29.1 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 | 29.1.1: A module is a file containing Python statements and definitions that can be used by other Python sources.

Animation captions:

1. A programmer writes scripts containing functions and code using those functions. Multiple scripts might define the same functions.
2. The functions can instead be defined in another file. The file can be imported as a 'module'.

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 | 29.1.2: 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.

>>>

Check Show answer

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

Check Show answer

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

Check Show answer

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 | 29.1.3: Importing a module.

Animation captions:

1. sys.modules checks for HTTPServer. A new module object is created. The module is then inserted into sys.modules.
2. HTTPServer's code is executed in module namespace. sys.modules checks for webpage. The new module object is created and inserted in sys.modules.
3. webpage's code is executed in module namespace. webpage is added to HTTPServer namespace.
4. HTTPServer's code continues executing.
5. webpage has already been imported. Existing module is loaded.

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 | 29.1.4: The importing process.

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

If unable to drag and drop, refresh the page.

HTTPServer code executed **HTTPServer added to sys.modules**

HTTPServer added to importer's namespace **sys.modules checked for HTTPServer**

Module object created

1st event

2nd event

3rd event

4th event

5th event

Reset

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 29.1.1: 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
```

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

Figure 29.1.2: 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=' ')
    if i != n:
        print('*', end=' ')

print('=', fact)
```

```
Enter number: 5
1 * 2 * 3 * 4 * 5 =
120
...
Enter number: 3
1 * 2 * 3 = 6
```


PARTICIPATION ACTIVITY 29.1.5: 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`

Check Show answer

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

shapes.

Check Show answer

3) Write a statement that changes the output to use '$' when drawing shapes. (Change the value of shapes.cr.)

Check Show answer

CHALLENGE ACTIVITY | 29.1.1: Enter the output of modules.

422102.2723990.qx3zqy7

Start

Type the program's output

main.py | arithmetic.py

```
import arithmetic

def calculate(number):
    return number - 5

print(calculate(4))
print(arithmetic.calculate(4))
```

1 2 3

Check Next

29.2 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 environment variable PYTHONPATH.
3. The directory where Python is installed.

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 environment variable ***PYTHONPATH*** in the operating system. An operating system ***environment variable*** is much like a variable in a Python script, except that an environment 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 | 29.2.1: Finding modules.

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

- True
- False

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

- True
- False

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

- True
- False

29.3 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 29.3.1: Importing specific names from a module.

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

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 | 29.3.1: 'import x' vs 'from x import y'.

Animation captions:

1. import my_mod adds my_mod into the global namespace.
2. calc can be accessed using attribute reference operations.
3. From my_mod import, calc only copies calc from the my_mod namespace into the global namespace.

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

Table 29.3.1: '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)`

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 the message was changed.

Figure 29.3.1: 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.

zyDE 29.3.1: Extending the hash example.

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

Load default template...

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

text = input("Enter text to hash ('q' to qu

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

print('\nHash value:', output.hexdigest())
```

"Simplicity is the key to brilliance."
md5

Run

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 29.3.2: 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.

Check Show answer

2) The following code uses functions defined in

my_funcs.py. Complete the import statement at the top of the program.

```
a = 5

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

print('a^2 =',
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.

```
a = 5

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

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

Check Show answer

29.4 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 web_search.py, which contains functions for performing searches that "scrape" the results from a fictional web search engine, like Yahoo or Google. Executing the file as a script produces the following output:

Figure 29.4.1: 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.
    """
    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])
```



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

...

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

www.personalgrowthcourses.net/video
/baby_laughing
```

©zyBooks 12/15/22 00:53 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

```
        start_tag_loc =
html.find(start_tag, link_end)

    return links

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

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


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

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

Figure 29.4.2: domain_freq.py: Importing web_search causes unintended search to occur.

```
# Tracks frequency of domains in
web searches
import web_search  # Causes
unintended search

domains = {}

terms = input("\nEnter search terms
('q' to quit): ")
while terms != 'q':
    results =
web_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
Found 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
```


A file can better support execution 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 web_search.__name__ would have the value "web_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 29.4.3: Checking if a file is the executing script or an imported module.


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

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., "web_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 web_search.py file is modified below to fix the unintentional search.

Figure 29.4.4: web_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
web searches
import web_search

domains = {}

terms = input("Enter search terms
('q' to quit): ")
while terms != 'q':
    results =
web_search.search(terms)

    # ...

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

```
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
```

web_search.py


```
import urllib.request

def search(terms):
    # ...

def _send_request(terms):
    # ...

def _get_results(html):
    # ...

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

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

```
Enter search terms: Music Videos
Found 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
```


The web_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 web_search, which now does not perform the initial search because __name__ is equal to "web_search".

PARTICIPATION ACTIVITY 29.4.1: Executing modules as scripts.

1) Importing a module executes the statements contained within the imported module.

- True
- False

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

- True
- False

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

- True
- False

29.5 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 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 importlib standard library module.

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

Figure 29.5.1: 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',
        text='Enjoy!')
```


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!
```

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

Figure 29.5.2: send_coupons.py: Automates emails to loyal customers.

```
import os
from importlib 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)
```


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_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 29.5.3: 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!
```

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

```
from importlib 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!
```

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 | 29.5.1: Reloading modules.

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

- True
- False

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

- True
- False

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

- True
- False

29.6 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 29.6.1: Packages group related modules together.

Animation content:

`undefined`

Animation captions:

1. Packages, such as 'sound', contain subpackages, such as 'music' and 'effects'. Once subpackages are imported, modules and definitions in the subpackages are reached via dot notation.
2. The 'game' package contains subpackages 'sound' and 'graphics'.

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 include import statements that import subpackages or modules. 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 29.6.1: 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
               ...
```


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

Figure 29.6.2: Importing the ASCIIArt package.

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

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 29.6.3: Importing the canvas module.

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

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


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 29.6.4: Import cow module from figures subpackage.

```
from ASCIIArt.figures import cow  # import cow
module

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


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

Figure 29.6.5: Import the draw function from the cow module.

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

draw()  # Can omit ASCIIArt.figures.cow
```

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.y import 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 | 29.6.2: Importing packages.

Consider the directory structure of the ASCIIArt package above.

1) Write a statement to import the figures subpackage.

`import`

Check Show answer

2) Write a statement to import the cow module.

```
import
```

Check Show answer

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

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

Check Show answer

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

Check Show answer

29.7 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 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 29.7.1: 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/library/datetime.html
random	Functions for working with random numbers	https://docs.python.org/3/library/random.html
copy	Create complete copies of objects	https://docs.python.org/3/library/copy.html
time	Get the current time, convert time zones, sleep for a number of seconds	https://docs.python.org/3/library/time.html
math	Mathematical functions	https://docs.python.org/3/library/math.html
os	Operating system informational and management helpers	https://docs.python.org/3/library/os.html
sys	System specific environment or configuration helpers	https://docs.python.org/3/library/sys.html
pdb	The Python interactive debugger	https://docs.python.org/3/library/pdb.html
urllib	URL handling functions, such as requesting web pages	https://docs.python.org/3/library/urllib.html

Examples of standard library module usage is provided below.

Figure 29.7.1: Using the datetime module.

The following program uses the datetime module to print 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
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'))
```



```
Enter number of days
from now: 30
30 days from now is
June 29, 2016
```

Figure 29.7.2: Using the random module.

The following program uses the *random* module 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) were drawn to
find an ace.')
```



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


PARTICIPATION ACTIVITY 29.7.1: A few standard library modules.

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

If unable to drag and drop, refresh the page.

os **urllib** **math** **random**

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

Retrieve the contents of the webpage zybooks.com.

Get the name of the current operating system.

Compute the mathematical cosine function of a user-entered angle in radians.

Reset

Review all of the standard library

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

29.8 LAB: Artwork label (modules)

Define the `Artist` class in Artist.py with a constructor to initialize an artist's information. The constructor should by default initialize the artist's name to "unknown" and the years of birth and death to -1.

Define the `Artwork` class in Artwork.py with a constructor to initialize an artwork's information. The constructor should by default initialize the title to "unknown", the year created to -1, and the artist to use the `Artist` default constructor parameter values. Add an import statement to import the `Artist` class.

Add import statements to main.py to import the `Artist` and `Artwork` classes.

Ex: If the input is:

```
Pablo Picasso
1881
1973
Three Musicians
1921
```

the output is:


```
Artist: Pablo Picasso (1881 to 1973)
Title: Three Musicians, 1921
```

Ex: If the input is:

```
Brice Marden
1938
-1
Distant Muses
2000
```

the output is:

```
Artist: Brice Marden (1938 to present)
Title: Distant Muses, 2000
```

Ex: If the input is:

```
Banksy
-1
-1
Balloon Girl
2002
```

the output is:

```
Artist: Banksy (unknown)
Title: Balloon Girl, 2002
```


422102.2723990.qx3zqy7

LAB ACTIVITY | 29.8.1: LAB: Artwork label (modules) 0 / 10

Current file: **main.py** ▾ Load default template...

```python
# TODO: Import Artist from Artist.py and Artwork from Artwork.py

if __name__ == "__main__":
    user_artist_name = input()
    user_birth_year = int(input())
    user_death_year = int(input())
    user_title = input()
    user_year_created = int(input())

    user_artist = Artist(user_artist_name, user_birth_year, user_death_year)

    new_artwork = Artwork(user_title, user_year_created, user_artist)

    new_artwork.print_info()
```


Develop mode **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **main.py** (Your program) → O

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

29.9 LAB: Guess the random number

Given the code that reads a list of integers, complete the number_guess() function, which should choose a random number between 1 and 100 by calling random.randint() and then output if the guessed number is too low, too high, or correct.

Import the `random` module to use the random.seed() and random.randint() functions.

- random.seed(*seed_value*) seeds the random number generator using the given *seed_value*.
- random.randint(*a*, *b*) returns a random number between *a* and *b* (inclusive).

For testing purposes, use the seed value 900, which will cause the computer to choose the same random number every time the program runs.

Ex: If the input is:

```
32 45 48 80
```

the output is:

```
32 is too low. Random number was 80.
45 is too high. Random number was 30.
48 is correct!
80 is too low. Random number was 97.
```

422102.2723990.qx3zqy7

LAB ACTIVITY | 29.9.1: LAB: Guess the random number | 0 / 10

main.py

```
Loading latest submission...
```

Develop mode Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **main.py** (Your program) → O

Program output displayed here

Coding trail of your work What is this?

Retrieving signature

29.10 LAB: Quadratic formula

Implement the quadratic_formula() function. The function takes 3 arguments, a, b, and c, and computes the two results of the quadratic formula:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

The quadratic_formula() function returns the tuple `(x1, x2)`. Ex: When a = 1, b = -5, and c = 6, quadratic_formula() returns `(3, 2)`.

Code provided in main.py reads a single input line containing values for a, b, and c, separated by spaces. Each input is converted to a float and passed to the quadratic_formula() function.

Ex: If the input is:

```
2 -3 -77
```

the output is:

```
Solutions to 2x^2 + -3x + -77 = 0
x1 = 7
x2 = -5.50
```


422102.2723990.qx3zqy7

LAB ACTIVITY 29.10.1: LAB: Quadratic formula 0 / 10

main.py Load default template...

```python
# TODO: Import math module

def quadratic_formula(a, b, c):
    # TODO: Compute the quadratic formula results in variables x1 and x2
    return (x1, x2)


def print_number(number, prefix_str):
    if float(int(number)) == number:
        print(f'{prefix_str}{number:.0f}')
    else:
        print(f'{prefix_str}{number:.2f}')


if __name__ == "__main__":
    input_line = input()
    split_line = input_line.split(" ")
```

Develop mode **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **main.py** (Your program) → 0

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

29.11 Lab: Unique random numbers (random module)

This section's content is not available for print.

29.12 LAB: Dates

This section's content is not available for print.

29.13 LAB: Radioactive decay

This section's content is not available for print.