

KEEPING TIME, SCHEDULING TASKS, THREADING AND MULTIPROCESSING

CS 3080: Python Programming



University of Colorado
Colorado Springs

Keeping Time, Scheduling Tasks, and Launching Programs

- Running programs while you're sitting at your computer is fine, but it's also useful to have programs run without your direct supervision.
- Your computer's clock can schedule programs to run code at some specified time and date or at regular intervals.
- For example:
 - *Your program could scrape a website every hour to check for changes*
 - *Do a CPU-intensive task at 4 am while you sleep.*

Keeping Time, Scheduling Tasks, and Launching Programs

- Modules
 - *time*
 - *datetime*
 - *threading*
 - *multiprocessing*



TIME MODULE



The time module

- Your computer's system clock is set to a specific date, time, and time zone.
- The built-in time module allows your Python programs to read the system clock for the current time.
- Most useful functions in the time module:
 - *time.time()*
 - *time.perf_counter()*
 - *time.sleep()*
 - *time.gmtime(x)*
 - *time.localtime(x)*

time.time()

- The **Unix epoch** is a time reference commonly used in programming: 12 am on January 1, 1970, Coordinated Universal Time (UTC).
- The time.time() function returns the number of seconds since that moment as a float value. This number is called an **epoch timestamp**.

```
import time
```

```
print(time.time())
```

```
# 1553297898.254009
```

time.perf_counter()

- Return the value (in fractional seconds) of a performance counter: **a clock with the highest available resolution to measure a short duration.**
- The reference point of the returned value is undefined, so that **only the difference between the results of consecutive calls is valid.**

```
import time
```

```
print(time.perf_counter())
```

```
# 70872.493310016
```

time.perf_counter() to profile code

- Timestamps can **profile** code, i.e., measure how long a piece of code takes to run.
 - *Example: the timer decorator*

```
def timer(func):  
    """Print the runtime of the decorated function"""  
    @functools.wraps(func)  
    def wrapperTimer(*args, **kwargs):  
        startTime = time.perf_counter()  
        value = func(*args, **kwargs)  
        endTime = time.perf_counter()  
        runTime = endTime - startTime  
        print("Finished {} in {:.4f} secs".format(func.__name__, runTime))  
        return value  
    return wrapperTimer
```


`time.gmtime(secs)` and `time.localtime(secs)`

```
secs = time.time()
```

If `secs` is not provided or `None`, the current time as returned by `time()` is used.

```
print(time.gmtime(secs)) # in UTC
```

```
# time.struct_time(tm_year=2019, tm_mon=3, tm_mday=22, tm_hour=23,  
# tm_min=51, tm_sec=7, tm_wday=4, tm_yday=81, tm_isdst=0)
```

```
print(time.localtime(secs)) # in localtime
```

```
# time.struct_time(tm_year=2019, tm_mon=3, tm_mday=22, tm_hour=17,  
# tm_min=51, tm_sec=7, tm_wday=4, tm_yday=81, tm_isdst=1)
```

time.sleep(secs)

- Suspend execution of the calling thread for the given number of seconds.
 - *If you need to pause your program for a while*

```
for i in range(3):  
    print('Tick')  
    time.sleep(1)  
    print('Tock')  
    time.sleep(1)
```

round(secs)

```
secs = time.time()
```

```
print(secs)
```

```
# 1554182343.6270301
```

```
print(round(secs, 2))
```

```
# 1554182343.63
```

```
print(round(secs, 5))
```

```
# 1554182343.62703
```

```
print(round(secs))
```

```
# 1554182344
```



DATETIME MODULE



datetime module

- The time module is useful for getting a Unix epoch timestamp to work with. But if you want to display a date in a more convenient format, or do arithmetic with dates (for example, figuring out what date was 205 days ago or what date is 123 days from now), you should use the **datetime module**.
- The datetime module has its own datetime data type. datetime values represent a specific moment in time.

datetime module

```
import datetime
```

```
print(type(datetime.datetime.now())) # <class  
'datetime.datetime'>
```

```
print(datetime.datetime.now())      # 2020-11-21 19:12:36.045976
```

```
dt = datetime.datetime(2015, 10, 21, 16, 29, 0)
```

```
print(dt.year, dt.month, dt.day)    # 2015 10 21
```

```
print(dt.hour, dt.minute, dt.second) # 16 29 0
```

datetime module

- A Unix epoch timestamp can be converted to a datetime object with the **datetime.datetime.fromtimestamp()** function. The date and time of the datetime object will be converted for the local time zone.

```
import datetime
```

```
import time
```

```
print(datetime.datetime.fromtimestamp(1000000))
```

```
# 1970-01-12 06:46:40
```

```
print(datetime.datetime.fromtimestamp(time.time()))
```

```
# 2020-11-21 19:14:42.959364
```

datetime module

- datetime objects can be compared with each other using comparison operators to find out which one precedes the other

```
halloween2020 = datetime.datetime(2020, 10, 31, 0, 0, 0)
```

```
newyears2021 = datetime.datetime(2021, 1, 1, 0, 0, 0)
```

```
oct31_2020 = datetime.datetime(2020, 10, 31, 0, 0, 0)
```

```
print(halloween2020 == oct31_2020) # True
```

```
print(halloween2020 > newyears2021) # False
```

```
print(newyears2021 > halloween2020) # True
```

```
print(newyears2021 != oct31_2020) # True
```


The `timedelta` Data Type

- The `datetime` module also provides a **`timedelta`** data type, which represents a duration of time rather than a moment in time.

```
delta = datetime.timedelta(days=11, hours=10, minutes=9, seconds=8)
```

```
print(delta.days, delta.seconds, delta.microseconds)
```

```
# (11, 36548, 0)
```

```
print(delta.total_seconds())
```

```
# 986948.0
```

```
print(str(delta))
```

```
# '11 days, 10:09:08'
```

The `timedelta` Data Type

- The `datetime` module also provides a **`timedelta`** data type, which represents a duration of time rather than a moment in time.

```
dt = datetime.datetime.now()
print(dt)                                # 2020-11-21 19:19:39.552334
```

```
threeYears = datetime.timedelta(days=365*3)
print(dt + threeYears * 2)               # 2026-11-20 19:19:39.552334
```

Pause code until a specific date

```
import datetime
```

```
import time
```

```
halloween2020 = datetime.datetime(2020, 10, 31, 0, 0, 0)
```

```
while datetime.datetime.now() < halloween2020:
```

```
    time.sleep(1)
```

```
# --code-- do something after that date
```

Converting datetime Objects into Strings

```
oct21st = datetime.datetime(2015, 10, 21, 16, 29, 0)
```

```
print(oct21st.strftime('%Y/%m/%d %H:%M:%S'))
```

```
# '2015/10/21 16:29:00'
```

```
print(oct21st.strftime('%I:%M %p'))
```

```
# '04:29 PM'
```

```
print(oct21st.strftime("%B of '%y"))
```

```
# "October of '15"
```

Converting datetime Objects into Strings

Table 15-1: strftime() Directives

strftime directive	Meaning
%Y	Year with century, as in '2014'
%y	Year without century, '00' to '99' (1970 to 2069)
%m	Month as a decimal number, '01' to '12'
%B	Full month name, as in 'November'
%b	Abbreviated month name, as in 'Nov'
%d	Day of the month, '01' to '31'
%j	Day of the year, '001' to '366'
%w	Day of the week, '0' (Sunday) to '6' (Saturday)
%A	Full weekday name, as in 'Monday'
%a	Abbreviated weekday name, as in 'Mon'
%H	Hour (24-hour clock), '00' to '23'
%I	Hour (12-hour clock), '01' to '12'
%M	Minute, '00' to '59'
%S	Second, '00' to '59'
%p	'AM' or 'PM'
%%	Literal '%' character

Converting Strings into datetime Objects

```
print(datetime.datetime.strptime('October 21, 2015', '%B %d, %Y'))  
# 2015-10-21 00:00:00  
print(datetime.datetime.strptime('2015/10/21 16:29:00', '%Y/%m/%d %H:%M:%S'))  
# 2015-10-21 16:29:00  
print(datetime.datetime.strptime("October of '15", "%B of '%y"))  
# 2015-10-01 00:00:00  
print(datetime.datetime.strptime("November of '63", "%B of '%y"))  
# 2063-11-01 00:00:00
```



THREADING MODULE

Parallelism

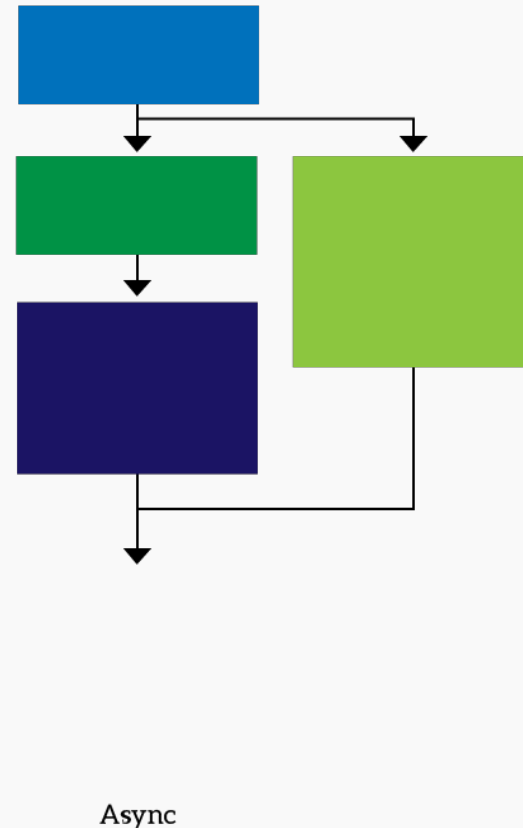
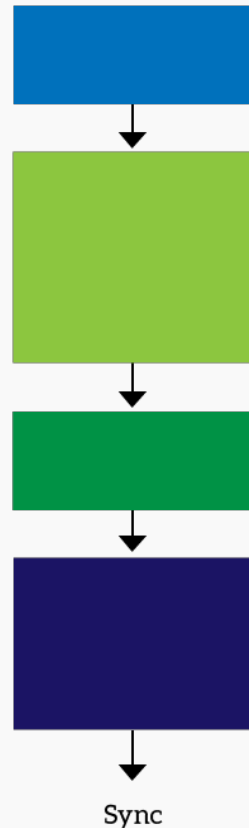
- **Parallelism** means that your program runs some of its parts at the same time. They might be different parts, or even multiple instances of the same part. However, each one is running independently from the other.
- Parallelism is at the core of modern programming. In fact, the reasons for running different pieces of code at the same time are endless. Scalability, efficiency – you name it. No matter the reason, multithreading is a simple way to achieve parallelism in Python.

Synchronous vs Asynchronous

- A simple program is **synchronous**, that means instructions will be executed in order. The program will run each instruction after another, in the order you prepared.
- However, with threading you can run parts of the programs **asynchronously**. This means parts of the code may run before or after some other parts, as they are now unrelated. **Asynchronous** is the buzzword here.

Synchronous vs Asynchronous

- With an **async** thread (= **background** thread = **parallel** thread), some instructions may be executed concurrently.



Real world examples

- Online ticket booking
 - *Many users trying to book available ticket at same time, the application needs to handle different threads for each different user request, if tickets sold out/not available then the rest of the users will get correct responses as not available to book.*
- Any program with GUI
 - You are building a GUI program that uses network and a DB. Usually opening network connections or db connections are costly in terms of resources and execution time. To avoid freezing of your GUI you can use threads where the main thread will run your GUI code while other threads will do DB or Network jobs asynchronously.

Single thread

```
import time
import datetime
```

```
halloween2029 = datetime.datetime(2029, 10, 31, 0, 0, 0)
while datetime.datetime.now() < halloween2029:
    time.sleep(1)
```

```
print('Program now starting on Halloween 2029')
# --code--
```

This program cannot do anything while waiting for the loop of `time.sleep()` calls to finish; it just sits around until Halloween 2029. This is because Python programs by default have a **single thread of execution**.

Single thread

```
import time
import datetime
```

```
halloween2029 = datetime.datetime(2029, 10, 31, 0, 0, 0)
while datetime.datetime.now() < halloween2029:
    time.sleep(1)
```

```
print('Program now starting on Halloween 2029')
# --code--
```

Rather than having all of your code wait until the while loop finishes, **you can execute the delayed or scheduled code in a separate thread using Python's threading module**. The separate thread will pause for the `time.sleep` calls. Meanwhile, your program can do other work in the original thread.

Multithreading example

```
import threading
import time

print('Start of program.')
```



```
def takeANap():
    time.sleep(5)
    print('Wake up!')
```



```
threadObj = threading.Thread(target=takeANap)
threadObj.start()
print('End of program.')
```

Multithreading example

```
import threading
```

```
import time
```

```
print('Start of program.')
```

```
def takeANap():  
    time.sleep(5)  
    print('Wake up!')
```

```
threadObj = threading.Thread(target=takeANap)
```

```
threadObj.start()
```

```
print('End of program.')
```

Output:

Start of program.
End of program.
Wake up!

Multithreading example

- Normally a program terminates when the last line of code in the file has run (or `sys.exit()` is called).
- But this code has two threads.
 - *The first is the original thread that began at the start of the program and ends after `print('End of program.')`.*
 - *The second thread is created when `threadObj.start()` is called, begins at the start of the `takeANap()` function, and ends after `takeANap()` returns.*
- **A Python program will not terminate until all its threads have terminated. When you ran that code, even though the original thread had terminated, the second thread was still executing the `time.sleep(5)` call.**
 - *Different from C*

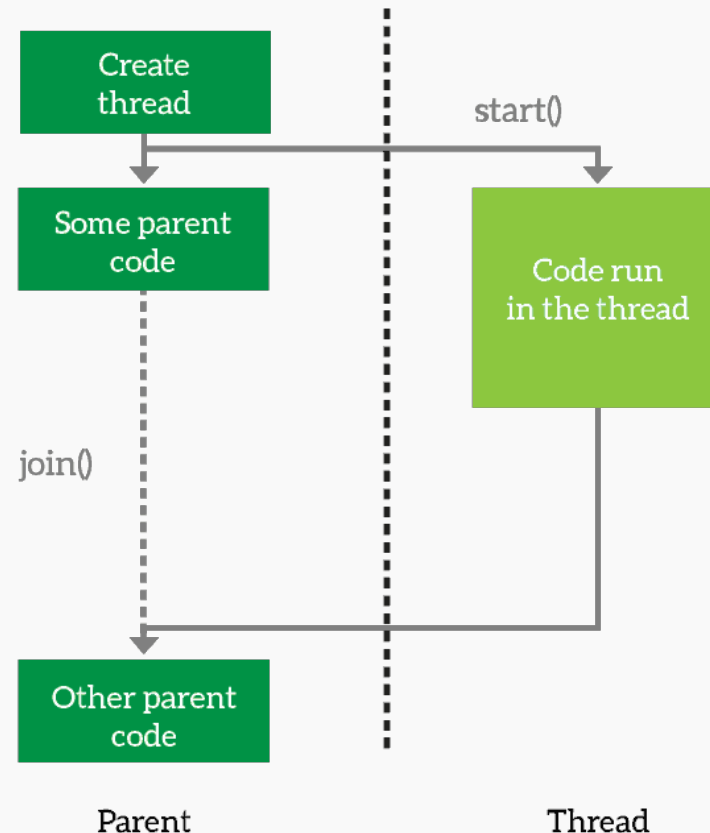
Passing arguments to the Thread target's function

```
def wakeUp(firstname, lastname, hours=8):  
    time.sleep(2)  
    print('Wake up {} {}! You have slept {} hours'.format(firstname,  
                                                             lastname, hours))
```

```
threadObj = threading.Thread(target=wakeUp, args=['Jim', 'Carrey'],  
                             kwargs={'hours': 10})  
  
threadObj.start()  
print('New thread started.')
```

Sync the threads

- We might need to have some thread waiting the execution of another. In that case, we need to **sync the threads**. Once you have a thread object, you can decide to start it with the **start()** function, and to join it with the **join()** function.



Sync the threads

```
print('Start of program.')
```

```
def someFunc():  
    print('Doing something in an async thread')  
    time.sleep(5)  
    print('Async thread finished')
```

```
threadObj = threading.Thread(target=someFunc)  
threadObj.start()
```

```
print('Doing something in the main thread')  
time.sleep(2)  
print('Main thread task finished. Wait for the async thread to finish')
```

```
threadObj.join()  
print('End of program.')
```

Sync the threads

```
print('Start of program.')
```

```
def someFunc():  
    print('Doing something in an async thread')  
    time.sleep(5)  
    print('Async thread finished')
```

```
threadObj = threading.Thread(target=someFunc)  
threadObj.start()
```

```
print('Doing something in the main thread')  
time.sleep(2)  
print('Main thread task finished. Wait for the async thread to finish')
```

```
threadObj.join()  
print('End of program.')
```

Output:


Start of program.
Doing something in an async thread
Doing something in the main thread
Main thread task finished. Wait for the async thread to finish
Async thread finished
End of program.

Inter-thread communication

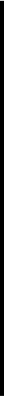

- A function that only executes code but that does not return anything has limited applications.
- Why can't a thread return something upon finish, just like a function does? The reason for that is ***the asynchronous nature of threads***. When does the return should happen? What should the parent script do until it gets a return? If you just wait for the whole thread to be executed, you could use a synchronous function as well.
- **Threads are designed to *not* be alternatives to functions.** They serve a whole different purpose, and **they do not strictly return something. Instead, they can communicate with the script that generated the thread.**
 - Here we have another challenge: **concurrency**

Concurrency issues

- You can easily create new threads and have them running at the same time. But multiple threads can also cause a problem called concurrency issues (aka race conditions). **These issues happen when threads read and write variables at the same time**, causing the threads to trip over each other. Concurrency issues can be hard to reproduce consistently, making them hard to debug.
- To avoid concurrency issues, **when possible** never let multiple threads read or write the same variables. When you create a new Thread object, make sure its **target function uses only local variables in that function**. This will avoid hard-to-debug concurrency issues in your programs.
- But, if you have to use global variables in the function: Python implements some libraries to be able to do this and avoid concurrency:
 - *queue module*



QUEUE MODULE



queue module

- The queue.Queue object is a special item that we can use to handle concurrency.
- In a common setup, you will have some part of the code populating the queue, and some other reading it and processing its data.
- The function, or piece of code, that adds items into the queue is the **producer**. Instead, the function processing (and thus removing) items from the queue is the **consumer** in background threads.
 - *Without the Queue object we would have a list where some parts of the code will be adding items to the list and others parts will be reading and deleting. Doing this in multiple threads will create concurrency issues.*

queue module - sentinel

- Sometimes, the consumer may be faster than the producer and empty the list. This doesn't mean the producer has finished, and thus the consumer shouldn't shut-down yet. We can solve this with a sentinel.
- A **sentinel** is a value of our choice that the producer puts in the queue to tell all the consumers it is done producing.
- A common value for a sentinel may be None.

queue module – example

- Show code

Multithreaded programming



Multithreading not that fast

- In many cases, python's threading doesn't improve execution speed very well... sometimes, it makes it worse.
- Multi-threading does not "speed up" an application. In Python multi-threading is a way to keep one processor busier with your code.
 - *Threads run in the same memory space. So threads cannot run "truly parallel".*
- If you want to speed up your program you should go for **multiprocessing**.
 - Processes have separate memory
 - Only real way to achieve true parallelism.



MULTIPROCESSING MODULE

Multiprocessing module

- API similar to the threading module.
- Uses subprocesses instead of threads. Due to this, the multiprocessing module allows the programmer to fully leverage multiple processors on a given machine.
 - *Each process is run in a different core. For example, Intel Core i7 have 4 cores. So I can run 4 different processes at the same time.*
- The module has his own Queue class.

Multiprocessing - example

- Show code

Threading vs multiprocessing

- Threading vs multiprocessing pros and cons:
 - <https://stackoverflow.com/a/3046201/5125212>
- Recommendation:
 - *Use threading if your program is network bound or have a GUI*
 - *Use multiprocessing if it's CPU bound and your machine has multiple cores*