Modules and Packages

In programming, a module is a piece of software that has a specific functionality. For example, when building a ping pong game, one module would be responsible for the game logic, and  
another module would be responsible for drawing the game on the screen. Each module is a different file, which can be edited separately.

### **Writing modules**

Modules in Python are simply Python files with a .py extension. The name of the module will be the name of the file. A Python module can have a set of functions, classes or variables defined and implemented. In the example above, we will have two files, we will have:

college/

college/cgec.py

college/cse.py

The Python script cgec.py will implement the game. It will use the function cse\_cgec from the file cse.py, or in other words, the draw module, that implements the logic for drawing the game on the screen.

Modules are imported from other modules using the import command. In this example, the cgec.py script may look something like this:

# cgec.py

# import the cse module

import cse

def play\_cgec():

...

def main():

result = play\_cgec()

cse.cse\_cgec(result)

# this means that if this script is executed, then

# main() will be executed

if \_\_name\_\_ == '\_\_main\_\_':

main()

The draw module may look something like this:

# cse.py

def cse\_cgec():

...

def clear\_screen(screen):

...

In this example, the cgec module imports the load module, which enables it to use functions implemented in that module. The main function would use the local function play\_cgec to run the cgec, and then cse the result of the cgec using a function implemented in the cse module called cse\_cgec. To use the function cse\_cgec from the cse module, we would need to specify in which module the function is implemented, using the dot operator. To reference the cse\_cgec function from the cgec module, we would need to import the cse module and only then call cse.cse\_cgec().

When the import cse directive will run, the Python interpreter will look for a file in the directory which the script was executed from, by the name of the module with a .py prefix, so in our case it will try to look for cse.py. If it will find one, it will import it. If not, he will continue to look for built-in modules.

You may have noticed that when importing a module, a .pyc file appears, which is a compiled Python file. Python compiles files into Python bytecode so that it won't have to parse the files each time modules are loaded. If a .pyc file exists, it gets loaded instead of the .py file, but this process is transparent to the user.

### **Importing module objects to the current namespace**

We may also import the function cse\_cgec directly into the main script's namespace, by using the from command.

# cgec.py

# import the cse module

from cse import cse\_cgec

def main():

result = play\_cgec()

cse\_cgec(result)

You may have noticed that in this example, cse\_cgec does not precede with the name of the module it is imported from, because we've specified the module name in the import command.

The advantages of using this notation is that it is easier to use the functions inside the current module because you don't need to specify which module the function comes from. However, any namespace cannot have two objects with the exact same name, so the import command may replace an existing object in the namespace.

### **Importing all objects from a module**

We may also use the import \* command to import all objects from a specific module, like this:

# cgec.py

# import the cse module

from cse import \*

def main():

result = play\_cgec()

cse\_cgec(result)

This might be a bit risky as changes in the module might affect the module which imports it, but it is shorter and also does not require you to specify which objects you wish to import from the module.

### **Custom import name**

We may also load modules under any name we want. This is useful when we want to import a module conditionally to use the same name in the rest of the code.

For example, if you have two cse modules with slighty different names - you may do the following:

# cgec.py

# import the cse module

if visual\_mode:

# in visual mode, we cse using graphics

import cse\_visual as cse

else:

# in textual mode, we print out text

import cse\_textual as cse

def main():

result = play\_cgec()

# this can either be visual or textual depending on visual\_mode

cse.cse\_cgec(result)

### **Module initialization**

The first time a module is loaded into a running Python script, it is initialized by executing the code in the module once. If another module in your code imports the same module again, it will not be loaded twice but once only - so local variables inside the module act as a "singleton" - they are initialized only once.

This is useful to know, because this means that you can rely on this behavior for initializing objects. For example:

# cse.py

def cse\_cgec():

# when clearing the screen we can use the main screen object initialized in this module

clear\_screen(main\_screen)

...

def clear\_screen(screen):

...

class Screen():

...

# initialize main\_screen as a singleton

main\_screen = Screen()

### **Extending module load path**

There are a couple of ways we could tell the Python interpreter where to look for modules, aside from the default, which is the local directory and the built-in modules. You could either use the environment variable PYTHONPATH to specify additional directories to look for modules in, like this:

PYTHONPATH=/foo python cgec.py

This will execute cgec.py, and will enable the script to load modules from the foo directory as well as the local directory.

Another method is the sys.path.append function. You may execute it before running an import command:

sys.path.append("/foo")

This will add the foo directory to the list of paths to look for modules in as well.

### **Exploring built-in modules**

Check out the full list of built-in modules in the Python standard library [here](https://docs.python.org/3/library/).

Two very important functions come in handy when exploring modules in Python - the dir and help functions.

If we want to import the module urllib, which enables us to create read data from URLs, we simply import the module:

# import the library

import urllib

# use it

urllib.urlopen(...)

We can look for which functions are implemented in each module by using the dir function:

>>> import urllib

>>> dir(urllib)

['ContentTooShortError', 'FancyURLopener', 'MAXFTPCACHE', 'URLopener', '\_\_all\_\_', '\_\_builtins\_\_',

'\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_version\_\_', '\_ftperrors', '\_get\_proxies',

'\_get\_proxy\_settings', '\_have\_ssl', '\_hexdig', '\_hextochr', '\_hostprog', '\_is\_unicode', '\_localhost',

'\_noheaders', '\_nportprog', '\_passwdprog', '\_portprog', '\_queryprog', '\_safe\_map', '\_safe\_quoters',

'\_tagprog', '\_thishost', '\_typeprog', '\_urlopener', '\_userprog', '\_valueprog', 'addbase', 'addclosehook',

'addinfo', 'addinfourl', 'always\_safe', 'basejoin', 'c', 'ftpcache', 'ftperrors', 'ftpwrapper', 'getproxies',

'getproxies\_environment', 'getproxies\_macosx\_sysconf', 'i', 'localhost', 'main', 'noheaders', 'os',

'pathname2url', 'proxy\_bypass', 'proxy\_bypass\_environment', 'proxy\_bypass\_macosx\_sysconf', 'quote',

'quote\_plus', 'reporthook', 'socket', 'splitattr', 'splithost', 'splitnport', 'splitpasswd', 'splitport',

'splitquery', 'splittag', 'splittype', 'splituser', 'splitvalue', 'ssl', 'string', 'sys', 'test', 'test1',

'thishost', 'time', 'toBytes', 'unquote', 'unquote\_plus', 'unwrap', 'url2pathname', 'urlcleanup', 'urlencode',

'urlopen', 'urlretrieve']

When we find the function in the module we want to use, we can read about it more using the help function, inside the Python interpreter:

help(urllib.urlopen)

### **Writing packages**

Packages are namespaces which contain multiple packages and modules themselves. They are simply directories, but with a twist.

Each package in Python is a directory which **MUST** contain a special file called \_\_init\_\_.py. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.

If we create a directory called foo, which marks the package name, we can then create a module inside that package called bar. We also must not forget to add the \_\_init\_\_.py file inside the foo directory.

To use the module bar, we can import it in two ways:

import foo.bar

or:

from foo import bar

In the first method, we must use the foo prefix whenever we access the module bar. In the second method, we don't, because we import the module to our module's namespace.

The \_\_init\_\_.py file can also decide which modules the package exports as the API, while keeping other modules internal, by overriding the \_\_all\_\_ variable, like so:

\_\_init\_\_.py:

\_\_all\_\_ = ["bar"]

**In Our Projects Uses Packages**

***# Linear algebra***

Import pandas as pd

***# Data processing***

Import numpy as np

***#data Visulazation***

Import matplotlib.pyplot as plt

***# regression***

from sklearn.feature\_selection import RFE

from sklearn.linear\_model import LogisticRegression

***#Logistic regression model***

from sklearn.cross\_validation import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn import metrics

### **#Random Forest**

from sklearn.ensemble import RandomForestClassifier

***#Support Vector Machine***

from sklearn.svm import SVC

***# Sklearn selection model***

from sklearn import model\_selection

from sklearn.model\_selection import cross\_val\_score

***# Precision***

from sklearn.metrics import classification\_report