CIS 41B
Advanced Python Programming

Threads

De Anza College
Instructor: Clare Nguyen

Threads in a Process

- A process is an executable instance of a program.
- When we run the file example.py or when we open this Powerpoint file to read it, the executable that runs is a process.
- When a process runs, the OS gives it some memory space that it can use. This memory space is not shared with any other process.
- A process can use multiple threads when it has multiple tasks that can or should be done in parallel or at the same time.
- A thread is an independent sequence of execution within a process: one thread can run its task in parallel (or concurrently[1]) with another thread that runs its own different task. Each thread doesn't have to wait for the other to finish or to start.
- Threads within one process share the same memory space, which is the memory space allocated to the process.
- An example of using threads in Python is when an application with GUI needs to fetch data at the start of the application, but the data fetching takes time to finish. In that case we want a GUI thread that runs immediately to bring up the main window so the user doesn't have to wait, while a second thread starts the data fetching.

[1]For Python: it only *appears* as if the threads run concurrently.

Working with Threads

- Advantage:
  - The code may run faster since different tasks run concurrently.
- Disadvantage:
  - The improvement in speed is significant only if the code has tasks that take a long time to finish.
  - It takes extra coding to coordinate the threads so that they don't interfere with each other.
- When a process starts a thread:
  - The starting process is the main thread.
  - The newly created thread is called the child thread.
  - The main thread can create multiple child threads.
- When a thread runs:
  - It can run independently of other threads.
  - Or it can work together with another thread, such as when:
    - It uses the same resource (such as a file) as another thread.
    - It can only do one of its tasks only after another thread has completed some other task.

Coordinating Threads

In a program with threads, care must be taken to maintain each thread and to coordinate threads that work together.

1. End each child thread properly.
   - Depending on what the child thread does, the main thread may have to wait for the child thread to end, before the main thread itself can terminate.
   - Otherwise, when the main thread ends, any remaining child thread can hang and possibly keep resources locked up.
2. Coordinate shared resources.
   - When 2 threads use the same resource, then the resource must be accessible by only one thread at a time.
   - If both threads try to write to the same memory location at the same time, for example, they could end up overwriting each other's data.
3. Communicate between threads.

   Because each thread runs independently but they may still need to work together toward one common goal, there are various ways for threads to communicate their status with each other.

Threads in Python

- To work with threads:
  1. Import the thread module so we can use the Thread class:

```
import  threading
```

  2. To create a child thread to do some task, create a Thread object:

```
t = threading.Thread(target = aFunction, args = tuple_for_args )
```

     where: aFunction is the function we want the thread to run
     tuple_for_args is a tuple of input arguments for aFunction

  3. To run the thread: `t.start()`
  4. To wait for the child thread to end: `t.join()`

- If we have multiple different tasks that we want to run with threads, we can create multiple Thread objects, one for each task.
- If we have one task that needs to run multiple times, we can also create multiple Thread objects and give each of them the same function to run.

Naming Threads

- We can give a name to a thread when we create it:

```
t = Thread(target = aFunction, name='thread name')
```

- In the target function we can check for the name of the thread that's running the function

```
print( threading.currentThread( ).getName() )
```

- Naming threads can be useful when there are many threads and we need to identify them.

Daemon Threads

- Typically the main thread should wait for all the child threads to end with the join method before the main thread ends.
- But sometimes it's okay to start a thread and let it run on its own, without having to wait for it to finish before ending the main thread.
- These independently run threads are marked as daemon threads.
- Daemon threads are automatically killed when the main thread ends.
- Therefore daemon threads should only be used to run tasks where they don't access shared resources, so that if they're abruptly killed (when the main task ends), then they don't leave resources opened or leave data partially modified.
- We set a thread as a daemon thread after creating the thread:

```
d = threading.Thread(target = aFunction)
d.setDaemon(True)
```

Checking Thread Status

- When the main thread runs the join method and the child thread has not finished running, the main thread is suspended or blocked until the child thread is finished.
- It is possible to set a timer for the join method so that after a certain amount of time, the join will happen even if the child thread is not finished, and the main thread can stop being blocked.
- To set the timer:

```
t.join(3.0)      # set a timer for 3.0 sec
```

the timer is always set with a floating point value

- If a timer is set, then when the main thread is no longer blocked, it can check whether it becomes unblocked due to the timer timing out or due to the child thread being done. The check:

```
t.is_alive( )
```

will return: True if the thread t is not done, False if t is done.

- To check the status of all currently running thread, use enumerate of the threading module:

```
for  t  in  threading.enumerate( ) :
    print(t.getName( ))
```

In the example code, we print the names of all currently running threads: main thread, child threads, and daemon threads.

Event

- When 2 threads work together and one thread needs to wait for another thread to do some task before it can run, then we can use an Event object.
- An Event object acts like a Boolean. It maintains an internal flag that a thread A can set or clear to signal that it has finished some work. The other thread B can wait for this flag condition before doing some of its own tasks.
- To create an Event object:

```
e = threading.Event()
```

- When thread A needs to set or clear the Event flag:

```
e.set()
```
or
```
e.clear()
```

- Thread B can check the status of the flag: which returns True or False.

```
e.is_set()
```

- Thread B can also wait for the flag to be set: which means B is blocked until the flag is set.

```
e.wait()
```

- Thread B can also wait for a certain time: which means B is blocked for 2.5 seconds, and after 2.5 seconds it will resume running.

```
e.wait(2.5)
```

Lock (1 of 2)

- When 2 threads use the same resource, then a race condition can happen.
- A race condition is when 2 threads compete or race with each other to get to the same resource. If they both get to the resource at the same time and both make modification to the data at the resource, then the resulting data may not be correct.
- To prevent a race condition and possible data corruption, a Lock is used.
- To create a Lock object: `lock = threading.Lock()`
- There are 2 states for the Lock object: locked and unlocked.
  A newly created lock is in the unlocked state.
- Before accessing the shared resource, a thread locks the resource so that no other thread can access the same resource. When it's done with the resource, then it unlocks the resource so that other threads can access it.
- To lock: `lock.acquire()` To unlock: `lock.release()`
- If an acquire is attempted during an unlocked state, then the lock happens immediately and the return value is True.
- If an acquire is attempted during a locked state, then the thread that runs acquire is blocked until the lock is released, at which time the lock is locked again immediately for the requesting thread.

Lock (2 of 2)

- A thread can prevent being blocked while waiting for the lock by using:

```
lock.acquire(blocking=False)
```

  If the return value is True, then it has successfully locked.
  If the return value is False, then it is not blocked and can do some other task that doesn't require the shared resource.

- A thread can also set a timer when requesting a lock:

```
lock.acquire(timeout=2.5)
```

  This means the requesting thread is only blocked for 2.5 seconds while waiting for the unlock. If the unlock doesn't occur before the timer times out, then the thread is unblocked when the timer times out.

- If there's a good chance that the thread won't be blocked when getting a lock, then we can use the with syntax, just as with file open:

```
lock.acquire()
do some task
lock.release()
```

is equivalent to

```
with lock :
    do some task
```

- When given a lock, the with construct runs the acquire, and at the end of the block, it automatically runs the release.

Thread Safe Containers

- A thread safe container is one that guarantees that only one thread can access it at a time. This means multiple threads can access a thread safe container without having to worry about locking it.
- Python's built-in containers (list, dictionary, etc) are thread safe if data are stored into or removed from the container in a single operation.
- Example:

```
L.append(data)
val = L.pop()
D[k] = val
```

thread safe, each line is one operation

```
L.append(L[-1] + 1)
```

not thread safe, 3 operations on one line

- Each operation is considered one atomic operation by Python, which means it will completely finish running before another thread can access the container. This prevents any race condition.
- When multiple threads access a thread safe container by using single operations, it is not necessary to use a lock.
- But it is considered acceptable to still use a lock with the built-in containers.

Queue (1 of 2)

- When 2 threads work together such that one thread produces output data and the other thread uses that data as input, we can use a Queue to coordinate the 2 threads.
- The thread that produces output data is called the producer, and the thread that uses the data as input is called the consumer.
- The producer puts data in the Queue object, and the consumer fetches data from the Queue. In this way the Queue is the buffer between the 2 threads that work asynchronously.

  producer → Queue → consumer

- To use a Queue:

```
import queue
```

- To create a Queue:

```
q = queue.Queue()
```

- The Queue object is a FIFO queue, and it has a built-in lock so that only one thread can access it at one time.

Queue (2 of 2)

- An important status of a queue is whether the queue is empty.
  If the queue is empty and the consumer attempts to get data from the queue, then it will be blocked until there is data in the queue and it can get the data.
- To check for the queue status: returns True or False

```
q.empty()
```

- To put data in the queue:

```
q.put(data)
```

- To get data from the queue: returns the data

```
q.get()
```

Threads in Python (1 of 2)

- The majority of Python code runs a version of Python called CPython, which means the interpreter is written in C. CPython is the original version of the Python language and is the version we get by downloading from python.org.
- In CPython (and some other Python versions) all the threads of one Python process run on one processor, which is the same processor that the process itself runs on. This occurs even on a multicore or multi-processor system.
- In addition, if a process has multiple threads, then only one thread can run at a time. This is enforced by the Python Global Interpreter Lock, aka the GIL. The lock allows only one Python thread in a process to be run at a time.
- This means that threads cannot be used to implement true parallel execution of Python code. In other words, parallel CPU operations is not possible with multithreading in Python.
- The decision to implement the GIL is controversial. On one hand, having the GIL means the interpreter code is simpler and can run faster. On the other hand, it means no true parallel processing with multithreading. (Here is a more detailed discussion of the pros and cons of the GIL.)

Threads in Python (2 of 2)

- In spite of the GIL, threads are very much in use in Python.
- Threads are most often used for slow operations such as IO-bound tasks:
  - GUI: threads enable the GUI to stay responsive to the user while a time consuming task is running.
  - Network or file IO: the program can create threads to do other tasks while waiting for data.
- For tasks that require intensive CPU operations, such as rendering an image or doing calculations on a large multi-dimensional matrix, using threads can actually slow down the task, not just because only one thread can run at a time, but also because it takes additional coordination between the threads.
- Therefore, for time intensive CPU-bound tasks, we can:
  - Use specific Python modules for the task. For example, for numerical calculations, use numpy, scipy, etc., which are implemented directly in C and Fortran and are not affected by the GIL.
  - Use Python multiprocessing, which means multiple processes can run in parallel. And since the GIL controls the threads within one process only, the GIL cannot affect multiple processes.

Threads and Tkinter (1 of 2)

- For Tkinter, the mainloop is the main thread that processes events.
- The mainloop blocks, which means once it starts, it does not let us directly call any function while it's running. Any call within the mainloop is a callback function, a result of an event.
- To run Tkinter code in a process that uses threads, Tkinter provides the after method, which lets us interrupt the mainloop with a callback function that's *not* due to an event:

```
id = aTkObject.after(delayTime, functionName)
```

  - after is a method of all Tkinter objects.
  - after puts the functionName in a queue of the mainloop, with a delayTime in milliseconds. The return value is a unique ID for the functionName that's in the queue.
- When the timer with the delayTime times out, the function runs as a callback function. This callback is run due to a timer instead of due to a user event.
- The functionName should be a short function that can run quickly so that it doesn't slow down the GUI response time.
- If we need to run the function more than once, we call after inside the function so that the function name can be put in the queue again.

Threads and Tkinter (2)

- To cancel a function that's already in the queue:

```
aTkObject.after_cancel(id)
```

  where id is the id of the function in the queue.
- Tkinter and most GUI packages are *not* thread safe. This is because GUIs are event controlled, and managing threads that run callback functions will add layers of coordination that slow down the GUI code.
- Therefore all GUI code should be run within one thread, and this thread is preferably the main thread. This means the GUI is the driver for the application and it creates child threads to do multiple tasks.
- If a child thread has data to update the GUI window, then after the data is fetched, the window can be updated with the new data by using:

```
aTkObject.update( )
```

- The main thread of the GUI can start another thread with the after method, and the 2 threads can communicate with each other through an Event object or a Queue object or a Python built-in container.

Going further…

- Threads can be useful in making certain applications run faster or be more responsive.
- The Python threading documentation describes more methods and objects in addition to the ones we've covered in the notes.
- Threading in Python doesn't provide true parallel execution because all threads run on the same processor, we can't take advantage of multiple processors on the same system.
- For parallel processing with multiple processors, we use processes instead of threads.

Up next: Processes