

# 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.



## 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**

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.

>>>

**Check**

[Show answer](#)

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

- 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.

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

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.



## Animation captions:

1. sys.module checks for HTTPServer. A new module object is created. The module is then inserted into sys.module.
2. HTTPServer's code is executed in module namespace. sys.module 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

### 11.1.4: The importing process.



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

**HTTPServer code executed**

**HTTPServer added to importer's namespace**

**HTTPServer added to sys.modules**

**sys.modules checked for HTTPServer**

**Module object created**

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

1

2

3

4

5

**Reset**

©zyBooks 11/11/19 20:42 569464

Dien Vu  
UNIONCSC103OrhanFall2019

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
```

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

print('=', fact)
```

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

©zyBooks 11/11/19 20:42 569464  
Dien Vu  
UNIONCSC103OrhanFall2019

**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()
```

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

- 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](#)



©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

## 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:

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

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
- 2) The environmental 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

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

False

## 11.3 Importing specific names from a module

©zyBooks 11/11/19 20:42 569464  
Diеп Vu  
UNIONCSC103OrhanFall2019

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, ...
```

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[b][c][d]` 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'.



### 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.

©zyBooks 11/11/19 20:42 569464  
Diеп Vu

Using "from" changes how an imported name is used in a program. UNIONCSC103OrhanFall2019

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

Description	Example import statement	Using imported names

Import an entire module	<code>import HTTPServer</code>	<code>print(HTTPServer.address)</code>
Import specific name from a module	<code>from HTTPServer import address</code>	<code>print(address)</code>

©zyBooks 11/11/19 20:42 569464

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.

zyDE 11.3.1: Extending the hash example.

Improve the hashing example from above by adding a new algorithm. Import the `md5` function from `hashlib`, and extend the user interface to allow that function to be called.

Load default template...

```

1
2 # FIXME: Import sha224 also
3 from hashlib import md5, sha1
4
5 text = input("Enter text to hash ('q' to quit)
6
7 # Add sha224 to prompt
8 algorithm = input('\nEnter algorithm (md5/sha1)
9 if algorithm == 'md5':
10     output = md5(text.encode('utf-8'))
11 elif algorithm == 'sha1':
12     output = sha1(text.encode('utf-8'))
13     # FIXME: Add check for sha224
14 else:
15     output = 'Invalid algorithm selection'
16
17 print('\nHash value:', output.hexdigest())
18
19

```

Run

"Simplicity is the key to brilliance." - Steve Jobs  
md5

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

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`.



[Show answer](#)

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

- 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))
```

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

**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**

## 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: google\_search.py: Get the 1st page of results for a Google search.

Enter search terms: Funny pictures of cat  
Google returned 7 links:

icanhas.cheezburger.com/lolcats 569464  
icanhas.cheezburger.com/Vu  
www.funnycatpix.com/103OrhanFall2019  
www.lolcats.com/  
www.buzzfeed.com/expresident/best-cat-p:  
photobucket.com/images/lol%20cat  
<https://www.facebook.com/pages/Funny-Cat-Pics/204188529615813>

...

Enter search terms: Videos of laughing baby  
Google returned 4 links:

[www.godtube.com/watch/?v=W7ZP6WNX](http://www.godtube.com/watch/?v=W7ZP6WNX)  
[afv.com/funniest-videos-week-laughing-baby-give-you-giggles-t22521](http://afv.com/funniest-videos-week-laughing-baby-give-you-giggles-t22521)  
[www.today.com/.../laughing-baby-video-w](http://www.today.com/.../laughing-baby-video-w)

[www.personalgrowthcourses.net/video/baby\\_](http://www.personalgrowthcourses.net/video/baby_)

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

```

import urllib.request

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

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

    url = 'http://www.google.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 _strip_html(string):
    new_string = ''
    in_tag = False
    for character in string:
        if character == '<':
            in_tag = True
        if not in_tag:
            new_string += character
        if character == '>':
            in_tag = False
    return new_string

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

```

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

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. However, the program will work on your own local Python installation.

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.

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

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
```

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:

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

Figure 11.4.3: 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., "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): ")
while terms != 'q':
    results =
    google_search.search(terms)

    # ...

print('\nNumber of search results for
```

### google\_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)
```

```
each site:")
for domain, num in domains.items():
    print(domain + ':', num )
```

```
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
```

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 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

©zyBooks 11/11/19 20:42 569464

Diep Vu  
UNIONCSC103OrhanFall2019

## 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',
        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 11.5.2: send\_coupons.py: Automates emails to loyal customers.

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

```
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[i][j][k]'
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 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!

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!

©zyBooks 11/11/19 20:42 569464  
 Diep Vu  
 UNIONCSC103OrhanFall2019

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.**

©zyBooks 11/11/19 20:42 569464

Diep Vu

UNIONCSC103OrhanFall2019

- 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

## 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.

©zyBooks 11/11/19 20:42 569464

Diep Vu

UNIONCSC103OrhanFall2019

**PARTICIPATION ACTIVITY****11.6.1: Packages group related modules together.**

### Animation captions:

1. The sound package is imported. Modules and definitions in the package are reached via dot notation.
2. Packages can contain subpackages. 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 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\
    __init__.py        Top-level package
    canvas.py
    figures\
        __init__.py     Subpackage for figures art
        man.py
        cow.py
        ...
    buildings\
        __init__.py     Subpackage for buildings art
        barn.py
        house.py
        ...
    ...

```

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
```

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

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
```

©zyBooks 11/11/19 20:42 569464

Diep Vu

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
```

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
```

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.

Consider the directory structure of the ASCIIArt package above.

- 1) Write a statement to import the figures subpackage.

import

**Check**

[Show answer](#)

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

- 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](#)

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

## 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.

©zyBooks 11/11/19 20:42 569464  
Diep Vu

UNIONCSC103OrhanFall2019

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

urllib

functions, such as  
requesting web pages<https://docs.python.org/3/library/urllib.html>

Examples of standard library module usage is provided below.

Figure 11.7.1: Using the datetime module.

©zyBooks 11/11/19 20:42 569464

Diep Vu

UNIONCSC103OrhanFall2019

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
# 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 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.

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

Card drawn: A card  
5 cards were drawn to find an ace.

```
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.')
```

©zyBooks 11/11/19 20:42 569464

Diep Vu

UNIONCSC103OrhanFall2019

### 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.

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

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

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 number of degrees.

©zyBooks 11/11/19 20:42 569464

Diep Vu  
UNIONCSC103OrhanFall2019

Reset

## 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.*

©zyBooks 11/11/19 20:42 569464  
Diep Vu  
UNIONCSC103OrhanFall2019