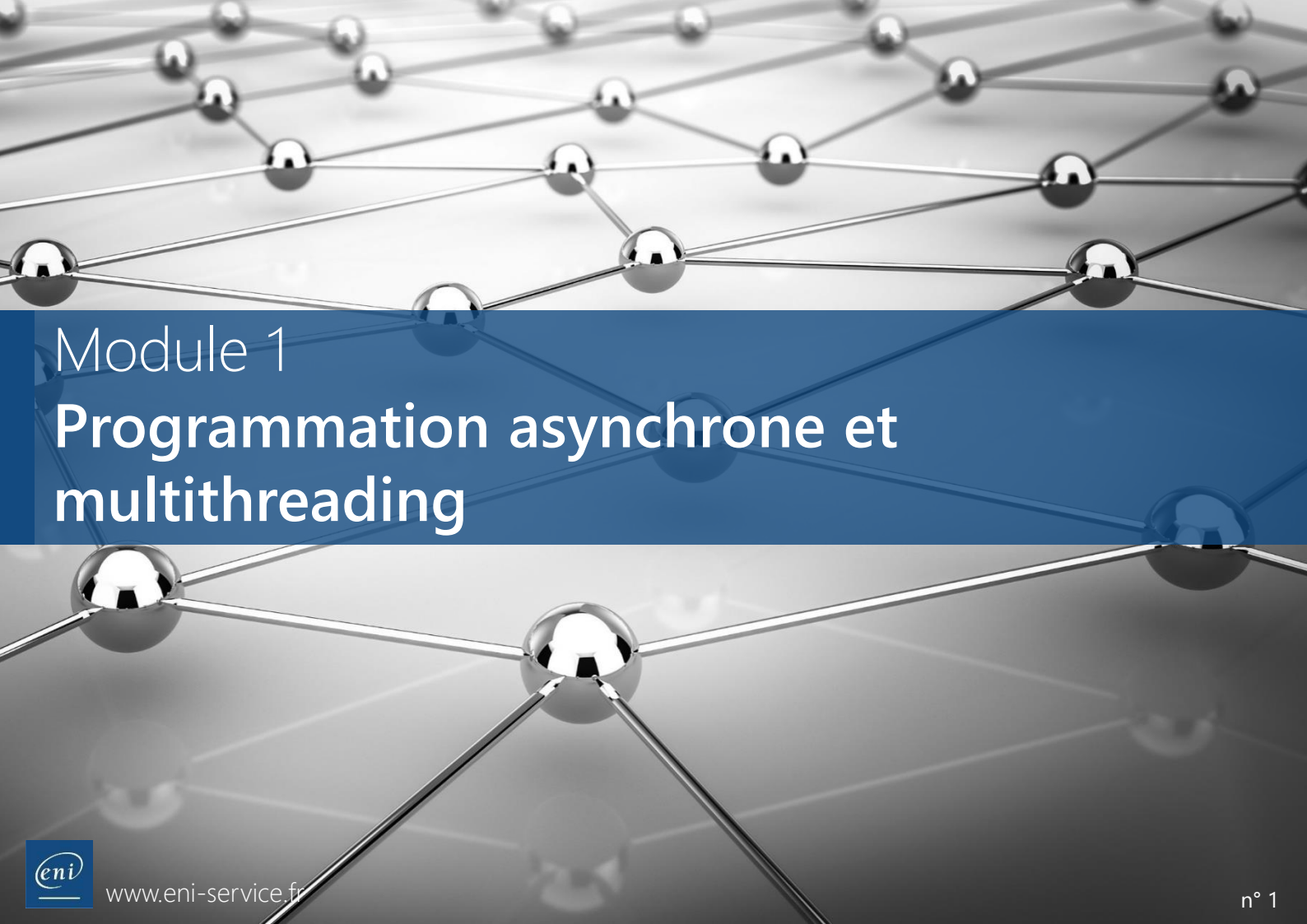

Module 1

Programmation asynchrone et multithreading

Contenu du module

- Utilité de la programmation asynchrone
  - Notion de thread et de sous-processus
- Les threads
  - Modèle de programmation
  - Définition d'une tâche asynchrone et de son traitement
- Le module `asyncio`
  - Les mot-clés `async` et `await`
  - Modèle de programmation
  - Fonction de l'API
- Les tâches cycliques avec « Event Scheduler »
  - Le module `sched` de Python 3
  - Le module `schedule`

Utilité de la programmation asynchrone

- La programmation asynchrone est une façon de concevoir des programmes qui s'exécutent de façon **concurrente** :
  - Programme qui évite au maximum de passer du temps à *attendre sans rien faire*.
  - Programme qui s'arrange pour *s'occuper autant que possible pendant qu'il attend*.
- Attention à ne pas confondre programmation concurrente et programmation parallèle :
  - Un programme qui s'exécute de façon **concurrente**, c'est un programme qui, à un instant T, est en train de réaliser **plusieurs tâches en même temps**,
  - Un programme qui s'exécute de façon **parallèle**, c'est **UNE** tâche qui a été découpée en plusieurs morceaux pour être réalisée par **PLUSIEURS** acteurs en même temps, la plupart du temps pour qu'elle se termine plus vite.

Processus, threads et sous-processus

- Un processus est l'exécution d'un ensemble d'instructions par l'utilisation de ressources physiques
  - Mémoire vive : il y stocke son environnement d'exécution
  - Processeur
- La création, gestion et destruction d'un processus sont gérées par le système d'exploitation, pas directement par Python.
  - Python propose des outils de haut niveau pour créer, gérer et détruire ces processus qui restent relativement simples, mais il utilise en réalité le système d'exploitation qui fait l'essentiel du travail.
- Le processus est géré par un ordonnanceur dépendant du système d'exploitation.
  - Il met à disposition les ressources (mémoire, temps processeurs...)
  - Veille à ce que chaque processus accède à ces ressources de manière équitable
  - S'il y a plusieurs processeurs, les processus sont également distribués entre eux de manière équitable.

Processus, threads et sous-processus

- Chaque processus contient par défaut un fil d'exécution.
  - Mais il est possible d'en créer plusieurs.
- Un fil d'exécution permet d'exécuter des instructions. Lorsque l'on a plusieurs fils d'exécution, tous utilisent le même contexte d'exécution, celui du processus auquel ils appartiennent.
  - Cependant, ils possèdent leur propre pile d'appel, liée à l'exécution de leurs instructions.
- Entre fils d'exécution, que certains appellent processus légers, le parallélisme n'est pas réel en tant que tel.
  - Par contre, il est possible de concevoir une délégation de travail vers des fils d'exécution secondaires pour fluidifier le fil d'exécution principal et de gérer au mieux les temps d'attente éventuels.
  - Eux aussi sont gérés par les systèmes d'exploitation.
- Le fil d'exécution se nomme ***thread*** en anglais et le module Python `threading` contient tous les outils de haut niveau pour le gérer.

Le fonctionnement asynchrone en python

- L'interpréteur Python implémente ce que l'on appelle le GIL
  - *Global Interpreter Lock*
- Un code en Python ne peut être exécuté par l'interpréteur que si celui-ci est possesseur du GIL, et un seul thread peut posséder le GIL à un instant donné.
- Si deux threads d'un même programme en Python sont exécutés sur deux cœurs de processeur distincts, le GIL les contraint à ne jamais pouvoir s'exécuter en parallèle (ce qui apporte de nombreuses garanties dans le code interne de l'interpréteur).
- Un seul thread peut être en état d'exécution à un moment donné, il peut être un goulot d'étranglement des performances dans le code lié au processeur et à plusieurs threads.

Intérêt du GIL

- Python utilise le comptage de références pour la gestion de la mémoire.
  - Les objets créés en Python ont une variable de comptage de références qui garde une trace du nombre de références qui pointent vers l'objet. Lorsque ce décompte atteint zéro, la mémoire occupée par l'objet est libérée.
  - Cette variable de comptage de référence a besoin d'une protection contre les conditions de concurrence où deux threads augmentent ou diminuent sa valeur simultanément.
  - Cette variable de comptage de référence peut être conservée en toute sécurité en ajoutant des verrous à toutes les structures de données qui sont partagées entre les threads afin qu'elles ne soient pas modifiées de manière incohérente.
  - L'ajout d'un verrou à chaque objet ou groupe d'objets signifie que plusieurs verrous existeront, ce qui peut provoquer un autre problème: les verrous mortels et une diminution des performances provoquée par l'acquisition et le déverrouillage répétés de verrous.
- Le GIL est un verrou unique sur l'interpréteur lui-même qui ajoute une règle selon laquelle l'exécution de tout *bytecode* Python nécessite l'acquisition du verrou d'interpréteur.
- Cela empêche les blocages (car il n'y a qu'un seul verrou) et n'introduit pas beaucoup de surcharge de performances. Mais cela rend tout programme Python lié au CPU simple thread.

Les threads

Avantages du multithreading

- Les programmes multithread peuvent s'exécuter plus rapidement sur les systèmes informatiques avec plusieurs processeurs, car ces threads peuvent être exécutés de manière vraiment simultanée.
- Un programme peut rester réactif à la saisie. Cela est vrai à la fois sur un seul et sur plusieurs processeurs
- Les threads d'un processus peuvent partager la mémoire des variables globales. Si une variable globale est modifiée dans un thread, cette modification est valide pour tous les threads. Un thread peut avoir des variables locales.
- La gestion des threads est plus simple que la gestion des processus pour un système d'exploitation.
  - C'est pourquoi ils sont parfois appelés processus légers.

Les threads en Python

- Python a longtemps proposé un module nommé `thread` qui proposait une API de bas niveau et qui permettait de créer des threads selon différentes solutions qui pouvaient correspondre aux capacités de Python en ce domaine.
- L'implémentation CPython ayant des pans entiers de son code incompatibles avec une création propre de threads, ce module a évolué et a finalement été déprécié avec la finalisation d'une API de haut niveau `threading` qui est à la fois plus simple à utiliser et plus proche des besoins, mais aussi plus fiable.
- La problématique n'est pas tellement la création et l'exécution en parallèle de threads, mais plutôt la gestion des ressources qu'ils partagent et de leurs communications.

Déclaration d'un thread

- Pour créer un thread, il faut :
  - Ecrire une classe qui hérite de la classe `Thread` du module `threading`
  - Définir un constructeur permettant de transmettre le nom du thread à la superclasse
  - Redéfinir la méthode `run()` contenant l'implémentation du traitement

```
class Worker(Thread):

    def __init__(self, name):
        Thread.__init__(self, name=name)

    def run(self):
        for i in range(5):
            print(f"{self.name} : Appel {i}, {time.strftime('%X')}")
            time.sleep(1)
```

Lancement d'un thread

- Pour démarrer un thread, il faut instancier sa classe et invoquer la méthode start() sur l'objet obtenu

```
if __name__ == "__main__":
    try:
        t1 = Worker("T1")
        t2 = Worker("T2")
        t1.start()
        t2.start()
        print("2 Threads démarrés")
    except:
        print("Error: unable to start threads")
```

```
T1 : Appel 0, 17:38:23
T2 : Appel 0, 17:38:23
2 Threads démarrés
T2 : Appel 1, 17:38:24
T1 : Appel 1, 17:38:24
T1 : Appel 2, 17:38:25
T2 : Appel 2, 17:38:25
T2 : Appel 3, 17:38:26
T1 : Appel 3, 17:38:26
T1 : Appel 4, 17:38:27
T2 : Appel 4, 17:38:27
```

Limites du module `threading`

- Le module threading permet de créer rapidement des threads pour déclencher des traitements parallèles simples
- Cependant, il devient plus difficile d'implémenter des traitements complexes multiples avec cette approche
- Ce module offre une approche bas niveau néanmoins nécessaire puisque qu'elle est la base de modules plus évolués tels que `asyncio`

Le module asyncio

Programmation asynchrone : les coroutines

- La programmation est basé sur le concept de *coroutine*.
- Une *coroutine* :
  - Est une tâche
  - Est déclarée comme une fonction avec le mot-clé `def`
  - L'ajout du mot-clé `async` la distingue d'une simple fonction
  - Elle ne peut être appelée directement contrairement à une fonction

```
async def say_after(delay, what):
    print(f"before {what} {time.strftime('%X')}")
    await asyncio.sleep(delay)
    print(f"after  {what} {time.strftime('%X')}")
```

Invocation d'une coroutine

- Lorsque l'on écrit du code, on écrit par défaut du code synchrone.
- La fonction `asyncio.run()` va permettre d'appeler du code asynchrone et d'attendre que toutes les instructions de ce dernier soient terminées avant de reprendre la suite des instructions synchrones.

```
asyncio.run(say_after(1, "Hello"))
```

- L'affichage produit sera alors le suivant :

```
before Hello 15:31:04
after  Hello 15:31:05
```

Le mot-clé `await`

- Le mot-clé `await` permet de bloquer l'exécution de la coroutine courante, en attendant la fin de l'appel, mais également permettre de faire en sorte que d'autres coroutines puissent être exécutées pendant ce temps.

- Il est impossible d'utiliser `await` en dehors d'une coroutine

- Dans l'exemple précédent, la fonction `asyncio.sleep(1)` est une fonctionnalité non bloquante.
  - Il s'agit d'une coroutine que l'on exécute et dont on attend qu'elle soit terminée par l'utilisation du mot-clé `await`.
  - On ne peut pas prévoir quelles autres coroutines seront appelées, ni dans quel ordre, mais ce que l'on sait d'une manière certaine, c'est que la coroutine ne reprendra qu'après que la coroutine `sleep()` ait terminé.

Exécution simultanée

- Lorsque l'on a un code qui a des parties bloquantes, nous avons vu qu'il faut identifier ces parties, les isoler en créant des coroutines utilisant des fonctionnalités non bloquantes.
- Ce que nous souhaitons faire est d'exécuter plusieurs coroutines en même temps pour faire en sorte qu'il y en ait toujours au moins une qui tourne.

```
async def print_message_long():
    await asyncio.sleep(2)
    print("Bienvenue en Python")

async def main():
    start_time = datetime.now()
    await print_message_long()
    await print_message_long()
    duree = datetime.now() - start_time
    print(duree)
```

Bien essayé mais…

```
Bienvenue en Python
Bienvenue en Python
0:00:04.014359
```

Les tâches - Principes

- Il est possible de paralléliser l'exécution des coroutines par la création de tâches asynchrones
- La création d'une tâche se fait grâce à la méthode `create_task()` du module `asyncio`
- Principe :
  - On crée une tâche pour chaque appel coroutine à exécuter (soit chaque appel de fonction asynchrone), puis on lance les tâches et on attend que chaque tâche soit terminée.
  - Les tâches sont toutes lancées en même temps par le premier `await`.

Les tâches asynchrones

```python
async def print_message_long():
    await asyncio.sleep(2)
    print("Bienvenue en Python")
```

```python
async def main():
    start_time = datetime.now()
    t1 = asyncio.create_task(print_message_long())
    t2 = asyncio.create_task(print_message_long())
    await t1
    await t2
    duree = datetime.now() - start_time
    print(duree)
```

```
Bienvenue en Python
Bienvenue en Python
0:00:02.035300
```

```python
async def main():
    start_time = datetime.now()
    await print_message_long()
    await print_message_long()
    duree = datetime.now() - start_time
    print(duree)
```

```
Bienvenue en Python
Bienvenue en Python
0:00:04.014359
```

Exécution concurrente de tâche asynchrones

- Une autre approche pour la gestion des tâches asynchrones concurrentes est d'utiliser la méthode `gather()` du module `asyncio`
- Ainsi l'exemple de code précédent pourrait être écrit comme ceci :

```
async def main():
    start_time = datetime.now()
    await asyncio.gather(
        print_message_long(), print_message_long()
    )
    duree = datetime.now() - start_time
    print(duree)
```

Les méthodes « attendables » (*awaitable*)

- Une méthode ou est un objet est « attendable » si elle peut être utilisée avec le mot-clé `await`
- Il existe trois principaux types d'objets :
  - Les coroutines : Traitement qui peut se suspendre
  - Les tâches : Permettant d'exécuter simultanément des coroutines
  - Les « futurs » : Objets de bas niveau qui représente le résultat éventuel d'une opération asynchrone.

Les résultats

- Une tâche terminée se voit affectée son résultat. Il est accessible via la méthode `result()` de la tâche
  - La méthode `done()` renvoi un booléen indiquant si la tâche est terminée ou non

```
async def get_long_message():
    await asyncio.sleep(2)
    return "Bienvenue en Python"


async def main():
    task = asyncio.create_task(get_long_message())
    print(task.done())
    await task
    print(task.done(), task.result())
```

```
False
True Bienvenue en Python
```

Les exceptions

- Une tâche se voit toujours associer une exception.
  - Si la coroutine se termine normalement, alors `task.exception()` renvoie None.
  - Si la coroutine a levé une exception, alors `task.exception()` renvoie cette exception.
- Toute exception se produisant dans une coroutine va se propager jusqu'au fil du programme principal
- Il est cependant possible de faire en sorte qu'une exception soit traitée comme un retour, de manière à ne pas impacter les autres tâches
  - Il faut pour cela utiliser la méthode `asyncio.gather()` pour créer les tâches
  - Le paramètre `return_exceptions=True` permettra de récupérer les exceptions comme un retour

Récupération des exceptions

- Exemple :

```python
async def get_long_message():
    await asyncio.sleep(2)
    return "Bienvenue en Python"


async def get_exception():
    await asyncio.sleep(1)
    raise Exception("Ceci est une erreur !")


async def main():
    task = asyncio.gather(
        get_long_message(),
        get_exception(),
        get_long_message(),
        return_exceptions=True
    )
    print(task.done())
    await task
    print(task.done(), task.result())
```

Résultat produit :

```
False
True ['Bienvenue en Python', Exception('Ceci est une erreur !'), 'Bienvenue en Python']
```

Les verrous

- Il est possible de verrouiller une ressource de manière à ce qu'une seule tâche puisse y accéder à la fois, en utilisant ce que l'on appelle un verrou
- En d'autres termes : un verrou asyncio peut être utilisé pour garantir un accès exclusif à une ressource partagée.

```
lock = asyncio.Lock()

# ...
async with lock:
    """
    Accès à l'état partagé...
    """
```

Ce qui équivaut à :

```
lock = asyncio.Lock()

# ...
await lock.acquire()
try:
    """
    Accès à l'état partagé...
    """
finally:
    lock.release()
```

Les sémaphores

- Supposons qu'une tâche ne peut être exécutée que 3 fois simultanément…
- On peut utiliser un sémaphore qui peut être vu comme un verrou multiple
- Un sémaphore gère un compteur interne
  - Il est décrémenté à chaque appel de `acquire()`
  - Incrémenté à chaque appel de `release()`
  - Le compteur ne peut jamais descendre en dessous de zéro
    - Lorsque `acquire()` trouve qu'il est à zéro, il bloque, attendant qu'une tâche appelle `release()`

- L'argument optionnel du constructeur donne la valeur initiale du compteur interne (1 par défaut).
  - Si la valeur donnée est inférieure à 0, une `ValueError` est levée.

Implémentation d'un sémaphore

- Avec `with` :

```
sem = asyncio.Semaphore(3)

# ...
async with sem:
    """
    Accès à l'état partagé...
    """
```

- Code équivalent :

```
sem = asyncio.Semaphore(3)

# ...
await sem.acquire()
try:
    """
    Accès à l'état partagé...
    """
finally:
    sem.release()
```

Le module sched

Le module `sched`

- Python possède un module nommé `sched` qui permet de programmer des événements dans le futur en exécutant des fonctions *callable* à un moment donné.
- Il fournit une API qui permet d'exécuter un événement :
  - A une heure spécifiée
  - Après un délai spécifié à partir de l'heure actuelle
- Le processus de planification d'événements via le module `sched` passe par les étapes suivantes :
  - Créez une instance de planificateur à l'aide du constructeur de la classe `scheduler()`
  - Créez des événements à l'aide de l'une des méthodes `enter()` ou `enterabs()` de l'instance du planificateur.
  - Appelez la méthode `run()` sur l'instance du planificateur pour démarrer l'exécution des événements.

Créer l'objet du planificateur

- Il s'agit de créer une instance de la classe `scheduler`
- `sched.scheduler(timefunc=time.monotonic, delayfunc=time.sleep)`
  - `timefunc` : Il prend comme référence d'entrée toute fonction qui ne prend aucun paramètre et renvoie le temps (toutes les unités). La valeur par défaut est définie sur la fonction `monotonic` du module de temps. Elle est basée sur l'unité des secondes.
  - `delayfunc` : Ce paramètre prend comme référence d'entrée une fonction qui reçoit un nombre en paramètre et retarde l'exécution de cette valeur.

- La classe `scheduler` est thread-safe et peut donc être utilisée dans un context multi-tâche.

Créer des événements avec un délai

- Les événements peuvent être programmés pour s'exécuter après un délai ou à une heure précise.
- Pour les programmer avec un délai, utilisez la méthode `enter()`, qui prend 4 arguments
- `scheduler.enter(delay, priority, action, argument=())`
  - `delay` : Un nombre représentant le retard, transmis à la fonction du constructeur de la classe `scheduler`
  - `priority` : Une valeur de priorité
  - `action` : La fonction à appeler
  - `argument` : Un tuple d'arguments pour la fonction
- Elle renvoie l'objet de type `Event` qui contient des informations sur l'heure à laquelle l'événement est programmé pour s'exécuter ainsi que des informations sur la priorité, la référence à la fonction d'action et ses arguments.

Implémentation

- Exemple :

```
import sched
from datetime import datetime
import time


def addition(a,b):
    print("Inside Addition : ", datetime.now())
    print("Time : ", time.monotonic())
    print("Result : ", a+b)


s = sched.scheduler()
print("Start Time : ", datetime.now(), "\n")
event1 = s.enter(3, 1, addition, argument=(10, 20))
print("Event Created : ", event1)
s.run()
print("End Time : ", datetime.now())
```

Créer des événements à une date précise

- La méthode `eventabs()` permet de planifier des événements à une date precise. Elle possède la même signature que `enter()` à l'exception du premier paramètre.
- `scheduler.eventabs(time, priority, action, argument=())`
  - `time` : Accepte des numériques et éxécute l'événement à ce moment. Les unites sont basés sur celle de la fonction passée en premier paramètre du constructeur de la classe.
  - `priority` : Une valeur de priorité
  - `action` : La fonction à appeler
  - `argument` : Un tuple d'arguments pour la fonction

Gestion de la file d'attente

- L'objet `scheduler` possède un attribut en lecture seule permettant d'avoir des informations sur les événements en attente.
  - queue
  - Il s'agit s'une file d'attente.
- Elle contient une liste des objets `Event` qui n'ont pas été exécutés et qui apparaissent dans l'ordre de leur exécution future
  - Chaque événement est affiché sous la forme d'un tuple nommé avec les champs suivants: heure, priorité, action, argument
- La méthode `empty()` renvoi `True` si la file est vide

Alternative : Le module `schedule`

- Le module `schedule` ne fait pas parti de la bibliothèque standard. Il implémente une interface de haut niveau pour la planification d'événements.
- Il s'installe avec `pip`

```python
import schedule
import time


def job():
    print("I'm working...")


schedule.every(10).minutes.do(job)
schedule.every().hour.do(job)
schedule.every().day.at("10:30").do(job)
schedule.every().monday.do(job)
schedule.every().wednesday.at("13:15").do(job)
schedule.every().minute.at(":17").do(job)

while True:
    schedule.run_pending()
    time.sleep(1)
```

Travaux Pratiques