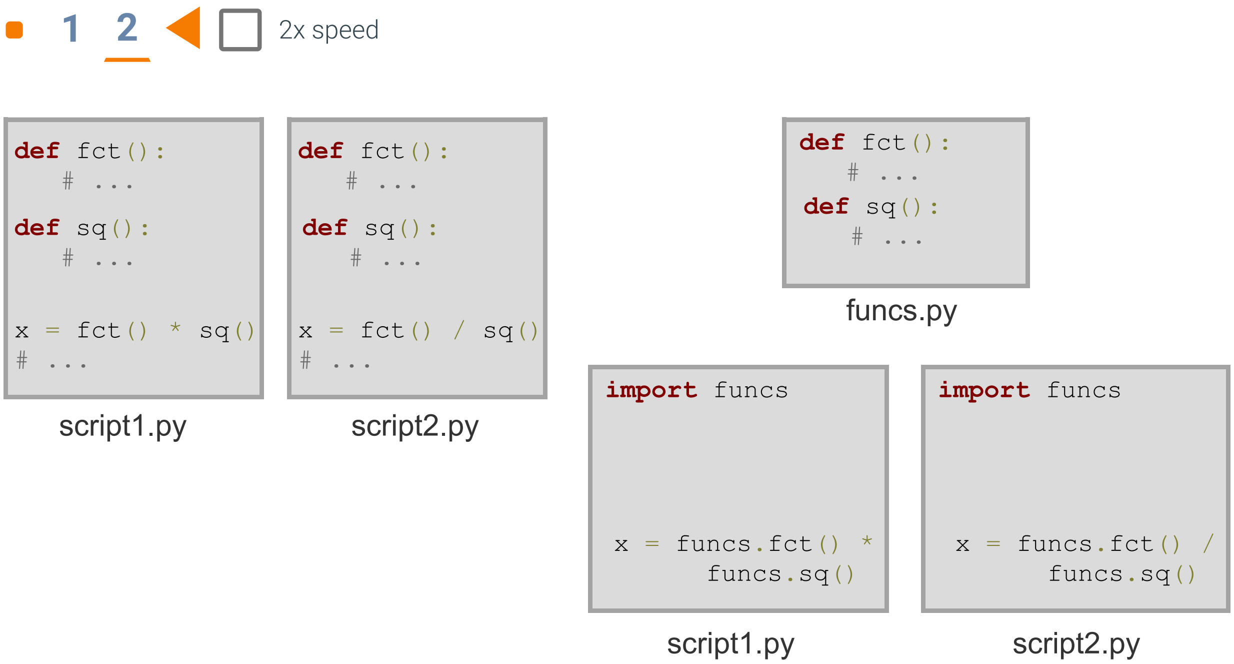
11.1 Modules

The interactive Python interpreter provides the most basic way to execute Python code. However, all of the defined variables, functions, classes, etc., are lost when a programmer closes the interpreter. Thus, a programmer will typically write Python code in a file, and then pass that file as input to the interpreter. Such a file is called a ***script***.

A programmer may find themselves writing the same function over and over again in multiple scripts, or creating very long and difficult to maintain scripts. A solution is to use a ***module***, which is a file containing Python code that can be imported and used by scripts, other modules, or the interactive interpreter. To ***import*** a module means to execute the code contained by the module, and make the definitions within that module available for use by the importing program.

PARTICIPATION ACTIVITY 11.1.1: A module is a file containing Python statements and definitions that can be used by other Python sources. ✓



```
def fct():
    # ...
def sq():
    # ...

x = fct() * sq()
# ...
```

```
def fct():
    # ...
def sq():
    # ...

x = fct() / sq()
# ...
```

```
def fct():
    # ...
def sq():
    # ...
```

```
import funcs


 x = funcs.fct() *
       funcs.sq()
```

```
import funcs


 x = funcs.fct() /
       funcs.sq()
```

The functions can instead be defined in another file. The file can be imported as a 'module'.

Feedback?

A module's filename should end with ".py"; otherwise, the interpreter will not be able to import the module. The module_name item should match the filename of the module, but without the .py extension. Ex: If a programmer wants to import a module whose filename is HTTPServer.py, the import statement `import HTTPServer` would be used. Note that import statements are case-sensitive; thus, `import ABC` is distinct from `import aBc`.

The interpreter must also be able to find the module to import. The simplest solution is to keep modules in the same directory as the executing script; however, the interpreter can also search the computer's file system for the modules. Later material covers these search mechanisms.

Good practice is to place import statements at the top of a file. There are few useful instances of placing import statements in any location other than the top. The benefit of placing import statements at the top is that a reader of the program can quickly identify the modules required for the program to run. A module being required by another program is often called a ***dependency***.

PARTICIPATION ACTIVITY 11.1.2: Basic importing of modules. ✓

1) A programmer using the interactive interpreter wants to use a function defined in the file tools.py. Write a statement that imports the content of tools.py into the interpreter. ✓

>>>

import tools

Check Show answer

Correct

import tools

The name of the file is used to identify the module.

2) A file containing Python code that is passed as input to the interpreter is called a ______? ✓

script

Check Show answer

Correct

Script

A script is the file being executed by the interpreter, which may import other files as modules.

3) A ______ is a file containing Python code that can be imported by a script.

module

Check Show answer

Correct

module

Modules are files containing Python code, and end with the .py extension. Modules can be imported by scripts, the interactive interpreter, and other modules.

Feedback?

Evaluating an import statement initiates the following process to load the module:

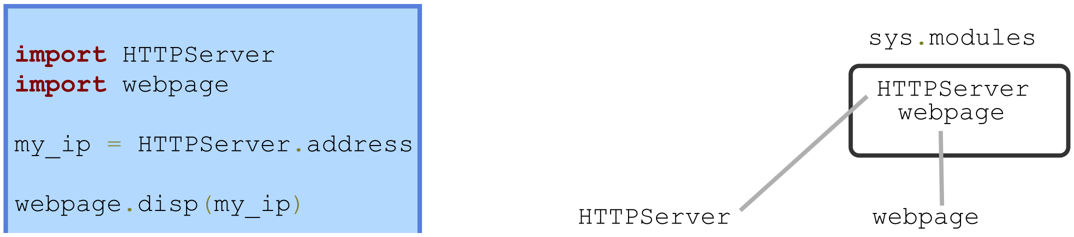
1. A check is conducted to determine whether the module has already been imported. If already imported, then the loaded module is used.
2. If not already imported, a new module object is created and inserted in sys.modules
3. The code in the module is executed in the new module object's namespace.

When importing a module, the interpreter first checks to see if that module has already been imported. A dictionary of the loaded modules is stored in ***sys.modules*** (available from the sys standard library module). If the module has not yet been loaded, then a new module object is created. A ***module object*** is simply a namespace that contains definitions from the module. If the module has already been loaded, then the existing module object is used.

If a module is not found in sys.modules, then the module is added and the statements within the module's code are executed. Definitions in the module's code, e.g., variable assignments and function definitions, are placed in the module's namespace. The module is then added to the importing script or module's namespace, so that the importer can access the definitions. The below animation illustrates.

PARTICIPATION ACTIVITY 11.1.3: Importing a module.

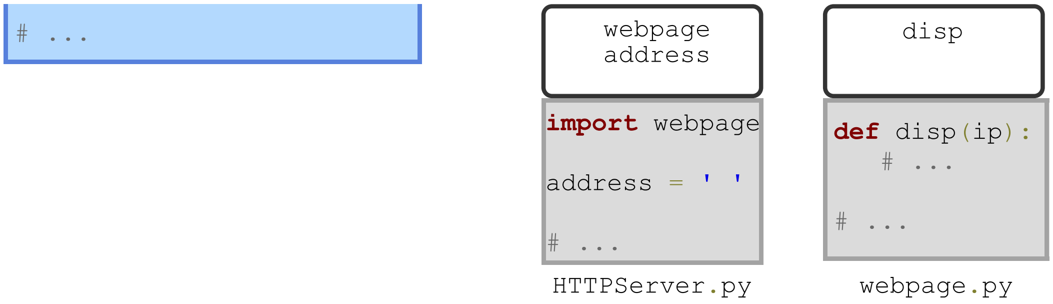
■ 1 2 3 4 5 ◀ □ 2x speed



```
import HTTPServer
import webpage

my_ip = HTTPServer.address

webpage.disp(my_ip)
```



```
# ...
```

```
import webpage

address = ' '

# ...
```

```
def disp(ip):
    # ...

# ...
```

webpage has already been imported. Existing module is loaded.

Feedback?

Executing `import HTTPServer` causes a new module object to be created and added to sys.modules. The code of HTTPServer is executed, which contains another import statement `import webpage`. Since webpage has not yet been imported, a second new module object is created and added to sys.modules. Execution of the webpage code occurs, which defines a function within the webpage module's namespace. Once the webpage module is successfully imported, the execution of HTTPServer's code continues, creating new definitions in the HTTPServer module's namespace. If the script attempts to import webpage, the already created module object is used.

PARTICIPATION ACTIVITY | 11.1.4: The importing process. ✓

Order the events as they occur when the statement `import HTTPServer` executes, assuming HTTPServer has not been previously imported.

sys.modules checked for HTTPServer	1st event	Correct
Module object created	2nd event	Correct
HTTPServer added to sys.modules	3rd event	Correct
HTTPServer code executed	4th event	Correct

HTTPServer added to importer's namespace	5th event	**Correct**

Reset

Feedback?

Once a module has been imported, the program can access the definitions of a module using attribute reference operations, e.g., `my_ip = HTTPServer.address` sets my_ip to address defined in HTTPServer.py. The definitions can also be overwritten, e.g., `HTTPServer.address = "www.yahoo.com"` binds address in HTTPServer to 'www.yahoo.com'. Note that such changes are temporary and restricted to the current executing Python instance. Ending the program and then re-importing the module would reload the original value of HTTPServer.address.

Consider a file my_funcs.py that contains the following:

Figure 11.1.1: Contents of my_funcs.py.

```
def factorial(num):
    """Calculates and returns the factorial (num!)"""
    x = 1
    for i in range(1, num + 1):
        x *= i

    return x
```

Feedback?

A programmer can then import my_funcs and use the factorial function as shown below:

Figure 11.1.2: Using factorial from my_funcs.py.

```
import my_funcs

n = int(input('Enter number: '))
fact = my_funcs.factorial(n)

for i in range(1, n + 1):
    print(i, end=' ')
```

```
Enter number: 5
1 * 2 * 3 * 4 * 5 = 120
...
Enter number: 3
```

```
    if i != n:
        print('*', end=' ')

print('=', fact)
```

```
Enter number: 3
1 * 2 * 3 = 6
```

Feedback?

PARTICIPATION ACTIVITY | 11.1.5: Basic usage of imported modules.

Consider a file shapes.py with the following contents:

```
cr = '#'

def draw_square(size):
    for h in range(size):
        for w in range(size):
            print(cr, end='')
        print()

def draw_rect(height, width):
    for h in range(height):
        for w in range(width):
            print(cr, end='')
        print()
```

1) Complete the import statement to import shapes.py

`import` shapes

Check Show answer

Correct

shapes

After importing, the definitions in shapes.py become available for use.

2) Complete the statement to call the draw_square function from the shapes module, passing an argument of 3.

`shapes.` draw_square(3)

Check Show answer

Correct

draw_square(3)

Dot notation is used to access items in the namespace of the imported module.

3) Write a statement that changes the output to use '$' when

Correct

the output to use '$' when drawing shapes. (Change the value of shapes.cr.)

shapes.cr = '$'

Check Show answer

```
shapes.cr = '$'
```

The character will be changed to '$'. Restarting the program and importing the module again would cause cr to have a value of '#' again.

Feedback?

11.2 Finding modules

Importing a module begins a search to find the corresponding file on the computer's file system. The interpreter first checks for a matching built-in module. A ***built-in module*** is a module that comes pre-installed with Python; examples of built-in modules include sys, time, and math. If no matching built-in module is found, then the interpreter searches the list of directories contained by ***sys.path***, located in the sys module. A programmer must be careful to not give a name to a module that is already used by a built-in module. In such cases, the interpreter would load the built-in module because built-in names are checked first.

The sys.path variable initially contains the following directories:

1. The directory of the executing script.
2. A list of directories specified by the environmental variable PYTHONPATH.
3. The directory where Python is installed (typically C:\Python27 or similar on Windows).

For simple programs, a module might simply be placed in the same directory. Larger projects might contain tens or hundreds of modules, or use third-party modules located in different directories. In such cases, a programmer might set the environmental variable ***PYTHONPATH*** in the operating system. An operating system ***environmental variable*** is much like a variable in a Python script, except that an environmental variable is stored by the computer's operating system and can be accessed by every program running on the computer. In Windows, a user can set the value of PYTHONPATH permanently through the control panel, or temporarily on a single instance of a command terminal (cmd.exe) using the command `set PYTHONPATH="c:\dir1;c:\other\directory"`.

PARTICIPATION ACTIVITY | 11.2.1: Finding modules. ✓

1) When an import statement executes, the interpreter immediately checks the current directory for a matching file. ✓

○ True

● False

Correct

The Python interpreter first checks for built-in modules with the same name.

● False

2) The environmental variable PYTHONPATH can be set to specify optional directories where modules are located.

● True

○ False

Correct

Those directories are checked only if no matching built-in module is found, and no module exists in the current directory.

✓

3) math.py is a good name for a new module.

○ True

● False

Correct

math is a built-in module, so a conflict would occur using that name.

✓

Feedback?

11.3 Importing specific names from a module

A programmer can specify names to import from a module by using the ***from*** keyword in an import statement:

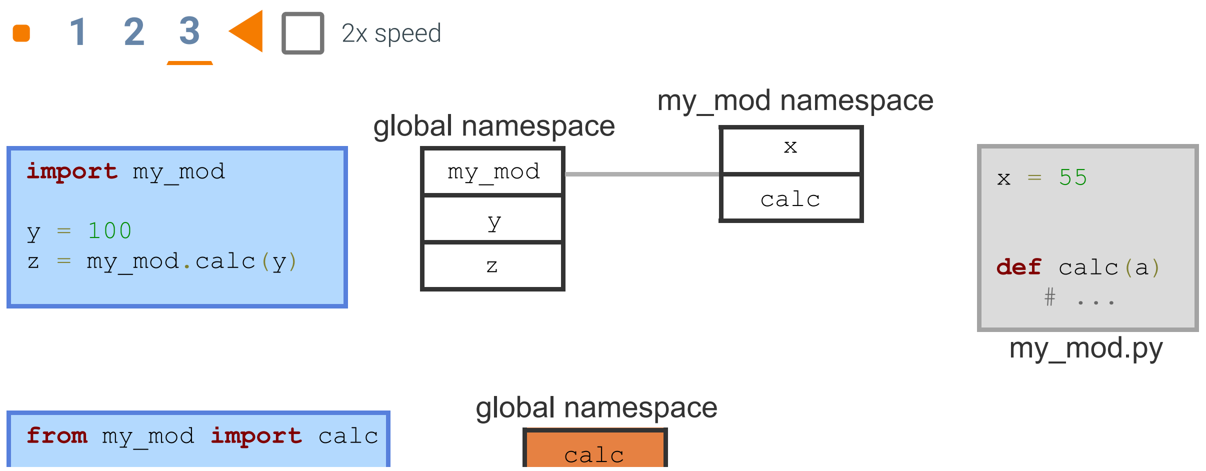
Construct 11.3.1: Importing specific names from a module.

```
from module_name import name1, name2, ...
```

Feedback?

A normal import statement, such as `import HTTPServer`, adds the new module into the global namespace, after which a programmer can access names through attribute reference operations (e.g., HTTPServer.address). In contrast, using `from` adds only the specified names. A statement such as `from HTTPServer import address` copies only the address variable from HTTPServer into the importing module's namespace. The following animation illustrates.

PARTICIPATION ACTIVITY 11.3.1: 'import x' vs 'from x import y'. ✓



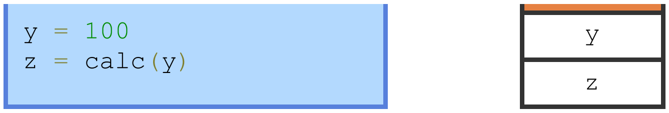
```
import my_mod

y = 100
z = my_mod.calc(y)
```

```
x = 55


def calc(a)
  # ...
```

```
from my_mod import calc
```



```
y = 100
z = calc(y)
```

From my_mod import, calc only copies calc from the my_mod namespace into the global namespace.

Feedback?

Using "from" changes how an imported name is used in a program.

Table 11.3.1: 'import module' vs. 'from module import names'.

Description	Example import statement	Using imported names
Import an entire module	`import HTTPServer`	`print(HTTPServer.address)`
Import specific name from a module	`from HTTPServer import address`	`print(address)`

Feedback?

The program below imports names from the ***hashlib*** module, a Python standard library module that contains a number of algorithms for creating a secure ***hash*** of a text message. A secure hash correlates exactly to a single series of characters. A sender of an email might create and send a secure hash along with the contents of the message. The email's recipient creates their own secure hash from the message contents and compares it to the received hash to detect if any the message was changed.

Figure 11.3.1: Using the from keyword to import specific names.

```
from hashlib import md5, sha1

text = input("Enter text to hash ('q' to quit): ")

while text != 'q':
    algorithm = input('Enter algorithm (md5/sha1): ')
    if algorithm == 'md5':
        output = md5(text.encode('utf-8'))
    elif algorithm == 'sha1':
        output = sha1(text.encode('utf-8'))
    else:
        output = 'Invalid algorithm selection'
    print('Hash value:', output.hexdigest())

    text = input("\nEnter text to hash ('q' to quit): ")
```

```
Enter text to hash ('q' to quit): Whether 'tis nobler in the mind to
suffer...
Enter algorithm (md5/sha1): md5
Hash value: 5b39e6686305363a2d60a4162fe3d012

Enter text to hash ('q' to quit): ...the slings and arrows of outrageous
fortune,...
Enter algorithm (md5/sha1): sha1
Hash value: 70c137974ad24691c1bb6cf8114aa2e3172ef910

Enter text to hash ('q' to quit): q
```

The hashlib library requires argument strings to md5 and sha1 be encoded; above, we encode the text using UTF-8 before passing to one of the hashing algorithms.

Feedback?

zyDE 11.3.1: Extending the hash example.

Improve the hashing example from above by adding a new algorithm. Import the sha224() function from hashlib, and extend the user interface to allow that function to be called.

Load default template...

```

# FIXME: Import sha224 also
from hashlib import md5, sha1, sha224

text = input("Enter text to hash ('q' to quit): ")

# Add sha224 to prompt
algorithm = input('\nEnter algorithm (md5/sha1): ')
if algorithm == 'md5':
    output = md5(text.encode('utf-8'))
elif algorithm == 'sha1':
    output = sha1(text.encode('utf-8'))
    # FIXME: Add check for sha224
```

```
"Simplicity is the key to brillia
sha224
```

Run

```
Enter text to hash (
Enter algorithm (md5
Hash value:
6a0b450dfcae1f7f894b
```

```
    # FIXME: Add check for sha224
elif algorithm == 'sha224':
    output = sha224(text.encode('utf-8'))
else:
    output = 'Invalid algorithm selection'

```

Feedback?

All names from a module can be imported directly by using a "*" character, as in the statement `from HTTPServer import *`. A common error is to use the import * syntax in modules and scripts, which makes identification of dependencies and the origins of variables difficult for a reader of the code to understand. Good practice is to limit the use of import * syntax to interactive interpreter sessions.

PARTICIPATION ACTIVITY 11.3.2: Importing specific names. ✓

my_funcs.py contains definitions for the factorial() and squared() functions.

1) Write a statement that imports only the function factorial from my_funcs.py.

factorial

Check Show answer

Correct

from my_funcs import factorial

factorial is added to the global namespace of the importing module, but not the entire my_funcs module.

✓

2) The following code uses functions defined in my_funcs.py. Complete the import statement at the top of the program.

import my_funcs

```
a = 5

print('a! =',
my_funcs.factorial(a))

print('a^2 =',
```

Correct

import my_funcs

The module is imported, and imported names are accessed using dot notation.

✓

```
my_funcs.squared(a))
```

Check Show answer

3) The following code uses functions defined in my_funcs.py. Complete the import statement at the top of the program.

factorial, squared

```
a = 5

print('a! =',
factorial(a))

print('a^2 =',
squared(a))
```

Check Show answer

✓

Correct

from my_funcs import factorial, squared

Only the specified names are imported from the module.

Feedback?

11.4 Executing modules as scripts

An import statement executes the code contained within the imported module. Thus, any statements in the global scope of a module, like printing or getting user input, will be executed when that module is imported. Execution of those statements may be an unintended side effect of the import. Commonly a programmer wants to treat a Python file as both a script executed by the interpreter and as an importable module. When used as an importable module, the file should not produce side effects when imported.

Ex: Consider the following Python file google_search.py, which contains functions for performing searches using the Google search engine. Executing the file as a script produces the following output:

Figure 11.4.1: web_search.py: Get the 1st page of results for a web search.

```
import urllib.request

def search(terms):
    """Do a fictional web engine search and
return the results"""
    html = _send_request(terms)
    results = _get_results(html)
    return results

def _send_request(terms):
    """Send search to fictional web search
engine and receive HTML response"""
    terms = terms.replace(' ', '%20')
#replace spaces

    url =
'http://www.search.fake.zybooks.com/search?
q=' + terms
    info = {'User-Agent': 'Mozilla/5.0'}
    req = urllib.request.Request(url,
headers=info)

    response = urllib.request.urlopen(req)
    html = str(response.read())
    return html

def _get_results(html):
    """
    Finds the links returned in 1st page of
results.
```

```
Enter search terms: Funny pictures of c
Found 7 links:
  icanhas.cheezburger.com/lolcats
  icanhas.cheezburger.com/
  www.funnycatpix.com/
  www.lolcats.com/
  www.buzzfeed.com/expresident/best-cat
  photobucket.com/images/lol%20cat
  https://www.facebook.com/pages/Funny-
Pics/204188529615813
```

```
    """
    start_tag = '<cite>'  # start of
results
    end_tag = '</cite>'   # Results end
with this tag
    links = []            # list of result
links

    start_tag_loc = html.find(start_tag)  #
find 1st link

    while start_tag_loc > -1:
        link_start = start_tag_loc +
len(start_tag)
        link_end = html.find(end_tag,
link_start)

links.append(html[link_start:link_end])
        start_tag_loc =
html.find(start_tag, link_end)

    return links

search_term  = input('Enter search terms:
')
result = search(search_term)

print('Found {}
links:'.format(len(result)))
for link in result:
    print(' ', link)
```

```
...

Enter search terms:  Videos of laughing
Found 4 links:
  www.godtube.com/watch/?v=W7ZP6WNX
  afv.com/funniest-videos-week-laughing
  www.today.com/.../laughing-baby-video
give-
you-giggles-t22521

www.personalgrowthcourses.net/video/bab
```

Note that the above program imports and uses the urllib module, which provides functions f
fetching URLs. urllib is not supported in the online interpreter of this material and the examp
for demo purposes only.

Feedback?

If another script imports google_search.py to use the search() function, the statements at the bottom of google_search.py will also execute[e]. The domain_freq.py file below tracks the frequency of specific domains in search results; however, importing google_search.py causes a search and listing of each site to unintentionally occur, because that search is called at the global scope of google_search.py.

Figure 11.4.2: domain_freq.py: Importing google_search causes unintended search to occur.

```
# Tracks frequency of
domains in Google searches
import google_search  #
Causes unintended search

domains = {}

terms = input("\nEnter
search terms ('q' to quit):
")
while terms != 'q':
    results =
google_search.search(terms)

    for link in results:
        if '.com' in link:
            domain_end =
link.find('.com')
        elif '.net' in
link:
            domain_end =
link.find('.net')
        elif '.org' in
link:
            domain_end =
link.find('.org')
        else:
            print('Unknown
top level domain')
            continue

        dom =
link[:domain_end + 4]
        if dom not in
domains:
            domains[dom] =
1
        else:
            domains[dom] +=
1

    terms = input("Enter
search terms ('q' to quit):
")

print('\nNumber of search
results for each site:')
for domain, num in
domains.items():
    print(domain + ':',
num)
```

```
Enter search terms: Music Videos
Google returned 9 links:
  http://www.mtv.com/music/videos/
  http://music.yahoo.com/videos/
  http://www.vh1.com/video/
  http://www.vevo.com/videos
  http://en.wikipedia.org/wiki/Music_video
  http://www.music.com/

http://www.youtube.com/watch%3Fv%3DSMpL6JKF5Ww
  http://www.bet.com/music/music-videos.html
  http://www.dailymotion.com/us/channel/music

Enter search terms ('q' to quit): Britney
Spears
Enter search terms ('q' to quit): Michael
Jackson
Enter search terms ('q' to quit): q

Number of search results for each site:
  http://www.people.com: 1
  http://www.britneyspears.com: 1
  http://www.imdb.com: 1
  http://www.michaeljackson.com: 1
  https://twitter.com: 1
  http://www.youtube.com: 3
  http://perezhilton.com: 1
  http://en.wikipedia.org: 2
  http://www.tmz.com: 2
  http://www.mtv.com: 2
  http://www.biography.com: 1
  https://www.facebook.com: 1
```

Feedback?

A file can better support being executed as both a script and an importable module by utilizing the __name__ special name. ***__name__*** is a global string variable automatically added to every module that contains the name of the module. Ex: my_funcs.__name__ would have the value "my_funcs", and google_search.__name__ would have the value "google_search". (Note that __name__ has two underscores before name and two underscores after.) However, the value of __name__ for the executing script is always set to "__main__" to differentiate the script from imported modules. The following comparison can be performed:

Figure 11.4.3: Checking if a file is the executing script or an imported module.

```
if __name__ == "__main__":
    # File executed as a script
```

Feedback?

If `if __name__ == "__main__"` is true, then the file is being executed as a script and the branch is taken. Otherwise, the file was imported and thus __name__ is equal to the module name, e.g., "google_search".

The contents of the branch typically include a user interface to functions or class definitions within the file. A user can execute the file as a script and interact with the user interface, or another script can import the file just to use the definitions. The google_search.py file is modified below to fix the unintentional search.

Figure 11.4.4: google_search.py modified to run as either script or module.

Each file below is executed as a script.

domain_freq.py

```
# Tracks frequency of domains
in Google searches
import google_search

domains = {}

terms = input("Enter search
terms ('q' to quit): ")
```

google_search.py

```
import urllib.request

def search(terms):
    # ...

def _send_request(terms):
    # ...
```

```
while terms != 'q':
    results =
google_search.search(terms)

    # ...

print('\nNumber of search
results for each site:')
for domain, num in
domains.items():
    print(domain + ':', num   )
```

```
def _get_results(html):
    # ...

if __name__ == "__main__":
    search_term  = input('Enter search
terms:\n')
    result = search(search_term)

    print('Google returned %d links:' %
len(result))
    for link in result:
        print(' ', link)
```

```
Enter search terms ('q' to
quit): Britney Spears
Enter search terms ('q' to
quit): Michael Jackson
Enter search terms ('q' to
quit): q

Number of search results for
each site:
  http://www.people.com: 1

http://www.britneyspears.com:
1
  http://www.imdb.com: 1

http://www.michaeljackson.com:
1
  https://twitter.com: 1
  http://www.youtube.com: 3
  http://perezhilton.com: 1
  http://en.wikipedia.org: 2
  http://www.tmz.com: 2
  http://www.mtv.com: 2
  http://www.biography.com: 1
  https://www.facebook.com: 1
```

```
Enter search terms: Music Videos
Google returned 9 links:
  http://www.mtv.com/music/videos/
  http://music.yahoo.com/videos/
  http://www.vh1.com/video/
  http://www.vevo.com/videos
  http://en.wikipedia.org/wiki/Music_video
  http://www.music.com/

http://www.youtube.com/watch%3Fv%3DSMpL6JKF5Ww
  http://www.bet.com/music/music-videos.html
  http://www.dailymotion.com/us/channel/music
```

Feedback?

The google_search.py file has been modified to compare __name__ to "__main__". When the file is executed as a script, a single search request is made and the results are displayed. Executing domain_freq.py imports google_search, which now does not perform the initial search because __name__ is equal to "google_search".

PARTICIPATION ACTIVITY 11.4.1: Executing modules as scripts.

1) Importing a module executes the

Correct

executes the statements contained within the imported module.

● True

○ False

A file imported as a module should not contain statements at the global scope that create side effects when importing that module, such as printing output.

2) The value of the __name__ variable of the executing script is always "__main__".

● True

○ False

✓

Correct

__main__ is added automatically to every module.

3) If a module is imported with the statement `import my_mod`, then my_mod.__name__ is equal to "__main__".

○ True

● False

✓

Correct

The __name__ variable of any imported module contains the name of the module, thus my_mod.__name__ is equal to "my_mod".

Feedback?

11.5 Reloading modules

Sometimes a Python program imports a module, but then the source code of the imported module needs to be changed. Since modules are executed only once when imported, changing the module's source does not immediately affect the running Python instance. Instead of restarting the entire Python program, the ***reload()*** function can be used to reload and re-execute the changed module. The reload() function is located in the imp standard library module.

Consider the following module, which can send an email using a Google gmail account:

Figure 11.5.1: send_gmail.py: Sends a single email through gmail.

```
import smtplib
from email.mime.text import
MIMEText

header = 'Hello. This is an
automated email.\n\n'

def send(subject, to, frm, text):
    # The message to send
    msg = MIMEText(header + text)
    msg['Subject'] = subject
    msg['To'] = to
    msg['From'] = frm

    # Connect to gmail's email
server and send
    s =
smtplib.SMTP('smtp.gmail.com',
port=587)
    s.ehlo()
    s.starttls()
    s.login(user=frm,
password='password')
    s.sendmail(frm, [to],
msg.as_string())
    s.quit()

if __name__ == "__main__":
    send(
        subject='A coupon for
you!',

to='billgates@microsoft.com',

frm='JohnnysHotDogs1@gmail.com',
```

Executing send_gmail.py as a script sends the message:

```
To: billgates@microsoft.com
From: JohnnysHotDogs1@gmail.com
Subject: A coupon for you!

Hello. This is an automated email.

Enjoy!
```

```
        text='Enjoy!')
```

Feedback?

The send_coupons.py script below imports send_gmail.py as a module, using the send function to deliver important messages to customers.

Figure 11.5.2: send_coupons.py: Automates emails to loyal customers.

```
import os
from imp import reload
import send_gmail

mod_time = os.path.getmtime(send_gmail.__file__)

emails = [  # Could be large list or stored in file
    'billgates@microsoft.com',
    'president@whitehouse.gov',
    'benedictxvi@vatican.va'
]

my_email = 'JohnnysHotDogs1@gmail.com'
subject = 'A coupon for you!'
text = ("As a loyal customer of Johnny's HotDogs, "
        "here is a coupon for 1 free bratwurst!")

for addr in emails:
    send_gmail.send(subject, addr, my_email, text)

    # Check if file has been modified
    last_mod = os.path.getmtime(send_gmail.__file__)
    if last_mod > mod_time:
        mod_time = last_mod
        reload(send_gmail)
```

Feedback?

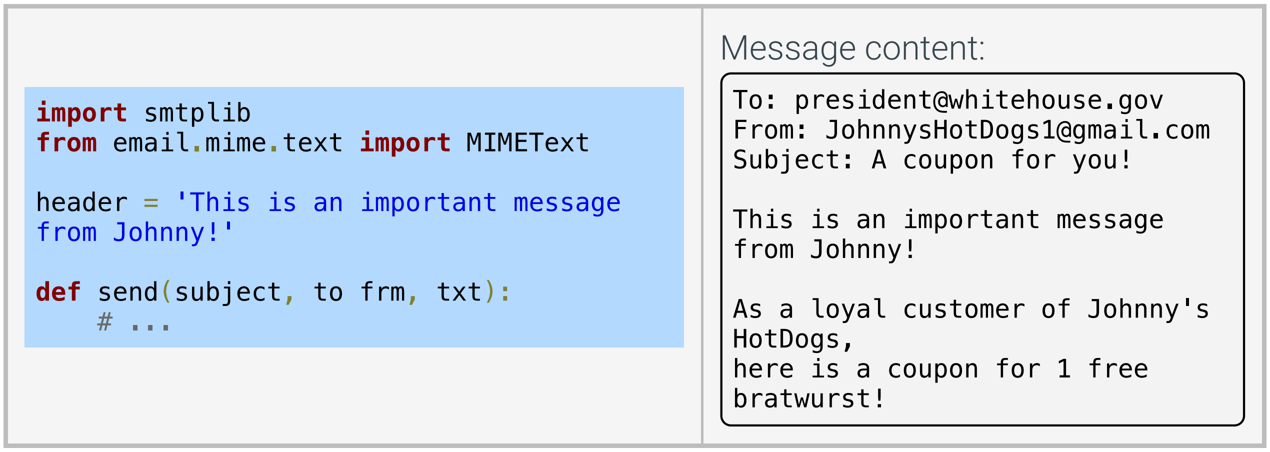
If thousands of emails are being sent, the program should not be stopped because rerunning the program could cause duplicate emails to be sent to some users, and Johnny's HotDogs might annoy their customers. If Johnny wants to change the content of the header string in the send_gmail module without stopping the program, then the variable's value in send_gmail.py's *source code* can be updated and reloaded.

When send_coupons.py imports send_gmail, a global variable mod_time stores the time

When send_coupons.py imports send_gmail, a global variable mod_time stores the time when send_gmail.py was last modified, using the os.path.getmtime() function. The ***__file__*** special name contains the path to a module in the computer file system, e.g., the value of send_gmail.__file__ might be "C:\\Users\\Johnny\\send_gmail.py". A comparison is made to the original modification time at the end of the for loop – if the modification time is greater than the original, then the module's source code has been updated and the module should be reloaded.

Modifying the header string in send_gmail.py to *"This is an important message from Johnny"* while the program is running causes the module to be reloaded, which alters the contents of the emails.

Figure 11.5.3: Modifying send_gmail.py while the program is running updates the email contents.



```
import smtplib
from email.mime.text import MIMEText

header = 'This is an important message
from Johnny!'

def send(subject, to frm, txt):
    # ...
```

Message content:

```
To: president@whitehouse.gov
From: JohnnysHotDogs1@gmail.com
Subject: A coupon for you!

This is an important message
from Johnny!

As a loyal customer of Johnny's
HotDogs,
here is a coupon for 1 free
bratwurst!
```

Feedback?

The reload function reloads a module in-place. When reload(send_gmail) returns, the namespace of the send_gmail module will contain updated definitions. The call to send_gmail.send() still accesses the same send_gmail module object, but the definition of send() will have been updated.

Importing attributes directly using "from", and then reloading the corresponding module, will *not* update the imported attributes.

Figure 11.5.4: Reloading modules doesn't affect attributes imported using 'from'.

```
from imp import reload
import send_gmail
from send_gmail import header

print('Original value of header:', header)

print('\n(---- send_gmail.py source code edited ----)')

print('\nReloading send_gmail\n')
reload(send_gmail)

print('header:', header)
print('send_gmail.header:', send_gmail.header)
```

```
Original value of header: Hello. This is an automated email.

(---- send_gmail.py edited ----)

Reloading send_gmail.

header: Hello. This is an automated email.
send_gmail.header: Hello from Johnny's Hotdogs!
```

Feedback?

Reloading modules is typically useful in long-running programs, when restarting and initializing the entire program may be an expensive operation. A common scenario is a web server that is communicating with multiple clients on the internet. Instead of restarting the server and disconnecting all of the clients, a single module can be reloaded dynamically as the server runs.

PARTICIPATION ACTIVITY 11.5.1: Reloading modules. ✓

1) Modules cannot be reloaded if they have already been imported. ✓

○ True

● False

Correct

The purpose of the reload function is to reload modules that have changed since being imported.

2) The reload function modifies a module in-place. ✓

Correct

reload() also does return the module, but it can be ignored

● True

○ False

ignored.

3) Reloading a module is useful when restarting a program is prohibitively costly.

● True

○ False

Correct

Reloading is often done in server programs, where shutting down a server may disconnect clients.

✓

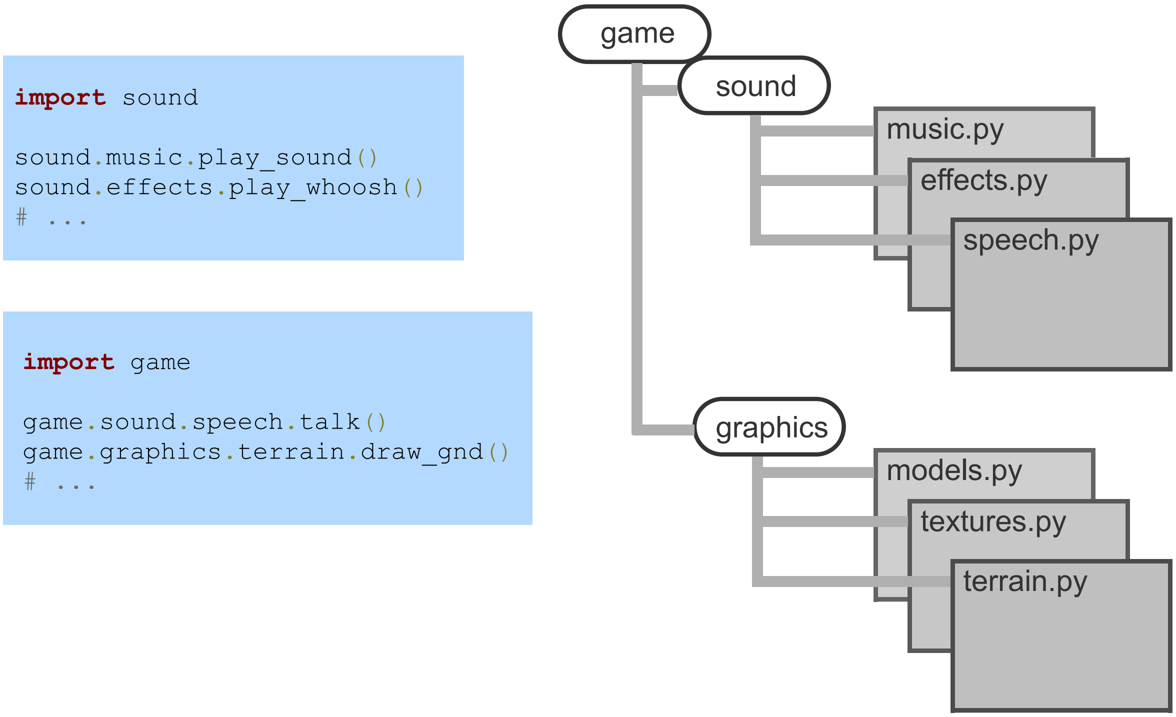
Feedback?

11.6 Packages

Instead of importing a single module at a time, an entire directory of modules can be imported all at once. A ***package*** is a directory that, when imported, gives access to all of the modules stored in the directory. Large projects are often organized using packages to group related modules.

PARTICIPATION ACTIVITY 11.6.1: Packages group related modules together. ✓

1 2 □ 2x speed



```
import sound

sound.music.play_sound()
sound.effects.play_whoosh()
# ...
```

```
import game

game.sound.speech.talk()
game.graphics.terrain.draw_gnd()
# ...
```

Packages can contain subpackages. The 'game' package contains subpackages 'sound' and 'graphics'.

Feedback?

To import a package, a programmer writes an import statement and gives the name of the directory where the package is located. To indicate that a directory is a package, the directory must include a file called ***__init__*.py_**. The __init__.py file often is empty, but may contain code to initialize the package when imported. The interpreter searches for the package in the directories listed in sys.path.

Consider the following directory structure. A package ASCIIArt contains a canvas module, as well as the subpackages figures and buildings.

Figure 11.6.1: Directory structure.

```
draw_scene.py            Script that imports ASCIIArt package
ASCIIArt\                     Top-level package
        __init__.py
        canvas.py
        figures\              Subpackage for figures art
               __init__.py
               man.py
               cow.py
               ...
        buildings\            Subpackage for buildings art
               __init__.py
               barn.py
               house.py
               ...
```

Feedback?

The draw_scene.py script can import the ASCIIArt package using the following statement:

Figure 11.6.2: Importing the ASCIIArt package.

```
import ASCIIArt  # import ASCIIArt package
```

Feedback?

Specific modules or subpackages can be imported individually by specifying the path to the item, using periods in the import name. References to names within the imported module require that the entire path is specified:

Figure 11.6.3: Importing the canvas module.

```
import ASCIIArt.canvas  # imports the canvas.py module
```

```
ASCIIArt.canvas.draw_canvas()  # Definitions in canvas.py have full name specified
```

Feedback?

The *from* technique of importing also works with packages, allowing individual modules or subpackages to be directly imported into the global namespace. A benefit of this method is that higher level package names need not be specified.

Figure 11.6.4: Import cow module from figures subpackage.

```
from ASCIIArt.figures import cow  # import cow module
cow.draw()  # Can omit ASCIIArt.figures prefix
```

Feedback?

Even individual names from a module can be imported, making that name directly available.

Figure 11.6.5: Import the draw function from the cow module.

```
from ASCIIArt.figures.cow import draw  # import draw() function
draw()  # Can omit ASCIIArt.figures.cow
```

Feedback?

When using syntax such as "import y.z", the last item z must be a package, a module, or a subpackage. In contrast, when using "from x import y.z", the last item z can also be a name from y, such as a function, class, or global variable.

Using packages helps to avoid module name collisions. For example, consider if another package called 3DGraphics also contained a module called canvas.py. Though both modules share a name, they are differentiated by the package that contains them, i.e., ASCIIArt.canvas is different from 3DGraphics.canvas. A programmer should take care when using the *from* technique of importing. A common error is to overwrite an imported module with another package's identically named module.

PARTICIPATION ACTIVITY | 11.6.2: Importing packages. ✓

Consider the directory structure of the ASCIIArt package above.

1) Write a statement to import the figures subpackage.

```
import
```

ASCIIArt.figures

Check Show answer

Correct

ASCIIArt.figures

The buildings subpackage is imported.

✓

2) Write a statement to import the cow module.

```
import
```

ASCIIArt.figures.cow

Check Show answer

Correct

ASCIIArt.figures.cow

The cow module is imported.

✓

3) Write a statement that calls the draw() function of the imported house module.

```
from
ASCIIArt.buildings.house
import draw
```

draw()

Check Show answer

Correct

draw()

Importing a module's function using 'from' makes that function directly callable; module and package dot notation is omitted.

✓

4) Write a statement that imports the barn module directly using the 'from' technique of importing.

from ASCIIArt.buildings import barn

Check

Show answer

Feedback?

11.7 Standard library

Python includes by default a collection of modules that can be imported into new programs. The ***Python standard library*** includes various utilities and tools for performing common program behaviors. Ex: The *math* module provides progress mathematical functions, the *datetime* module provides date and calendar capabilities, the *random* module can produce random numbers, the *sqlite3* module can be used to connect to SQL databases, and so on. Before starting any new project, good practice is to research what is available in the standard library, or on the internet, to help complete the task. Methods to find many more useful modules made available on the internet by other programmers are discussed in another section.

Commonly used standard library modules are listed below.

Table 11.7.1: Some commonly used Python standard library modules.

Module name	Description	Documentation link
datetime	Creation and editing of dates and times objects	https://docs.python.org/3.5/library/datetime.html
random	Functions for working with random numbers	https://docs.python.org/3.5/library/random.html
copy	Create complete copies of objects	https://docs.python.org/3.5/library/copy.html
time	Get the current time, convert time zones, sleep for a number of seconds	https://docs.python.org/3.5/library/time.html

math	Mathematical functions	https://docs.python.org/3.5/library/math.html
os	Operating system informational and management helpers	https://docs.python.org/3.5/library/os.html
sys	System specific environment or configuration helpers	https://docs.python.org/3.5/library/sys.html
pdb	The Python interactive debugger	https://docs.python.org/3.5/library/pdb.html
urllib	URL handling functions, such as requesting web pages	https://docs.python.org/3.5/library/urllib.html

Feedback?

Examples of standard library module usage is provided below.

Figure 11.7.1: Using the datetime module.

The *datetime* module prints the day, month, and year of a date that is a user-entered number of days in the future.

```
import datetime

# Create a new date object representing the current
date (May 30, 2016)
today   = datetime.date.today()

days_from_now = int(input('Enter number of days
from now: '))

# Create a new timedelta object that represents a
```

```
Enter number of days
from now: 30
```

```
difference in the
# number of days between dates.
day_difference = datetime.timedelta(days =
days_from_now)

# Calculate new date
future_date = today + day_difference

print(days_from_now, 'days from now is',
future_date.strftime('%B %d, %Y'))
```

```
30 days from now is
June 29, 2016
```

Feedback?

Figure 11.7.2: Using the random module.

The *random* module is used to implement a simple game where a user continues to draw from a deck of cards until an ace card is found.

```
import random

# Create a shuffled card deck with 4 suites of
cards 2-10, and face cards
deck = [2, 3, 4, 5, 6, 7, 8, 9, 10, 'J', 'Q',
'K', 'A'] * 4
random.shuffle(deck)

num_drawn = 0
game_over = False
user_input = input("Press any key to draw a
card ('q' to quit): ")
while user_input != 'q' and not game_over:

    # Draw a random card, and remove card from
the deck
    card = random.choice(deck)
    deck.remove(card)
    num_drawn += 1

    print('\nCard drawn:', card)

    # Game is over if an ace was drawn
    if card == 'A':
        game_over = True
    else:
        user_input = input("Press any key to
draw a card ('q' to quit): ")

if user_input == 'q':
    print("\nGame was quit")
else:
    print(num_drawn, 'card(s) drawn to find an
ace.')
```

```
Press any key to draw a
card ('q' to quit): g
Card drawn: 10 card
Press any key to draw a
card ('q' to quit): g
Card drawn: 5 card
Press any key to draw a
card ('q' to quit): g
Card drawn: K card
Press any key to draw a
card ('q' to quit): g
Card drawn: 9 card
Press any key to draw a
card ('q' to quit): g
Card drawn: A card
5 cards were drawn to find
an ace.
```

Feedback?

PARTICIPATION ACTIVITY

11.7.1: A few standard library modules.

Match the program behavior to a standard library module that might be used to implement the desired program.

random

A trivia game generates a new question at random time intervals.

random.random() or random.randint() would be useful to generate random time intervals.

Correct

urllib

Retrieve the contents of the webpage zybooks.com.

urllib.request can be used to open a URL and retrieve contents of a webpage. Review the documentation of the module for more details.

Correct

os

Get the name of the current operating system.

The os module has many useful functions, such as changing file permissions, creating new directories, and so on. In this case, os.name would be useful to retrieve the name of the operating system.

Correct

math

Compute the mathematical cosine function of a user-entered number of degrees.

The math module implements math.cos().

Correct

Reset

Feedback?

Review all of the standard library

This section describes a small subset of the features provided by the standard library. The standard library documentation provides a full list of available modules.

11.8 Third-party libraries

Third-party libraries

While the Python Standard Library includes extensive functionality, there are many specialized tasks that it does not support. There are numerous third-party libraries available that can be imported into Python to support such tasks. Of particular note are pandas, a high-performance data analytics library that includes dataframes, and NumPy, a powerful scientific computing package.

Pandas is widely used to analyze data in Python. It can take existing data and create a Python object with rows and columns called a data frame. The existing data could be a CSV file, SQL database, or a simple dictionary.

Figure 11.8.1: Example of creating a tabular data structure.

```python
import pandas
data = {'name': ['Connie', 'Jessica', 'Dana'], 'extension': [4682, 4198, 4351]}
dataF = pandas.DataFrame(data, columns=['name', 'extension'])
```

Feedback?

NumPy is used to provide multidimensional arrays and tools used to work with these arrays.

Figure 11.8.2: Example of creating a NumPy array and accessing elements within it.

```python
import numpy as np

a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))                # Prints "<class 'numpy.ndarray'>"
print(a.shape)                # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                      # Change an element of the array
print(a)                      # Prints "[5, 2, 3]"
```

Feedback?

You can install third-party libraries with a package manager such as pip (included with Python 3) or Anaconda (widely used among data scientists). Once you have installed the desired third-party libraries, simply initialize and use them as you would any other package.

When faced with a task you are unsure how to accomplish or the need to troubleshoot a problem, research how others have solved similar issues. There is likely a library or module that will help!

Python's tutorial and documentation are great starting places for research. Make sure you are looking at the documentation for the Python version you are using. Third-party libraries publish their own documentation on an associated website, so bookmark the source of any library you are working with so you have it readily available for research as well.

Coworkers, fellow programmers, experts, authors, and bloggers are often great sources of information: try to stay active in the Python community so you can learn to recognize reliable sources within the community. Stackoverflow is an active, well-populated forum, and often others will have previously asked the question that you have. Review multiple responses to ensure you find quality solutions.

Finally, the web contains plenty of other possible sources of information: bug trackers, short-lived blogs, various specialized forums, and so forth. Be careful of advice from sources with which you are unfamiliar.

11.9 Standard library practice

Standard Library examples

You learned about several commonly used Python Standard Library modules and will now practice with various library functions on your own. The following figures show ways that Python Standard Libraries may be imported and used. There are many functionalities and additional libraries not shown. It is important to use the Python documentation to research what is available.

Figure 11.9.1: Using time library to access current local time.

```
import time
localtime = time.asctime( time.localtime(time.time()) )
print( "Local current time is:", localtime )
```

Feedback?

The time library is used in figure 11.9.1 to access the current time as time.time() and convert it to the local time. This is not an easily readable format, so asctime() is used to convert to a more readable string.

In this example, the entire time library is imported and all functions of the library are available. It is also possible to only import a specific object from the library, limiting the size of the module imported and the available functionality.

Figure 11.9.2: Selecting randomized items from a list.

```
from random import choices
items = [12,6,4,18,3,5,16]
selection = choices(items, k=2)
print(selection)
```

Feedback?

In figure 11.9.2 the choices function is imported from the random library and used to select two numbers from the list at random. Run this code several times and notice that each time different values are printed for the selection.

Figure 11.9.3: Using datetime to access the current date.

```
import datetime
print(datetime.date.today())
```

Feedback?

Python's datetime library has objects available to access and calculate time- and date-related data. In figure 11.9.3 the date object is referenced first from within the datetime library. Then the today() object method is called to provide the current date.

Datetime has objects for date, time, datetime, and timedelta. Each of these objects has available attributes and methods. The Python Standard Library has a complete listing of objects, attributes, and methods. It is important to remember that you must first access the object, then the method.

Figure 11.9.4: Calculating a future date.

```
import datetime
span = datetime.timedelta(days=7)
today = datetime.date.today()
futuredate = today + span
print("One week from now will be: {}".format(futuredate))
```

Feedback?

In figure 11.9.4 the timedelta object is used to create a duration of seven days. That time duration is then added to the current date to accurately calculate the future date seven days from the current date from the current date.

Figure 11.9.5: Calculate area of a circle.

```
from math import pi, ceil
def calcArea(radius):
        area = pi * (radius**2)
        return ceil(area)
print(calcArea(7))
print(calcArea(3))
```

Feedback?

The math library is helpful in performing more advanced mathematical functions. Figure 11.9.5 shows how to use the math constant pi to calculate the area of a circle for any given radius. The ceil() function is used to round the decimal area up to the nearest integer.

Standard library details

For additional review of the Python Standard Library, watch this Lynda.com series: Learning the Python 3 Standard Library.

Help method

Previously you learned how the help() function can be used to find documentation associated with an object. This function is also useful for locating documentation related to objects within libraries.

Run the following code on your own and notice the full documentation it produces of all methods available within the math module.

Figure 11.9.6: Example of applying help to an object.

```
import math
help(math)
```

Feedback?

This is useful when you are not sure which method would solve your problem. Other times you may know which method you want to use but are unsure how to format and implement it. Run the following code to see how help() is used to look up the choice method from the random library.

Figure 11.9.7: Example of applying help to a specific library method.

```
import random
help(random.choices)
```

Feedback?

Practice problems

You may use a web-based version of Python such as PyFiddle, Repl.It, or PythonFiddle to complete the exercises. Just make sure you are using version 3 or greater. If you use a web-based Python environment, you can easily share code with course instructors if you need help. You may also use a local installation of Python.

You should copy the entire code snippet into your Python editor. Your code should be placed in the area that says "student code goes here" highlighted in yellow. This is inside the function. When you run this entire code snippet, there are test cases below your code. Your output should match the expected output.

Task 1

Complete the function that takes an integer as input and returns the factorial of that integer

```
from math import factorial

def calculate(x):
# Student code goes here

print(calculate(3)) #expected outcome: 6
print(calculate(9)) #expected outcome: 362880
```

Task 2

Complete the function that takes a list as input and returns a randomized item from that list

```
 import random as r

def selection(x):
# Student code goes here

print(selection(['apple','banana','orange','grape']))
print(selection([7,5,3,9,12,4,8,10]))
```

Task 3

Complete the function that takes as input an integer for a number of days and prints the total number of seconds in that number of days

```
 import datetime

def currentDate(x):
# Student code goes here

currentDate(4) #expected outcome: The total number of seconds is 345600.0.
currentDate(7) #expected outcome: The total number of seconds is 604800.0.
```

Task 4

Complete the function to return the current date

```
 import datetime as dt

def currentDate():
# Student code goes here

print(currentDate()) #Expected outcome will vary, but should follow this format: The
current date is 9-11-2018.
```

Task 5

Complete the function that takes an integer as input, multiplies by e, and returns result rounded up

```
 from math import e,ceil

def mathCalculation(x):
# Student code goes here

#expected outcome: 9
print(mathCalculation(3))

#expected outcome: 25
print(mathCalculation(9))
```

Task 6

Complete the function to return the number of leap years in the list

```
import calendar

# Complete the function to return the number of leap years in the list
def countLeapYears(yearList):
# Student code goes here

# expected output: 2
print(countLeapYears([2001, 2018, 2020, 2090, 2233, 2176, 2200, 2982]))

# expected output: 4
print(countLeapYears([2001, 2018, 2020, 2092, 2204, 2176, 2200, 2982]))
```

Task 7

Complete the function to print the full name of the month using the calendar library

```
import calendar

# Complete the function to print the full name of the month using the calendar library
def printMonthName(monthNum):
# Student code goes here

# expected output: March
printMonthName(3)

# expected output: November
printMonthName(11)
```

Task 8

Complete the function to print the full name of the day of the week

```
import calendar, datetime

# Complete the function to print the full name of the day of the week
def printWeekdayName(year, month, day):
# Student code goes here

# expected output: Friday
printWeekdayName(2001, 8, 31)

# expected output: Monday
printWeekdayName(2018, 10, 1)
```

Task 9

Complete the following function to return a random number between 5 and 8 exclusive

```
import random

# Complete the following function to return a random number
# between 5 and 8 exclusive
def getRandom():
# Student code goes here

# expected output: You should only get 5s, 6s, and 7s
for i in range(10):
    print(getRandom())
```

Task 10

Complete the function to add 90 days to the given date and return the new date

```
import datetime

# Complete the function to add 90 days to the given date and return the new date
def add90Days(someDate):
# Student code goes here

# expected output: 2018-12-30
print(add90Days(datetime.date(2018, 10, 1)))

# expected output: 2015-05-12
print(add90Days(datetime.date(2015, 2, 11)))
```