Python's `threading`

CPEN333 - 2023 W1

University of British Columbia

Introduction

- We have already discussed the thread concept and compared it with the process concept.

- The `threading` module of the Python's standard library provides the API for creating and managing threads.
  - It is technically a higher-level threading interface on top of its lower level `_thread` module. Python documentation states: "The `threading` module provides an easier to use and higher-level threading API built on top of this module"

- In this set of slides, we focus on using the `threading` module to write multithreading programs.

Objectives

- Learn the `threading` module of Python's standard library
- Use the API to implement programs that can create new threads to multi-task, and join
- Understand Cpython's GIL's implications when using `threading`

Multi-tasking

- Here we are using the term multi-tasking to signify the ability to run two or more task concurrently or in parallel.

- We have seen how to do multi-tasking with multiple processes using the `multiprocessing` module.

- A second alternative is to use two or more threads, known as multithreading.
  - Python's `threading` module provide the needed API.

Thread Objects

- The `Thread` class can be used to represent an activity that is run in a separate thread of control.
- The class can be used to create a thread object.
- Once created, its activity is started by calling the `start`() method.
- The initial thread of control is the “main thread” object.
  - There is always a “main thread” object.

Example

- A simple example that uses the Thread class to create a thread.

```
import threading

def worker():
    """ A function to be used as target for threading"""
    print("Hello from a thread")

if __name__ == "__main__":
    thread = threading.Thread(target = worker)
    thread.start()
    thread.join()
```

output:

```
Hello from a thread
```

Thread class

- The `threading.Thread` class can be used to create thread objects.
  - https://docs.python.org/3/library/threading.html#threading.Thread

class `threading.`**Thread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, *, *daemon=None*) ¶

This constructor should always be called with keyword arguments. Arguments are:

group should be `None`; reserved for future extension when a `ThreadGroup` class is implemented.

target is the callable object to be invoked by the `run()` method. Defaults to `None`, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-*N*" where *N* is a small decimal number, or "Thread-*N* (target)" where "target" is `target.__name__` if the *target* argument is specified.

args is the argument tuple for the target invocation. Defaults to `()`.

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to `{}`.

If not `None`, *daemon* explicitly sets whether the thread is daemonic. If `None` (the default), the daemonic property is inherited from the current thread.

Thread class

- `class threading.Thread(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None)`

- `group` is reserved for future use.
- `target` is the callable object that is invoked by object's run() method (itself arranged to be invoked by start()).
- `name` is the thread name.
- `args` is the argument tuple for invoking the target.
- `kwargs` is a dictionary of keyword argument for invoking the target.
- `daemon` : if not none, it explicitly sets whether the thread is daemonic.
- The constructor should always be called with keyword arguments.

Example (three worker threads)

- A simple example that uses the Thread class to create three worker threads.

```
def worker():
    """ A function to be used as target for threading"""
    print("Hello from a thread")

if __name__ == "__main__":
    for _ in range(3): #Three threads
        thread = threading.Thread(target = worker)
        thread.start()

    #main thread: joining by default here
```

output:

```
Hello from a thread
Hello from a thread
Hello from a thread
```

Example (cont.)

- Same as previous example, without the `if`:

```
import threading

def worker():
    """ A function to be used as target for threading"""
    print("Hello from a thread")

for _ in range(3): #Three threads
    thread = threading.Thread(target = worker)
    thread.start()
```

output:

```
Hello from a thread
Hello from a thread
Hello from a thread
```

Thread name

- A thread has a name.
- The name can be passed to the constructor. It is an optional argument, and if not provided, python will assign one automatically of the form “Thread-N”.
- The name can be read or changed using the name property.

Example (thread name)

- we can use the `name` argument to assign identifying names to thread.

```
import threading

def worker():
    """ A function to be used as target for threading"""
    print(f"Hello from thread {threading.current_thread().name}")

if __name__ == "__main__":
    for t in range(2): #Three threads
        thread = threading.Thread(target = worker, name=f"worker{t}")
        thread.start()
        thread1 = threading.Thread(target = worker) # the default
        thread1.start()
```

Alternatively, we could use the `getName()` method.

output:

```
Hello from thread worker0
Hello from thread Thread-1
Hello from thread worker1
Hello from thread Thread-2
```

Passing parameters

- We can use the args argument to pass parameters to the threads.
- Example:

```
import threading

def worker(num):
    """thread worker function"""
    print(f"Worker thread {num} says hello")

if __name__ == "__main__":
    threads = []
    for i in range(3):
        t = threading.Thread(target=worker, args=(i,))
        threads.append(t)
        t.start()
```

output:

```
Worker thread 0 says hello
Worker thread 1 says hello
Worker thread 2 says hello
```

Alternatively using kwargs

```
import threading

def worker(num, str1):
    """thread worker function"""
    print(f"Worker thread {num} says {str1}")

if __name__ == "__main__":
    threads = []
    msg = ["hello", "hey", "aloha"]
    for i in range(3):
        t = threading.Thread(target=worker, kwargs={"num": i+1, "str1": msg[i]})
        threads.append(t)
        t.start()
```

output:

```
Worker thread 1 says hello
Worker thread 2 says hey
Worker thread 3 says aloha
```

Important methods and properties

- `start`(): start the thread activity
  - must be called at most once per thread object.
  - will raise a RuntimeError if called more than once on the same thread object

- `join`(timeout=None): wait until the thread terminates.
  - This blocks the calling thread until the thread whose join() method is called terminates (normally or through an unhandled exception) or until the optional timeout occurs (in seconds, or a fraction of). Other threads can call a thread's join() method

- `run`(): method representing the thread's activity. When start() is called, it arranges for the object's run() method to be invoked in a separate thread of control.
  - invokes the callable object passed to the constructor as the target argument.

Methods and properties (cont.)

- `is_alive`(): Return whether the thread is alive
- `name`: A string used for identification purposes only
- `daemon`: a Boolean value indicating whether this thread is daemon or not
- TID (Thread ID) or `native_id`: assigned by the OS (kernel). It is a non-negative integer (or *None* if the thread has not been started) that can be used to uniquely identify the thread system-wide.
- See https://docs.python.org/3/library/threading.html#threading.Thread

Other useful methods

- `threading.active_count()`
  - Returns the number of Thread objects currently alive.
- `threading.current_thread()`
  - Returns the current Thread object, corresponding to the caller's thread of control.
- `threading.get_ident()`
  - Return the 'thread identifier' of the current thread.
- `threading.main_thread()`
  - Return the main Thread object.
- `threading.enumerate()`
  - Return a list of all Thread objects currently active.
- See https://docs.python.org/3/library/threading.html for more functions.

Daemon Thread

- Behaving similarly to a daemon process, a daemon thread will terminate immediately when the main program exits.

- That is, it does not block the main program from exiting.
  - Daemon threads are useful for services when there may not be an easy way to interrupt the thread, or letting the thread terminate in the middle of its works would be fine (e.g. no loss or corrupted data).

- To flag a thread as a daemon thread, we can set it through the daemon property or the daemon constructor argument.

Example (daemonic and non-daemonic)

```
import threading
import time

def daemon():
    print("Daemon thread starting")
    time.sleep(1)
    print("Daemon thread exiting")

def non_daemon():
    print("Non-daemon thread starting")
    print("Non-daemon thread exiting")

if __name__ == "__main__":
    d = threading.Thread(target=daemon, daemon=True)
    t = threading.Thread(target=non_daemon)
    d.start()
    t.start()
```

output:

```
Daemon thread starting
Non-daemon thread starting
Non-daemon thread exiting
```

Example (cont.)

- We can use `join()`, to wait until a daemon thread is completed.

```
import threading
import time

def daemon():
    print("Daemon thread starting")
    time.sleep(1)
    print("Daemon thread exiting")

def non_daemon():
    print("Non-daemon thread starting")
    print("Non-daemon thread exiting")

if __name__ == "__main__":
    d = threading.Thread(target=daemon, daemon=True)
    t = threading.Thread(target=non_daemon)
    d.start()
    t.start()
    d.join()  #waiting for a daemon thread to be done
    t.join()  #good practice for readability, explicitly stating join
```

output:

```
Daemon thread starting
Non-daemon thread starting
Non-daemon thread exiting
Daemon thread exiting
```

Thread-local data

- We can use the class `threading.local` to represent a thread local data.
  - The value of a thread-local data is thread specific (hidden from view in separate threads).
  - https://docs.python.org/3/library/threading.html#thread-local-data

- We create an instance of `threading.local` and store attributes on it.

- Example:

```
mydata = threading.local()
mydata.x = 1
```

Example

```
import random
import threading

def worker(data):
    try:
        print(data.value) #attribute not present until set in the thread
    except AttributeError:
        print(f"Thread {threading.current_thread().name}: No value yet")
    data.value = random.randint(1, 100)
    print(f"Thread {threading.current_thread().name}:", data.value)

if __name__ == "__main__":
    local_data = threading.local()
    local_data.value = 2022
    print("main thread value:", local_data.value)

    for _ in range(2):
        t = threading.Thread(target=worker, args=(local_data,))
        t.start()
```

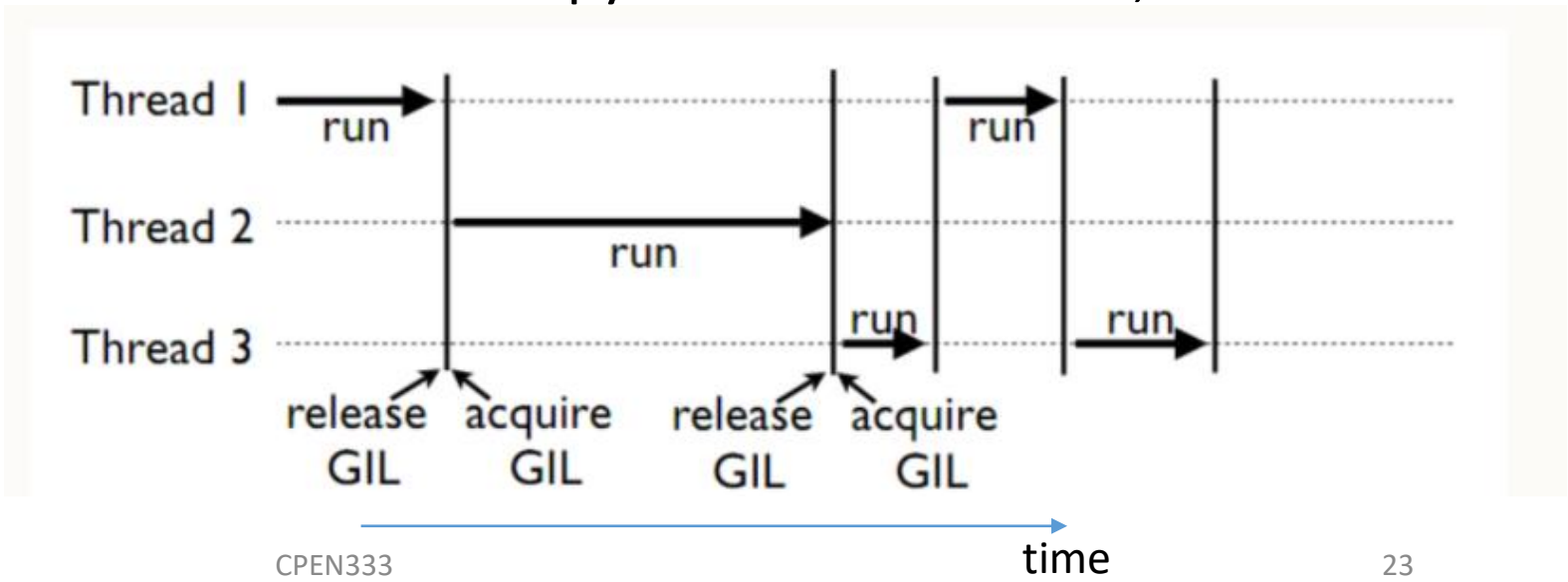
sample output:

```
main thread value: 2022
Thread Thread-1: No value yet
Thread Thread-1: 68
Thread Thread-2: No value yet
Thread Thread-2: 13
```

Global Interpreter lock

- CPython uses global interpreter lock (GIL) to ensure thread-safety, for example for built-in data structures (lists, dictionaries, …) by having atomic byte-code from manipulating them.
  - This is a benefit and the reason for the existence of GIL. It protects access to Python objects, preventing multiple threads from executing python bytecodes at once.
  - But due to GIL, only one thread can execute python code at a time, even in a multi-core machine.

We will talk more about locks in future lectures.



GIL and `threading`

- GIL limits where Python's `threading` is beneficial.

- When using `threading`, GIL can be performance-hindering when dealing with CPU-bound (spending time mainly doing computations) multitasking.
  - For CPU-bound multitasking, we can use the multiprocessing module or `concurrent.futures.ProcessPoolExecutor`.

- On the other hand, IO-bound tasks (e.g., spending much of their time waiting for IO and external events) are generally good candidates for Python's `threading`.
  - It will be an appropriate model when running multiple I/O-bound tasks simultaneously.

Related topics

- We postpone discussions on some related topics to near future, for example:
  - Communication between threads
  - Synchronization
  - Also some related python concepts: `ThreadPoolExecutor`, `concurrent.futures`, …

References

- Python documentation:
  - https://docs.python.org/3/library/threading.html

- *The Python Standard Library*, D. Hellmann