Ways to import modules:

* Import module\_name
* Import package.module
* Import module as m
* From module import method
* From module import \*
* From package.module import entity

Dir() function:

* Shows all methods available in the given namespace.
* Ex. Dir(\_\_builtins\_\_) shows all methods that are available by default.
* Ex. Dir(math) shows all methods available in the math module.

Sys Module:

* A built-in module that contains methods and variables that can manipulate the runtime environment.
* Sys.version:
  + Returns a string containing the version of the Python interpreter
* Sys.path:
  + Sys is a built-in module that contains variables and methods that interact with the system or interpreter.
  + Sys.path contains a list of directories that the interpreter will search for imported modules.
  + You can add a new search directory using:
    - Sys.path.append(‘C:/path/to/directory’)
* Stdin:
  + Used to get input directly from the command line:
  + Ex.

**import** sys

**for** line **in** sys.stdin:

**if** 'q' **==** line.rstrip():

**break**

    print(f'Input : {line}')

print("Exit")

* Stdout:
  + A built-in file object that is analogous to the interpreter’s standard output stream in Python.
  + Used to display output directly to the console.
  + Ex.

**import** sys

sys.stdout.write('Geeks')

* Stderr:
  + Whenever an exception occurs, it is written to sys.stderr.
  + Ex.

**import** sys

**def** print\_to\_stderr(**\***a):

    # Here a is the array holding the objects

    # passed as the argument of the function

**print**(**\***a, file **=** sys.stderr)

print\_to\_stderr("Hello World")

* Argv:
  + When calling a Python program, you can also provide arguments. Those are saved in sys.argv, which is a list.
  + Ex.

**import** sys

# total arguments

n **=** len(sys.argv)

**print**("Total arguments passed:", n)

# Arguments passed

**print**("\nName of Python script:", sys.argv[0])

**print**("\nArguments passed:", end **=** " ")

**for** i **in** range(1, n):

    print(sys.argv[i], end **=** " ")

# Addition of numbers

Sum **=** 0

**for** i **in** range(1, n):

    Sum **+=** int(sys.argv[i])

print("\n\nResult:", Sum)

* Sys.exit(“arg”)
* Sys.modules
  + A list containing all of the currently imported modules.

Math Module

Constants:

* .pi = 3.141592
* .e = 2.7182818
* .tau = 6.283185 (pi\*2)
* .inf = floating point infinity

Methods:

* .ceil – ceiling function
* .floor – floor function
* .fabs – absolute value
* .factorial – factorial
* .sqrt – square root
* .copysign(x,y) – returns x with the sign of y.
* .gcd(x,y) – greatest common divisor between x and y.
* .lcm(x,y) – least common multiple between x and y.
* .comb(x,y) – number of possible combinations selecting y items from x.
* .isfinite(x) – returns True or False depending on whether x is finite or infinite.
* .hypot(x,y) – returns the Euclidean norm (distance to reach a set of coordinates)
  + Can be given any number of inputs.
* .trunc – returns the integral part of a floating point number.
* .log(x, y) – returns the logarithm of x with base y or natural log if y is not passed.
* .degrees – converts radian value to degrees.
* .radians – converts degree value into radians.
* .dist – returns Euclidean distance between two sets of coordinates.

Random Module

Methods:

* .seed – initialize a pseudo-random number sequence.
  + Can be given 0-2 arguments.
* .random – creates a random float number from 0 to 1.
* .randint – creates a random integer within a range.
* .choice – selects a random item from a given list.
* .choices – selects a number of random items from a given list.
  + Choice and Choices can have some arguments:
    - weights = [] (picking certain choices at higher rates)
    - k = # (length of the returned list)
* .shuffle – randomly shuffles the elements of a given sequence.
* .sample(seq, x) – returns a given sample of size x of a sequence.

Platform Module

Methods:

* .platform – displays the underlying platform with as much useful information as possible
  + Possible arguments: aliased, terse
* .processor – displays information about the system processor
* .architecture – displays the bit architecture
* .machine – displays the machine type (size of registers used in the core)
* .system – displays the name of the operating system
* .version – displays the system’s release version
* .node – displays the network information
* .uname – displays system, node, release, version, and machine as a named tuple
* .python\_implementation
* .python\_version\_tuple
* .python\_build – displays Python build date and number

Creating and Using User-Defined modules and packages:

* A python file can be written simply as an importable module or act as a top-level script. It is even possible to write a Python file that can act as both at the same time.

If \_\_name\_\_ == ‘\_\_main\_\_’:

Do\_thing

* \_\_name\_\_ is a built-in variable for Python scripts that indicates what file is being run as a top-level script.
* If the file is being run as a script, then \_\_name\_\_ will be set equal to ‘\_\_main\_\_’. If it is simply being imported, then this if-statement will evaluate to False, and not run.
* Python uses a list of directories when looking for required modules. These are the typical locations:
  + The program’s home directory
  + PYTHONPATH directories
  + Standard library directories (installed with Python)
  + Directories listed in .pth files
  + The site-packages directory
* Python has a built-in ‘reload’ function that dynamically reloads an imported module in the case of changes. This ensures that the process doesn’t need to start over from the beginning.

from importlib import reload

reload(math)

* \_\_init\_\_.py is a file contained in regular packages.
  + It indicates that the current directory is a python module package.
  + In Python 2.x this file was required to be included in all packages, but in 3.x it is no longer necessary.
  + The \_\_init\_\_.py file is often just an empty file, but it can also contain code to initialize parameters for the import.

\_\_pycache\_\_

* This is a directory that is sometimes created when you execute Python scripts.
* It contains .pyc files which are comprised of the bytecode compilations of the scripts. The purpose of this is to make it faster to run programs after they have been executed once already.
* The .pyc files may contain sensitive information such as database access credentials among other things, so it is wise to include \_\_pycache\_\_ in the .gitignore when publishing it to a remote repository.

Nested Packages vs Directory Trees:

* Nested Packages are sub-packages inside a Python package. In order to import a sub-package, you must use the ‘.’ notation in the import, specifying the higher order package and then the sub-packages after.
* Ex. Import pyspark.sql.functions
* ‘sql’ is a sub-package of pyspark, and ‘functions’ is a sub-package of ‘sql’.