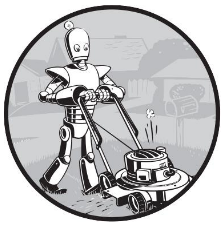
19

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 or do a CPU-intensive task at 4 AM while you sleep. Python's `time` and `datetime` modules provide these functions.

You can also write programs that launch other programs on a schedule by using the `subprocess` module. Often, the fastest way to program is to take advantage of applications that other people have already written.

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. The most useful of its functions are `time.time()`, which returns a value called the epoch timestamp, and `time.sleep()`, which pauses a program.

Returning the Epoch Timestamp

The *Unix epoch* is a time reference commonly used in programming: midnight on January 1, 1970, Coordinated Universal Time (UTC). The `time.time()` function returns the number of seconds since that moment as a float value. (Recall that a float is just a number with a decimal point.) This number is called an *epoch timestamp*. For example, enter the following into the interactive shell:

```
>>> import time
>>> time.time()
1773813875.3518236
```

Here, I'm calling `time.time()` on March 17, 2026, at 11:04 PM Pacific Standard Time. The return value is how many seconds have passed between the Unix epoch and the moment `time.time()` was called.

The return value from `time.time()` is useful, but is not human readable. The `time.ctime()` function returns a string description of the current time. You can also optionally pass the number of seconds since the Unix epoch, as returned by `time.time()`, to get a string value of that time. Enter the following into the interactive shell:

```
>>> import time
>>> time.ctime()
'Tue Mar 17 11:05:38 2026'
>>> this_moment = time.time()
>>> time.ctime(this_moment)
'Tue Mar 17 11:05:45 2026'
```

Epoch timestamps can be used to *profile* code: that is, measure how long a piece of code takes to run. If you call `time.time()` at the beginning of the code block you want to measure and again at the end, you can subtract the first timestamp from the second to find the elapsed time between those two calls. For example, open a new file editor tab and enter the following program:

```
# Measure how long it takes to multiply
100,000 numbers.
import time
❶ def calculate_product():
        # Calculate the product of the first
    100,000 numbers.
        product = 1
        for i in range(1, 100001):
            product = product * i
        return product

❷ start_time = time.time()
  result = calculate_product()
❸ end_time = time.time()
❹ print(f'It took {end_time – start_time}
  seconds to calculate.')
```

At ❶, we define a function `calculate_product()` to loop through the integers from 1 to 100,000 and return their product. At ❷, we call `time.time()` and store it in `start_time`. Right after calling `calculate_product()`, we call `time.time()` again and store it in `end_time` ❸. We end by printing how long it took to run `calculate_product()` ❹.

Save this program as *calcProd.py* and run it. The output will look something like this:

```
It took 2.844162940979004 seconds to
calculate.
```

Another way to profile your code is to use the `cProfile.run()` function, which provides a much more informative level of detail than the simple `time.time()` technique. You can read about `cProfile.run()` function in Chapter 13 of my other book, *Beyond the Basic Stuff with Python* (No Starch Press, 2020).

Pausing Programs

If you need to pause your program for a while, call the `time.sleep()` function and pass it the number of seconds you want your program to stay paused. For example, enter the following into the interactive shell:

```
  >>> import time
  >>> for i in range(3):
  ...   ❶ print('Tick')
  ...   ❷ time.sleep(1)
  ...   ❸ print('Tock')
  ...   ❹ time.sleep(1)
  ...
  Tick
  Tock
  Tick
  Tock
  Tick
  Tock
❺ >>> time.sleep(5)
  >>>
```

The `for` loop will print `Tick` ❶, pause for one second ❷, print `Tock` ❸, pause for one second ❹, print `Tick`, pause, and so on, until `Tick` and `Tock` have each been printed three times.

The `time.sleep()` function will *block* (that is, it won't return or release your program to execute other code) until after the number of seconds you passed to `time.sleep()` has elapsed. For example, if you enter `time.sleep(5)` ❺, you'll see that the next prompt (>>>) doesn't appear until five seconds have passed.

Project 14: Super Stopwatch

Say you want to track how much time you spend on boring tasks you haven't automated yet. You don't have a physical stopwatch, and it's surprisingly difficult to find a free stopwatch app for your laptop or smartphone that isn't covered in ads and doesn't send a copy of your browser history to marketers. (It says it can do this in the license agreement you agreed to. You did read the license agreement, didn't you?) You can write a simple stopwatch program yourself in Python.

At a high level, here's what your program will do:

- Find the current time by calling `time.time()` and store it as a timestamp at the start of the program, as well as at the start of each lap.
- Keep a lap counter and increment it every time the user presses ENTER.
- Calculate the elapsed time by subtracting timestamps.
- Handle the `KeyboardInterrupt` exception so that the user can press CTRL-C to quit.

Open a new file editor tab and save it as *stopwatch.py*.

Step 1: Set Up the Program to Track Times

The stopwatch program will need to use the current time, so you'll want to import the `time` module. Your program should also print some brief instructions to the user before calling `input()` so that the timer can begin after the user presses ENTER. Then, the code will start tracking lap times each time the user presses ENTER until they press CTRL-C to quit.

Enter the following code into the file editor, writing a `TODO` comment as a placeholder for the rest of the code:

```
# A simple stopwatch program
import time

# Display the program's instructions.
print('Press ENTER to begin and to mark laps.
Ctrl-C quits.')
input()  # Press Enter to begin.
print('Started.')
start_time = time.time()  # Get the first
lap's start time.
last_time = start_time
lap_number = 1

# TODO: Start tracking the lap times.
```

Now that you've written the code to display the instructions, start the first lap, note the time, and set the `lap_number` to 1.

Step 2: Track and Print Lap Times

Now let's write the code to start each new lap, calculate how long the previous lap took, and calculate the total time elapsed since starting the stopwatch. We'll display the lap time and total time and increase the lap count for each new lap. Add the following code to your program:

```
# A simple stopwatch program
import time

--snip--

# Start tracking the lap times.
❶ try:
    ❷ while True:
            input()
          ❸ lap_time = round(time.time() -
last_time, 2)
          ❹ total_time = round(time.time() -
start_time, 2)
          ❺ print('Lap #{lap_number}:
{total_time}({lap_time})', end='')
            lap_number += 1
            last_time = time.time() # Reset the
last lap time.
❻ except KeyboardInterrupt:
        # Handle the Ctrl-C exception to keep its
error message from displaying.
        print('\nDone.')
```

If the user presses CTRL-C to stop the stopwatch, the `KeyboardInterrupt` exception will be raised, and the program will crash. To prevent crashing, we wrap this part of the program in a `try` statement ❶. We'll handle the exception in the `except` clause ❻, which prints `Done` when the exception is raised instead of showing the `KeyboardInterrupt` error message. Until this happens, the execution occurs inside an infinite loop ❷ that calls `input()` and waits until the user presses ENTER to end a lap. When a lap ends, we calculate how long the lap took by subtracting the start time of the lap, `last_time`, from the current time, `time.time()` ❸. We calculate the total time elapsed by subtracting the overall start time of the stopwatch, `start_time`, from the current time ❹.

Because the results of these time calculations will have many digits after the decimal point (such as `4.766272783279419`), we use the `round()` function to round the float value to two digits at ❸ and ❹.

At ❺, we print the lap number, total time elapsed, and lap time. As the user pressing ENTER for the `input()` call will print a newline to the screen, pass `end=''` to the `print()` function to avoid double-spacing the output. After printing the lap information, we get ready for the next lap by adding 1 to the count `lap_number` and setting `last_time` to the current time, which is the start time of the next lap.

Ideas for Similar Programs

Time tracking opens up several possibilities for your programs. Although you can download apps to do some of these things, the benefit of writing programs yourself is that they will be free and not bloated with ads and useless features. You could write similar programs to do the following:

- Create a simple timesheet app that records when you type a person's name and uses the current time to clock them in or out.
- Add a feature to your program to display the elapsed time since a process started, such as a download that uses the `requests` module. (See Chapter 13.)
- Intermittently check how long a program has been running and offer the user a chance to cancel tasks that are taking too long.

The 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. The `datetime` values represent a specific moment in time. Enter the following into the interactive shell:

```
>>> import datetime
❶ >>> datetime.datetime.now()
❷ datetime.datetime(2026, 2, 27, 11, 10, 49,
727297)
❸ >>>
dt = datetime.datetime(2026, 10, 21, 16, 29,
0)
❹ >>> dt.year, dt.month, dt.day
(2026, 10, 21)
❺ >>> dt.hour, dt.minute, dt.second
(16, 29, 0)
```

Calling `datetime.datetime.now()` ❶ returns a `datetime` object ❷ for the current date and time, according to your computer's clock. This object includes the year, month, day, hour, minute, second, and microsecond of the current moment. You can also retrieve a `datetime` object for a specific moment by using the `datetime.datetime()` function ❸, passing it integers representing the year, month, day, hour, and second of the moment you want. These integers will be stored in the `datetime` object's `year`, `month`, `day` ❹, `hour`, `minute`, and `second` ❺ attributes.

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. Enter the following into the interactive shell:

```
>>> import datetime, time
>>> datetime.datetime.fromtimestamp(1000000)
datetime.datetime(1970, 1, 12, 5, 46, 40)
>>> datetime.datetime.fromtimestamp(time.time
())
datetime.datetime(2026, 10, 21, 16, 30, 0,
604980)
```

Calling `datetime.datetime.fromtimestamp()` and passing it `1000000` returns a `datetime` object for the moment

1,000,000 seconds after the Unix epoch. Passing `time.time()`, the Unix epoch timestamp for the current moment, returns a `datetime` object for the current moment. So, the expressions `datetime.datetime.now()` and `datetime.datetime.fromtimestamp(time.time())` do the same thing; they both give you a `datetime` object for the present moment.

You can compare `datetime` objects with each other using comparison operators to find out which one precedes the other. The later `datetime` object is the “greater” value. Enter the following into the interactive shell:

```
  >>> import datetime
❶ >>>
  halloween_2026 = datetime.datetime(2026, 10,
  31, 0, 0, 0)
❷ >>>
  new_years_2027 = datetime.datetime(2027, 1,
  1, 0, 0, 0)
  >>>
  oct_31_2026 = datetime.datetime(2026, 10, 31,
  0, 0, 0)
❸ >>> halloween_2026 == oct_31_2026
  True
❹ >>> halloween_2026 > new_years_2027
  False
❺ >>> new_years_2027 > halloween_2026
  True
  >>> new_years_2027 != oct_31_2026
  True
```

This code makes a `datetime` object for the first moment (midnight) of October 31, 2026, and stores it in `halloween_2026` ❶. Then, it makes a `datetime` object for the first moment of January 1, 2027, and stores it in `new_years_2027` ❷. It creates another object for midnight on October 31, 2026, and stores it in `oct_31_2026`. Comparing `halloween_2026` and `oct_31_2026` shows that they’re equal ❸. Comparing `new_years_2027` and `halloween_2026` shows that `new_years_2027` is greater (later) than `halloween_2026` ❹ ❺.

Representing Duration

The `datetime` module also provides a `timedelta` data type, which represents a *duration* of time rather than a *moment* in time. Enter the following into the interactive shell:

```
>>> import datetime
❶ >>> delta = datetime.timedelta(days=11,
hours=10, minutes=9, seconds=8)
❷ >>> delta.days, delta.seconds,
delta.microseconds
(11, 36548, 0)
>>> delta.total_seconds()
986948.0
>>> str(delta)
'11 days, 10:09:08'
```

To create a `timedelta` object, use the `datetime.timedelta()` function. The `datetime.timedelta()` function takes the keyword arguments `weeks`, `days`, `hours`, `minutes`, `seconds`, `milliseconds`, and `microseconds`. There is no `month` or `year` keyword argument, because “a month” or “a year” is a variable amount of time depending on the particular month or year. A `timedelta` object has the total duration represented in days, seconds, and microseconds. These numbers are stored in the `days`, `seconds`, and `microseconds` attributes, respectively. The `total_seconds()` method will return the duration in number of seconds alone. Passing a `timedelta` object to `str()` will return a nicely formatted, human-readable string representation of the object.

In this example, we pass keyword arguments to `datetime.delta()` to specify a duration of 11 days, 10 hours, 9 minutes, and 8 seconds and store the returned `timedelta` object in `delta` ❶. This `timedelta` object’s `days` attribute stores `11`, and its `seconds` attribute stores `36548` (10 hours, 9 minutes, and 8 seconds, expressed in seconds) ❷. Calling `total_seconds()` tells us that 11 days, 10 hours, 9 minutes, and 8 seconds is 986,948 seconds. Finally, passing the `timedelta` object to `str()` returns a string that plainly describes the duration.

The arithmetic operators can be used to perform *date arithmetic* on `datetime` values. For example, to calculate the date 1,000 days from now, enter the following into the interactive shell:

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> now
datetime.datetime(2026, 12, 2, 18, 38, 50,
636181)
>>> thousand_days =
datetime.timedelta(days=1000)
>>> now + thousand_days
datetime.datetime(2029, 8, 28, 18, 38, 50,
636181)
```

First, make a `datetime` object for the current moment and store it in `now`. Then, make a `timedelta` object for a duration of 1,000 days and store it in `thousand_days`. Add `now` and `thousand_days` together to get a `datetime` object for the date 1,000 days from the date and time in `now`. Python will do the date arithmetic to figure out that 1,000 days after December 2, 2026, will be August 28, 2029. When calculating 1,000 days from a given date, you have to remember how many days are in each month and factor in leap years and other tricky details. The `datetime` module handles all of this for you.

You can add or subtract `timedelta` objects with `datetime` objects or other `timedelta` objects using the + and − operators. A `timedelta` object can be multiplied or divided by integer or float values with the * and / operators. Enter the following into the interactive shell:

```
  >>> import datetime
❶ >>> oct_21st = datetime.datetime(2026, 10,
  21, 16, 29, 0)
❷ >>> about_thirty_years =
  datetime.timedelta(days=365 * 30)
  >>> oct_21st
  datetime.datetime(2026, 10, 21, 16, 29)
  >>> oct_21st - about_thirty_years
  datetime.datetime(1996, 10, 28, 16, 29)
  >>> oct_21st - (2 * about_thirty_years)
  datetime.datetime(1966, 11, 5, 16, 29)
```

Here, we make a `datetime` object for October 21, 2026, ❶ and a `timedelta` object for a duration of about 30 years ❷. (We're using 365 days for each of those years and ignoring leap years.) Subtracting `about_thirty_years` from `oct_21st` gives us a `datetime` object for the date 30 years before October 21, 2026. Subtracting `2 * about_thirty_years` from `oct_21st` returns a `datetime` object for the date about 60 years before: the late afternoon of November 5, 1966.

Pausing Until a Specific Date

The `time.sleep()` method lets you pause a program for a certain number of seconds. By using a `while` loop, you can pause your programs until a specific date. For example, the following code will continue to loop until Halloween 2039:

```
import datetime
import time
halloween_2039 = datetime.datetime(2039, 10,
31, 0, 0, 0)
while datetime.datetime.now() <
halloween_2039:
    time.sleep(1)  # Wait 1 second before
looping to check again.
```

The `time.sleep(1)` call will pause your Python program so that the computer doesn't waste CPU processing cycles by checking the time over and over as fast as possible. Rather, the `while` loop will just check the condition once per second and continue with the rest of the program after Halloween 2039 (or whenever you program it to stop).

Converting datetime Objects into Strings

Epoch timestamps and `datetime` objects aren't very friendly to the human eye. Use the `strftime()` method to display a `datetime` object as a string. (The *f* in the name of the `strftime()` function stands for *format*.)

The `strftime()` method uses directives similar to Python's string formatting. Table 19-1 has a full list of `strftime()` directives. You can also consult the helpful *https://strftime.org* website for this information.

Table 19-1: `strftime()` Directives

`strftime()` directive	**Meaning**
`%Y`	Year with century, as in `'2026'`
`%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

Pass `strftime()` a custom format string containing formatting directives (along with any desired slashes, colons, and so on), and `strftime()` will return the `datetime` object's information as a formatted string. Enter the following into the interactive shell:

```
>>> oct_21st = datetime.datetime(2026, 10,
21, 16, 29, 0)
>>> oct_21st.strftime('%Y/%m/%d %H:%M:%S')
'2026/10/21 16:29:00'
>>> oct_21st.strftime('%I:%M %p')
'04:29 PM'
>>> oct_21st.strftime("%B of '%y")
"October of '26"
```

Here, we have a `datetime` object for October 21, 2026, at 4:29 PM, stored in `oct_21st`. Passing the custom format string `'%Y/%m/%d %H:%M:%S'` to `strftime()` returns a string containing 2026, 10, and 21 separated by slashes and 16, 29, and 00 separated by colons. Passing `'%I:%M %p'` returns `'04:29 PM'`, and passing `"%B of '%y"` returns `"October of '26"`.

Converting Strings into datetime Objects

If you have a string of date information, such as `'2026/10/21 16:29:00'` or `'October 21, 2026'`, and you need to convert it to a `datetime` object, use the `datetime.datetime.strptime()` function. The `strptime()` function is the inverse of the `strftime()` method, and you must pass it a custom format string using the same directives as `strftime()` so that the function knows how to parse and understand the string. (The *p* in the name of the `strptime()` function stands for *parse*.)

Enter the following into the interactive shell:

```
❶ >>> datetime.datetime.strptime('October 21,
2026', '%B %d, %Y')
datetime.datetime(2026, 10, 21, 0, 0)
>>> datetime.datetime.strptime('2026/10/21
16:29:00', '%Y/%m/%d %H:%M:%S')
datetime.datetime(2026, 10, 21, 16, 29)
>>> datetime.datetime.strptime("October of
'26", "%B of '%y")
datetime.datetime(2026, 10, 1, 0, 0)
```

```
   >>> datetime.datetime.strptime("November of
   '63", "%B of '%y")
❷ datetime.datetime(2063, 11, 1, 0, 0)
   >>> datetime.datetime.strptime("November of
   '73", "%B of '%y")
❸ datetime.datetime(1973, 11, 1, 0, 0)
```

To get a `datetime` object from the string `'October 21, 2026'`, pass that string as the first argument to `strptime()` and the custom format string that corresponds to `'October 21, 2026'` as the second argument ❶. The string with the date information must match the custom format string exactly, or Python will raise a `ValueError` exception. Notice that `"November of '63"` is interpreted as 2063 ❷ while `"November of '73"` is interpreted as 1973 `3` because the `%y` directive spans from 1970 to 2069.

A REVIEW OF PYTHON'S TIME FUNCTIONS

Dates and times in Python can involve quite a few different data types and functions. Here's a review of the three different types of values used to represent time:

- A Unix epoch timestamp (used by the `time` module) is a float or integer value representing the number of seconds since midnight on January 1, 1970, UTC.
- A `datetime` object (of the `datetime` module) has integers stored in the attributes `year`, `month`, `day`, `hour`, `minute`, `second`, and `microsecond`.
- A `timedelta` object (of the `datetime` module) represents a time duration, rather than a specific moment.

Here's a review of time functions, their parameters, and their return values:

`time.time()` This function returns an epoch timestamp of the current moment as a float value.

`time.sleep(`*`seconds`*`)` This function stops the program for the number of seconds specified by the *`seconds`* argument.

`datetime.datetime(`*`year, month, day, hour, minute, second`*`)` This function returns a `datetime` object of the moment specified by the arguments. If *`hour`*, *`minute`*, or *`second`* arguments are not provided, they default to `0`.

`datetime.datetime.now()` This function returns a `datetime` object of the current moment.

`datetime.datetime.fromtimestamp(`*`epoch`*`)` This function returns a `datetime` object of the moment represented by the *`epoch`* timestamp argument.

`datetime.timedelta(`*`weeks, days, hours, minutes, seconds, milliseconds, microseconds`*`)` This function returns a `timedelta` object representing a duration of time. The function's keyword arguments are all optional and do not include *`month`* or *`year`*.

`total_seconds()` This `timedelta` method returns the number of seconds the `timedelta` object represents.

`strftime(`*`format`*`)` This `datetime` method returns a string of the time in a custom format based on the *`format`* string. See Table 19-1 for the format details.

`datetime.datetime.strptime(`*`time_string, format`*`)` This function returns a `datetime` object representing the moment specified by *`time_string`*, parsed using the *`format`* string argument. See Table 19-1 for the format details.

Launching Other Programs from Python

Your Python program can start other programs on your computer with the `run()` function in the built-in `subprocess` module. If you have multiple instances of an application open, each of those instances is a separate process of the same program. For example, each open window of the calculator app shown in Figure 19-1 is a different process.

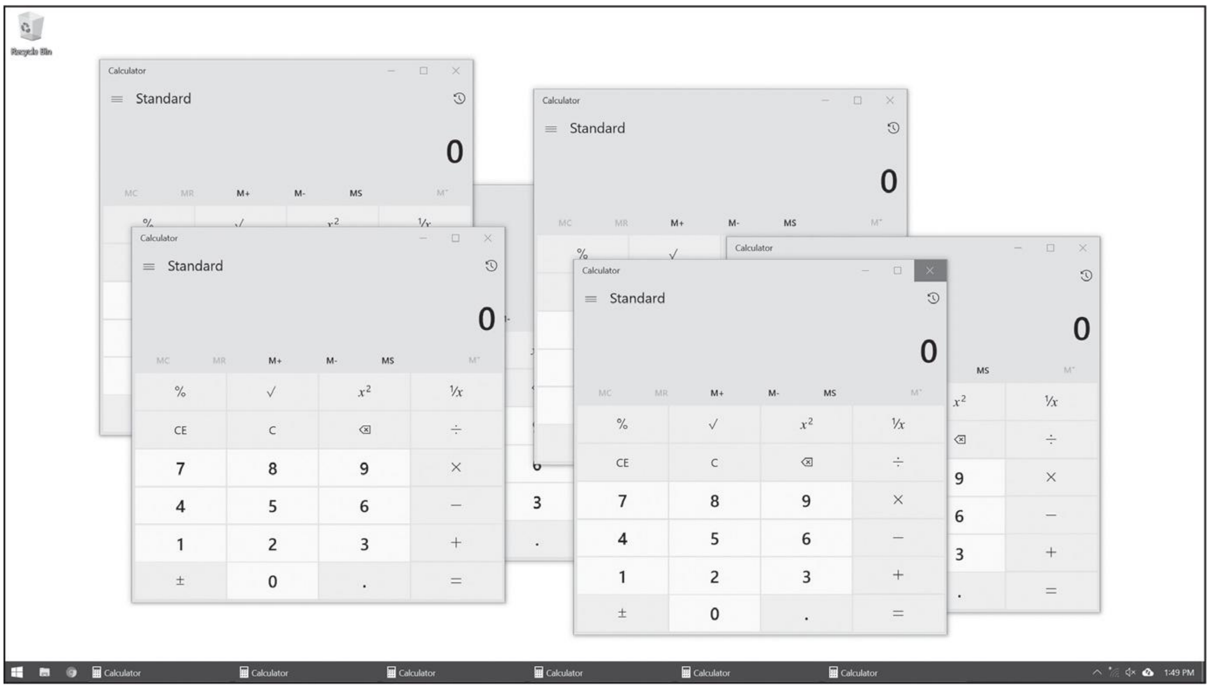


Figure 19-1: Six running processes of the same calculator program

If you want to start an external program from your Python script, pass the program's filename to `subprocess.run()`. (On Windows, right-click the application's **Start** menu item and select **Properties** to view the application's filename. On macOS, CTRL-click the application and select **Show Package Contents** to find the path to the executable file.) The `run()` function will block until the launched program closes. Pass the program to launch as a string of the executable program's filepath inside a list, keeping in mind that the launched program will be run in a separate process, not in the same process as your Python program.

On a Windows computer, enter the following into the interactive shell:

```
>>> import subprocess
>>> subprocess.run(['C:\\Windows\\System32\
\calc.exe'])
CompletedProcess(args=['C:\\Windows\
\System32\\calc.exe'], returncode=0)
```

On Ubuntu Linux, enter the following:

```
>>> import subprocess
>>> subprocess.run(['/usr/bin/gnome-
calculator'])
CompletedProcess(args=['/usr/bin/gnome-
calculator'], returncode=0)
```

On macOS, enter the following:

```
>>> import subprocess
>>> subprocess.run(['open', '/System/
Applications/Calculator.app'])
CompletedProcess(args=['open', '/System/
Applications/Calculator.app'], returncode=0)
```

Notice that macOS requires you to run the `open` program and pass it a command line argument of the program you want to launch.

In these examples, our Python code launched the program, waited for the program to close, and then continued. If you want your Python code to launch a program and then immediately continue without waiting for the program to close, call the `subprocess.Popen()` ("process open") function instead:

```
>>> import subprocess
>>> calc_proc = subprocess.Popen(['C:\
\Windows\\System32\\calc.exe'])
```

The return value is a `Popen` object, which has two useful methods: `poll()` and `wait()`.

You can think of the `poll()` method as asking your driver "Are we there yet?" over and over until you arrive. The `poll()` method will return `None` if the process is still running at the time `poll()` is called. If the program has terminated, it will return the process's integer *exit code*. An exit code indicates whether the process terminated without errors (represented by an exit code of `0`) or whether an error caused the process to terminate (represented by a nonzero exit code—generally `1`, but it may vary depending on the program).

The `wait()` method is like waiting until the driver has arrived at your destination. The method will block until the launched process has terminated. This is helpful if you want your program to pause until the user finishes interacting with the other program. The return value of `wait()` is the process's integer exit code.

On Windows, enter the following into the interactive shell. Note that the `wait()` call may block until you quit the launched Calculator program:

```
>>> import subprocess
❶ >>> calc_proc = subprocess.Popen(['c:\
\Windows\\System32\\calc.exe'])
❷ >>> calc_proc.poll() == None
True
❸ >>> calc_proc.wait() # Doesn't return until
Calculator closes
0
```

```
>>> calc_proc.poll()
0
```

Here, we open a Calculator process ❶. On older versions of Windows, `poll()` returns `None` ❷ if the process is still running. Then, we close the Calculator application's window, and back in the interactive shell, we call `wait()` on the terminated process ❸. Now `wait()` and `poll()` return `0`, indicating that the process terminated without errors.

If you run *calc.exe* on Windows 10 and later using `subprocess.Popen()`, you'll notice that `wait()` instantly returns even though the calculator app is still running. This is because *calc.exe* launches the calculator app and then instantly closes itself. The calculator program in Windows is a "Trusted Microsoft Store app," and its specifics are beyond the scope of this book. Suffice it to say that programs can run in many application-specific and operating system–specific ways.

If you want to close a process you've launched with `subprocess.Popen()`, call the `kill()` method of the `Popen` object the function returned. If you have MS Paint on Windows, enter the following into the interactive shell:

```
>>> import subprocess
>>> paint_proc = subprocess.Popen('c:\
\Windows\\System32\\mspaint.exe')
>>> paint_proc.kill()
```

Note that the `kill()` method immediately terminates a program and bypasses any "Are you sure you want to quit?" confirmation window. Any unsaved work in the program will be lost.

Passing Command Line Arguments to Processes

You can pass command line arguments to processes you create with `run()`. To do so, pass a list as the sole argument to `run()`. The first string in this list will be the executable filename of the program you want to launch; all the subsequent strings will be the command line arguments to pass to the program when it starts. In effect, this list will be the value of `sys.argv` for the launched program.

Most applications with a graphical user interface (GUI) don't use command line arguments as extensively as command line–based or terminal- based programs do. But most GUI applications will accept a single argument for a file that the applications will immediately open when they start. For example, if you're using Windows, create a text file named *C:\Users\Al\hello.txt* and then enter the following into the interactive shell:

```
>>> subprocess.run(['C:\\Windows\
\notepad.exe', 'C:\\Users\Al\\hello.txt'])
CompletedProcess(args=['C:\\Windows\
\notepad.exe', 'C:\\Users\\Al\\hello.txt'],
returncode=0)
```

This will not only launch the Notepad application but also have it immediately open the *C:\Users\Al\hello.txt* file. Every program has its own set of command line arguments, and some programs (especially GUI applications) don't use command line arguments at all.

The `subprocess.Popen()` function handles command line arguments in a similar way, and you should add them to the end of the list you pass the function.

Receiving Output Text from Launched Commands

You can also launch terminal commands using `subprocess.run()` and `subprocess.Popen()`. You may want your Python code to receive the text output of these commands or simulate keyboard input to them. For example, let's launch the `ping` command from Python and receive the text it produces. (The details of the `ping` command are beyond the scope of this book.) On Windows, you'll use the `-n 4` arguments to send four network "ping" requests that check whether the Nostarch.com server is online. If you're on macOS and Linux, replace `-n` with `-c`. This command takes a few seconds to run:

```
>>> import subprocess
>>>
proc = subprocess.run(['ping', '-n', '4',
'nostarch.com'], capture_output=True,
text=True)
```

```
>>> print(proc.stdout)
Pinging nostarch.com [104.20.120.46] with 32
bytes of data:
Reply from 104.20.120.46: bytes=32 time=19ms
TTL=59
Reply from 104.20.120.46: bytes=32 time=17ms
TTL=59
Reply from 104.20.120.46: bytes=32 time=97ms
TTL=59
Reply from 104.20.120.46: bytes=32 time=217ms
TTL=59

Ping statistics for 104.20.120.46:
    Packets: Sent = 4, Received = 4, Lost = 0
(0% loss),
Approximate round trip times in milli-
seconds:
    Minimum = 17ms, Maximum = 217ms, Average
= 87ms
```

When you pass the `capture_output=True` and `text=True` arguments to `subprocess.run()`, the text output of the launched program is stored as a string in the returned `CompletedProcess` object's `stdout` ("standard output") attribute. Your Python script can use the features of other programs and then parse the text output as a string.

Running Task Scheduler, launchd, and cron

If you're computer savvy, you may know about Task Scheduler on Windows, launchd on macOS, or the cron scheduler on Linux. These well-documented and reliable tools all allow you to schedule applications to launch at specific times.

Using your operating system's built-in scheduler saves you from writing your own clock-checking code to schedule your programs. If you just need your program to pause only briefly, however, it's best to instead loop until a certain date and time, calling `time.sleep(1)` on each iteration through the loop.

Opening Files with Default Applications

Double-clicking a *.txt* file on your computer will automatically launch the application associated with the *.txt* file extension. Each operating system has a program that performs the equivalent of this double-clicking action. On Windows, this is the `start` command. On macOS and Linux, this is the `open` command. Enter the following into the interactive shell, passing either `'start'` or `'open'` to `run()`, depending on your system. Also, on Windows, you should pass the `shell=True` keyword argument, as shown here:

```
>>> file_obj = open('hello.txt', 'w')  # Create a hello.txt file.
>>> file_obj.write('Hello, world!')
13
>>> file_obj.close()
>>> import subprocess
>>> subprocess.run(['start', 'hello.txt'], shell=True)
```

Here, we write `Hello, world!` to a new *hello.txt* file. Then, we call `run()`, passing it a list containing the program or command name (in this example, `'start'` for Windows) and the filename. The operating system knows all of the file associations and can figure out that it should launch, say, *Notepad.exe* to handle the *hello.txt* file on Windows.

Project 15: Simple Countdown

Just as it's hard to find a simple stopwatch application, it can be hard to find a simple countdown application. Let's write a countdown program that plays an alarm at the end of the countdown.

At a high level, here's what your program will do:

- Pause for one second in between displaying each number in the countdown by calling `time.sleep()`.
- Call `subprocess.run()` to open an *alarm.wav* sound file with the default application.

Open a new file editor tab and save it as *simplecountdown.py*.

Step 1: Count Down

This program will require the `time` module for the `time.sleep()` function and the `subprocess` module for the `subprocess.run()` function. Enter the following code and save the file as *simplecountdown.py*:

```
# https://autbor.com/simplecountdown.py - A
simple countdown script

import time, subprocess

❶ time_left = 60
while time_left > 0:
  ❷ print(time_left)
  ❸ time.sleep(1)
  ❹ time_left = time_left - 1

# TODO: At the end of the countdown, play a
sound file.
```

After importing `time` and `subprocess`, make a variable called `time_left` to hold the number of seconds left in the countdown ❶. We set it to `60` here, but you can change the value to whatever you'd like, or even set it based on a command line argument.

In a `while` loop, display the remaining count ❷, pause for one second ❸, and then decrement the `time_left` variable ❹ before the loop starts over again. The loop will keep looping as long as `time_left` is greater than `0`. After that, the countdown will be over. Next, let's fill in the `TODO` comment with code that plays the sound file.

Step 2: Play the Sound File

While Chapter 12 covers the `playsound3` module to play sound files of various formats, the quick and easy way to do this is to launch whatever application the user already uses to play sound files. The operating system will figure out from the *.wav* file extension which application it should launch to play the file. This *.wav* file could easily be some other sound file format, such as *.mp3* or *.ogg*. You can use any sound file on your computer to play at the end of the countdown, or download *alarm.wav* from *https://autbor.com/alarm.wav*.

Add the following to your code:

```
# https://autbor.com/simplecountdown.py - A
simple countdown script

--snip--

# At the end of the countdown, play a sound
file.
#subprocess.run(['start', 'alarm.wav'],
shell=True)  # Windows
#subprocess.run(['open', 'alarm.wav'])  #
macOS and Linux
```

After the `while` loop finishes, *alarm.wav* (or the sound file you choose) will play to notify the user that the countdown is over. Uncomment the `subprocess.run()` call for your operating system. On Windows, be sure to include `'start'` in the list you pass to `run()`. On macOS and Linux, pass `'open'` instead of `'start'` and remove `shell=True`.

Instead of playing a sound file, you could save a text file somewhere with a message like *Break time!* and use `subprocess.run()` to open it at the end of the countdown. This will effectively create a pop-up window with a message. Or you could use the `webbrowser.open()` function to open a specific website at the end of the countdown. Unlike some free countdown application you'd find online, your own countdown program's alarm can be anything you want!

Ideas for Similar Programs

A countdown essentially produces a simple delay before continuing the program's execution. You could use the same approach for other applications and features, such as the following:

- Use `time.sleep()` to give the user a chance to press CTRL-C to cancel an action, such as deleting files. Your program can print a "Press CTRL-C to cancel" message and then handle any

`KeyboardInterrupt` exceptions with `try` and `except` statements.

- For a long-term countdown, you can use `timedelta` objects to measure the number of days, hours, minutes, and seconds until some point (a birthday? an anniversary?) in the future.

Summary

The Unix epoch (January 1, 1970, at midnight, UTC) is a standard reference time for many programming languages, including Python. While the `time.time()` function returns an epoch timestamp (that is, a float value of the number of seconds since the Unix epoch), the `datetime` module is better for performing date arithmetic and formatting or parsing strings with date information.

The `time.sleep()` function will block (that is, not return) for a certain number of seconds. It can be used to add pauses to your program. But if you want to schedule your programs to start at a certain time, the instructions at *https://nostarch.com/automate-boring-stuff-python-3rd-edition* can tell you how to use the scheduler already provided by your operating system.

Finally, your Python programs can launch other applications with the `subprocess.run()` function. Command line arguments can be passed to the `run()` call to open specific documents with the application. Alternatively, you can use the `start` or `open` program with `run()` and use your computer's file associations to automatically figure out which application to use to open a document. By using the other applications on your computer, your Python programs can leverage their capabilities for your automation needs.

Practice Questions

1. What is the Unix epoch?
2. What function returns the number of seconds since the Unix epoch?
3. What `time` module function returns a human-readable string of the current time, like `'Mon Jun 15 14:00:38 2026'`?
4. How can you pause your program for exactly five seconds?
5. What does the `round()` function return?
6. What is the difference between a `datetime` object and a `timedelta` object?
7. Using the `datetime` module, what day of the week was January 7, 2019?

Practice Programs

For practice, write programs to do the following tasks.

Prettified Stopwatch

Expand the stopwatch project from this chapter so that it uses the `rjust()` and `ljust()` string methods to "prettify" the output. (These methods were covered in Chapter 8.) Instead of output such as this

```
Lap #1: 3.56 (3.56)
Lap #2: 8.63 (5.07)
Lap #3: 17.68 (9.05)
Lap #4: 19.11 (1.43)
```

the output should look like this:

```
Lap # 1:   3.56 (  3.56)
Lap # 2:   8.63 (  5.07)
Lap # 3:  17.68 (  9.05)
Lap # 4:  19.11 (  1.43)
```

Next, use the `pyperclip` module introduced in Chapter 8 to copy the text output to the clipboard so that the user can quickly paste the output to a text file or email.

Friday the 13th Finder

Friday the 13th is considered an unlucky day by those with triskaidekaphobia (though personally I celebrate it as a lucky day). With leap years and months of varying lengths, it can be hard to figure out when the next Friday the 13th is coming.

Write two programs. The first program should create a `datetime` object for the current day and a `timedelta` object of one day. If the current day is a Friday the 13th, it should print the month and year. Then, it should add the `timedelta` object to the `datetime` object to set it to the next day, and repeat the check. Have it repeat until it has found the next ten Friday the 13th dates.

The second program should do the same thing except subtract the `timedelta` object. This program will find all of the past months and years with a Friday the 13th, and stop when it reaches the year 1.