Python Programming - V

- By Nimesh Kumar Dagur

Python Modules

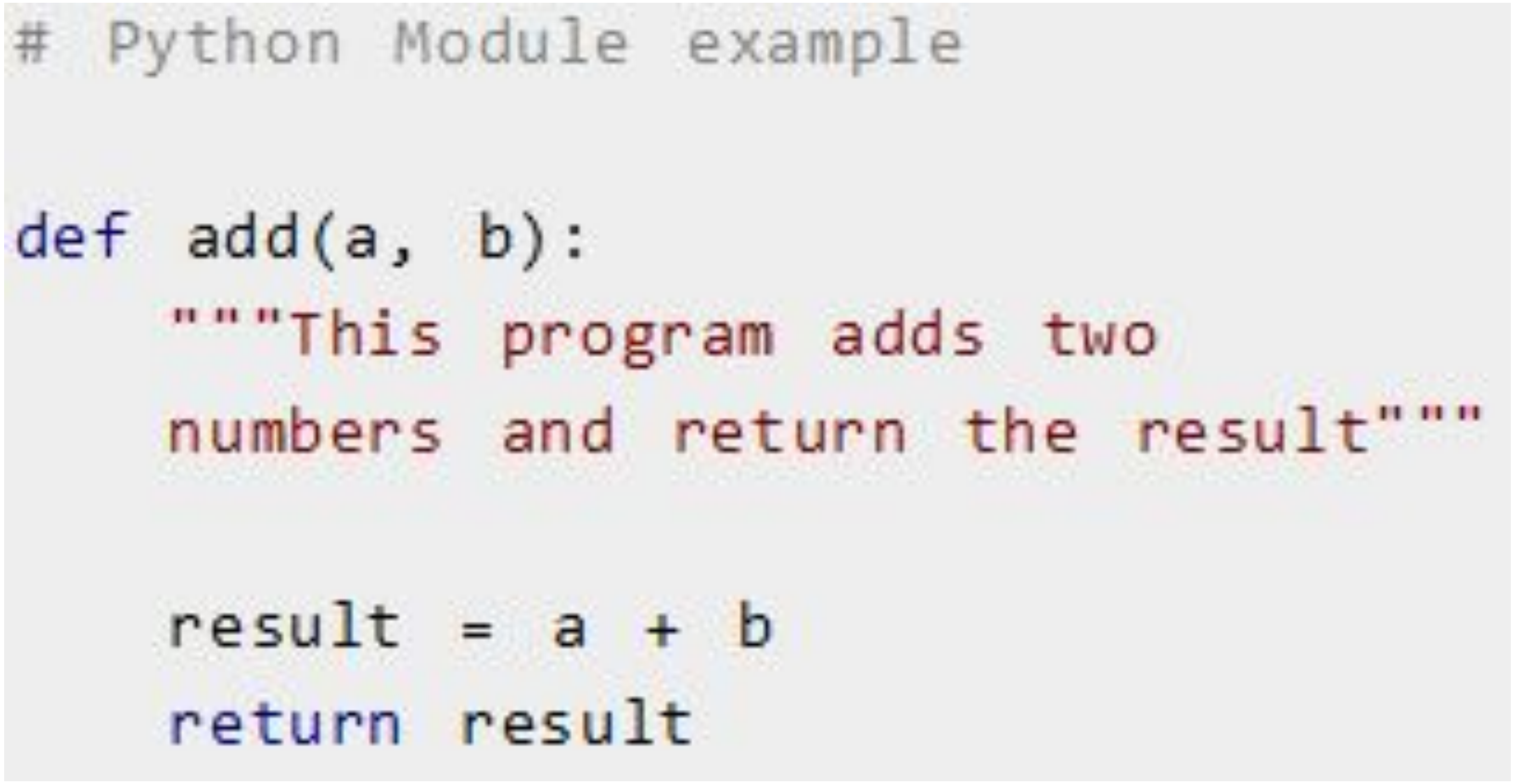
- A module allows you to logically organize your Python code.
- Grouping related code into a module makes the code easier to understand and use.
- A module is a Python object with arbitrarily named attributes that you can bind and reference.
- Simply, a module is a file consisting of Python code. A module can define functions, classes and variables.

Python Modules

- Modules refer to a file containing Python statements and definitions.
- A file containing Python code, for e.g.: example.py, is called a module and its module name would be example.
- We use modules to break down large programs into small manageable and organized files.
- Furthermore, modules provide reusability of code.
- We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Example

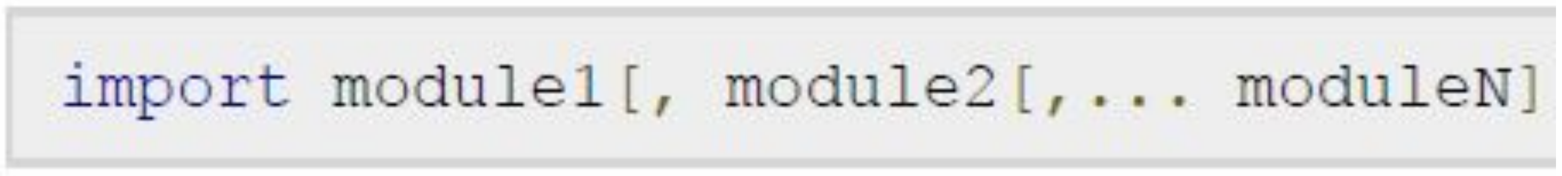
- Type the following and save it as **example.py**.
- Here, we have defined **a function add() inside a module named example**



```python
# Python Module example

def add(a, b):
    """This program adds two
    numbers and return the result"""

    result = a + b
    return result
```

Importing modules

- We can import the definitions inside a module to another module or the interactive interpreter in Python.
- You can use any Python source file as a module by executing an import statement in some other Python source file.
- We use the import keyword to do this.



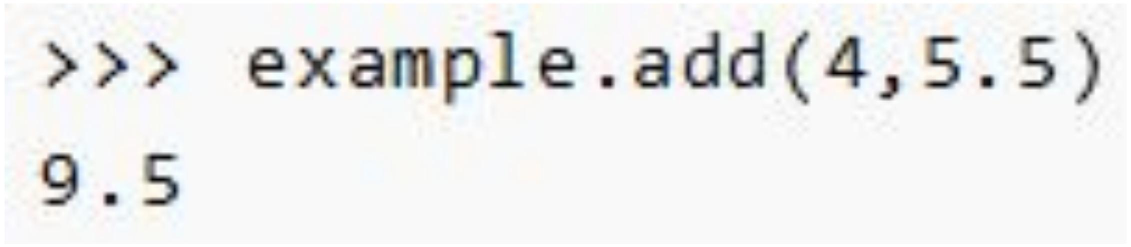
```
import module1[, module2[,... moduleN]
```

Importing modules: example

- To import our previously defined module example we type the following in the Python prompt.

>>> import example

- This does not enter the names of the functions defined in example directly in the current symbol table.
- It only enters the module name example there.
- Using the module name we can access the function using dot (.) operation.



```
>>> example.add(4,5.5)
9.5
```

Importing modules

- When the interpreter encounters an import statement, it imports the module if the module is present in the search path.
- A search path is a list of directories that the interpreter searches before importing a module.
- A module is loaded only once, regardless of the number of times it is imported.
- This prevents the module execution from happening over and over again if multiple imports occur.

Importing modules

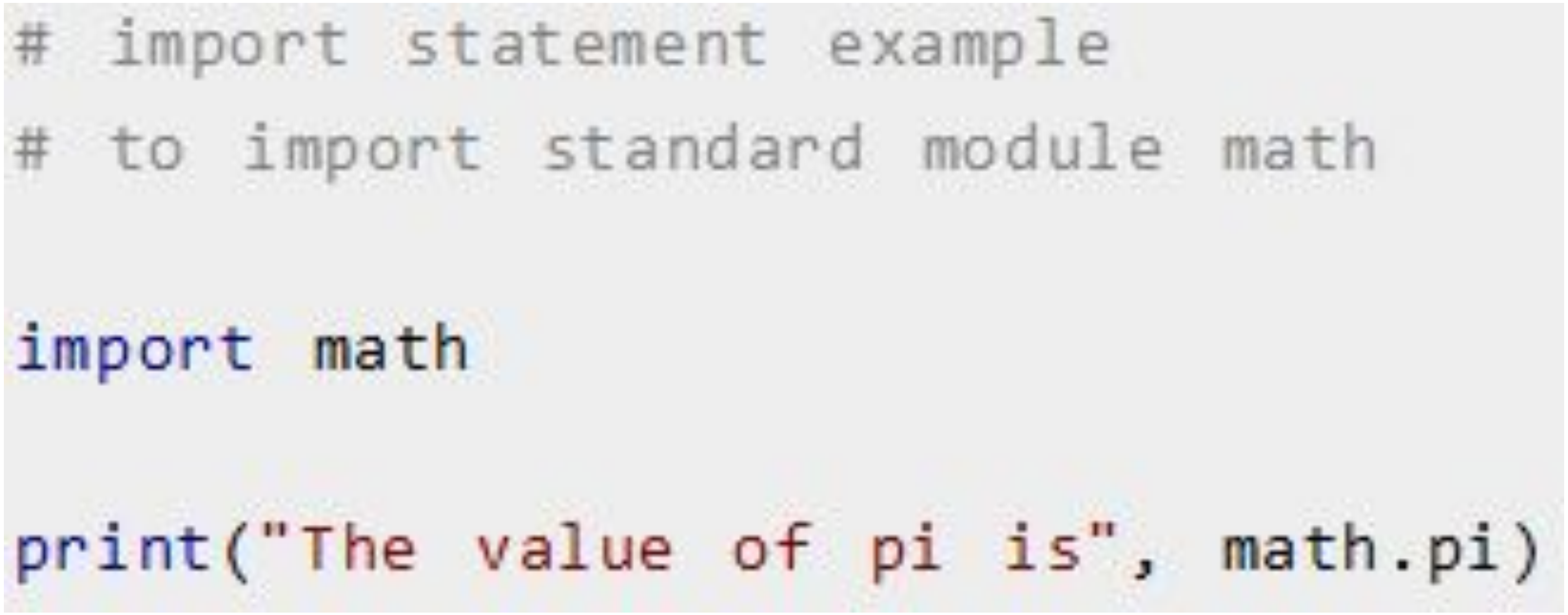
- Python has a ton of standard modules available.
- You can check out the full list of Python standard modules and what they are for.
- These files are in the Lib directory inside the location where you installed Python.
- Standard modules can be imported the same way as we import our user-defined modules.

Importing modules

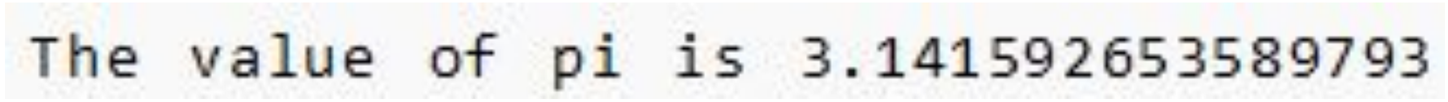
There are various ways to import modules:

- **The import statement**
- **Import with renaming**
- **The from...import statement**
- **Import all names by using *the from...import * Statement***

The import statement

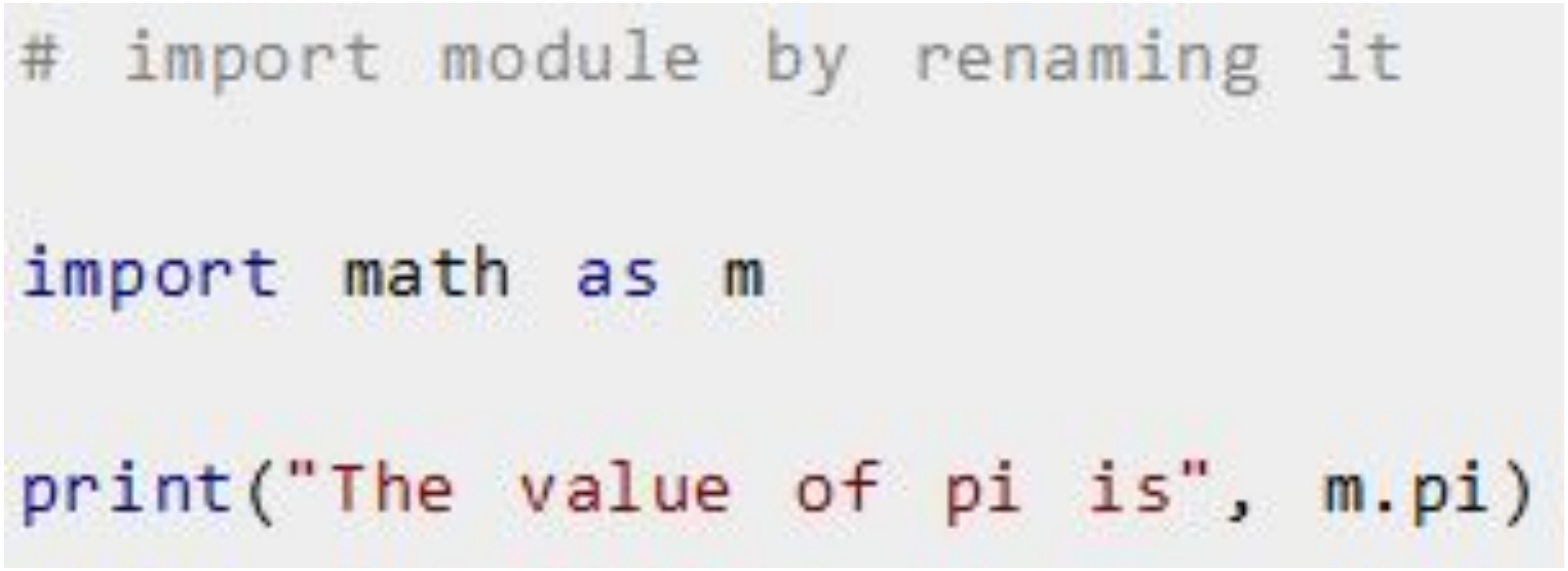


```python
# import statement example
# to import standard module math

import math

print("The value of pi is", math.pi)
```



```
The value of pi is 3.141592653589793
```

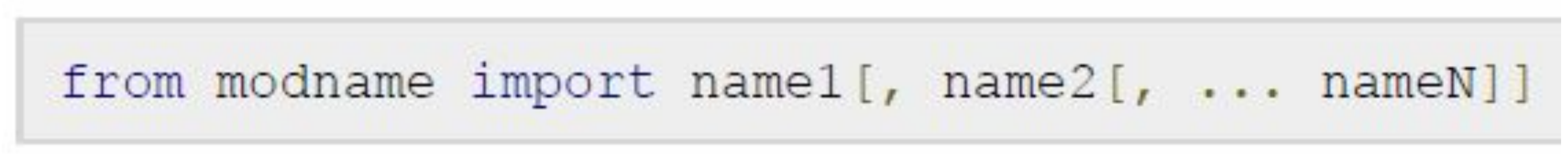
Import with renaming



```
# import module by renaming it

import math as m

print("The value of pi is", m.pi)
```

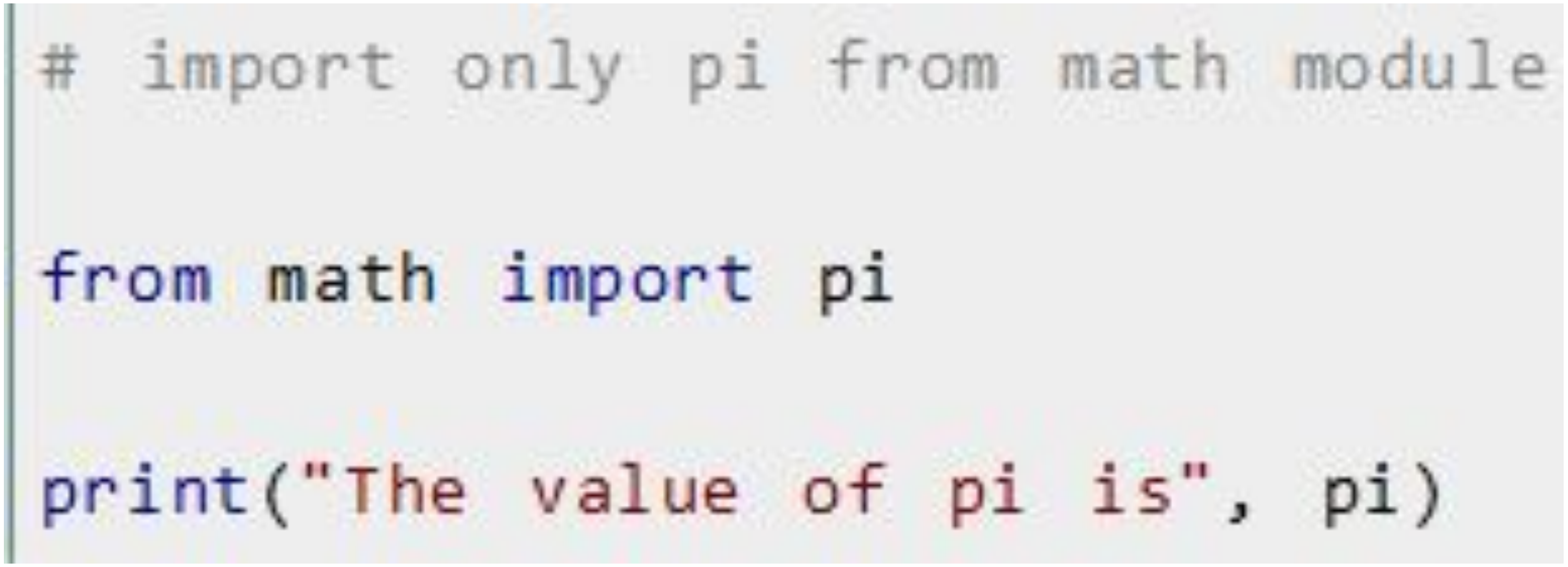
The from...import statement

- Python's from statement lets you import specific attributes from a module into the current namespace.
- **Syntax:**



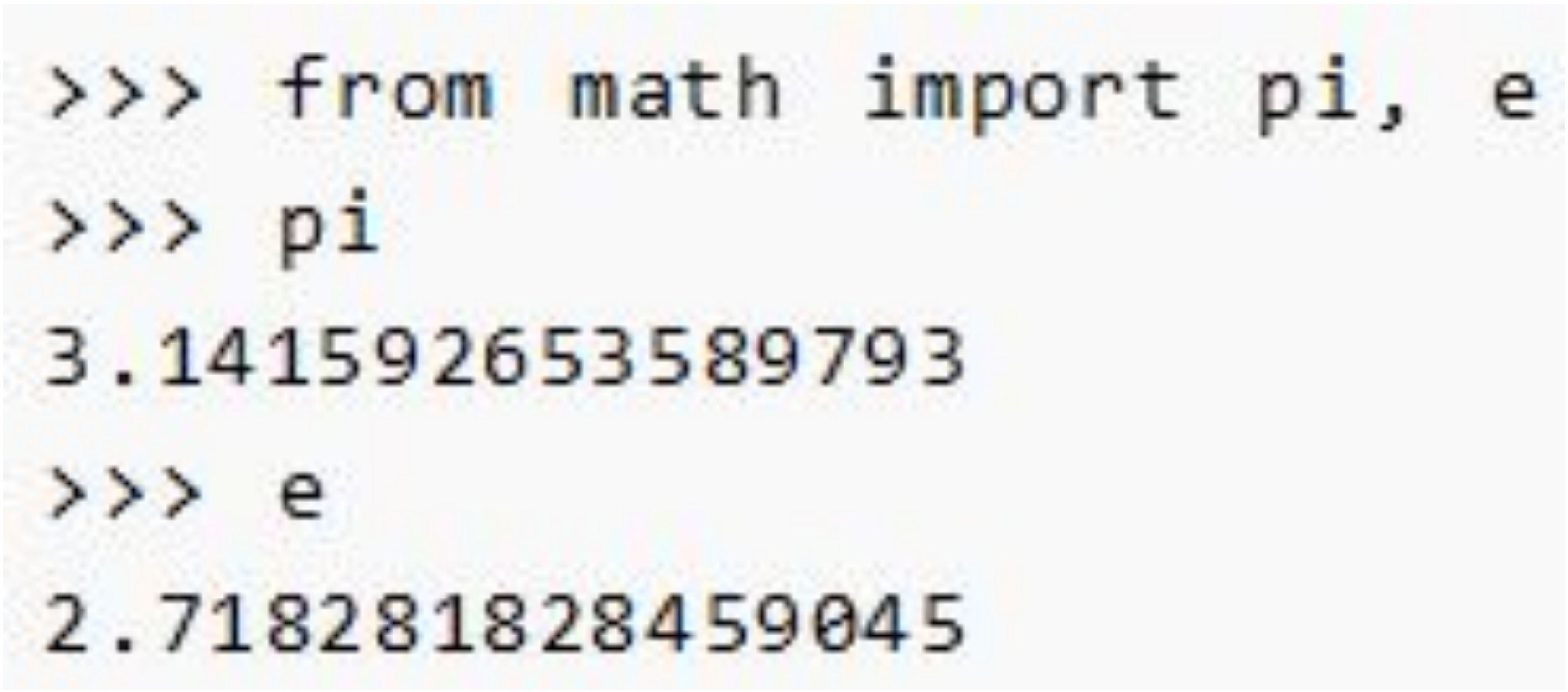
```
from modname import name1[, name2[, ... nameN]]
```

The from...import statement



```
# import only pi from math module

from math import pi

print("The value of pi is", pi)
```

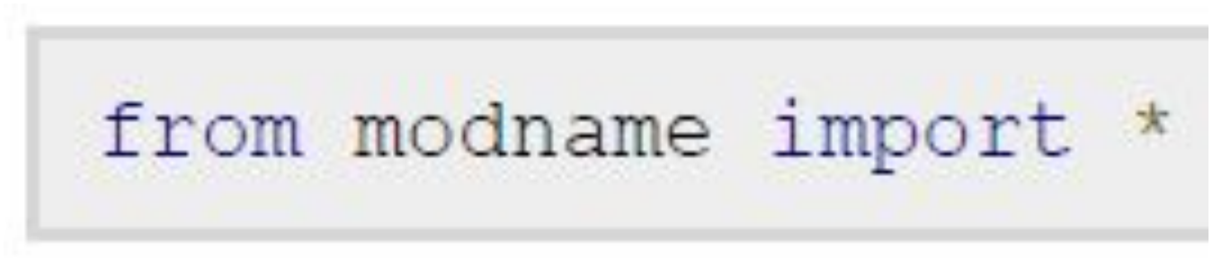
The from...import statement



```
>>> from math import pi, e
>>> pi
3.141592653589793
>>> e
2.718281828459045
```

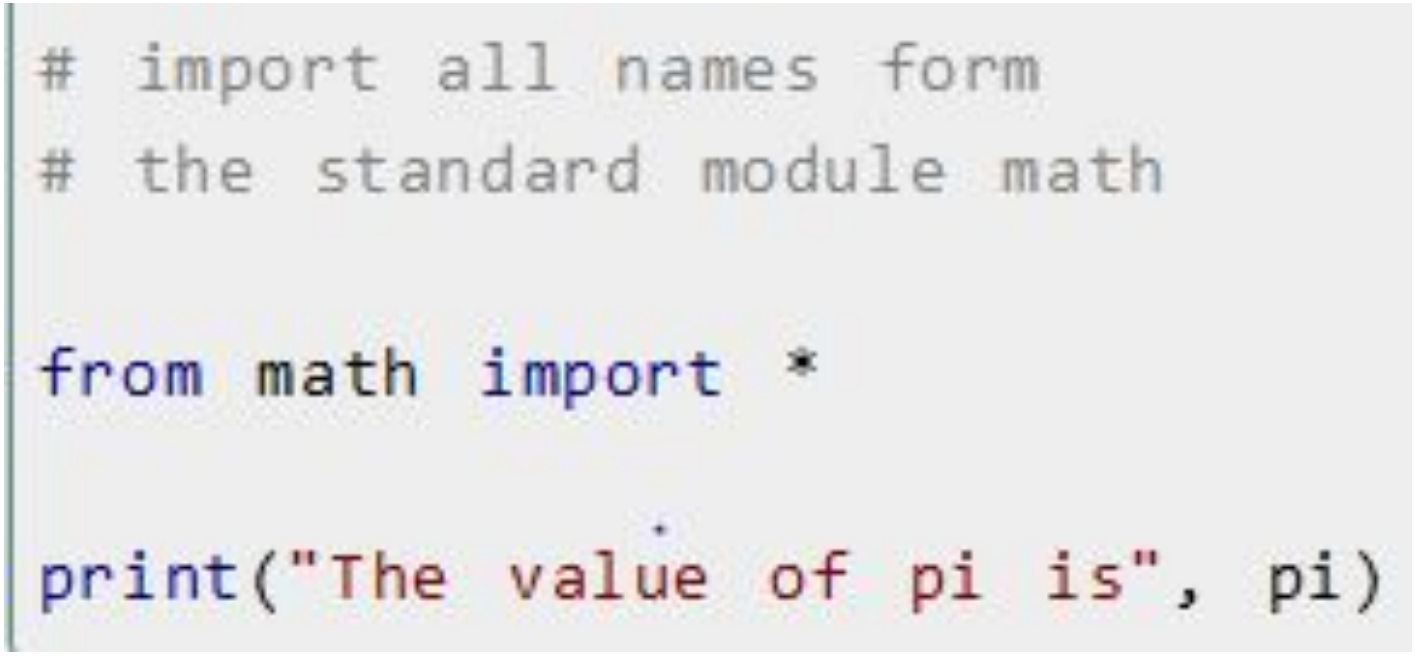
Import all names

- It is also possible to import all names from a module into the current namespace by using the following import statement:



```
from modname import *
```

Import all names



```
# import all names form
# the standard module math

from math import *

print("The value of pi is", pi)
```

We imported all the definitions from the math module. This makes all names except those beginning with an underscore, visible in our scope.

Python Module Search Path – Locating Module

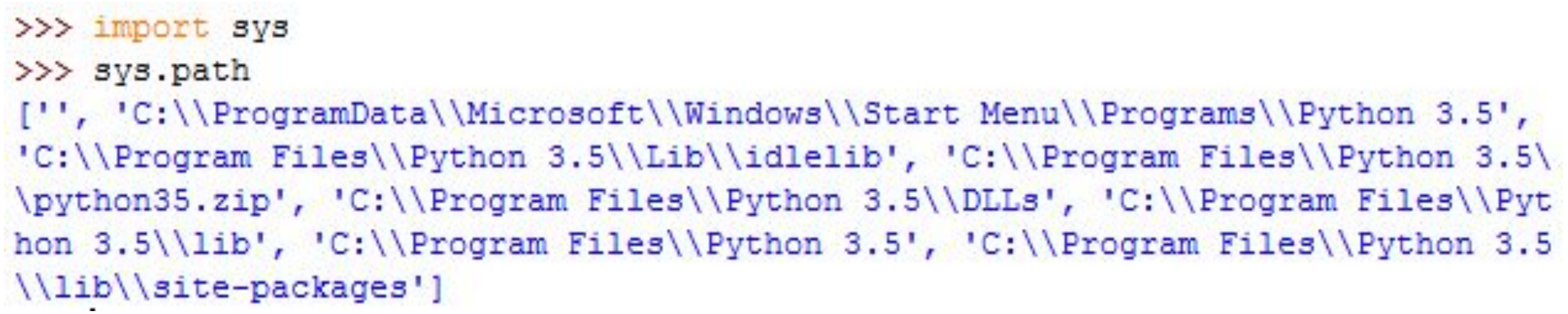
- While importing a module, Python looks at several places.
- Interpreter first looks for a built-in module then (if not found) into a list of directories defined in sys.path.
- The search is in this order:

✔ The current directory.

✔ PYTHONPATH (an environment variable with a list of directory).

✔ The installation-dependent default directory.

Python Module Search Path – Locating Module



```
>>> import sys
>>> sys.path
['', 'C:\\ProgramData\\Microsoft\\Windows\\Start Menu\\Programs\\Python 3.5',
'C:\\Program Files\\Python 3.5\\Lib\\idlelib', 'C:\\Program Files\\Python 3.5\
\python35.zip', 'C:\\Program Files\\Python 3.5\\DLLs', 'C:\\Program Files\\Pyt
hon 3.5\\lib', 'C:\\Program Files\\Python 3.5', 'C:\\Program Files\\Python 3.5
\\lib\\site-packages']
```

Reloading a module

- The Python interpreter imports a module only once during a session.
- This makes things more efficient.
- **Example:** code in a module named my_module

```
print("This code got executed")
```

Module1.py

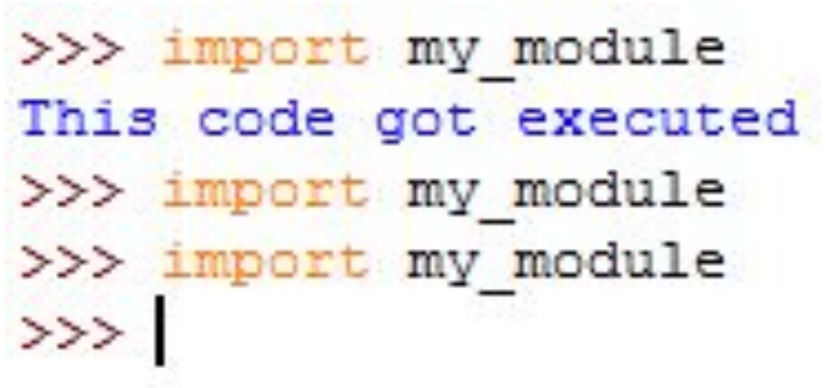
```
Add(x,y)
Sub(x,y)

      Mul()
      Div()
```

Test.py

```
import Module1
Module1.
Import Module1
```

Reloading a module: example

Effect of multiple imports:



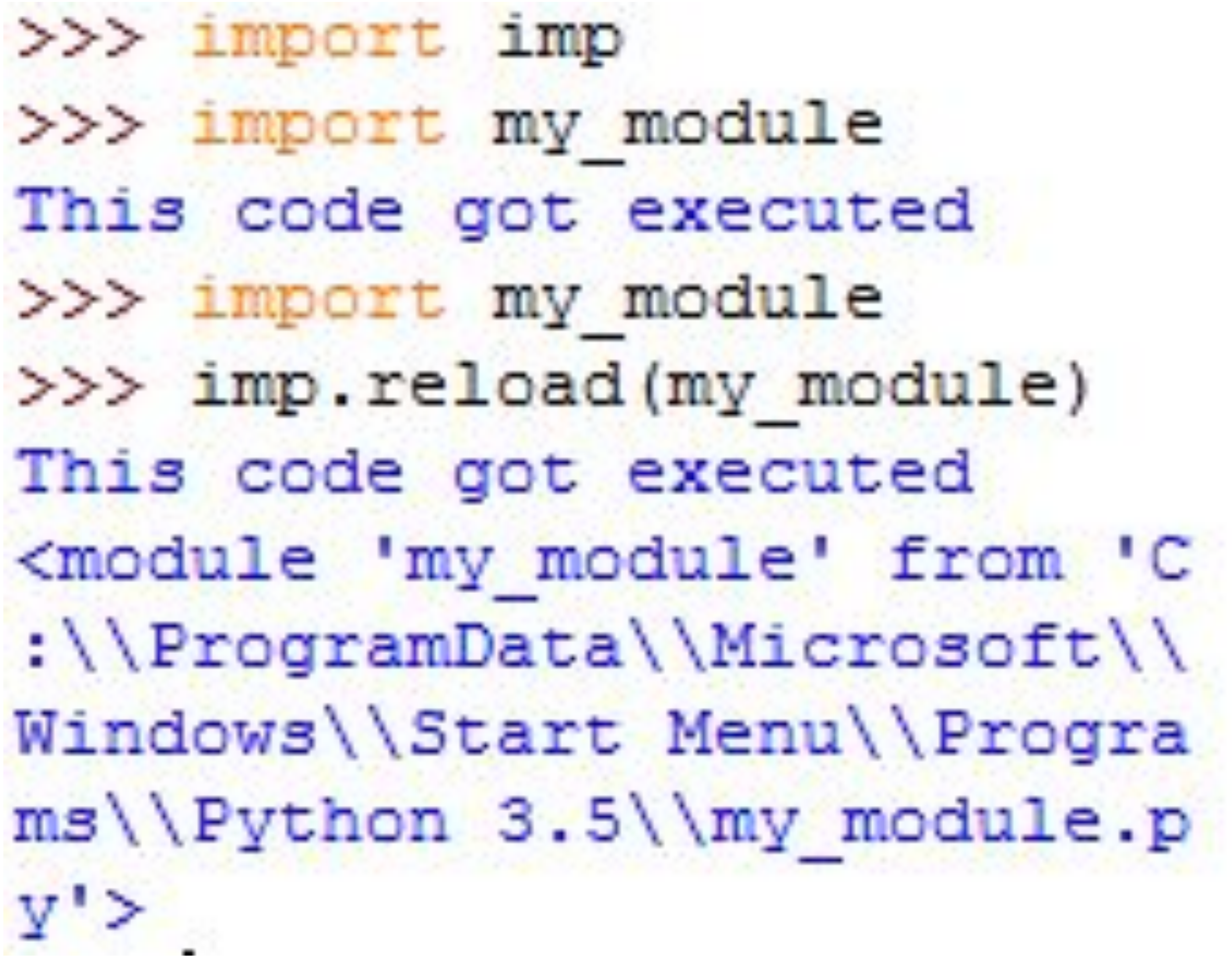
```
>>> import my_module
This code got executed
>>> import my_module
>>> import my_module
>>> |
```

- We can see that our code got executed only once.
- This goes to say that our module was imported only once.

Reloading a module

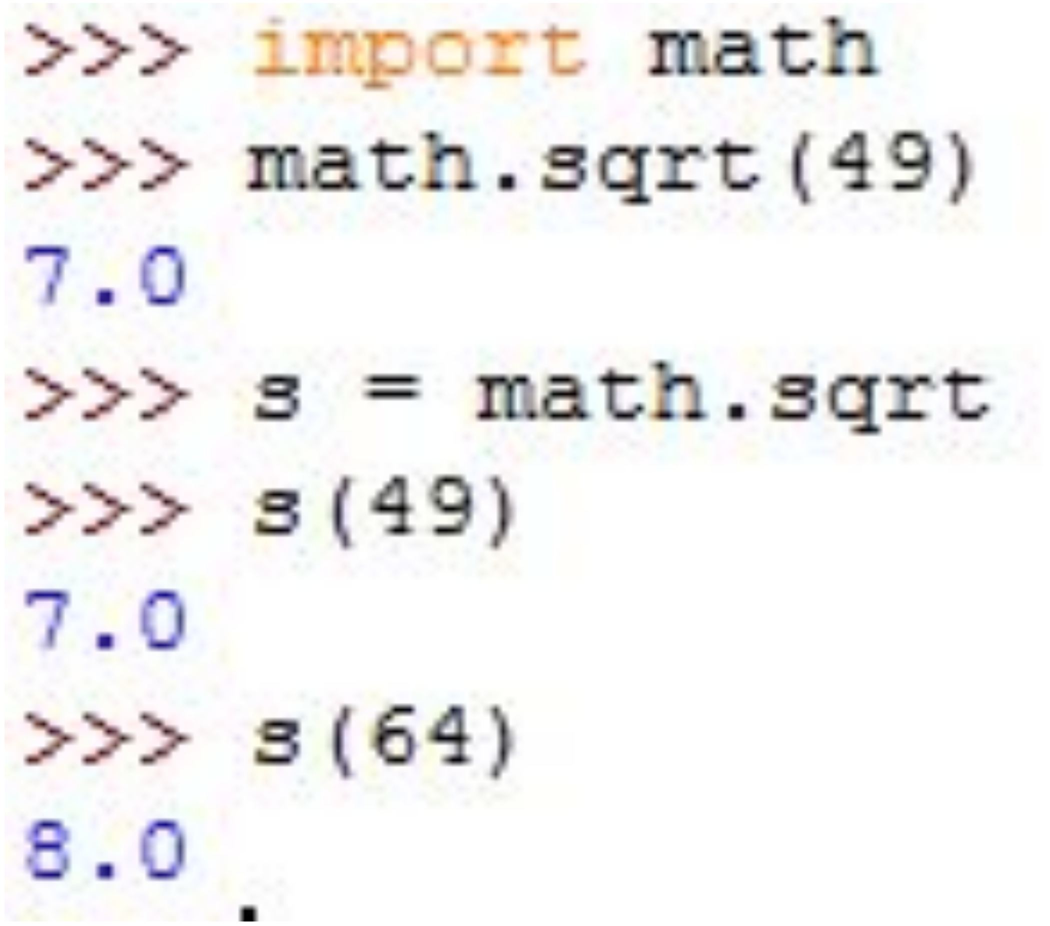
- Now if our module changed during the course of the program, we would have to reload it.
- One way to do this is to restart the interpreter.
- But this does not help much.
- Python provides a neat way of doing this.
- Use the **reload() function** inside the **imp module** to reload a module.
- But Now in latest Python 3 version, we use **importlib** in place of imp.

Reloading a module: Example



```
>>> import imp
>>> import my_module
This code got executed
>>> import my_module
>>> imp.reload(my_module)
This code got executed
<module 'my_module' from 'C
:\\ProgramData\\Microsoft\\
Windows\\Start Menu\\Progra
ms\\Python 3.5\\my_module.p
y'>
```

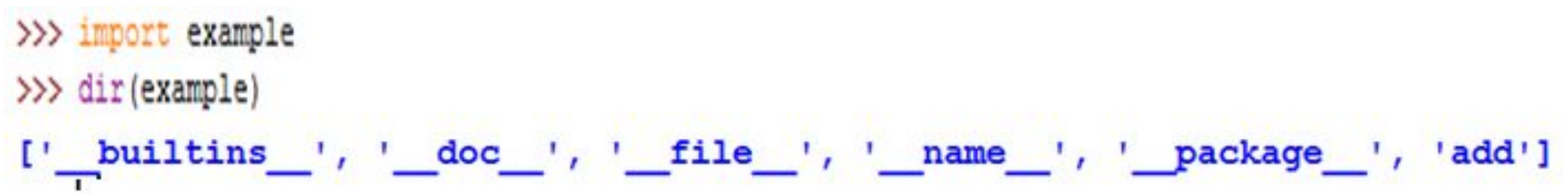
Assigning functions to variables



```
>>> import math
>>> math.sqrt(49)
7.0
>>> s = math.sqrt
>>> s(49)
7.0
>>> s(64)
8.0
```

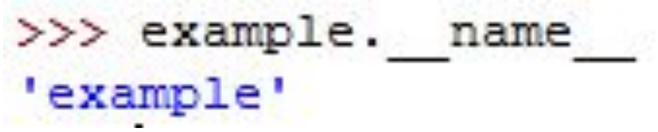
The dir() built-in function

- We can use the dir() function to find out names that are defined inside a module
- **Example:** we have defined a function add() in the module example in previous slides.



```
>>> import example
>>> dir(example)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'add']
```

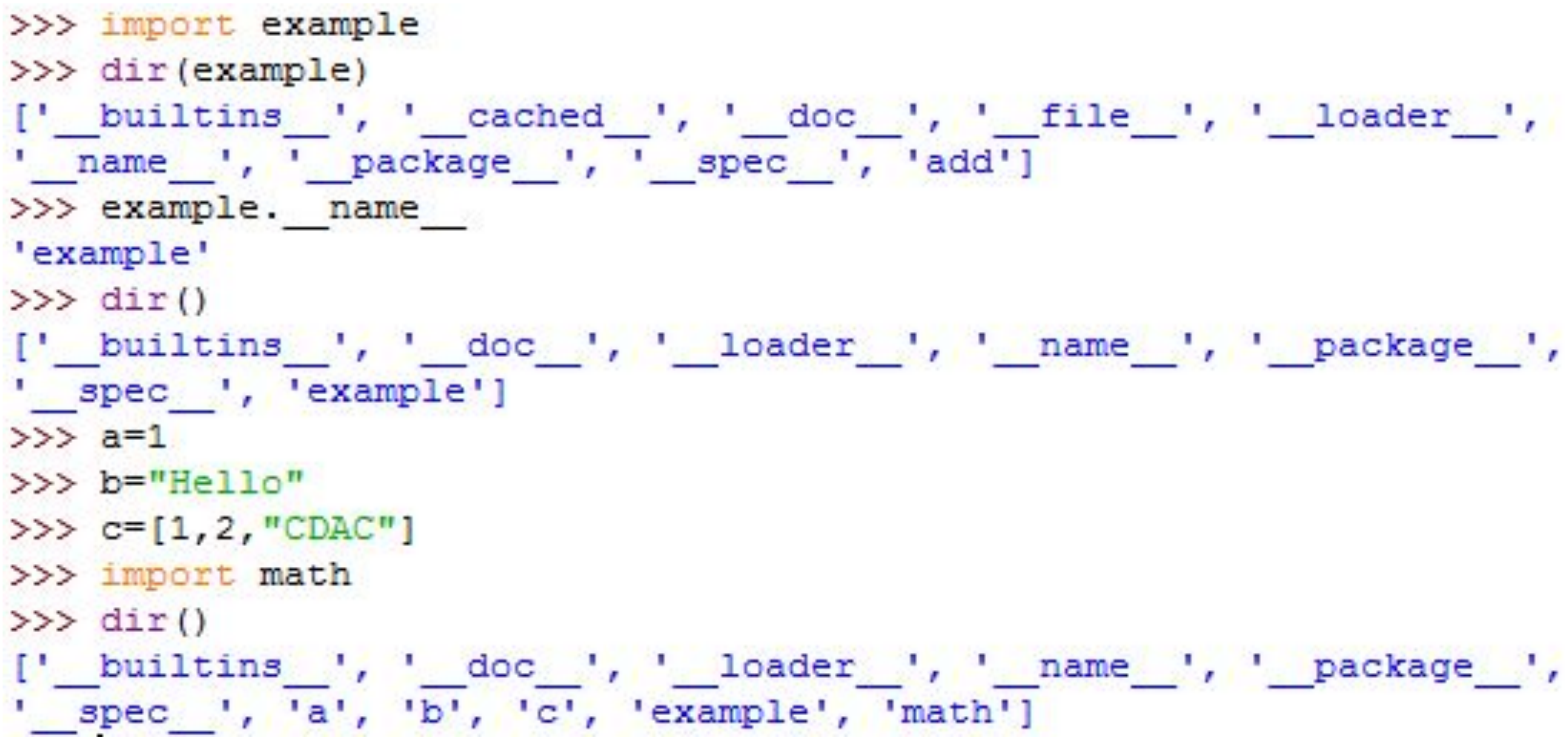
- Here, we can see a sorted list of names (along with add).
- All other names that begin with an underscore are default Python attributes associated with the module .
- For example, the **__name__** attribute contains the name of the module



```
>>> example.__name__
'example'
```

The dir() built-in function

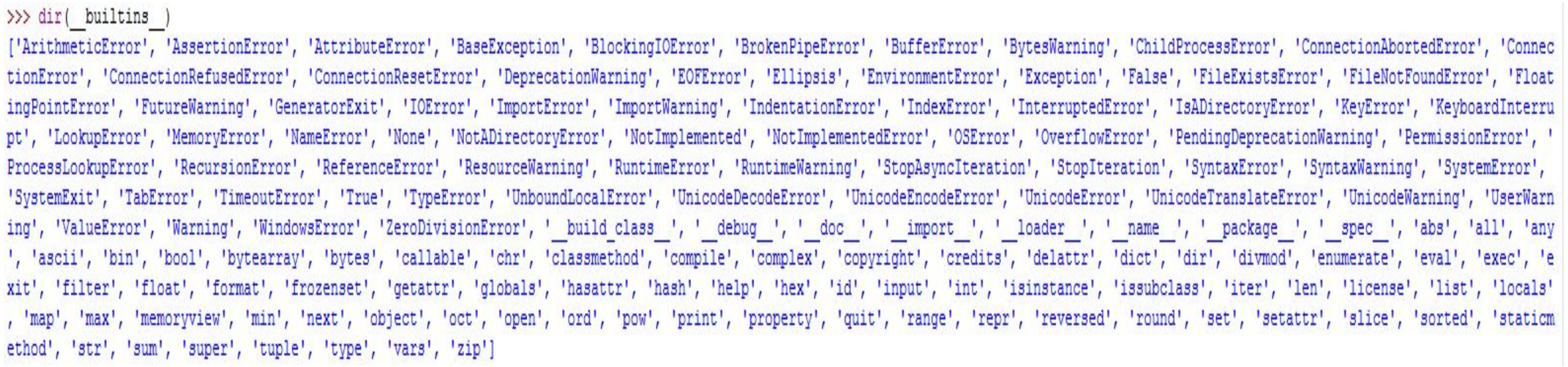
- All the names defined in our current namespace can be found out using the dir() function without any arguments.



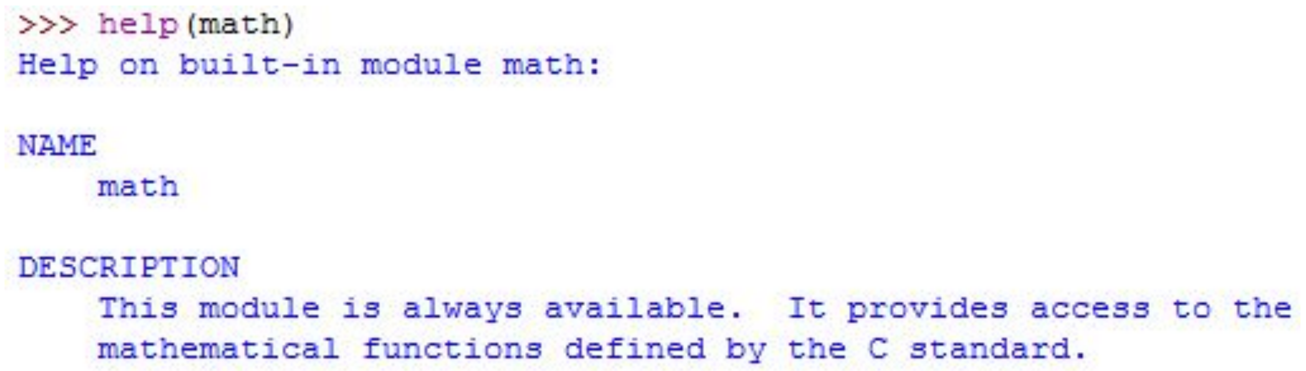
```
>>> import example
>>> dir(example)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', 'add']
>>> example.__name__
'example'
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__', 'example']
>>> a=1
>>> b="Hello"
>>> c=[1,2,"CDAC"]
>>> import math
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__', 'a', 'b', 'c', 'example', 'math']
```

The dir() and help() built-in function

- **dir(__builtins__)** is used to list all built-in functions
- **help()** is used to Show help menu for an imported module.



```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'Connec
tionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'Float
ingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterru
pt', 'LookupError', 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', '
ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarn
ing', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', '__build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__', '__package__', '__spec__', 'abs', 'all', 'any
', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'e
xit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals'
, 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticm
ethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```



```
>>> help(math)
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.
```

__name__ in Python

- Since there is no main() function in Python, when the command to run a python program is given to the interpreter, the code that is at level 0 indentation is to be executed. However, before doing that, it will define a few special variables.
- **__name__** is one such special variable.
- If the source file is executed as the main program, the interpreter sets the __name__ variable to have a value **"__main__".**
- If this file is being imported from another module, __name__ will be set to the module's name.
- **__name__ is a built-in variable which evaluates to the name of the current module.**
- Thus it can be used to check whether the current script is being run on its own or being imported somewhere else by combining it with if statement,

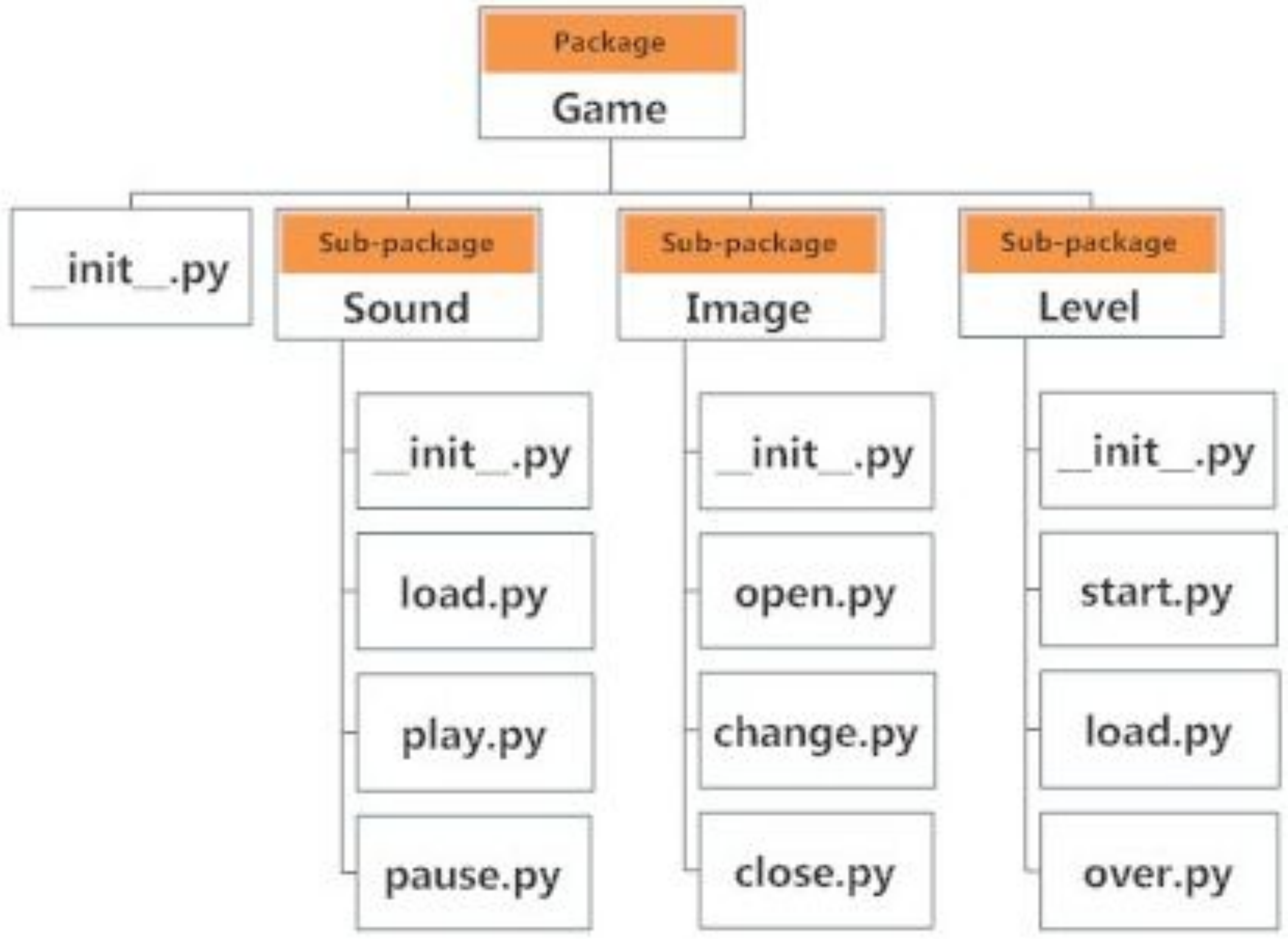
Python Package

Python Package

- We don't usually store all of our files in our computer in the same location.
- We use a well-organized hierarchy of directories for easier access.
- Similar files are kept in the same directory, for example, we may keep all the songs in the "music" directory.
- Analogous to this, Python has packages for directories and modules for files.
- As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages.
- This makes a project (program) easy to manage and conceptually clear.
- Similar, as a directory can contain sub-directories and files, a Python package can have sub-packages and modules.
- A directory must contain a file named **__init__.py** in order for Python to consider it as a package.
- This file can be left empty but we generally place the initialization code for that package in this file.

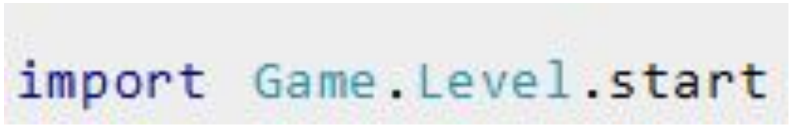
Example

- Suppose we are developing a game, one possible organization of packages and modules could be as shown in the figure below.



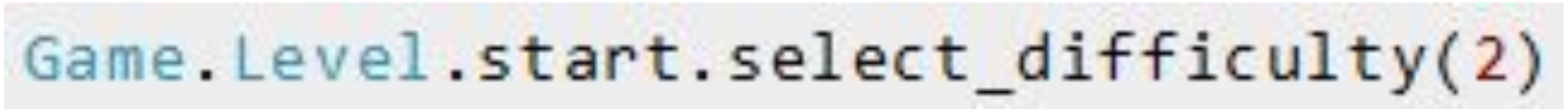
Importing module from a package

- We can import modules from packages using the dot (.) operator
- if want to import the start module in the above example, it is done as follows.



```
import Game.Level.start
```

- Now if this module contains a function named **select_difficulty(),** we must use the full name to reference it.



```
Game.Level.start.select_difficulty(2)
```

- While importing packages, Python looks in the list of directories defined in **sys.path**, similar as for module search path.

Working with Files in Python Programming

- File is a named location on disk to store related information
- While a program is running, its data is in memory.
- When the program ends, or the computer shuts down, data in memory disappears.
- To store data permanently, you have to put it in a File.
- file is used to permanently store data in a non-volatile memory (e.g. hard disk).
- Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.

- Files are usually stored on a hard drive, floppy drive, or CD-ROM.
- When there are a large number of files, they are often organized into directories (also called \folders").
- Each file is identified by a unique name, or a combination of a file name and a directory name.
- By reading and writing files, programs can exchange information with each other and generate printable formats like PDF.

Working with Files in Python Programming

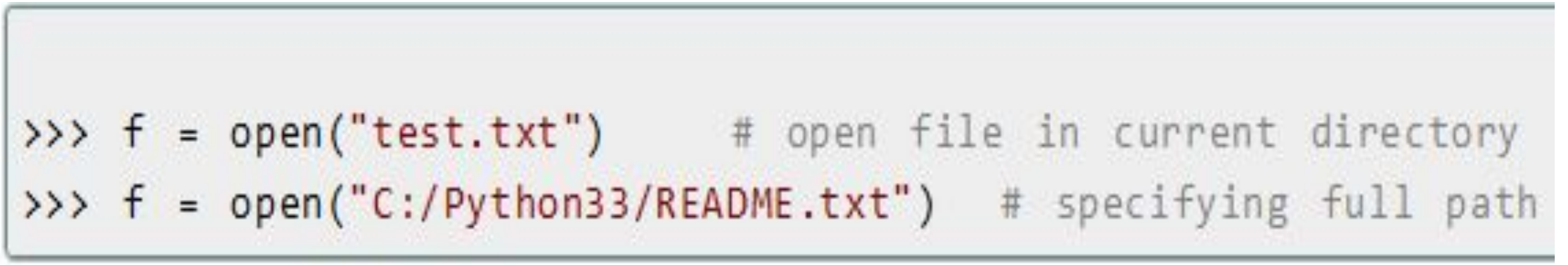
- Working with files is a lot like working with books.
- To use a book, you have to open it. When you're done, you have to close it.
- While the book is open, you can either write in it or read from it.
- In either case, you know where you are in the book.
- Most of the time, you read the whole book in its natural order, but you can also skip around.
- All of this applies to files as well. To open a file, you specify its name and indicate whether you want to read or write.
- Opening a file creates a file object.

Working with Files in Python Programming

- In Python, a file operation takes place in the following order.
  - Open a file
  - Read or write (perform operation)
  - Close the file

Opening a File

- Python has a built-in function open() to open a file.
- This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.



```
>>> f = open("test.txt")    # open file in current directory
>>> f = open("C:/Python33/README.txt")  # specifying full path
```

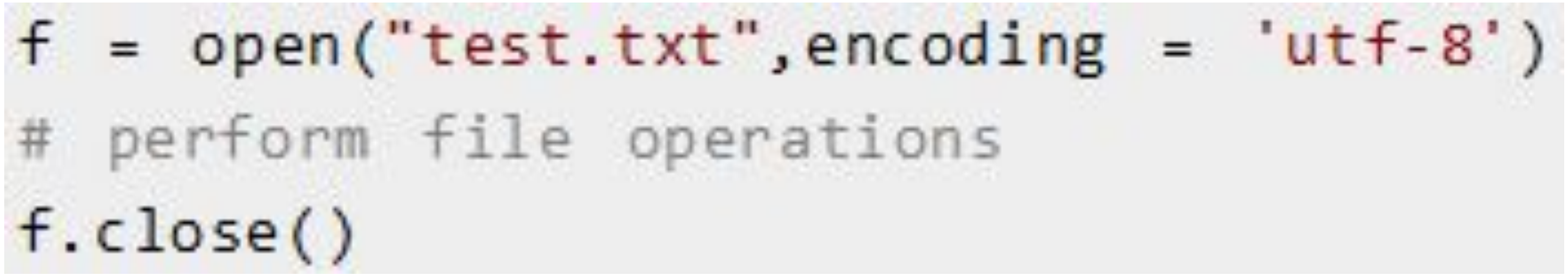
- Opening a file creates what we call a file **handle**.
- In this example, the variable **f** refers to the new handle object.
- Our program calls methods on the handle, and this makes changes to the actual file which is usually located on our disk.

Opening a File

- We can specify the mode while opening a file.
- In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file.
- We also specify if we want to open the file in text mode or binary mode.
- The default is reading in text mode.
- In text mode, we get strings when reading from the file.
- On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files.

Closing a File

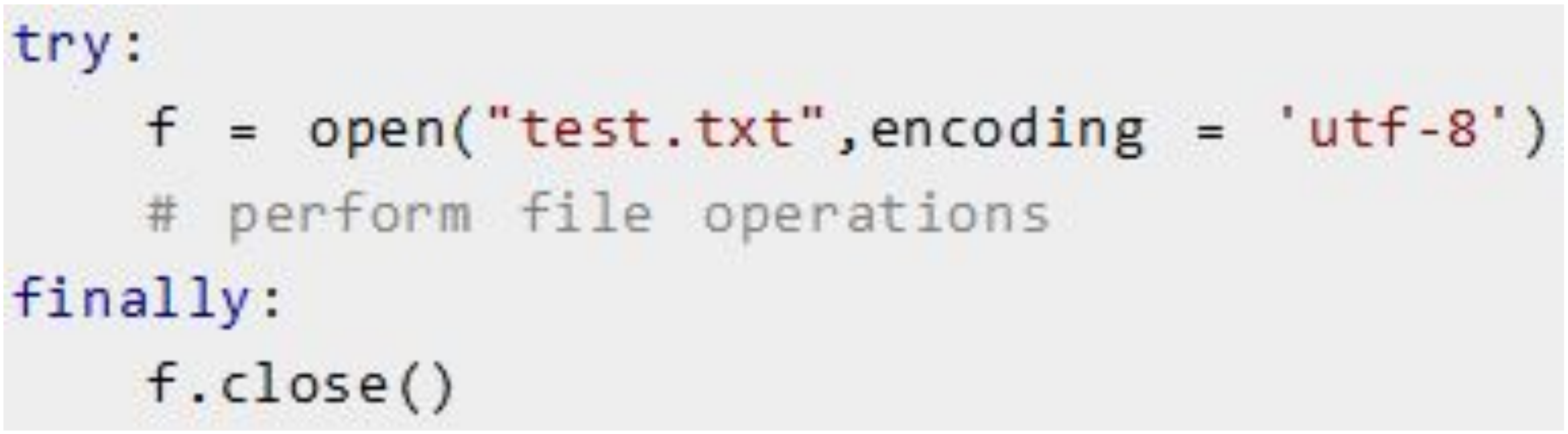
- When we are done with operations to the file, we need to properly close it.
- Python has a garbage collector to clean up unreferenced objects.
- But we must not rely on it to close the file.
- Closing a file will free up the resources that were tied with the file and is done using the close() method.



```
f = open("test.txt",encoding = 'utf-8')
# perform file operations
f.close()
```

Closing a File

- Previous method is not entirely safe.
- If an exception occurs when we are performing some operation with the file, the code exits without closing the file.
- A safer way is to use a try...finally block.

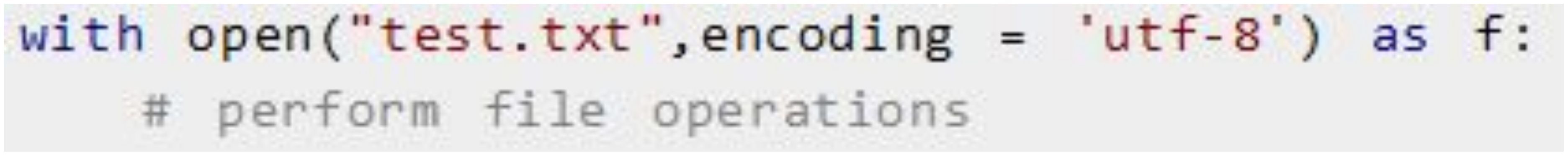


```
try:
   f = open("test.txt",encoding = 'utf-8')
   # perform file operations
finally:
   f.close()
```

This way, we are guaranteed that the file is properly closed even if an exception is raised, causing program flow to stop.

Closing a File

- The best way to do this is using the with statement.
- This ensures that the file is closed when the block inside with is exited.
- We don't need to explicitly call the close() method. It is done internally.



```
with open("test.txt",encoding = 'utf-8') as f:
   # perform file operations
```