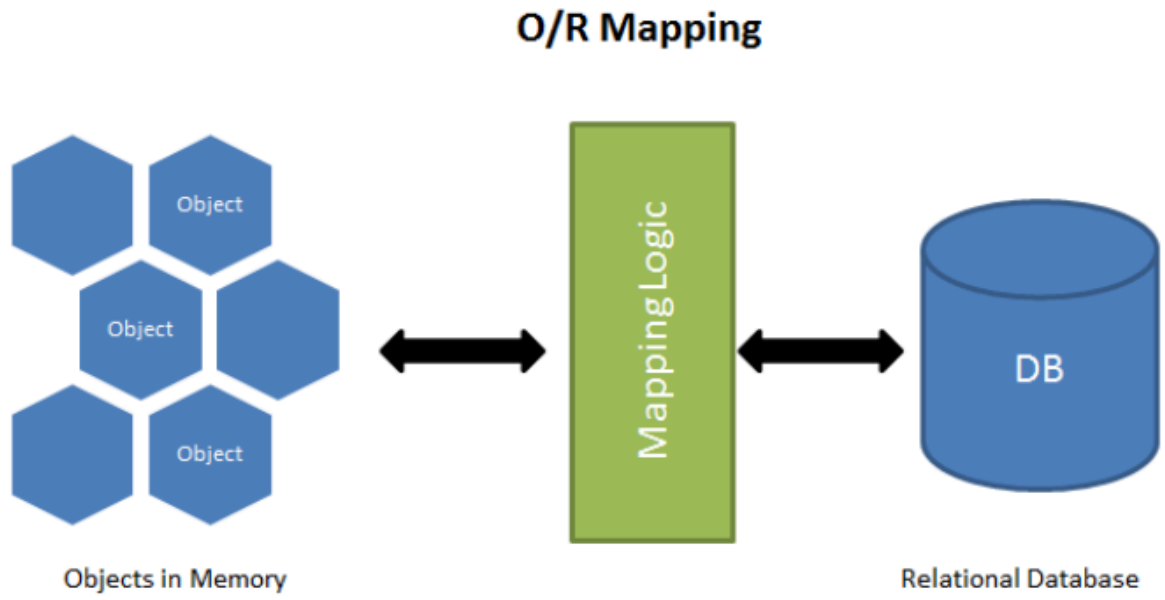
- Chien (Id, Nom, Age, RaceId)
- Race (Id, Nom)
  Ajouter en amont quelques chiens et races via un 2e script python `init_db.py`
  Le programme permettra via un menu :
- D'**ajouter** des chiens en choisissant l'id d'une race existante
- De **consulter** tout les chiens
- De **consulter** les chiens avec un race précise (id)
- De **supprimer** un chien via son Id

ORM

Un ORM (**Object Relational Mapper**) est une technologie **encapsulant les interactions avec la base de données** à travers la **manipulation d'objets** (POO).

C'est une **sur-couche** qui **utilise les connecteurs/API du langage**

On ne manipule que des objets, généralement sans avoir besoins de faire de requêtes SQL



Chaque table correspondra à une classe et les **instances** de cette classes pourront êtres **liées** (mappées) à leur **entrée** dans la table correspondante

SQLAlchemy

- SQLAlchemy est une bibliothèque Python.
- SQLAlchemy fournit une suite complète de fonctionnalités, conçus pour un accès efficace et performant à la base de données, adapté pour Python.
- SQLAlchemy offre une version Core et un ORM
- Documentation

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Intégration Python/C

Interfaçage avec d'autres langages

- Python est avantageux car il est **simple à écrire**, il nous fait gagner en efficacité, il est aussi **léger et facile à lancer** car il est **interprété**
- Seul chose que l'on peut lui reprocher : ses **performances**
- Comment y remédier ?
- Faire **appel à un autre langage** plus **puissant** car fortement typé, compilé, plus bas-niveau (proche de la machine), avec plus de contrôle de sécurité
- D'où l'intérêt de la possibilité d'appel du **C** dans python

Interfaçage avec C

Quand est ce que on utilise l'interface de Python en C ?

- Pour étendre les fonctionnalités de Python
- Pour améliorer les performances
- Pour utiliser Python comme langage de collage (assembler différentes partie d'un système)
- Pour créer des liaisons Python pour une bibliothèque C

Exemple comparatif

Soit un script qui fait 10 fois la moyenne des entiers de 0 à 9999999 :

C

Exécution : 0.09s

```c
#include <stdio.h>

int main(int argc, char **argv){

  int i, j, total;
  double avg;
  total = 10000000;
  for (i = 0; i < 10; i++){
        avg = 0;
        for (j = 0; j < total; j++){
          avg += j;
          }
        avg = avg/total;
        }
  printf("Average is %f\n", avg);
  return 0;
  }
```

Python

Exécution : 20.17s

```python
total = 10000000

for i in xrange(10):
  avg = 0.0
  for j in xrange(total):
        avg += j
  avg = avg/total

print "Average is {0}".format(avg)
```

Python avec **Numpy**

Exécution : 0.17s

```python
from numpy import mean, arange

total = 10000000

a = arange(total)

for i in xrange(10):
        avg = mean(a)
print "Average is {0}".format(avg)
```

- Interfaçage avec d'autres langages
- Pour l'interaction avec du C:

Python C API
Ctypes
Cython

Boost
Swig
pybind11

Ctypes

- Ctypes est une bibliothèque de fonctions pour Python
- Il fournit des types de données compatibles C et permet d'appeler des fonctions dans des fichiers assembly (projets compilés) avec les extensions `.dll` ou `.so`
- Pour l'utiliser, il nous faudra donc le fichier `.so` fait par le développeur C ainsi que la documentation correspondante.

Ctypes Demo

Fichier `main.c` d'origine =>

Télécharger directement la version compilée .so :

- x86
- ARM64

```c
#include <stdio.h>
#include <stdlib.h>

int int_add(int a, int b) {
    return a + b;
}

float  float_add(float a, float b){
    return a + b;
}

void float_add_ref(float *a, float *b, float *c) {
    *c = *a + *b;
}

void add_int_vecteur(int *a, int *b, int *c, int taille) {
    for(int i = 0; i < taille; i++) {
        c[i] = a[i] + b[i];
    }
}
int write_in_file(char* todo) {
    FILE *file_pointer = fopen("data.txt", "a");
    if(file_pointer != NULL)
    {
        fputs(todo, file_pointer);
        fclose(file_pointer);
        return 1;
    }
    return 0;
}
unsigned long * fib(unsigned n)
{
    unsigned long * fib_arr = (unsigned long *) malloc(n * sizeof(unsigned long));

    if (n >= 1)
        fib_arr[0] = 0;
    if (n >= 2)
        fib_arr[1] = 1;

    for (unsigned i = 2; i < n; ++i)
    {
        fib_arr[i] = fib_arr[i-2] + fib_arr[i-1];
    }

    return fib_arr;
}

void freeme(unsigned long * ptr) {
    free(ptr);
}

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Ctypes Demo

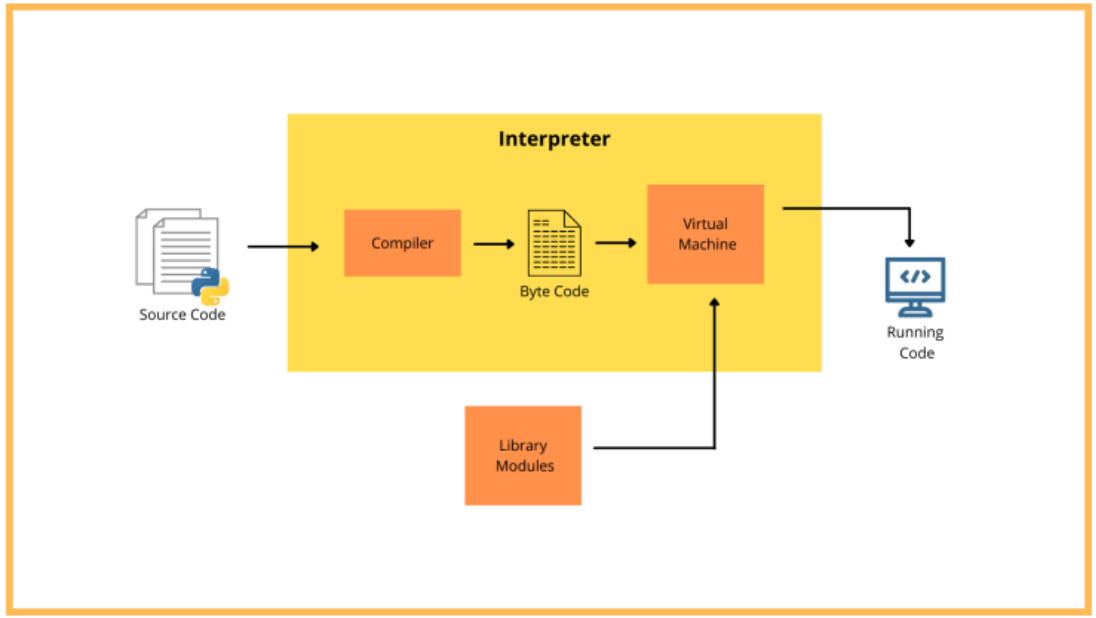
```python
import ctypes


def use_c_module():
    c_module = ctypes.CDLL('./lib1.so')
    # a = ctypes.c_float(10.0)
    # b = ctypes.c_float(30.0)
    # c = ctypes.c_float()
    # a = 10.0
    # b = 30.0
    # c_module.float_add.restype = ctypes.c_float
    # c_module.float_add.argtypes = [ctypes.c_float, ctypes.c_float]
    # result = c_module.float_add(a, b)
    # result2 = c_module.float_add(30.0, 20.0)
    # print(result)
    # print(result2)
    # c_module.float_add_ref(ctypes.byref(a), ctypes.byref(b), ctypes.byref(c))
    # print(c.value)
    # n = 3
    # v1 = (ctypes.c_int * n) (1, 3, 5)
    # v2 = (ctypes.c_int * n) (3, 5, 6)
    # res = (ctypes.c_int * n)(0,0,0)
    # taille = ctypes.c_int(n)
    # c_module.add_int_vecteur(v1, v2, res, taille)
    # print(res[1])

    #Avec un pointeur
    c_module.fib.restype = ctypes.POINTER(ctypes.c_ulong)
    a = 10
    res = c_module.fib(ctypes.c_uint(a))
    for i in range(10):
        print(res[i])

    c_module.freeme(res)
    #ctypes.cdll.LoadLibrary('msvcrt').free(res)

if __name__ == '__main__':
    use_c_module()
```

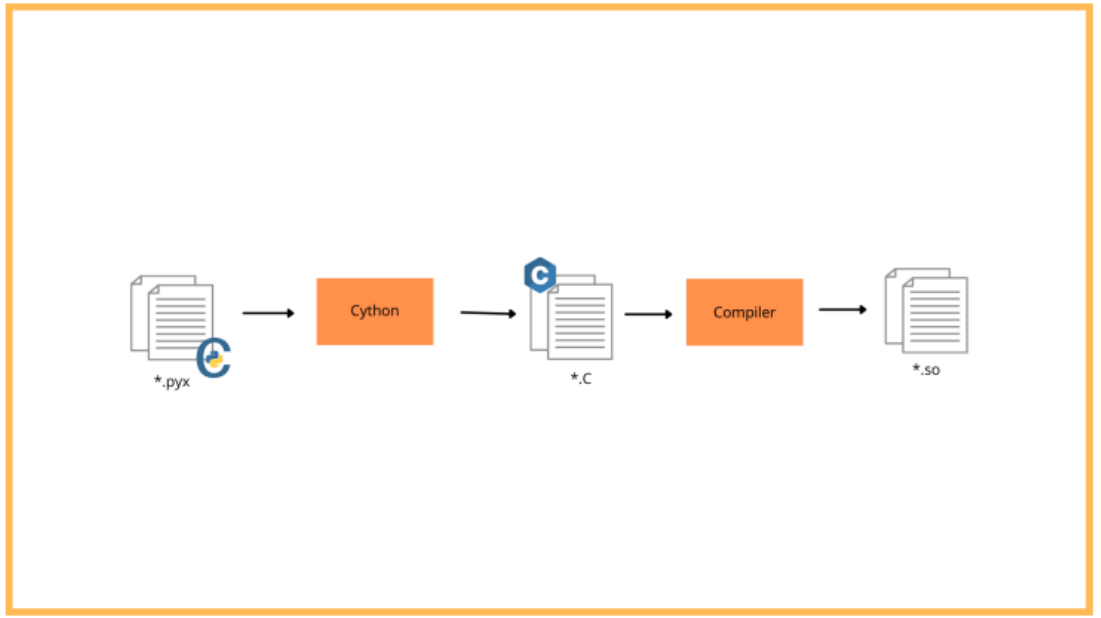
Cython

Cython peut être considéré à la fois comme un **module** et comme un **langage de programmation qui étend (en quelque sorte) Python** en permettant l'utilisation du typage statique emprunté à C/C++.

- Tout le code Python est valide Cython, mais pas l'inverse. Vous pouvez directement copier votre code Python existant dans un fichier Cython, puis le compiler pour améliorer les performances.
- Python est plus efficace à écrire que C étant donné qu'il s'agit d'un langage de haut niveau mais il reste lent.
- C en revanche, est moins efficace mais plus rapide que Python.
- Cython vise donc à apporter tous les avantages de C à Python tout en maintenant l'efficacité de Python.

Cython


Interpreter
Source Code
Compiler
Byte Code
Virtual Machine
Running Code
Library Modules


*.pyx
Cython
*.C
Compiler
*.so

Exemple Cython

```python
import time

def demo_python():
    t1 = time.time()
    total = 0
    for k in range(1000000000):
        total += k
    print(f"Total = {total}")
    t2 = time.time()
    t = t2 - t1
    print(f"{t:.20f}")
```

```python
import time

def demo_cython():
    cdef unsigned long long int total
    cdef int k
    cdef float t1, t2, t

    t1 = time.time()
    total = 0
    for k in range(1000000000):
        total += k
    print("Total =", total)
    t2 = time.time()
    t = t2 - t1
    print(f"{t:.100f}")
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Débogage et Analyse en Python

Débogage : Exécution pas à pas

- Le **débogage** en Python consiste à **examiner et corriger les erreurs** dans le code.
- L'**exécution pas à pas** est une technique qui permet de **parcourir le code ligne par ligne** pour **identifier les problèmes**.
- Un **Inspecteur/Espion** permet de **suivre l'évolution de vos variables et instances**
- LKe debug mode est correctement pris en charge par Visual Studio Code et PyCharm

Modes : Verbose et Trace

- **Mode Verbose**: Affiche des informations détaillées sur l'exécution du programme, telles que les variables utilisées et les actions effectuées.

```
python -v script.py
```

- **Mode Trace**: Permet de suivre le flux d'exécution du programme en affichant les différentes étapes franchies par le code.

```
python -m trace --trace script.py
```

Analyse des Performances et Profiling

- L'analyse des performances consiste à évaluer l'efficacité d'un programme en termes de vitesse d'exécution et d'utilisation des ressources.
- Le profiling est une technique qui permet d'identifier les parties du code qui prennent le plus de temps à s'exécuter, ce qui permet d'optimiser les performances.

```
python -m cProfile script.py
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Merci pour votre attention !

Des Questions ?