

Programmation Événementielle

Frédéric Drouhin – frederic.drouhin@uha.fr

R3.09 : Programmation événementielle

SAÉ3.02 : Développer des applications communicantes

Déroulé des séances

- FA :
 - R3.09 : 16,5h dont 1,5h de cours (aujourd'hui)
 - SAE 3.02 : 16h dont 7h d'autonomie
- FI :
 - R3.09 : 15h dont 1,5h de cours (aujourd'hui)
 - SAE 3.02 : 47,5h dont 29,5h d'autonomie

Référentiel de compétences

- Créer des outils et applications informatiques pour les R&T
 - AC23.02 | Développer une application à partir d'un cahier des charges donné, pour le Web ou les périphériques mobiles
 - AC23.03 | Utiliser un protocole réseau pour programmer une application client/serveur
- SAÉ 3.02 | Développer des applications communicantes
- Composantes essentielles :
 - CE3.01 | en étant à l'écoute des besoins du client
 - CE3.02 | en documentant le travail réalisé
 - CE3.03 | en utilisant les outils numériques à bon escient
 - CE3.04 | en choisissant les outils de développement adaptés
 - CE3.05 | en intégrant les problématiques de sécurité

Contenus

- Notions de programmation synchrone vs asynchrone,
- Les principes de la programmation réseau,
- La gestion des processus : Thread ...
- Ces notions peuvent être approfondies à partir d'un ou plusieurs des exemples suivants :
 - Interface homme machine : applications graphiques, web ou smartphone,
 - Boucle d'événements,
 - Socket, websocket,
 - Timer,
 - Programmation asynchrone.

Prérequis

- R1.07, SAÉ1.05, R2.08, {R2.09}, R3.08
- Algorithmie de base : condition, boucle, tableau
- Programmation orienté objets
 - Notion de classes
 - Être capable de créer et d'utiliser un objet
 - Être capable de créer une classe
- Programmation Python
 - Programmation de l'algorithmie de base
 - Structure de données complexe : liste, tuple, dictionnaire, set (et itération)
 - Savoir mettre en œuvre la programmation orienté objets

Programmation Événementielle

- Notions abordées par ce cours
 - *Exception*
 - Thread
 - Socket
 - Programmation événementielle
- Une belle SAÉ
 - SAÉ3.02 : Développer des applications communicantes

Exception en Python

C'est quoi une exception (en programmation) ?

- Evènement inattendu qui se produit *durant l'exécution*
 - Impossible de lire/ouvrir/écrire un fichier
 - Impossible d'accéder à une ressource
 - *Division par zéro* → est-ce si inattendu que ça ?
- Syntaxe spécifique de gestion de l'exception
 - Spécifique au langage
 - Séquence d'instructions spécifique en cas d'exception
- Concept qui existe dans de nombreux langages : C++, Java, Python, ...

Un premier exemple

```
if __name__ == '__main__':  
    a = float(input("a: "))  
    b = float(input("b: "))  
    res = a/b  
    print(res)
```

Quels sont les différents cas possibles ?

Coder cette exercice et trouver les 3 (4) cas possibles avec des valeurs exemples

Avant de continuer

- Coder l'exercice précédent
- Trouver les 3/4 jeux de valeurs possibles

Un premier exemple

```
if __name__ == '__main__':  
    a = float(input("a: "))  
    b = float(input("b: "))  
    res = a/b  
    print(res)
```

```
a: 12  
b: 0  
Traceback (most recent call last):  
  File ...  
    res = a/b  
ZeroDivisionError: float division by zero
```

Quels sont les différents cas possibles ?

```
a: 12  
b: 15  
0.8
```

```
a: aaa  
Traceback (most recent call last):  
  File ...  
    a = float(input("a: "))  
ValueError: could not convert string to float:  
'aaa'
```

Un premier exemple

```
if __name__ == '__main__':  
    a = float(input("a: "))  
    b = float(input("b: "))  
    res = a/b  
    print(res)
```

```
a: 12  
b: 0  
Traceback (most recent call last):  
  File ...  
    res = a/b  
ZeroDivisionError: float division by zero
```

Quelles sont les exceptions possibles ?

```
a: 12  
b: 15  
0.8
```

```
a: aaa  
Traceback (most recent call last):  
  File ...  
    a = float(input("a: "))  
ValueError: could not convert string to float:  
'aaa'
```

Un premier exemple

```
if __name__ == '__main__':  
    a = float(input("a: "))  
    b = float(input("b: "))  
    res = a/b  
    print(res)
```

```
a: 12  
b: 0  
Traceback (most recent call last):  
  File ...  
    res = a/b  
ZeroDivisionError: float division by zero
```

Quelles sont les exceptions possibles ?

```
a: 12  
b: 15  
0.8
```

```
a: aaa  
Traceback (most recent call last):  
  File ...  
    a = float(input("a: "))  
ValueError: could not convert string to float:  
'aaa'
```


Un premier exemple

```
if name == 'main':  
    a = float(input("a: "))  
    b = float(input("b: "))  
    if b != 0:  
        res = a/b  
        print(res)  
    else:  
        print("Divide by zero")
```

```
a: 12  
b: 0  
Trying to divide by zero
```

Mais :

```
a: aaa  
Traceback (most recent call last):  
  File ...  
a = float(input("a: "))  
ValueError: could not convert string to float:  
'aaa'
```

Expression régulière
C'est lourd ! 

Gestion de l'exception

```
try:
```

```
except:
```

```
try:
```

```
except:
```

```
else:
```

```
try:
```

```
except:
```

```
else:
```

```
finally:
```

```
try:
```

```
except ZeroDivisionError:
```

```
except ValueError:
```

```
else:
```

```
finally:
```

Gestion de l'exception

```
try:  
  
except:
```

```
try:  
  
except:  
  
else:
```

```
try:  
  
except:  
  
else:  
  
finally:
```

```
try:  
  
except ZeroDivisionError as err:  
  
except ValueError as err:  
  
else:  
  
finally:
```


Gestion de l'exception

`try:`

Code pouvant générer un arrêt inattendu - exception

`except:` **# et ses alternatives :**

 # `except ValueError: except ValueError as err except (ValeurError, Error):`

Code en cas d'arrêt inattendu - exception

`else:` **# considéré comme une bonne pratique de placer le code « non problématique »**

Code si aucune exception n'est produite

`finally:`

Code à exécuter qu'il y ait eu ou pas une exception

Un premier exemple

```
if __name__ == '__main__':  
    a = float(input("a: "))  
    b = float(input("b: "))  
    res = a/b  
    print(res)
```

```
a: 12  
b: 15  
0.8
```

```
a: 12  
b: 0  
Traceback (most recent call last):  
  File ...  
    res = a/b  
ZeroDivisionError: float division by zero
```

```
a: aaa  
Traceback (most recent call last):  
  File ...  
    a = float(input("a: "))  
ValueError: could not convert string to float:  
'aaa'
```

Gestion des exceptions par l'exemple

```
if __name__ == '__main__':  
    a = float(input("a: "))  
    b = float(input("b: "))  
    res = a/b  
    print(res)
```

```
a: 12  
b: 0  
Traceback (most recent call last):  
  File ...  
    res = a/b
```

Division by zero

Reprendre cet exemple de code et coder la gestion d'exception

- Une exception générale
- Une exception spécifique
- Une exception avec l'affichage de l'erreur
- (nous considérons que nous testons pas si b = 0)

```
a: 12  
b: 15  
0.8
```

```
Traceback (most recent call last):  
  File ...  
    a = float(input("a: "))  
ValueError: could not convert string to float:  
'aaa'
```

Solutions

```
if __name__ == '__main__':  
    try:  
        a = float(input("a: "))  
        b = float(input("b: "))  
        res = a/b  
    except:  
        print("An exception occurs")  
    else:  
        print(res)
```

```
if __name__ == '__main__':  
    try:  
        a = float(input("a: "))  
        b = float(input("b: "))  
        res = a/b  
    except ValueError:  
        print("Please enter a float")  
    except ZeroDivisionError:  
        print("b should not be 0")  
    else:  
        print(res)
```

Solutions

```
if __name__ == '__main__':  
    try:  
        a = float(input("a: "))  
        b = float(input("b: "))  
        res = a/b  
    except ValueError as err:  
        print(f>Please enter a float: {err}")  
    except ZeroDivisionError as err:  
        print("b should not be 0: {err}")  
    else:  
        print(res)
```

Et si je veux lever une exception

- Préférez une exception python
- Toujours possible de créer sa propre exception par héritage
- Appel au mot clé `raise`
- `traceback`

1. `raise exception` – pas d'argument, message par défaut
2. `raise exception(args)` – message pour le programmeur
3. `raise` – sans argument, renvoie la dernière exception
4. `raise exception(args) from original_exception` – renvoie l'exception avec le contenu de l'exception d'origine

```
a = int(input("Enter a positive number:"))
if a <= 0:
    raise ValueError("It's not a positive number")
```

Quelques tips

- `except ZeroDivisionError as err:`
 - N'a d'intérêt que si vous voulez donner l'erreur à l'utilisation ou mettre dans un log.
- Ne pas utiliser « `except:` »
 - Certaines exceptions ne doivent pas être gérées
 - Préférez mettre `except (ValueError, TypeError, NameError):`
- <https://docs.python.org/3/library/exceptions.html>
 - Pour une liste des exceptions qui sont généralement levées

- La clause « `finally:` »

```
try:
    ...
except MemoryError:
    # ok mais que faire ?
    ---
try:
    ...
finally:
    # je ferme les flux et l'exception
    # provoque l'arrêt du programme
```

Tips : la clause « else : »

```
try:
    a = float(input("a: "))
    b = float(input("b: "))
    res = a/b
except (ValueError, ZeroDivisionError):
    print("An error occurs")
else:
    print(res)
```

```
try:
    a = float(input("a: "))
    b = float(input("b: "))
    res = a/b
    print(res)
except (ValueError, ZeroDivisionError):
    print("An error occurs")
```

La clause « else : » est une bonne pratique donc ... vous devez l'utiliser

Exercice sur les exceptions : ouverture d'un fichier

```
f = open('testfile.txt', 'r')
```

A partir de l'ouverture d'un fichier en lecture gérer les exceptions :

- Lors de l'ouverture
- Lire la première du fichier
- Fermer le fichier

Reprenez cette exercice mais en écriture avec :

- Saisir une ligne et l'ajouter au fichier
- En gérant un finally si peu que le disque soit plein (par exemple)

• Exceptions possible :

- FileNotFoundError
- IOError
- FileExistsError
- PermissionError

Exercice lever sa propre exception

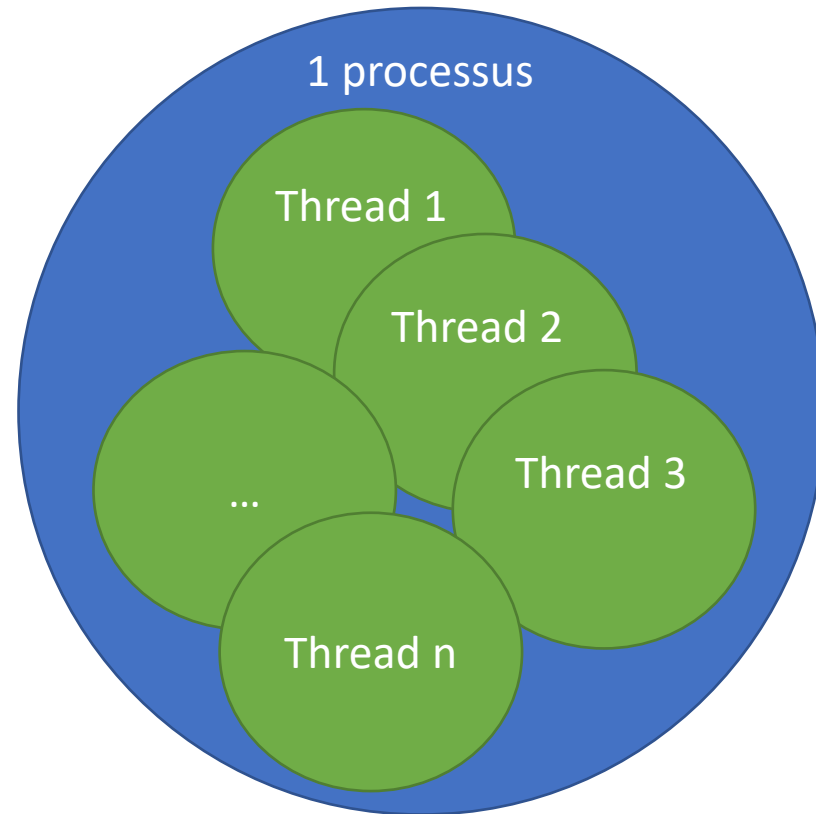
- A partir d'un fichier csv d'article :
nom;référence;tva;prix
 - Par exemple Pantalon;1;0.2;100
- Créer une classe Article
 - qui lève une exception si le prix TTC est ≤ 0
 - Avec les attributs correspondant au csv
 - (pas de prix TTC)
- Créer une classe Stock avec 100 articles
 - Stock de 100 articles = paramètre du constructeur
- Méthode de Stock :
 - add, delete by name – by ref, insert, sort by name - by ref
 - Créer une fonction de lecture des articles qui lit le fichier csv et qui crée les articles correspondant dans la limite de la taille du stock d'articles
 - Une fonction d'écriture dans un format parmi json, csv, xls
 - Une méthode de lecture du fichier json, csv, xls
- Gérer les exceptions liées aux deux classes
- Faire un main avec les tests de votre stock
- Si vous êtes à l'aise, utilisez unittest



Exception class hierarchy

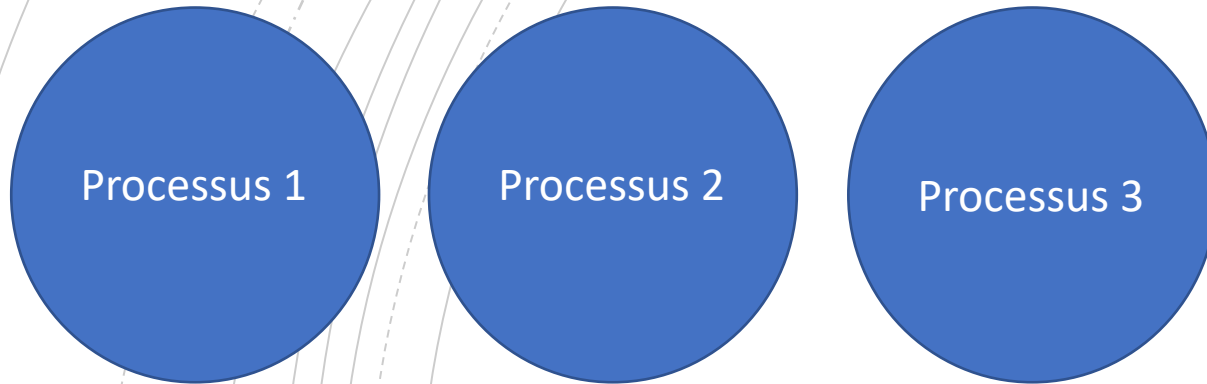
Thread : introduction

Thread

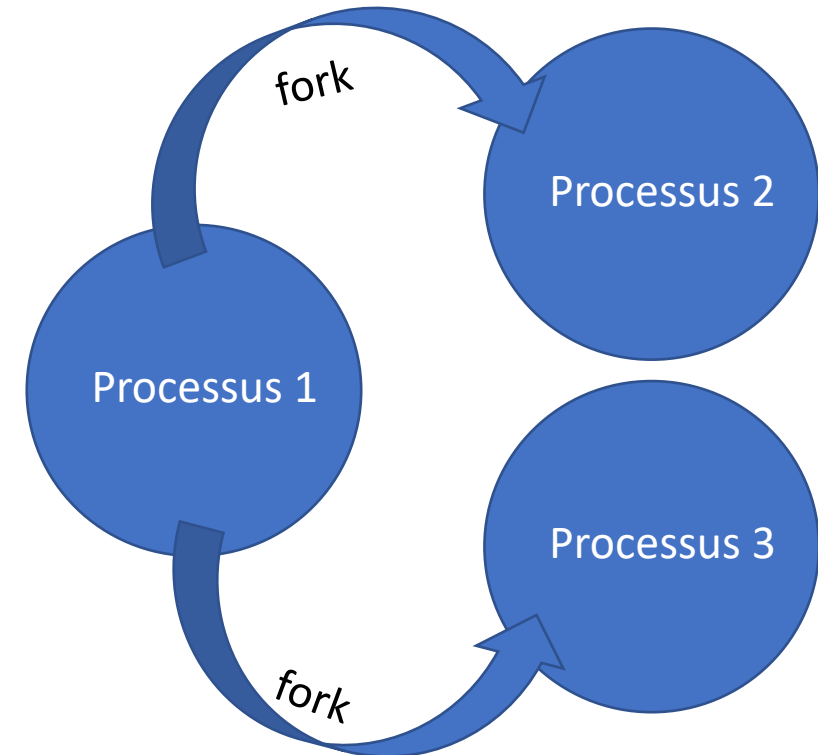


Concept de thread :
Même environnement d'exécution
Mémoire partagée

Processus



Environnement d'exécution indépendant
Pas de partage de mémoire



Et du coup : multi-coeurs / multi-processeurs

Multicœur	Multiprocesseur
Un seul CPU ou processeur avec au moins deux unités de traitement indépendantes appelées cœurs capables de lire et d'exécuter des instructions de programme.	Un système avec deux processeurs ou plus qui permet le traitement simultané de programmes.
Il exécute un seul programme plus rapidement.	Il exécute plusieurs programmes plus rapidement.
Pas aussi fiable qu'un multiprocesseur.	Plus fiable car la défaillance d'un processeur n'affectera pas les autres.
Il y a moins de trafic (intégré dans un seul processeur)	Il a plus de trafic (utilisation des bus, ...)
Il n'a pas besoin d'être configuré.	Il nécessite peu de configuration complexe.
C'est très moins cher (un seul processeur qui ne nécessite pas de système de prise en charge de plusieurs processeurs).	C'est cher (plusieurs processeurs séparés qui nécessitent un système prenant en charge plusieurs processeurs) par rapport à MultiCore.

Attention au système d'exploitation qui doit le supporter

Thread par l'exemple

Commençons par un petit programme

```
import time

def task(i):
    print(f"Task {i} starts")
    time.sleep(1)
    print(f"Task {i} ends")

start = time.perf_counter()

task(1)

end = time.perf_counter()

print(f"Tasks ended in {round(end -
start, 2)} second(s)")
```

```
Task 1 starts
Task 1 ends
Tasks ended in 1.0 second(s)
```

Commençons par un petit programme

```
import time

def task(i):
    print(f"Task {i} starts")
    time.sleep(1)
    print(f"Task {i} ends")

start = time.perf_counter()

task(1)
task(2)

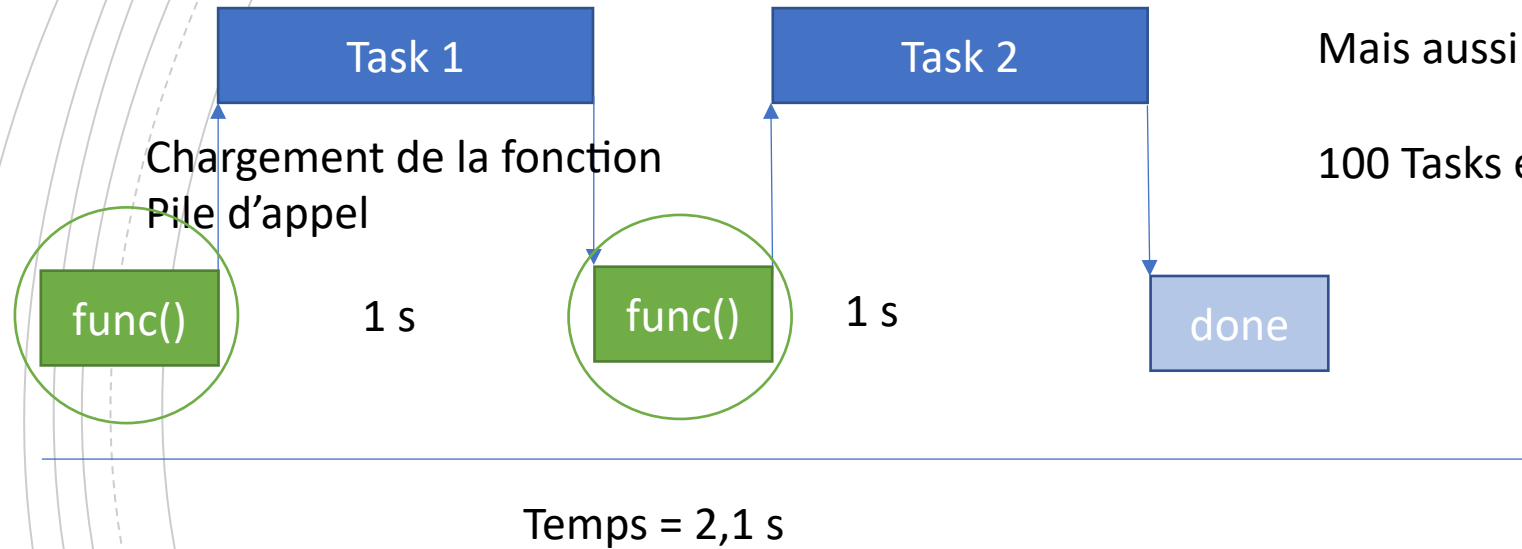
end = time.perf_counter()

print(f"Tasks ended in {round(end - start, 2)} second(s)")
```

```
Task 1 starts
Task 1 ends
Task 2 starts
Task 2 ends
Tasks ended in 2.01 second(s)
```

Bon je n'ai pas utilisé `__name__ == 'main'`
pour des raisons de place

Commençons par un petit programme



Mais aussi les appels à `perf_counter()`

100 Tasks ended in 100.37 second(s)

Programmation synchrone

- Chaque tâche est exécutée l'une après l'autre
- Elles peuvent être dépendante
 - Besoin du résultat pour continuer le programme
 - Par ex. : une somme avant de faire la moyenne
- Temps du programme $\approx \text{Nb} \times \text{temps}(\text{tâches}) + \epsilon$

Programmation asynchrone

- Lancer plusieurs tâches en parallèle
 - « gain de temps »
 - Utilisation des capacités de plusieurs cœurs ou de plusieurs processeurs
 - Time sharing, multitâche, multiprocessing, ...

Introduction aux Threads

```
import threading
import time

def task(i):
    print(f"Task {i} starts")
    time.sleep(1)
    print(f"Task {i} ends")

start = time.perf_counter()

t1 = threading.Thread(target=task, args=[1]) # créat
t1.start() # je démarre la thread
t1.join() # j'attends la fin de la thread

end = time.perf_counter()

print(f"Tasks ended in {round(end - start, 2)} second(s)")
```

Task 1 starts

Task 1 ends

Tasks ended in 1.01 second(s)

Pas de grand changement non ?

Introduction aux Threads

```
import threading
import time

...

start = time.perf_counter()

t1 = threading.Thread(target=task, args=[1])
t1.start()

t2 = threading.Thread(target=task, args=[2])
t2.start()

t1.join() # j'attends la fin de la thread
t2.join() # j'attends la fin de la thread

end = time.perf_counter()

print(f"Tasks ended in {round(end - start, 2)} second(s)")
```

Task 1 starts

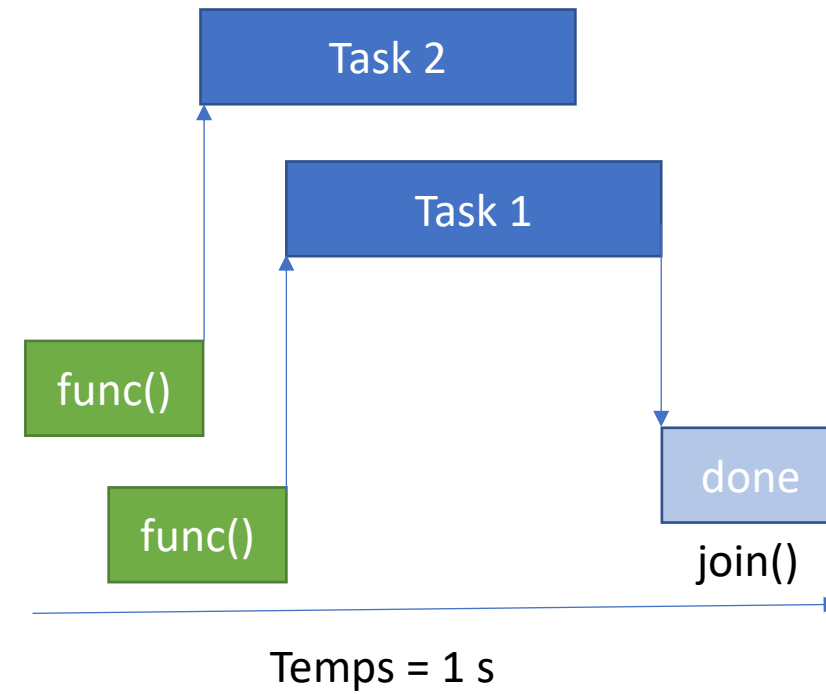
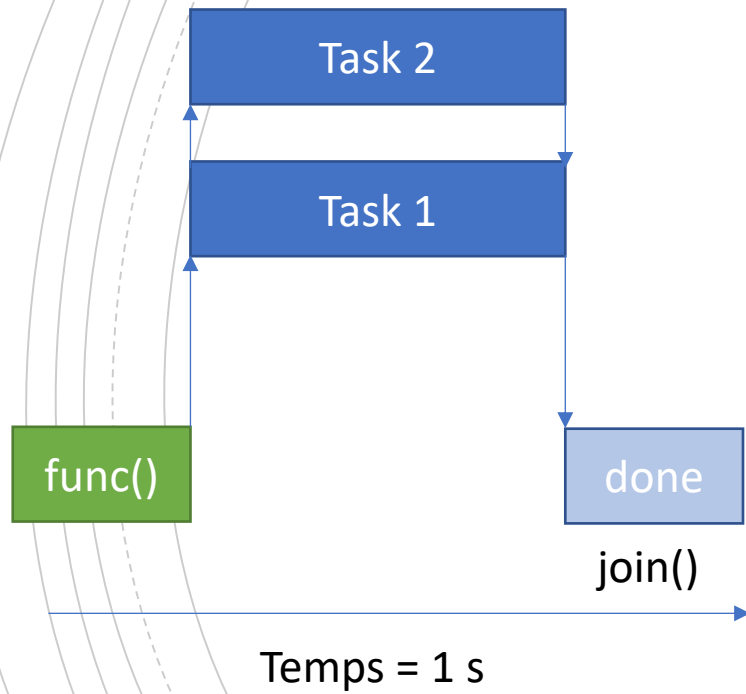
Task 2 starts

Task 1 ends

Task 2 ends

Tasks ended in 1.0 second(s)

Multithreading



Et pour 100 Threads

```
T = []

for i in range(100):
    T.append(threading.Thread(target=task, args=[i]))

for i in range(len(T)):
    T[i].start()

for i in range(len(T)):
    T[i].join()
```

```
Task 66 ends
Task 98 ends
Task 77 ends
Task 82 ends
Tasks ended in 1.01 second(s)
```

Et un tips :

```
T = []
print(dir(T))
```

Pas mal non ?

Asynchrone ➔ regarder le numéro des tâches

Nécessité de bien utiliser join si vous voulez synchroniser des tâches

Un point d'attention

```
t1 = threading.Thread(target=task, args=[1])
```

Ne pas mettre de parenthèse
Sinon vous déclenchez le fonctionnement de *task*
au moment de la création de la thread

```
def task(i):  
    print(f"Task {i} starts")  
    time.sleep(1)  
    print(f"Task {i} ends")
```

Arguments sous forme de liste

```
t1 = threading.Thread(target=task(1)) # création de la thread  
t1.start() # je démarre la thread  
  
t2 = threading.Thread(target=task(2)) # création de la thread  
t2.start() # je démarre la thread  
  
t1.join() # j'attends la fin de la thread  
t2.join() # j'attends la fin de la thread
```

```
Task 1 starts  
Task 1 ends  
Task 2 starts  
Task 2 ends  
Tasks ended in 2.01 second(s)
```

Et pour 100 Threads

```
T = []

for i in range(100):
    T.append(threading.Thread(target=task, args=[i]))

for i in range(len(T)):
    T[i].start()

for i in range(len(T)):
    T[i].join()
```

```
Task 66 ends
Task 98 ends
Task 77 ends
Task 82 ends
Tasks ended in 1.01 second(s)
```

Et un tips :

```
T = []
print(dir(T))
```

Pas mal non ?

Asynchrone ➔ regarder le numéro des tâches

Nécessité de bien utiliser join si vous voulez synchroniser des tâches

Avec une autre fonction

```
def task(i):  
    print(f"Task {i} starts for {i+1} second(s)")  
    time.sleep(i+1)  
    print(f"Task {i} ends")  
  
T = []  
  
for i in range(100):  
    T.append(threading.Thread(target=task, args=[i]))  
  
for i in range(len(T)):  
    T[i].start()  
  
for i in range(len(T)):  
    T[i].join()
```

Quel résultat à votre avis ?

Avec une autre fonction

```
def task(i):  
    print(f"Task {i} starts for {i+1} second(s)")  
    time.sleep(i+1)  
    print(f"Task {i} ends")
```

```
T = []
```

```
for i in range(10):  
    T.append(threading.Thread(target=task, args=(i,)))
```

```
for i in range(10):  
    T[i].start()
```

```
for i in range(10):  
    T[i].join()
```

Task 0 starts for 1 second(s)
Task 1 starts for 2 second(s)
Task 2 starts for 3 second(s)

Task 3 starts for 4 second(s)
Task 4 starts for 5 second(s)
Task 5 starts for 6 second(s)
Task 6 starts for 7 second(s)
Task 7 starts for 8 second(s)
Task 8 starts for 9 second(s)

Task 9 starts for 10 second(s)

Task 0 ends

Task 1 ends

Task 2 ends

Task 3 ends

Task 4 ends

Task 5 ends

Task 6 ends

Task 7 ends

Task 8 ends

Task 9 ends

Tasks ended in 10.0 second(s)

votre avis ?

Exercice

- Reprenez les codes proposés
- Testez ces codes

Sémaphore, mutex

- **Mutex** : permet l'accès synchrone à une ressource partagée
 - Par exemple, l'accès à un fichier (ou à un système de fichier)
 - Une fois qu'une thread est entrée dans la mutex
 - La thread suivante attend la libération de la mutex (accès à la ressource)
- **Sémaphore** : permet l'accès à un pool de ressources
 - Par exemple, l'accès à une base de données
 - 4 connexions à la BDD utilisables (et partagées)
 - Les 4 premières threads utilisent les connexions
 - Les autres sont mises en attente de la libération d'une connexion

Pool de threads

```
import time
import concurrent.futures
import requests

img_urls = [
    'https://pixabay.com/get/gf1a238e2982a8ca3133ba4a9f7cb4db8438be90b74885e72d16ec7e3f604d00e150c9b00b39a98a59942255bd9de38983077cfaa161d1b104bb404d273bae119b07fb9fbf160f3e1f8215081f04eaca2_1920.jpg',
    'https://pixabay.com/get/g00f4426320d22da4e94b9ba518ba83dc19a27e8b185a09f91e663d81ca35de6a1fc2d9a6e129497e8974b69e54642cf2bbd099e312460e15b2cdc82efe9cf1e4366f0774f95472d35180f92762342899_1920.jpg'
]

def download_image(img_url):
    img_bytes = requests.get(img_url).content
    img_name = img_url.split('/')[4]
    with open(img_name, 'wb') as img_file:
        img_file.write(img_bytes)
        print(f"{img_name} was downloaded")

start = time.perf_counter()

with concurrent.futures.ThreadPoolExecutor() as executor:
    executor.map(download_image, img_urls)

    #for img_url in img_urls:
    #    download_images(img_url)

end = time.perf_counter()
print(f"Tasks ended in {round(end - start, 2)} second(s)")
```


Multiprocessing

```
import time
import multiprocessing

def task():
    print(f"Task starts for 1 second")
    time.sleep(1)
    print(f"Task ends")

if __name__ == '__main__':
    start = time.perf_counter()
    p1 = multiprocessing.Process(target=task)
    p2 = multiprocessing.Process(target=task)
    p1.start()
    p2.start()
    end = time.perf_counter()
    print(f"Tasks ended in {round(end - start, 2)} second(s)")
```

Identique aux Threads non ?

Sauf que :

```
Tasks ended in 0.01 second(s)
Task starts for 1 second
Task starts for 1 second
Task ends
Task ends
```

Possibilité d'appeler `join()`

Exercice de validation

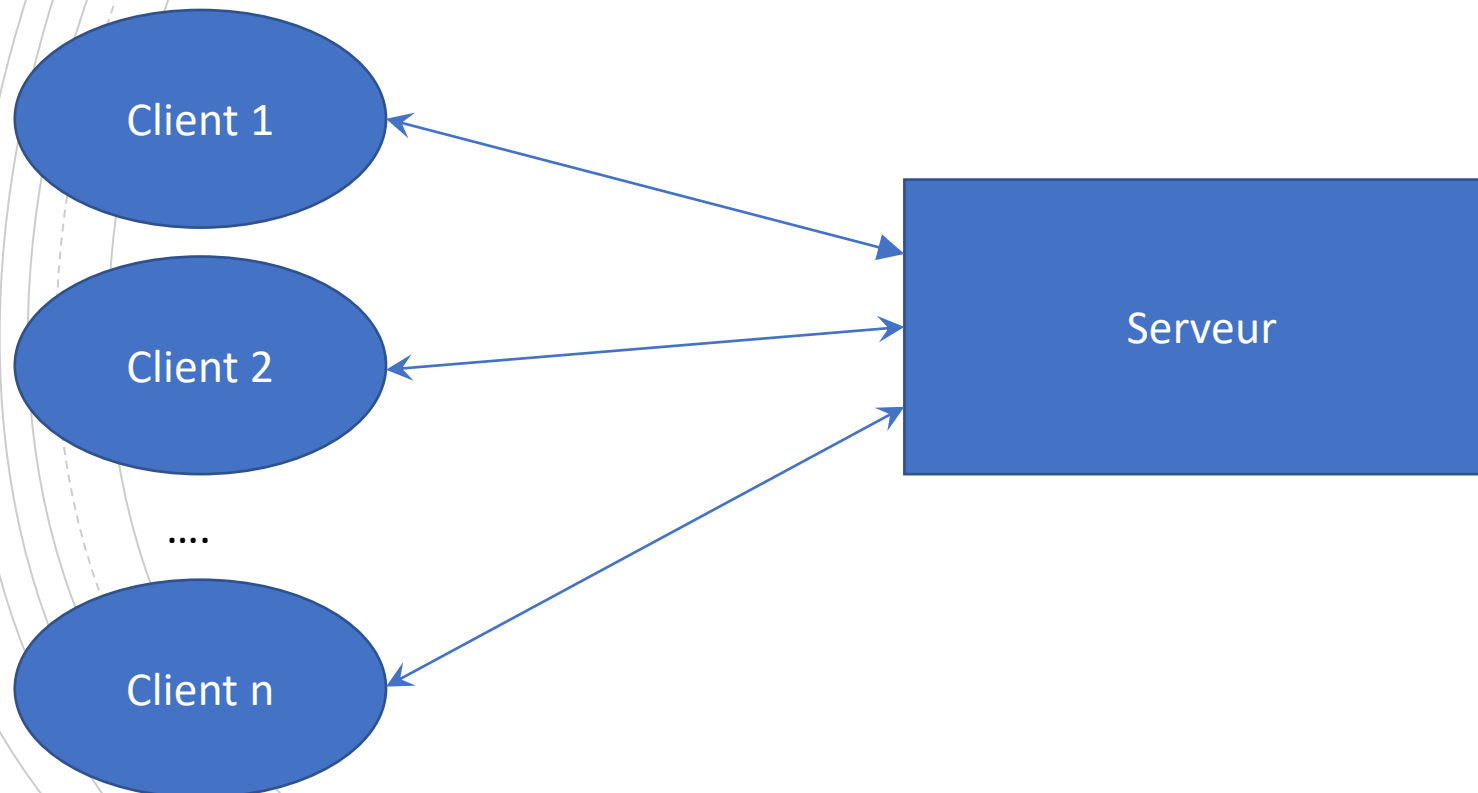
- En reprenant l'exercice de pool de threads, créer le téléchargement en parallèle des photos sur 3 codes différents :
 - Utilisation du mécanisme de `threading.Thread`
 - Utilisation du pool de threads
 - Utilisation du multiprocessing
 - A l'aide du suivi de processus, donnez la différence entre les threads et les processus
- Comparez les performances sur X tests à partir d'un petit code python
 - X est un argument de lignes de commande : `TestDownload.py --nb=10`
 - Il lance les trois codes à la suite
 - Il mesure les performances de chacun des codes
 - Gérer des exceptions si nécessaires

Programmation réseau

Les sockets

Programmation réseau

- Connexion à un serveur sur un port
 - Connexion à un serveur par exemple



Programmation réseau : les sockets

- Socket (BSD – *Berkeley Software Distribution*)
- Communication inter-processus : IPC - *Inter Process Communication*
- Processus de communication via un réseau TCP/IP
 - Y compris en local(host)
- Modes de communications
 - Protocole TCP : mode connecté pour des modes de communication durable (ACK)
 - Protocole UDP : mode non connecté pour des modes de communications (pas de ACK)

Programmation réseau : les sockets

Modèle OSI	Sockets
Application	Application utilisant les sockets
Présentation	
Session	
Transport	UDP / TCP
Réseau	IP / ARP
Liaison	Ethernet, ...
Physique	

Les sockets en python



```
client_socket = socket.socket()
client_socket.connect(host, port)
client_socket.send(message.encode())
data =
    client_socket.recv(1024).decode()
client_socket.close()
```

```
server_socket = socket.socket()
server_socket.bind(host, port)
server_socket.listen(1)
conn, address = server_socket.accept()
data = conn.recv(1024).decode()
conn.send(reply.encode())
conn.close()
```

Ne pas oublier l'import : `import socket`

Les sockets en python : côté serveur

Création de la socket

Association du host et du port

- Par exemple 127.0.0.1
- Le port 10000

Attente de la connexion

(nombre de communication simultanée)

Etablissement de la communication

Réception de données

Envoi de données

Fermeture de la communication

Serveur

`server_socket = socket.socket()`

`server_socket.bind(host, port)`

`server_socket.listen(1024)`

`conn, addr = server_socket.accept()`

**1024 correspond à la taille en bytes
→ buffer overflow**

**Si vous ne fermez pas la connexion le porte ouvert
sans possibilité de le fermer sauf par le système**

`conn.close()`

Les sockets en python : côté client

Client

```
client_socket = socket.socket()
```

Création de la socket

```
client_socket.connect(host, port)
```

Connexion au host et au port

```
client_socket.send(data)
```

Envoi de données

```
client_socket.recv(1024).decode()
```

Réception de données

```
client_socket.close()
```

Fermeture de la communication

Si vous ne fermez pas la connexion le port ouvert sans possibilité de le fermer sauf par le système

Les sockets en python

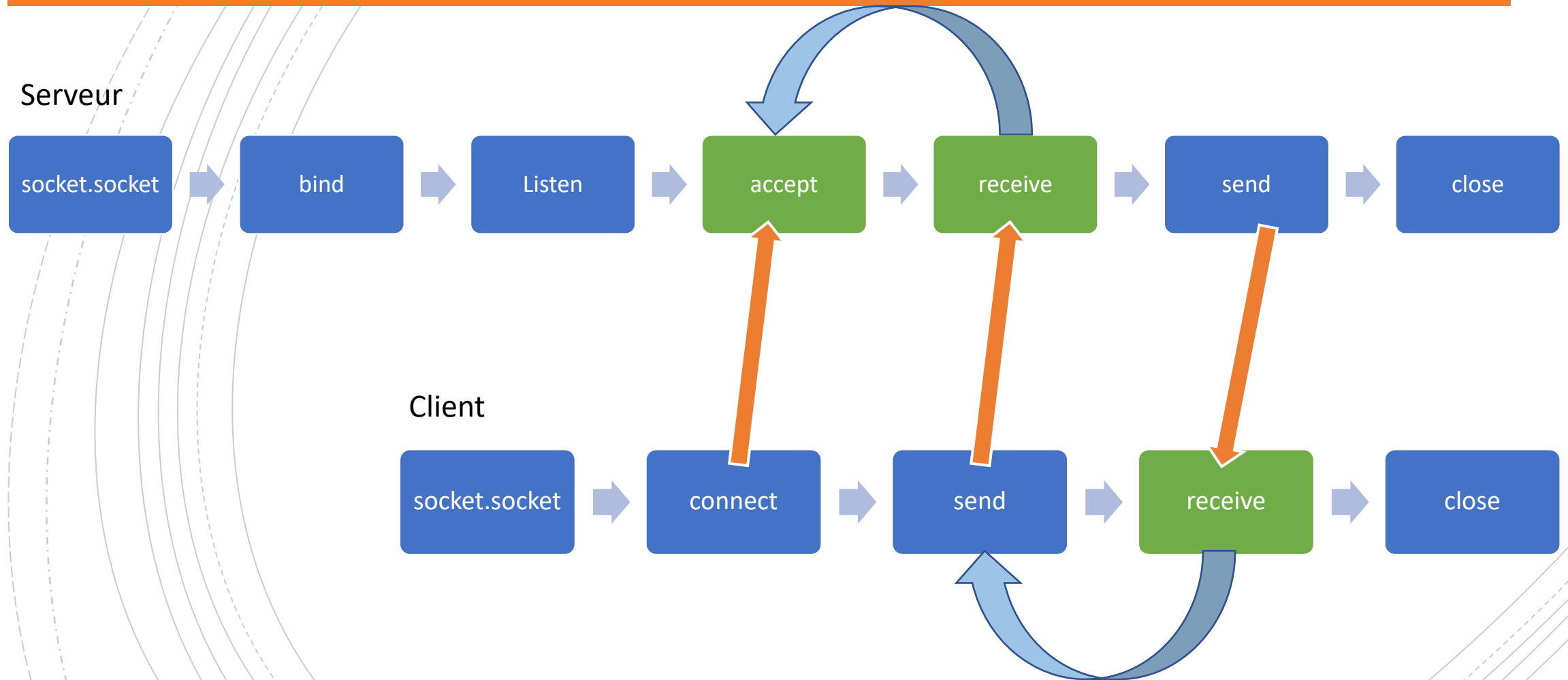


```
client_socket = socket.socket()
client_socket.connect(host, port)
client_socket.send(message.encode())
data =
    client_socket.recv(1024).decode()
client_socket.close()
```

```
server_socket = socket.socket()
server_socket.bind(host, port)
server_socket.listen(1)
conn, address = server_socket.accept()
data = conn.recv(1024).decode()
conn.send(reply.encode())
conn.close()
```

Pourquoi ai-je inversé ces deux commandes entre le client et le serveur ?

En résumé, le flow en TCP



Les exceptions

- `TimeoutError`
 - Comme son nom l'indique
- `ConnectionRefusedError`
 - Comme son nom l'indique
- `socket.gaierror`
 - Nom de l'hôte invalide
- `ConnectionResetError`
 - Connexion interrompue abruptement (sans close → RST PACKET)
- `BrokenPipeError`
 - Rupture de connexion avec le serveur (perte de réseau, ^C, ...) → signal

Les sockets : tips

- `host = socket.gethostname()`
 - Récupérer le nom de la machine
 - Plutôt côté serveur
- `udpSvr = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)`
 - Ouvrir une socket en UDP
- `tcpSvr = socket.socket()` →
`tcpSvr = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`
 - Ouvrir une socket en TCP

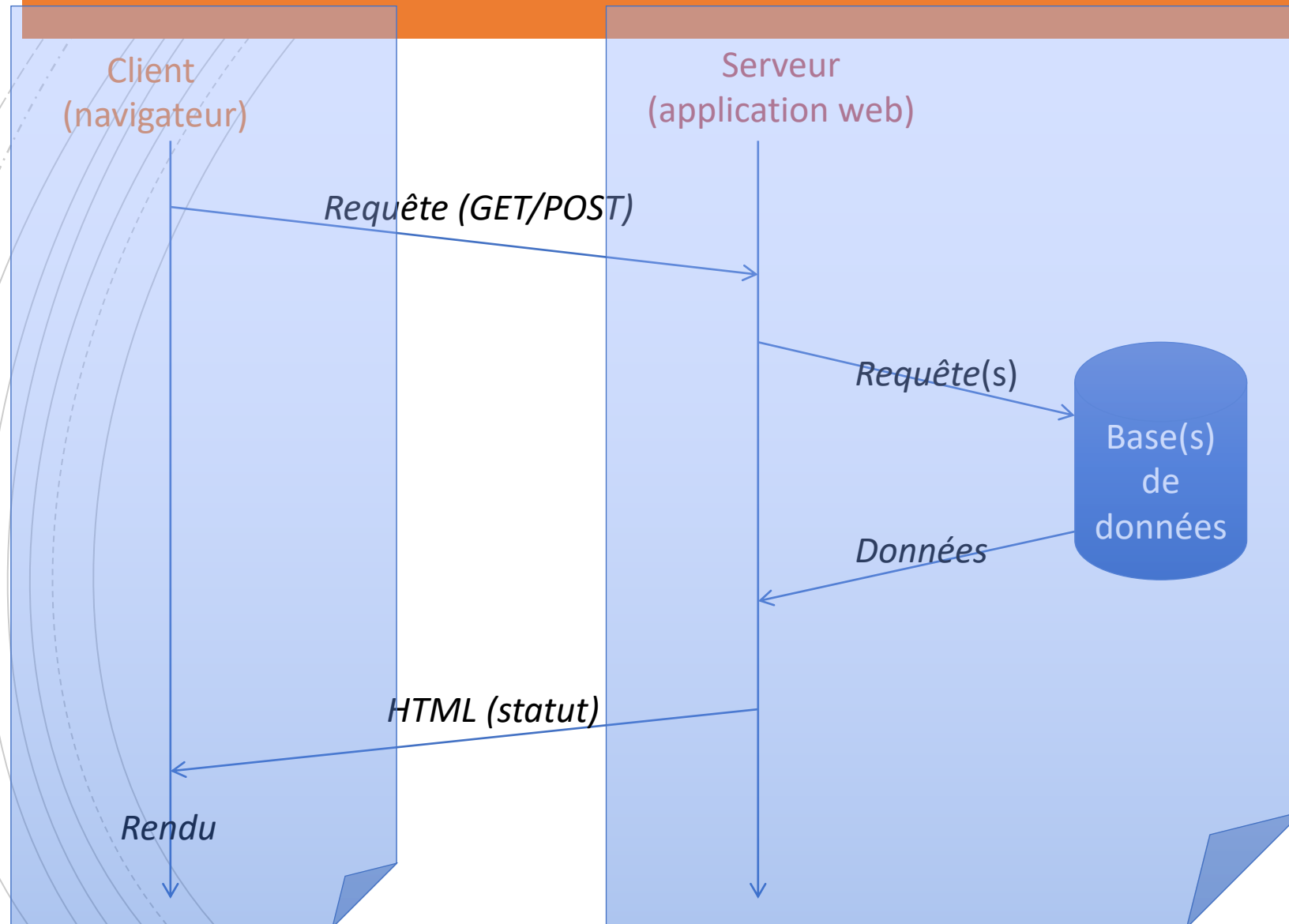
Créer un système client-serveur de chat

- Un client, un serveur
- De manière synchrone
 - Le client envoie un message et le serveur répond
 - Tester votre code avec votre voisin
- De manière asynchrone
 - Le client et le serveur peuvent envoyer des messages quand il le souhaite
 - En utilisant une mécanisme de Thread
 - Tester votre code avec votre voisin
- Si vous en avez la force, créer un serveur qui permet la discussion entre plusieurs clients
 - Les clients discutent entre eux
 - Le serveur renvoie les messages aux autres clients en spécifiant l'IP de la machine

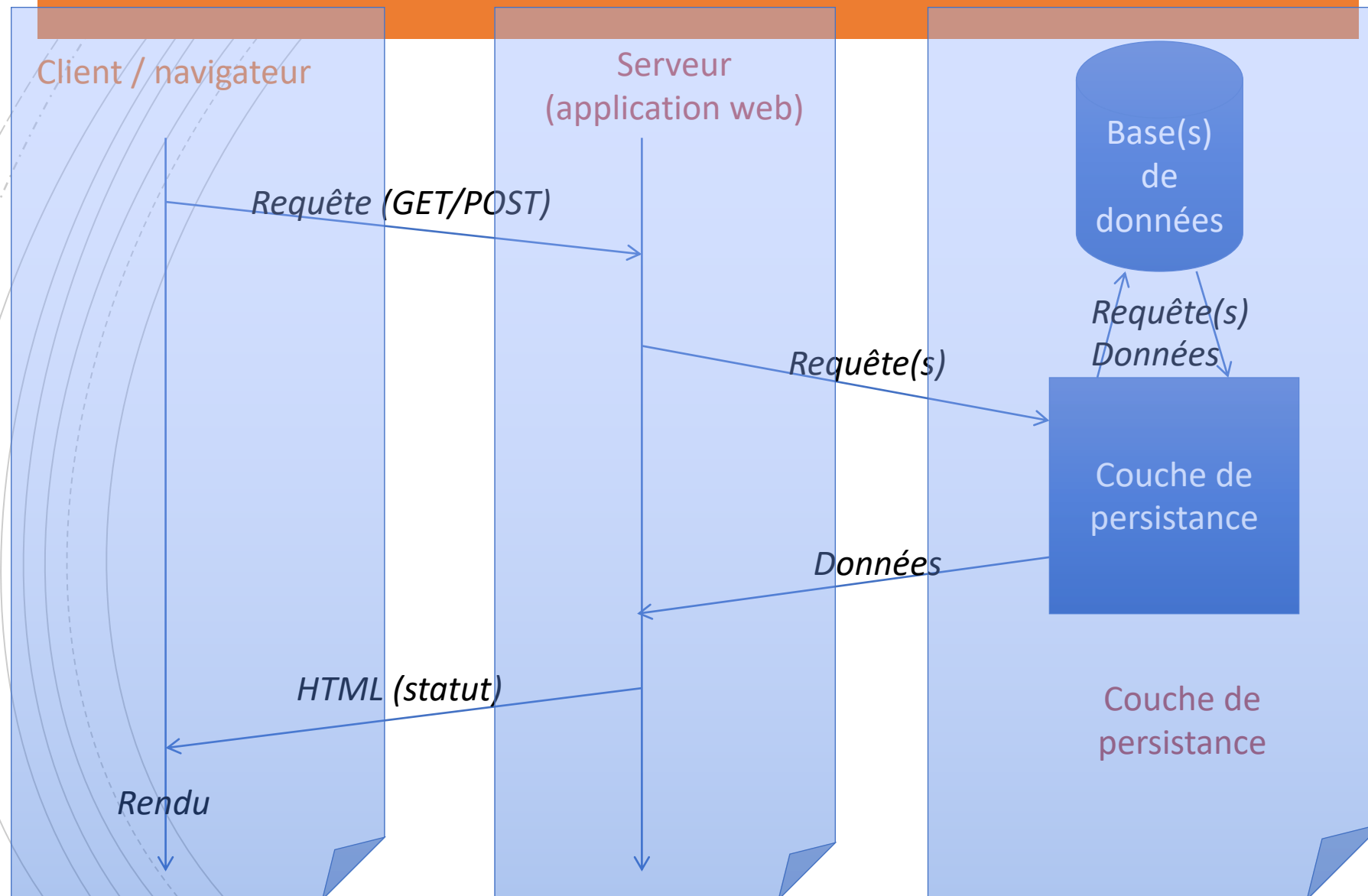
Architecture n-tiers

Une parenthèse

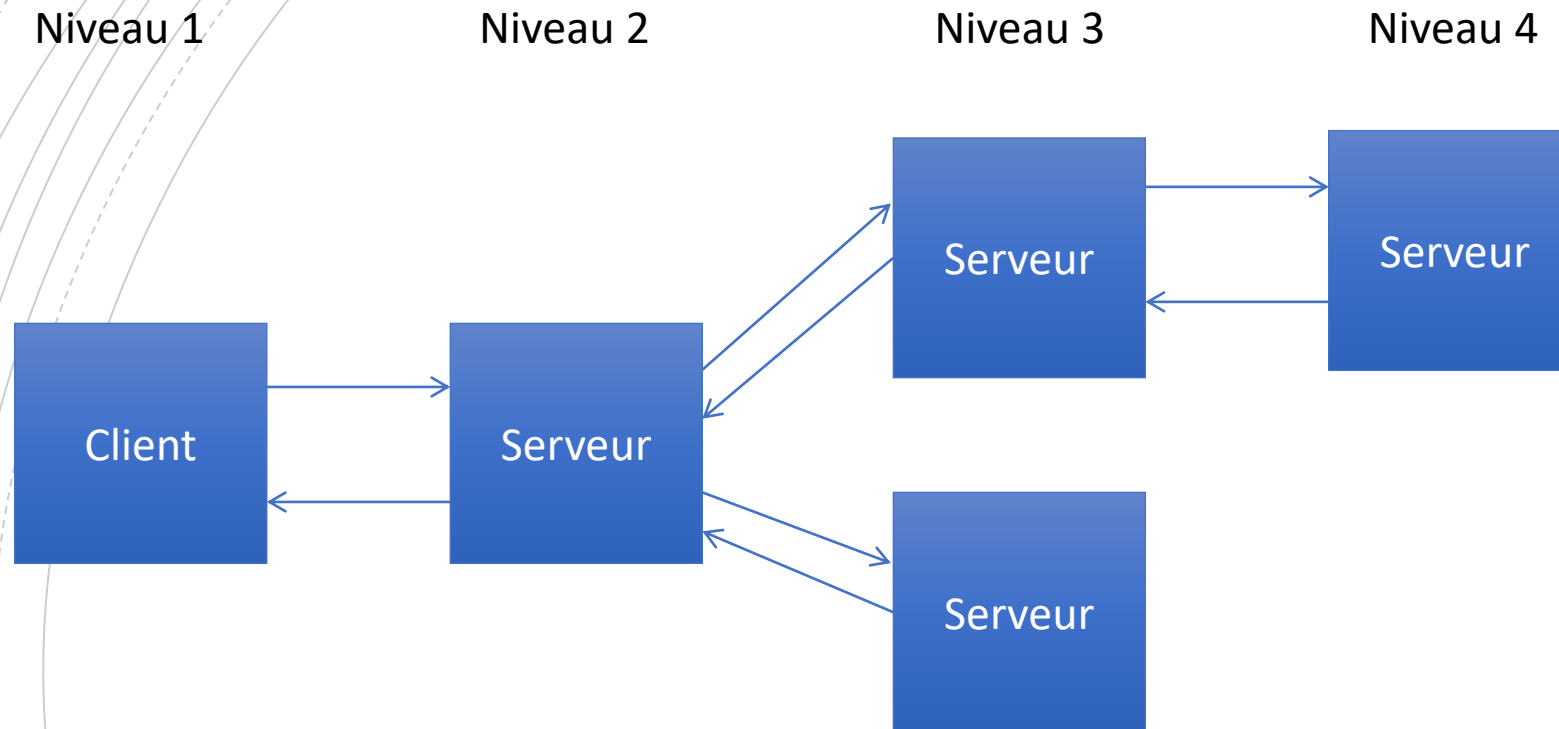
Architecture client-serveur (2 tier)



Architecture client-serveur (3-tier)

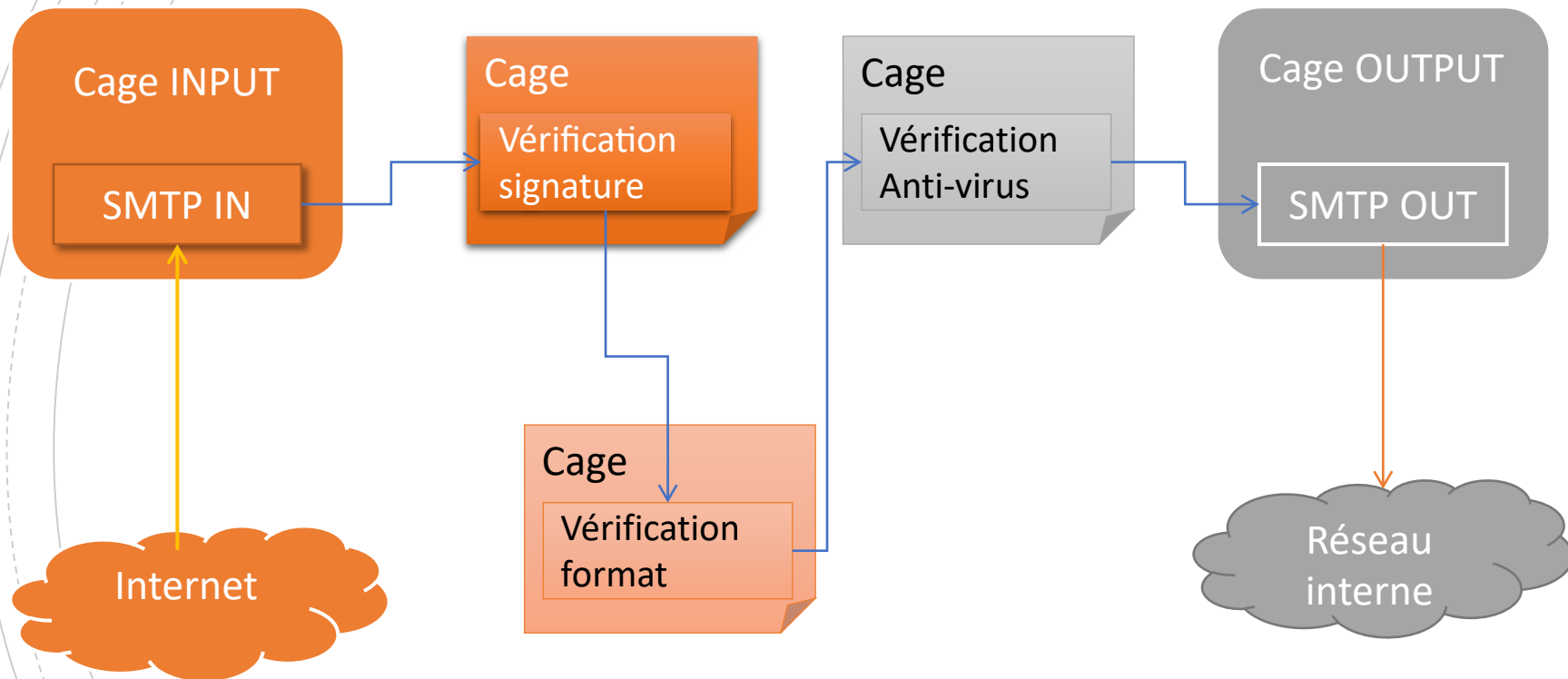


Architecture client-serveur (n-tier)



Attention à ne pas aboutir à des architectures trop complexes

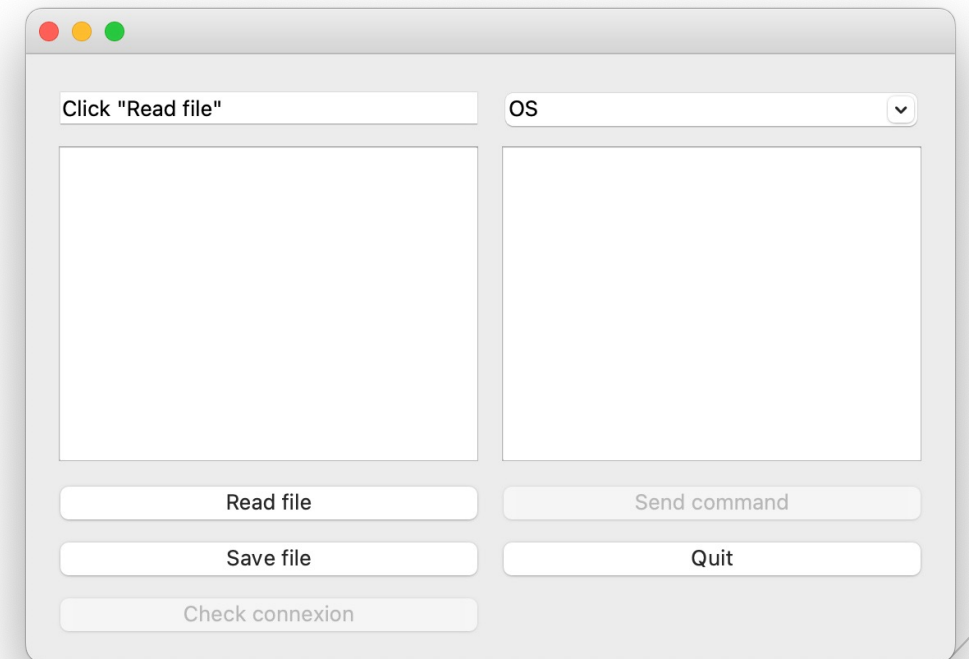
Exemple d'architecture n-tier en incluant des aspects sécurités



La programmation événementielle

(Graphic) User Interface

- User interface = UI
 - Par exemple, un menu texte
- Graphic User Interface = GUI
 - Par exemple, une belle interface graphique



Événement

Un objet

Propriétés
Méthodes
Événement

Objet : O Événement : E Traitement : T

Si l'événement E se produit sur l'objet O alors le traitement T est exécuté.

Exemple : un bouton (o) dans une interface graphique

- Attributs : une image, un label, une couleur de fond, une police de caractères
- Méthodes : se dessiner
- Événement : réagir quand on clique dessus
- Traitement : action effectuée lors du clic sur le bouton

Principe de développement

- Conception de l'interface
 - A la main → bof
 - A l'aide d'outil de placement et de génération de code (!)
- Mise en place des contrôles
 - Les traitements / les actions
- Ecriture du code
 - Le code du traitement
- Tests et exécution

Principe de développement dans ce cours

- Conception de l'interface
 - A la main – je sais c'est compliqué mais je vais (essayer) de vous aider
- Mise en place des contrôles et écriture du code
 - Les traitements / les actions
 - Le code du traitement
- Tests et exécution

Et en Python ?

- Librairie native : Tkinter (TCL/TK)
- Librairie PyQt sur la bibliothèque QT – licence GPL et commerciale
 - QT Designer = outil de génération de code d'interface et de code en Python
- PyQt :
 - **Mon avis :**
 - J'ai déjà créer des interfaces QT (C++)
 - Je préfère PyQt à Tkinter = j'ai trouvé ça plus beau
 - Utiliser par DropBox par exemple
 - Après faites vous votre propre idée
 - Même si dans ce cours, nous utiliserons PyQt5

<https://doc.qt.io/qt-5>

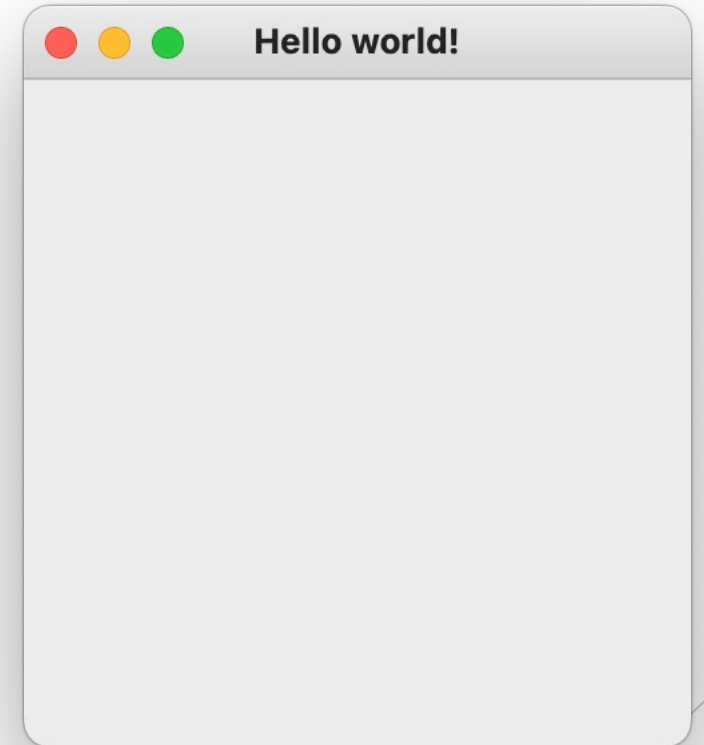
<https://pypi.org/project/PyQt5/>

Une première application PyQt

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget

app = QApplication(sys.argv)
root = QWidget()
root.resize(250, 250)
root.setWindowTitle("Hello world!")
root.show()

if __name__ == '__main__':
    sys.exit(app.exec_())
```



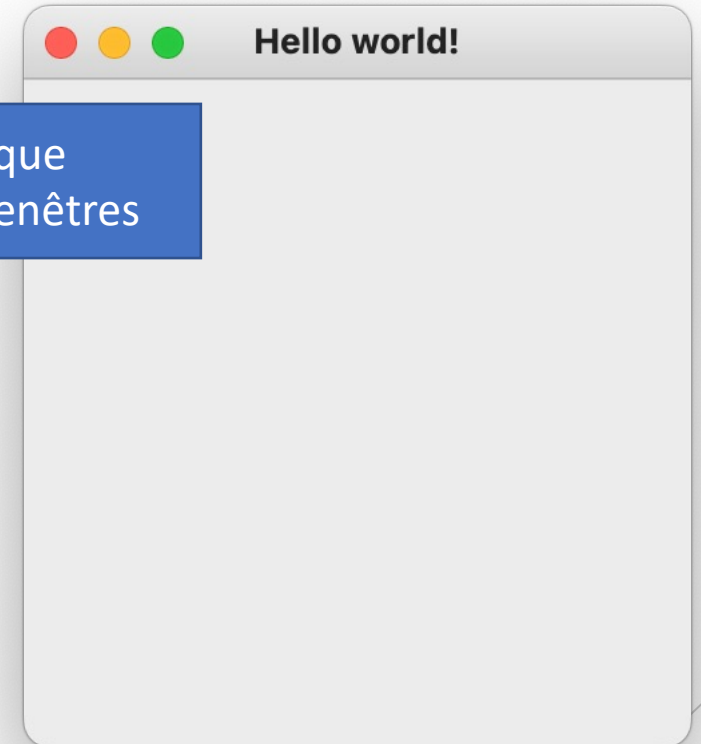
Une première application PyQt

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget

app = QApplication(sys.argv)
root = QWidget()
root.resize(250, 250)
root.setWindowTitle('Hello world!')
root.show()

if __name__ == '__main__':
    sys.exit(app.exec_())
```

Gère l'initialisation d'une application graphique (QWidget) et ce peu importe le nombre de fenêtres



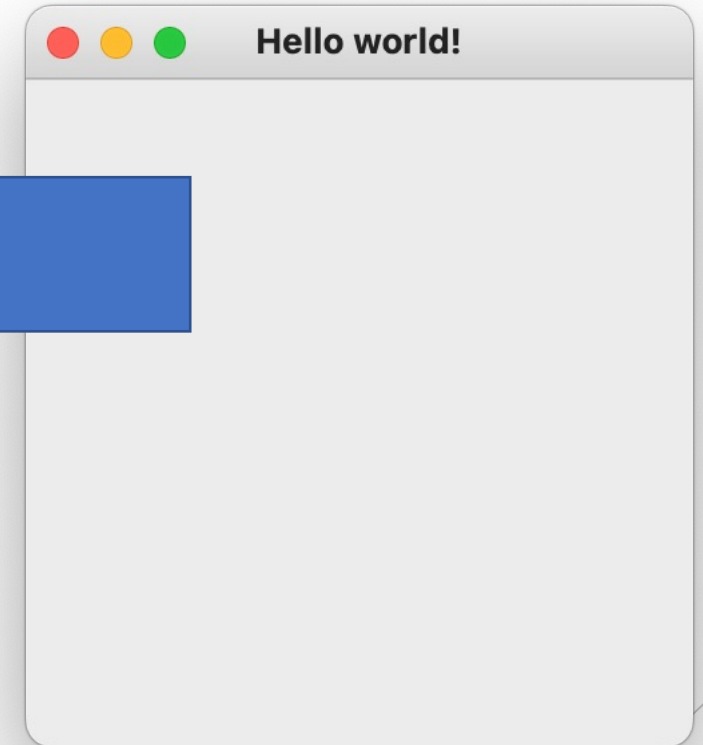
Une première application PyQt

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget

app = QApplication(sys.argv)
root = QWidget()
root.resize(250, 250)
root.setWindowTitle('Hello world!')
root.show()

if __name__ == '__main__':
    sys.exit(app.exec_())
```

Support de la partie graphique



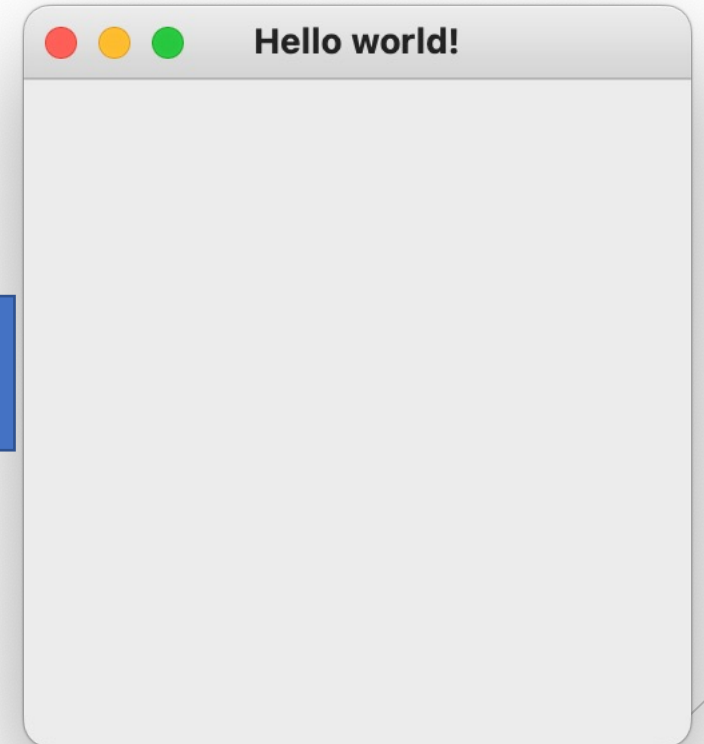
Une première application PyQt

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget

app = QApplication(sys.argv)
root = QWidget()
root.resize(250, 250)
root.setWindowTitle("Hello world!")
root.show()

if __name__ == '__main__':
    sys.exit(app.exec_())
```

Je paramètre ma fenêtre (QWidget)



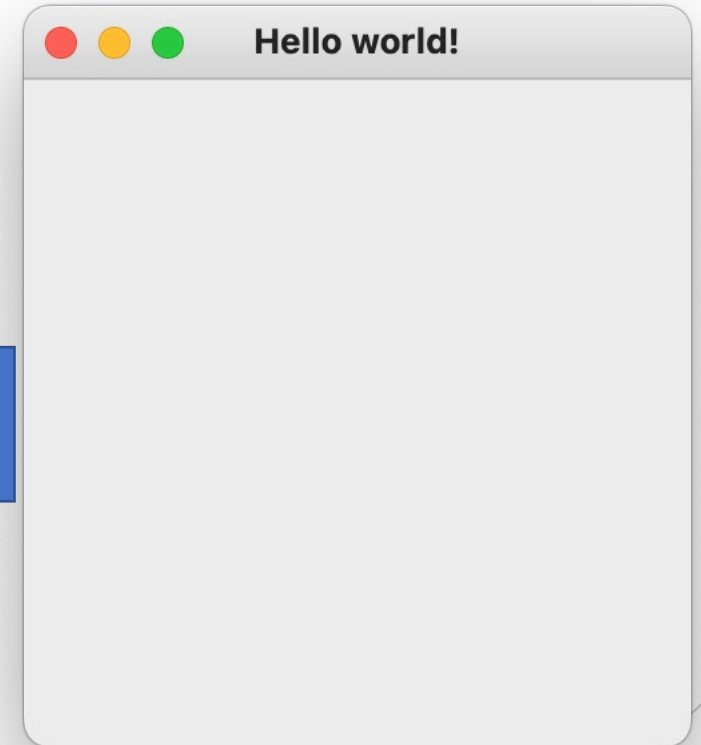
Une première application PyQt

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget

app = QApplication(sys.argv)
root = QWidget()
root.resize(250, 250)
root.setWindowTitle("Hello world!")
root.show()

if __name__ == '__main__':
    sys.exit(app.exec_())
```

Affichage de ma fenêtre



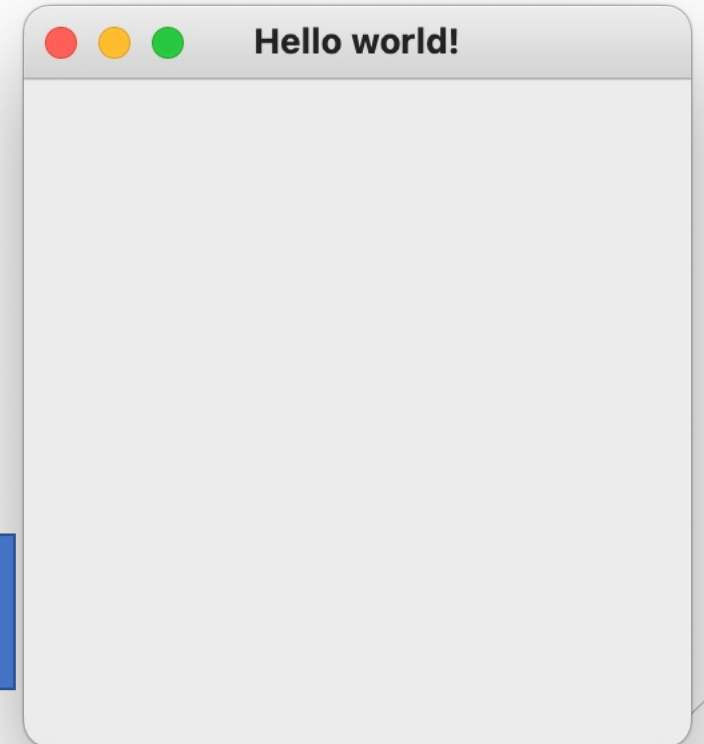
Une première application PyQt

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget

app = QApplication(sys.argv)
root = QWidget()
root.resize(250, 250)
root.setWindowTitle("Hello world!")
root.show()

if __name__ == '__main__':
    sys.exit(app.exec_())
```

Exécution de l'application



Une première application, en résumé

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget

app = QApplication(sys.argv)
root = QWidget()
root.resize(250, 250)
root.setWindowTitle("Hello world!")
root.show()

if __name__ == '__main__':
    sys.exit(app.exec_())
```

The diagram consists of blue arrows pointing from specific lines of code to their corresponding descriptions on the right. The arrows originate from the following code lines: `QApplication(sys.argv)`, `QWidget()`, `root.resize(250, 250)`, `root.setWindowTitle("Hello world!")`, `root.show()`, and `sys.exit(app.exec_())`. These arrows point to the following descriptions: "Une application", "Qwidget : reçoit tous les événements et Contient les autres composants", "La taille", "Un titre", "Affiche le widget (et les composants internes au widget)", and "Lancement de l'interface".

- Une application
- Qwidget : reçoit tous les événements et Contient les autres composants
- La taille
- Un titre
- Affiche le widget (et les composants internes au widget)
- Lancement de l'interface

Un layout

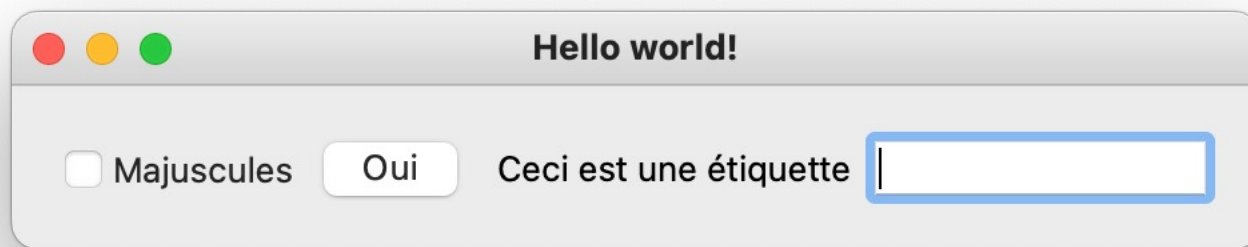
- Disposition des différents composants au sein du widget
- Plusieurs layouts possibles :
 - QHBoxLayout - Linear horizontal layout
 - QVBoxLayout - Linear vertical layout
 - QGridLayout - In indexable grid XxY
 - QStackedLayout - Stacked (z) in front of one another
- <https://www.pythonguis.com/tutorials/pyqt-layouts/>



A QVBoxLayout, filled from top to bottom



A QHBoxLayout, filled from left to right.



0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

A QGridLayout showing the grid positions for each location

			0,3
	1,1		
		2,2	
3,0			

A QGridLayout with unfilled slots

Hello world!

☐ Majuscules

Oui

Ceci est une étiquette

Hello world!

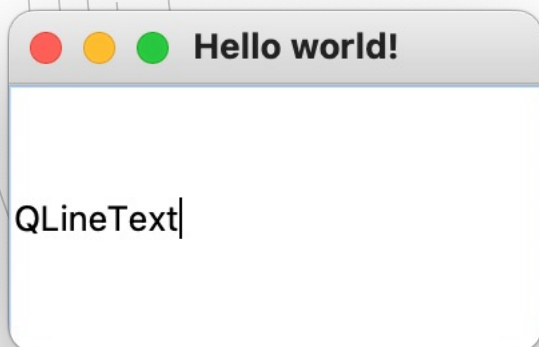
☐ Majuscules

Oui

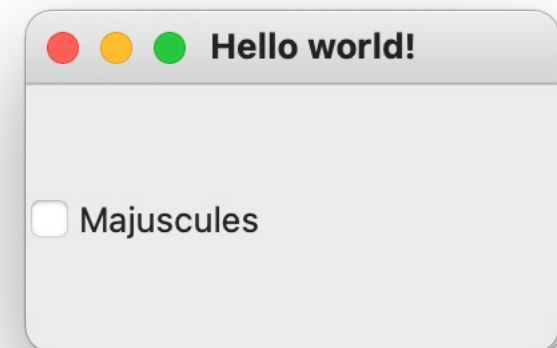
Ceci est une étiquette



QStackedLayout — in use only the uppermost widget is visible, which is by default the first widget added to the layout.



QStackedLayout, with the 2nd (1) widget selected and brought to the front.



Le layout QGridLayout

- Vous devez choisir le layout qui vous convient

```
grid = QGridLayout()
```

- Ajout au QWidget

```
root.setLayout(grid)
```

- Ajout des composants dans le grid layout

```
grid.addWidget(composant1, 0, 0) # composant, ligne, colonne  
grid.addWidget(composant2, 1, 0) # composant, ligne, colonne  
grid.addWidget(composant3, 0, 1) # composant, ligne, colonne  
grid.addWidget(composant4, 1, 1) # composant, ligne, colonne
```

- Composants :

```
QCheckBox  
QPushButton  
QLineEdit  
...
```

```
check = QCheckBox("Majuscules")  
btn = QPushButton("Bonjour")  
text = QLineEdit("")
```

Le layout

- Vous devez choisir le layout

```
grid = QGridLayout
```

- Ajout au Qwidget

```
root.setLayout(grid)
```

- Ajout des composants dans le layout

```
grid.addWidget(component, ligne, colonne)
```

```
grid.addWidget(component, ligne, colonne)
```

```
grid.addWidget(component, ligne, colonne)
```

```
grid.addWidget(component, ligne, colonne)
```

- Composants :

```
QCheckBox
```

```
QPushButton
```

```
QLineEdit
```

...



```
osant, ligne, colonne
```

```
osant, ligne, colonne
```

```
osant, ligne, colonne
```

```
osant, ligne, colonne
```

```
check = QCheckBox("Majuscules")
```

```
btn = QPushButton("Bonjour")
```

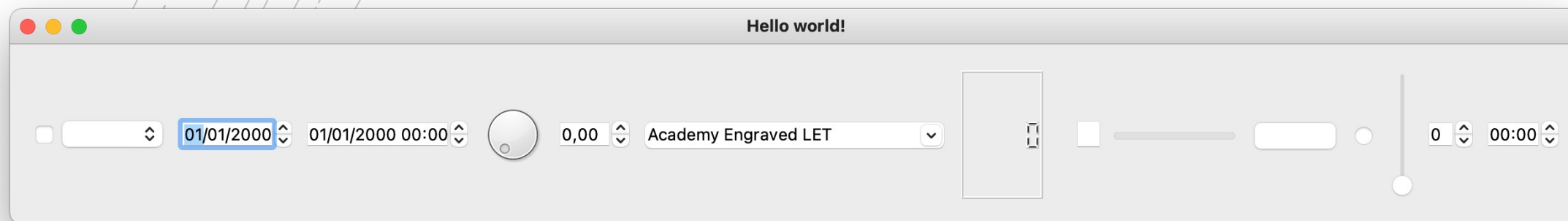
```
text = QLineEdit("")
```

Composants graphiques usuels avec QT5

- Case à cocher - [QCheckBox](#)
- Etiquette - [QLabel](#)
- Champ de texte – [QLineEdit](#)
- Bouton - [QPushButton](#)
- Case à sélectionner - [QRadioButton](#)
- Liste de choix - [QComboBox](#)
- Choix numérique- [QSpinBox](#)



Composants avec QT5



QCheckBox,
QComboBox,
QDateEdit,
QDateTimeEdit,
QDial,
QDoubleSpinBox,
QFontComboBox,
QLCDNumber,

QLabel,
QLineEdit,
QProgressBar,
QPushButton,
QRadioButton,
QSlider,
QSpinBox,
QTimeEdit

Les actions

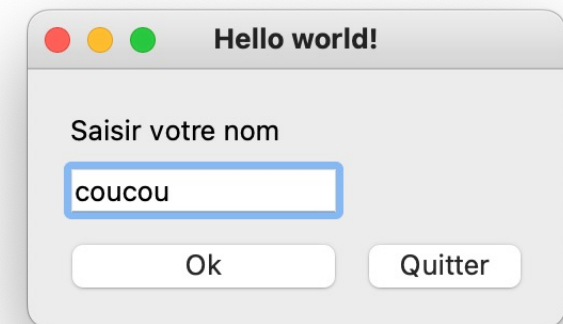
- Objet : O
- Événement : E
- Traitement : T
- Traitement :
 - Associer le composant à une action `clicked.connect(action)`
 - `clicked.connect` est une méthode permettant l'appel de la fonction action

Traitements associés aux boutons

```
def actionOk():  
    pass  
  
def actionQuitter():  
    QApplication.exit(0)
```

Association des traitements aux boutons

```
ok = QPushButton("Ok")  
quit = QPushButton("Quitter")  
  
ok.clicked.connect(actionOk)  
quit.clicked.connect(actionQuitter)
```



Les actions

- N'est pas limité aux boutons : 3 exemples

Syntax : `combo_box.activated.connect(self.do_something)`

Argument : It takes method name which should get called as argument

Return : None

QComboBox

Syntax : `line_edit.returnPressed.connect(self.do_something)`

Argument : adding action to the line edit when enter key is pressed

Return : None

QLineEdit

Syntax : `button_action.triggered.connect(self.do_something)`

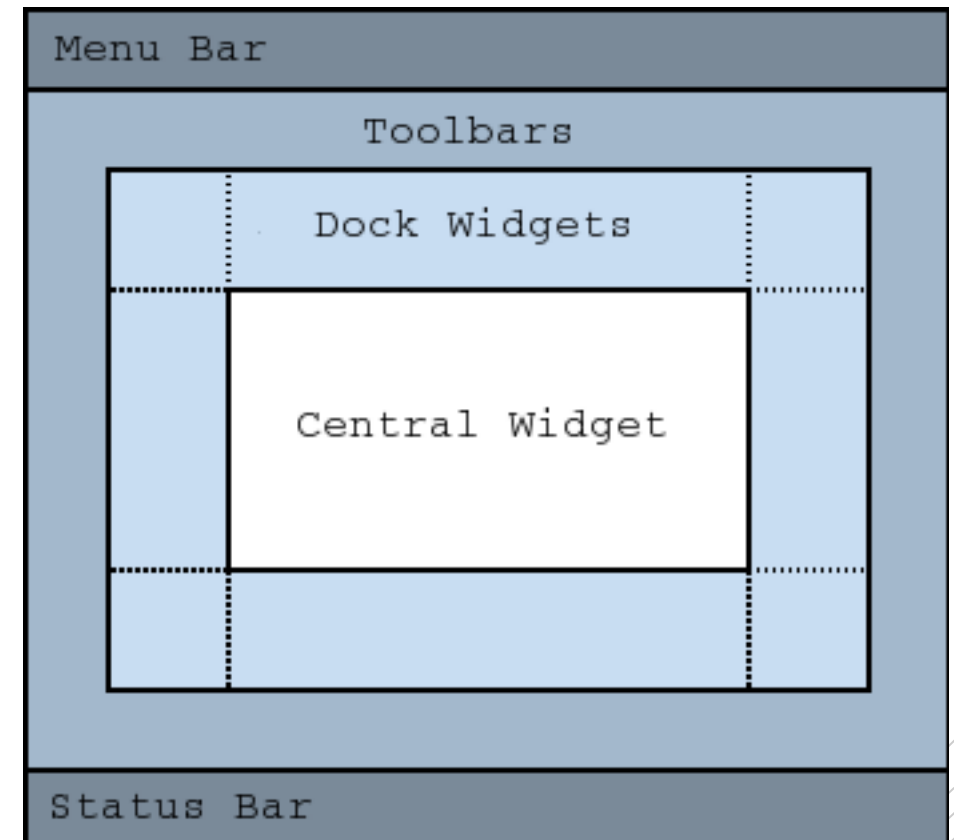
Argument : adding action on a menu

Return : None

QAction

Comment développer une application

- Une **classe** qui hérite de QMainWindow
- Ajout d'un constructeur avec :
 - Un widget
 - Votre choix de layout (ajouté au widget)
 - Vos différents composants
- Vos méthodes de gestion d'action
- Un main



Comment développer une application

- Une classe qui hérite de QMainWindow
- Ajout d'un constructeur avec :
 - Un widget
 - Votre choix de layout (ajouté au widget)
 - Vos différents composants
- Vos méthodes de gestion d'action
- Un main

```
class MainWindow(QMainWindow) :
```

Comment développer une application

- Une classe qui hérite de QMainWindow
- Ajout d'un constructeur avec :
 - Un widget
 - Votre choix de layout (ajouté au widget)
 - Vos différents composants
- Vos méthodes de gestion d'action
- Un main

```
def __init__(self):  
    super().__init__()   
  
    widget = QWidget()  
  
    self.setCentralWidget(widget)  
  
    grid = QGridLayout()  
    widget.setLayout(grid)
```

Comment développer une application

- Une classe qui hérite de QMainWindow
- Ajout d'un constructeur avec :
 - Un widget
 - Votre choix de layout (ajouté au widget)
 - Vos différents composants
- Vos méthodes de gestion d'action
- Un main

```
ok = QPushButton("Ok")
quit = QPushButton("Quitter")

ok.clicked.connect(self.actionOk)
quit.clicked.connect(self.actionQuitter
)
```

Comment développer une application

- Une classe qui hérite de QMainWindow
- Ajout d'un constructeur avec :
 - Un widget
 - Votre choix de layout (ajouté au widget)
 - Vos différents composants
- Vos méthodes de gestion d'action
- Un main

```
def _actionOk(self):  
    pass  
  
def _actionQuitter(self):  
    QApplication.exit(0)
```

Comment développer une application

- Une classe qui hérite de QMainWindow
- Ajout d'un constructeur avec :
 - Un widget
 - Votre choix de layout (ajouté au widget)
 - Vos différents composants
- Vos méthodes de gestion d'action
- Un main

```
if __name__ == '__main__':  
    app = QApplication(sys.argv)  
  
    window = MainWindow()  
    window.show()  
  
    app.exec()
```


Le code

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        widget = QWidget()
        self.setCentralWidget(widget)

        grid = QGridLayout()
        widget.setLayout(grid)

        lab = QLabel("Saisir votre nom")
        text = QLineEdit("")
        ok = QPushButton("Ok")
        quit = QPushButton("Quitter")

        ok.clicked.connect(self._actionOk)
        quit.clicked.connect(self._actionQuitter)

        self.setWindowTitle("Une première fenêtre")

    def _actionOk(self):
        pass

    def _actionQuitter(self):
        QApplication.exit(0)
```

```
if __name__ == '__main__':
    app = QApplication(sys.argv)

    window = MainWindow()
    window.show()

    app.exec()
```

Les menus

- Sur un widget QMainWindow

Une belle SAÉ

Créons notre propre système de surveillance des équipements

Dans le cadre de la SAÉ 2.02 | Développer des applications communicantes

Une application client-serveur

- Voir cahier des charges
- Une interface graphique
- Surveillance d'un certain nombre de machines
 - Synchrone ou asynchrone (à définir)
- Une liste d'adresses IP
 - Connexion aux machines
 - Envoi de requête de surveillance : OS, RAM, ...
 - Envoi de requêtes plus complexe : Powershell:get-process, DOS:dir, Linux:ls -al, python --version
- Le retour des commandes pour afficher les résultats