UNIT-2

1. What is a file?

✓File is a named location on disk to store related information. It 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.

File object:

✓***File objects*** contain methods and attributes that can be used to collect information about the file you **opened.** They can also be used to manipulate said file.

✓To get a ***file object we*** use Python's built-in ***open*** function

How Python Handle Files?

✓If you are working in a large software application where they process a large number of data, then we cannot expect those data to be stored **in a variable** as the variables are **volatile** in nature.

✓As files are **non-volatile** in nature, the data will be **stored permanently** in a secondary device like Hard Disk.

Types Of File in Python

✓There are **two types of files** in Python.

They are:

1. Binary file
2. Text file

1. Binary files in Python

✓Most of the files that we see in our computer system are **called binary files.**

Example:

Document files: .pdf, .doc, .xls etc.

Image files: .png, .jpg, .gif, .bmp etc.

Video files: .mp4, .3gp, .mkv, .avi etc.

Audio files: .mp3, .wav, .mka, .aac etc.

Database files: .mdb, .accde, .frm, .sqlite etc.

Archive files: .zip, .rar, .iso, .7z etc.

Executable files: .exe, .dll, .class etc.

All binary files follow a **specific format.** We can open some binary files in the normal text editor but **we can't read the content present inside the file**. That's because all the binary files will be encoded in the binary format, which can be understood only by a computer or machine. For handling such binary files we need a specific type of software to open it.

For Example, You need Microsoft word software to open .doc binary files. Likewise, you need a pdf reader software to open .pdf binary files.

2. Text files in Python

✓Text files don’t have any specific encoding and it can be opened in normal text editor itself.

Example:

Web standards: html, XML, CSS, JSON etc.

Source code: c, app, js, py, java etc.

Documents: txt, tex, RTF etc.

Tabular data: csv, tsv etc.

Configuration: ini, cfg, reg etc.

.

2. Python File Handling Operations

Most importantly there are **4 types of operations** that can be handled by Python on files:

1. Open
2. Read
3. Write
4. Close

Other operations include:

1. Rename
2. Delete

1. How to open a file?

✓Python provides the **open() function** which accepts two arguments, **file name** and **access mode** in which the file is accessed.

✓ The function returns a **file object** which can be used to perform various operations like reading, writing, etc.

Syntax:

file_object = open(“filename”, “mode”)

- where ***file_object*** is the variable to add the file object.
- ***mode*** – tells the interpreter and developer which way the file will be used.

Access Modes for File Objects

File	*Mode Operation*
r	Open for read
w	Open for write (truncate if necessary)
a	Open for append (always works from EOF, create if necessary)
r+	Open for read and write
w+	Open for read and write (see w above)
a+	Open for read and write (see a above)
rb	Open for binary read
Wb	Open for binary write (see w above)
ab	Open for binary append (see a above)
rb+	Open for binary read and write (see r+ above)
wb+	Open for binary read and write (see w+ above)
ab+	Open for binary read and write (see a+ above)

Ex:

#open file

```
>>> f=open("demo","w")
>>> f
<_io.TextIOWrapper name='demo' mode='w' encoding='cp1252'>
```

#Create a text file

```
>>> f=open("demo.txt","w")        or        with open('demo.txt', 'w') as f:
>>> f
<_io.TextIOWrapper name='demo.txt' mode='w' encoding='cp1252'>
```

#to know the current path

```
>>> import os
>>> os.path
    <module 'ntpath' from C:\\Users\\ajee\\AppData\\Local\\Programs\\Python\\Python37-32\\lib\\ntpath.py'>
```

2.Reading a Text File in Python:

✓If you need to extract a string that contains all characters in the file.

Syntax:

fileobject.read()

Ex;

```
>>> f=open("abc.txt","r")
>>>a= f.read()
>>>print(a)
'welcome to python programming
 python is a very powerful language
 python is also simple programming language compare to other language‘
```

✓For suppose, I want to read a file is to call **a certain number of characters.**

```
>>> f=open("abc.txt","r")
>>> f.read(22)
'welcome to python prog'
```

✓If you want to read a file line by line , then you use the **readline()** function.

Ex: **#Displaying first line from file**

```
>>> f=open("abc.txt","r")
>>> a=f.readline()
>>>print(a)
welcome to python programming
```

or

```
>>> print(open("abc.txt","r").readline())
welcome to python programming
```

Or

```
>>> f=open("abc.txt","r")
>>> a=f.readlines()
>>> print(a[0])
welcome to python programming
```

#Displaying second line from file

```
>>> f=open("abc.txt","r")
>>> a=f.readlines()
Print(a[1])
python is a very powerful language
```

Note: for d**isplaying last line from file use print(a[-1])**

3. How to write in to file:

```
>>> f=open("demo.txt","w")
>>> f.write("this is first line\n")
18
>>> f.write("this is second line\n")
19
>>> f.write("this is third line\n")
18
>>> f.close()
```

✓Then open demo.txt file in your path

How to append new data in to existing file:

```
>>> f=open("abc.txt","a")
>>> f.write(" \n This is file concepts in python\n")
35
>>> f.write(" \nThis is write method")
22
>>> f.close()
```

Note: Then open abc.txt file in path:

4. Python Close File

✓In order to close a file, we must **first open the file**. In python, we have an in-built method called **close()** to close the file which is opened.

✓**Whenever you open a file, it is important to close it**, especially, with write method. Because if we don't call the close function after the write method then whatever data we have written to a file will not be saved into the file.

Example 1:

```
my_file = open("C:/Documents/Python/test.txt", "r")
print(my_file.read())
my_file.close()
```

Example 2:

```
my_file = open("C:/Documents/Python/test.txt", "w")
my_file.write("Hello World")
my_file.close()
```

Python Rename or Delete File

✓Python provides us with an **“os” module** which has some in-built methods that would help us in performing the file operations **such as renaming and deleting the file.**

✓In order to use this module, first of all, we need to import the “os” module in our program and then call the related methods.

1. rename() method:

✓This rename() method accepts two arguments i.e. the current file name and the new file name.

Syntax:

os.rename(current_file_name, new_file_name)

Ex:

```
>>> import os
>>> os.rename("new.txt","new1.txt")
```

2. remove() method:

✓We use the **remove() method** to delete the file by supplying the file name or the file location that you want to delete.

Syntax:

```
os.remove(file_name)
```

Example :

```
import os
os.remove("abc.txt")
```

3. File Built-in Methods:

Method	Description
close()	Closes the file
detach()	Returns the separated raw stream from the buffer
fileno()	Returns a number that represents the stream, from the operating system's perspective
flush()	Flushes the internal buffer
isatty()	Returns whether the file stream is interactive or not
read()	Returns the file content
readable()	Returns whether the file stream can be read or not
readline()	Returns one line from the file
readlines()	Returns a list of lines from the file
seek()	Change the file position
seekable()	Returns whether the file allows us to change the file position
tell()	Returns the current file position
truncate()	Resizes the file to a specified size
writeable()	Returns whether the file can be written to or not
write()	Writes the specified string to the file
writelines()	Writes a list of strings to the file

1. close() in Python

✓With close(), we close a file to free up the resources held by it.

syntax:

f.close()

---

2. detach() in Python

✓This detaches the underlying binary buffer from TextIOBase and returns raw stream from buffer.

syntax:

f.detach()

Ex:

```
>>> f=open("ab.txt","r")
>>> f.detach()
<_io.BufferedReader name='ab.txt'>
>>> f.read()
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    f.read()
ValueError: underlying buffer has been detached
```

3. fileno() in Python

✓It returns the **integer file descriptor** that is used by the underlying implementation to request I/O operations from the operating system

✓An error will occur if the operator system **does not use a file descriptor.**

Ex:

```
>>> f=open("abc.txt","r")
>>> f.fileno()
4
```

---

4. flush() in Python:

✓The flush() **method cleans out the internal buffer**.

✓**flush()** writes the **specified content** to the **operating system buffer** from the program buffer in event of a **power cut.**

✓Python **automatically flushes** the files when closing them. But you may want to flush the data before closing any file.

Ex:

```
>>> f=open("abc.txt","r")
>>> f.flush()
>>> f.close()
```

5. isatty() in Python

✓The method **isatty()** returns True if the file is connected (is associated with a terminal device) to a tty(-like) device, else False

Ex:

```
>>> f=open("abc.txt","r")
>>> f.isatty()
False
```

Note: tty(teletype) is a terminal which refers to serial ports of a computer, to which terminals are attached

---

6. read() in Python

✓The read() method returns the specified number of bytes from the file.

Syntax:

file.read(size)

✓Where size is the number of bytes to be read from the file.

Ex:

```
>>> f=open("abc.txt","r")
>>> a=f.read()
>>> print(a)
```

7. readable() in Python

✓This returns True if the **object** is readable

Ex:

>>> f=open("abc.txt",”r")

>>> **f.readable()**

True

>>> f=open("abc.txt","w")

>>> **f.readable()**

False

>>> f=open("abc.txt","a")

>>> f.readable()

False

8. readline( ) in Python:

✓The readline() method returns **one line from the file.**

✓**Syntax**

file.readline(*size*)

Ex:

```
>>> f=open("abc.txt","r")
>>> a=f.readline()
>>>print(a)
welcome to python programming
```

✓You can also specified **how many bytes from the line to return,** by using the size parameter.

Return only the five first bytes from the first line:

```
>>>f = open("demofile.txt", "r")
>>>print(f.readline(5))
```

9. readlines() in Python:

✓readlines() method **returns a list containing each line in the file as a list item**.

Ex:

```
>>> f=open("demo.txt","r")
>>> a=f.readlines()
>>> print(a)
['hi...hello...python‘,’welcome to python world']
```

10. seek() in python:

✓The seek() method sets the current file position in a file stream.

Syntax

file.seek(*offset*)

Where ***offset*** *is a* number representing the position to set the current file stream position.

Ex:

```
>>> f=open("demo.txt","r")
>>> f.seek(4)
>>> print(f.readline())                    #output:  o van rossam
```

11. seekable() in Python

✓This returns **whether file allows us to change the file position**

Ex:

```
>>> f=open("demo.txt","r")
>>> f.seekable()
    True
```

---

12. tell() in Python

✓tell() tells us the **current position of the cursor**.

Ex:

```
>>> f=open("demo.txt","r")
>>> f.seek(8)
8
>>> print(f.readline())
n rossam
>>> print(f.tell())
8
```

13. truncate():

✓This method **truncates the file's size**. If the optional size argument is present, the file is truncated to (at most) that size.

✓The *size* defaults to the current position

✓This method would not work in case **file is opened in read-only mode.**

Syntax:

fileObject.truncate([size])

✓**Example:**

```
>>> f=open("wel.txt","w")
>>> f.write("truncate function")
        17
>>> f.close()
                                >>> #wel.txt size is 17bytes
>>> f.truncate()
0
                                 >>> #wel.txt size is 0 bytes
```

14. writable() in Python

✓This returns True if the stream can be written to.

Ex:

```
>>> f=open("wel.txt","w")
>>> f.writable()
True
>>> f=open("wel.txt","r")
>>> f.writable()
False
```

15. write(s) in Python

✓ This method takes **string** ‘s’, and writes it to the file. Then, it returns the **number** of characters written.

Ex:

```
>>> f=open("wel.txt","w")
>>> f.write("truncate function")
```

16. writelines():

- ✓The method **writelines()** writes a sequence of strings to the file.
- ✓ The sequence can be any iterable object producing strings, typically a list of strings.
- ✓There is no return value.

Syntax:

fileObject.writelines(sequence)

Where **sequence** − This is the Sequence of the strings.

Ex:

```
>>> f=open("new.txt","w")
>>> a=f.writelines(["this is writelines function\n","previous function is write function"])
>>>f.close()
```

Note: open new.txt file in your path and see the text

4. Python file Built-in attributes:

Attribute	Description
Name	Return the name of the file
Mode	Return mode of the file
Encoding	Return the encoding format of the file
Closed	Return true if the file closed else returns false

Ex:

```
>>> f=open("new.txt","w")
>>> print("Name of the File:",f.name)
    Name of the File: new.txt
>>> print("Mode of the File:",f.mode)
    Mode of the File: w
>>> print("closed?:",f.closed)
    closed?: False
>>> f.close()
>>> print("closed?:",f.closed)
    closed?: True
```

>>> print("Encoding of the File:",**f.encoding**)

Encoding of the File: cp1252

Encoding in Files

- ✓File encoding represents **converting characters into a specific format** which only a machine can understand.

Different machines have different encoding format as shown below.

- ✓Microsoft Windows OS uses **'cp1252'** encoding format by default.
- ✓Linux or Unix OS uses **'utf-8'** encoding format by default.
- ✓Apple's MAC OS uses **'utf-8' or 'utf-16'** encoding format by default.

5. Standard files:

✓There are generally **three standard files** that are made available to you when your program starts.

1. **Standard input** is the data that goes to the program. The standard input comes from a keyboard.
2. **Standard output** is where we print our data with the print keyword. Unless redirected, it is the terminal console.
3. **The standard error** is a stream where programs write their error messages. It is usually the text terminal.

✓These files are named **stdin, stdout, and stderr** and take their names from the C language.

✓The standard input and output in Python are objects located in the **sys** module.

Object	Description
sys.stdin	standard input
sys.stdout	standard output
sys.stderr	standard error

✓These three are standard streams and these are **abstractions** used to provide a general method for computer programs to communicate with the physical input (keyboard) and output (screen) hardware of a computer.

✓When a computer program want's to display text on the screen**, it doesn't interact with the computer screen directly.**

✓ The program instead writes the text it wants to display to the *automatically created* file, STDOUT.

✓The operating system then takes the **STDOUT file,** and writes the text it contains to the computer's screen.

✓The same applies to STDIN. When a program asks the user for input from the keyboard, the operating system grabs the user input from the computer's keyboard, and stores it in the **STDIN** file. The user input can then be accessed by the program *via* the STDIN file.

✓ Before standard streams, each computer program has to accesses a computer's hardware directly. This was a **tedious task** and, since the hardware of each brand of computer had to be accessed differently, made it nearly impossible to make a completely platform independent program

✓*In modern times* however, operating systems now provide ***standard streams*** (STDIN, STDOUT, STDERR) as a **platform independent method** for interacting with a computer's input and output hardware.

Ex:

Python standard input

✓Standard input **is the data that goes to the program**

```
import sys
print('Enter your name: ')
c = sys.stdin.readline()
print('Your name is:', c)
```

```
Output:
Enter your name:    rajeev
Your name is: rajeev
```

Python standard output

✓The standard output is where we print our data.

```
import sys
sys.stdout.write('Honore de Balzac, Father Goriot\n')
sys.stdout.write('Honore de Balzac, Lost Illusions\n')
```

6. Command line arguments in python:

✓Till now, we have taken input in python using raw_input() or input() [for integers].

✓There is another method that uses command line arguments.

✓The command line arguments must be given whenever we want to give the input before the start of the script,

✓while on the other hand, raw_input() is used to get the input while the python program / script is running.

✓There are many options to read python command line arguments. The three most common ones are:

1. Python sys.argv
2. Python getopt module
3. Python argparsemodule

1. Python sys module

✓Python sys module stores the command line arguments into a list, we can access it using sys.argv.

✓The argv variable contains an array of strings consisting of each argument from the command line

✓In Python, the value for argc is simply the number of items in the sys.argv list, and

✓the first element of the list, sys.argv[0], is always the program name

✓Each list element represents a single argument. The first one -- sys.argv[0] -- is the name of the Python script. The other list elements -- sys.argv[1] to sys.argv[n] -- are the command line arguments 2 to n.

✓sys.argv *is the list of command-line arguments*

✓ len(sys.argv) *is the number of command-line arguments*

```
# file save as test.py
        import sys
        print("you entered", len(sys.argv), "arguments...")
        print("they were:", str(sys.argv))


>>>python test.py 2 3 4 5
you entered 5 arguments...
they were: [‘cmd.py',  '2', '3', '4', '5']
```

7. File System:

✓In python using **os module,** we can access to your file system.

✓This module serves as the primary interface to your operating system facilities and services from Python.

✓ The os module is actually a front-end to the **real module** that is loaded, a module that is clearly operating system dependent.

✓This "real" module may be one of the following:

✓**posix** (Unix-based, i.e., Linux, MacOS X, *BSD, Solaris, etc.), nt (Win32), mac (old MacOS), dos (DOS), os2 (OS/2), etc.

✓You should never import those modules directly.

✓ Just **import os** and the appropriate module will be loaded.

✓In os module, **common file or directory operations available** and we can managing process and the process execution environment

What is Directory in Python?

A directory or folder is a **collection of files and sub directories**. Python has the os module, which provides us with many useful methods to work with directories (and files as well).

Get Current Directory

✓We can get the present working directory using the getcwd() method.

✓This method returns the current working directory in the form of a string. We can also use the getcwdb() method to get it as bytes object.

Ex:

```
>>> import os
>>> os.getcwd()
    'C:\\Users\\ajee\\AppData\\Local\\Programs\\Python\\Python37-32‘
>>> os.getcwdb()
    b'C:\\Users\\ajee\\AppData\\Local\\Programs\\Python\\Python37-32‘
```

✓The extra backslash implies escape sequence.

```
    >>> print(os.getcwd())
    C:\Users\ajee\AppData\Local\Programs\Python\Python37-32
```

Changing Directory

✓We can change the current working directory using the chdir() method.

Ex:

```
>>> os.chdir('C:\\Python3')
>>> os.getcwd()
        'C:\\Python3'
>>> print(os.getcwd())
        C:\Python3
```

List Directories and Files

✓All files and sub directories inside a directory can be known using the listdir() method.

✓This method takes in a path and returns a list of sub directories and files in that path. If no path is specified, it returns from the current working directory.

```
>>> import os
>>> print(os.getcwd())
C:\Users\ajee\AppData\Local\Programs\Python\Python37-32
>>> os.listdir()
['a.txt', 'ab.txt', 'data.txt', 'demo', 'demo.txt', 'demo1', 'DLLs', 'Doc', 'dw.txt', 'include', 'Lib', 'libs',
'LICENSE.txt', 'new1.txt', 'NEWS.txt', 'python.exe', 'python3.dll', 'python37.dll', 'pythonw.exe‘]
```

Making a New Directory

✓We can make a new directory using the mkdir() method. This method takes in the path of the new directory.

✓If the full path is not specified, the new directory is created in the current working directory.

```
>>> os.mkdir('test')
>>> os.listdir()
```

['a.txt', 'ab.txt', 'data.txt', 'demo', 'demo.txt', 'demo1', 'DLLs', 'Doc', 'dw.txt', 'include', 'Lib', 'libs', 'LICENSE.txt', 'new1.txt', 'NEWS.txt', 'python.exe', 'python3.dll', 'python37.dll', 'pythonw.exe', 'tcl', **'test**‘]

Renaming a Directory or a File

✓The rename() method can rename a directory or a file.

✓The first argument is the old name and the new name must be supplies as the second argument.

```
>>> os.rename('test','new_test')
>>> os.listdir()
```

['a.txt', 'ab.txt', 'data.txt', 'demo', 'demo.txt', 'demo1', 'DLLs', 'Doc', 'dw.txt', 'include', 'Lib', 'libs', 'LICENSE.txt', 'new1.txt', 'NEWS.txt', **'new_test‘]**

Removing Directory or File

- ✓A file can be removed (deleted) using the remove() method.
- ✓Similarly, the rmdir() method removes an empty directory.

```
>>> os.listdir()
['a.txt', 'ab.txt', 'data.txt', 'demo', 'demo.txt', 'demo1', 'DLLs‘, 'empty file‘, ‘non-empty file’]
>>> os.remove('demo.txt')
['a.txt', 'ab.txt', 'data.txt', 'demo', 'demo1', 'DLLs‘, 'empty file‘, ‘non-empty file’]
>>>os.rmdir('empty file')
['a.txt', 'ab.txt', 'data.txt', 'demo', 'demo1', 'DLLs‘, ‘non-empty file’]
```

- ✓However, note that rmdir() method can only remove empty directories.

✓In order to remove a non-empty directory we can use the rmtree() method and import shutil module.

```
>>>os.rmdir('non-empty file')
        Traceback (most recent call last):
          File "<pyshell#21>", line 1, in <module>
            os.rmdir('non-empty file')
        OSError: [WinError 145] The directory is not empty: 'non-empty file‘
```

```
>>> import shutil
>>> shutil.rmtree('non-empty file')
>>> os.listdir()
['a.txt', 'ab.txt', 'data.txt', 'demo', 'demo1', 'DLLs‘]
```

Access() function:

- ✓Verify permission modes.
- ✓**os.access()** method uses the real uid/gid to test for access to path.

Syntax:

os.access(path, mode)

path: *path to be tested for access or existence*

mode: *Should be F_OK to test the existence of path, or can be the inclusive OR of one or more of R_OK, W_OK, and X_OK to test permissions.*

os.F_OK: Tests existence of the path.

os.R_OK: Tests readability of the path

.**os.W_OK:** Tests writability of the path.

os.X_OK: Checks if path can be executed.

Ex:

```
>>> import os
>>> import sys
>>> path1 = os.access("data.txt", os.F_OK)
>>> print("Exists the path:", path1)
        Exists the path: True
>>> path2 = os.access("data.txt", os.R_OK)
>>> print("Access to read the file:", path2)
        Access to read the file: True
>>> path3 = os.access("data.txt", os.W_OK)
>>> print("Access to write the file:", path3)
        Access to write the file: True
>>> path4 = os.access("data.txt", os.X_OK)
>>> print("Check if path can be executed:", path4)
        Check if path can be executed: True
```

os.stat()

os.stat() method in Python performs **stat()** system call on the specified path. This method is used to get status of the specified path.

Syntax:

os.stat(path)

path*: A string or bytes object representing a valid path*

Return Type: *This method returns a 'stat_result' object of class 'os.stat_result' which represents the status of specified path. The returned 'stat-result' object has following attributes:*

Ex:

```
>>> import os
>>> path='C:\\Users\\ajee\\AppData\\Local\\Programs\\Python\\Python37-32\\lib\\ntpath.py'
>>> status = os.stat(path)
>>> print(status)
    os.stat_result(st_mode=33206, st_ino=1970324837047016, st_dev=1056808185,
    st_nlink=1, st_uid=0, st_gid=0, st_size=23009, st_atime=1565676535,
    st_mtime=1562594058, st_ctime=1562594058)
```

st_mode: It represents file type and file mode bits (permissions).

st_ino: It represents the inode number on Unix and the file index on Windows platform.

st_dev: It represents the identifier of the device on which this file resides.

st_nlink: It represents the number of hard links.

st_uid: It represents the user identifier of the file owner.

st_gid: It represents the group identifier of the file owner.

st_size: It represents the size of the file in bytes.

st_atime: It represents the time of most recent access. It is expressed in seconds.

st_mtime: It represents the time of most recent content modification. It is expressed in seconds.

st_ctime: It represents the time of most recent metadata change on Unix and creation time on Windows. It is expressed in seconds.

os.chmod()

It is used to change the mode of path to the numeric mode.

Syntax:

os.chmod(path, mode)

Parameters:

path – *path name of the file or directory path*

mode *– mode may take one of the following values:*

stat.S_ISUID : *Set user ID on execution*

stat.S_ISGID : *Set group ID on execution*

stat.S_ENFMT : *Record locking enforced*

stat.S_ISVTX : *Save text image after execution*

stat.S_IREAD : *Read by owner.*

stat.S_IWRITE : *Write by owner.*

stat.S_IEXEC : *Execute by owner.*

stat.S_IRWXU : *Read, write, and execute by owner*

stat.S_IRUSR : *Read by owner*

stat.S_IWUSR : *Write by owner.*

stat.S_IXUSR : *Execute by owner.*

stat.S_IRWXG : *Read, write, and execute by group*

stat.S_IRGRP : *Read by group*

stat.S_IWGRP : *Write by group*

stat.S_IXGRP : *Execute by group*

stat.S_IRWXO : *Read, write, and execute by others.*

stat.S_IROTH : *Read by others*

stat.S_IWOTH : *Write by others*

stat.S_IXOTH : *Execute by others*

Ex:

```
>>> import os, sys, stat
>>> os.chmod("data.txt", stat.S_IREAD)
>>> print("File can be read only by owner.")
File can be read only by owner.
>>> os.chmod("data.txt", stat.S_IROTH)
>>> print("File access changed, can be read by others now.")
File access changed, can be read by others now.
```

os.walk()

✓os.walk() function is used to generate the file names in a directory tree by walking the tree either top-down or bottom-up

Ex:

```
>>> import os
>>> path = "C:/Users/ajee/Desktop/py"
>>> files=os.walk(path)
>>> print(files)
                o/p:  <generator object walk at 0x01ECFE70>
>>> for files in os.walk(path):
        print(files)
```

```
('C:/Users/ajee/Desktop/py', ['New folder'], ['cmdline.py', 'cmdline.txt', 'hello.py',
'hello.txt', 'walk.py', 'walk.txt', 'ww..py', 'ww.py.txt'])
('C:/Users/ajee/Desktop/py\\New folder', [], [])
```

os.chown():

✓**os.chown()** method is used to change the owner and group id of the specified path to the specified numeric owner id (UID) and group id (GID).

✓**os.chown()** method is available only on UNIX platforms and the functionality of this method is typically available only to the superuser or a privileged user.

Syntax: os.chown(path, uid, gid)

Parameters:

path: A file descriptor representing the file whose uid and gid is to be set

uid: An integer value representing the owner id to be set for the path.

gid: An integer value representing the group id to be set for the path. To leave any one of the ids unchanged, set it to -1.

EX:

```
import os
path = "C:/Users/ajee/Desktop/py/chown.txt“
print("Owner id of the file:", os.stat(path).st_uid)      #1000
print("Group id of the file:", os.stat(path).st_gid)      #1000
uid = 2000                              # Change the owner id
gid = 2000                              # the group id of the file
os.chown(path, uid, gid)
print("\nOwner and group id of the file changed")
print("\nOwner id of the file:", os.stat(path).st_uid)     #2000
print("Group id of the file:", os.stat(path).st_gid)       #2000
```

NOTE: os.stat() method will return a 'stat_result’ object of ‘os.stat_result’ class whose 'st_uid' and 'st_gid' attributes will represent owner id and group id of the file respectively

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.

Creation of a module

Create a Module

- To create a module just save the code you want in a file with the file extension .py:

Example

- Save this code in a file named mymodule.py

```
def greeting(name):
    print("Hello, " + name)
```

Usage of a module

Use a Module

- Now we can use the module we just created, by using the import statement:

Example

- Import the module named mymodule, and call the greeting function:

```
import mymodule
mymodule.greeting("Jonathan")
```

Variables in Module

- The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Example:-

- Save this code in the file varmodule.py

```
person1 = {
    "name": "John",
    "age": 36,
    "country": "Norway"
  }
```

Example

- Import the module named varmodule, and access the person1 dictionary:

```
import mymodule
a = varmodule.person1["age"]
  print(a)
```

Re-naming a Module

- You can create an alias when you import a module, by using the as keyword:

Example

- Create an alias for mymodule called mx:

```
import mymodule as mx

a = mx.person1["age"]

print(a)
```

8. Persistent Storage Modules

✓python has modules that let you save Python objects. Saving an object actually takes two steps:

1. **serialization and**
2. **persistence**.

✓***Serialization*** (sometimes called marshaling) is the process of converting an object into a stream of bytes. The stream of bytes can be a textual or binary representation of the original object.

✓***Persistence*** means saving that representation to some sort of data store that lives beyond your program's execution time or interactive shell session. Keep in mind that before you *persist* an object, it must be serialized.

1. pickle *and marshal Modules:*

✓Python provides a variety of modules that implement **minimal persistent storage**. One set of modules **(marshal and pickle)** allows for **pickling** of Python objects.

✓ **Pickling** is the process whereby objects more complex than primitive types can be converted to a binary set of bytes that can be stored or transmitted across the network, then be converted back to their original object forms.

✓Pickling is also known as **flattening, serializing, or marshalling**.

✓Another set of modules **(dbhash/bsddb, dbm, gdbm, dumbdbm)** and their "manager" **(anydbm)** can **provide persistent storage of Python** strings only.

✓The last module (shelve) can do both.

✓But both marshal and pickle can **flatten Python objects.** These modules do not provide a **namespace for the objects**, nor **can they provide concurrent write access to persistent objects**. Storage, is sequential in nature(you store or transmit objects one after another).

✓The difference between **marshal and pickle** is that—

- ❑ **marshal** can handle only simple Python objects (numbers, sequences, mapping, and code).
- ❑while **pickl**e can transform recursive objects, objects that are multi-referenced from different places, and user-defined classes and instances.

✓The pickle module is also available in a turbo version called **cPickle,** which implements all functionality in C.

2. DBM-style Modules:

✓The db* series of modules **writes data in the traditional DBM format.** There are a large number of different implementations: **dbhash/bsddb, dbm, gdbm, and dumbdbm.**

✓ If you are particular about any specific DBM module, feel free to use your favorite, but if you are not sure or do not care, the generic **anydbm module** detects which DBM-compatible modules are installed on your system and uses the "best" one at its disposal.

✓**anydbm** is a **front-end for DBM-style databases** that **use simple string values as keys** to access records containing strings. It uses the **whichdb module** to identify dbhash, gdbm, and dbm databases, then opens them with the appropriate module. It is also used as a **backend for shelve,** which knows how to store objects using pickle.

✓The **dumbdbm module** is the most limited one, and is the default **used if none of the other packages is available.**

3. shelve *Module:*

✓Shelve is a python module used **to store objects in a file**.

✓The shelve module **implements persistent storage for arbitrary Python objects** which can be pickled, using a **dictionary**-like API.

✓The shelve module can be used as a **simple persistent storage option for Python objects** when a relational database is overkill. The shelf is accessed by **keys**, just as with a dictionary.

✓ The values are pickled and written to a database created and managed by **anydbm.**

✓The shelve module uses the **anydbm module** to **find a suitable DBM module,** then uses **cPickle** to perform the **pickling process.**

✓The shelve module permits **concurrent read access to the database file, but not shared read/write access.**

The diagram shows the relationship between the **pickling modules** and **the persistent storage modules**, and **how the shelve object appears to be the best of both worlds**.

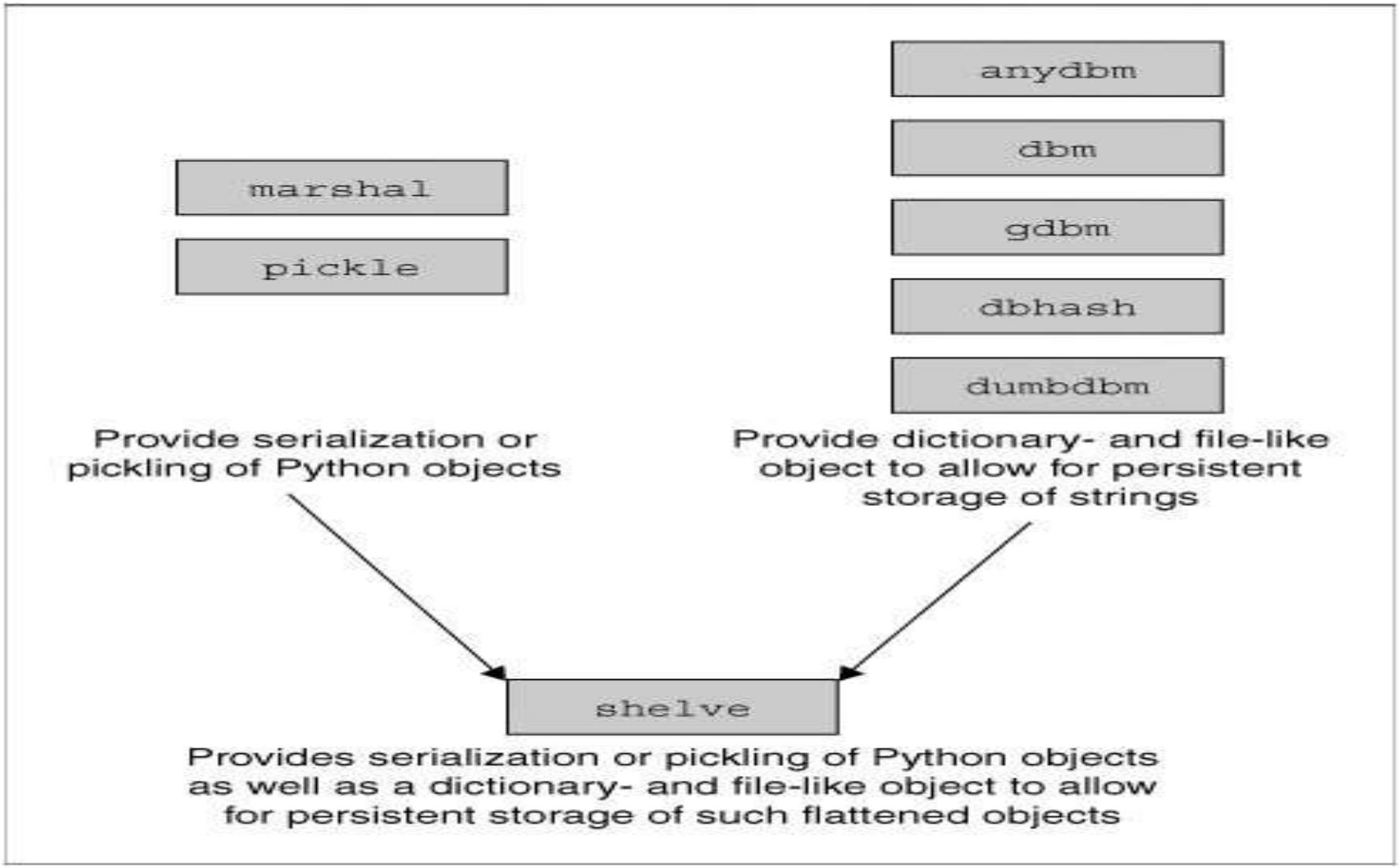


Fig: Python modules for serialization and persistency

Error and Exceptions

What Are Exceptions?

Errors:

- ✓ In the context of software, errors are either syntactical or logical in nature.
- ✓ **Syntax errors** indicate errors with the construct of the software and cannot be executed by the interpreter or compiled correctly.
- ✓ These errors must be repaired before execution can occur.
- ✓ Once programs are semantically correct, the only errors that remain are logical.
- ✓ **Logical errors/ Run time errors** can either be caused by lack of or invalid input, or by the inability of the logic to generate, calculate, or otherwise produce the desired results based on the input.
- ✓ **Run time errors** are also called **Exceptions**
- ✓ These errors are sometimes known as **domain and range failures**, respectively.

Exceptions:

✓An exception can be defined as **an abnormal condition in a program resulting in the disruption in the flow of the program**.

✓Whenever an exception occurs, the program halts the execution, and thus the further code is not executed. Therefore, an exception is the error which python script is unable to tackle with.

✓Python provides us with the way to handle the Exception so that the other part of the code can be executed without any disruption. However, if we do not handle the exception, the interpreter doesn't execute all the code that exists after the that.

There are **two types of exception:**

1. System defined exceptions
2. User-defined exceptions

1. System defined exceptions:

A list of System defined exceptions that can be thrown from a normal python program is given below.

ZeroDivisionError: Occurs when a number is divided by zero.

NameError: It occurs when a name is not found. It may be local or global.

IndentationError: If incorrect indentation is given.

IOError: It occurs when Input Output operation fails.

EOFError: It occurs when the end of the file is reached, and yet operations are being performed.

TypeError

1. **ZeroDivisionError:**

   Occurs when a number is divided by zero

```
>>> a=eval(input("enter a value"))
enter a value4
>>> b=eval(input("enter b value"))
enter b value0
>>> c=a/b


Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    c=a/b
ZeroDivisionError: division by zero
```

2.Type error:

Ex:

```
>>> a=eval(input("enter a value"))
enter a value4
>>> b=eval(input("enter b value"))
enter b value5
>>> c=a/'b'
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    c=a/'b'
TypeError: unsupported operand type(s) for /: 'int' and 'str‘
```

3. NameError:

Ex:

```
>>> a=eval(input("enter a value"))
enter a value4
>>> b=eval(input("enter b value"))
enter b value6
>>> c=a/d
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    c=a/d
NameError: name 'd' is not defined
```

4. SyntaxError: ***Python interpreter syntax error***

```
>>> for
File "<string>", line 1
for
^
SyntaxError: invalid syntax
```

✓**SyntaxError** exceptions are the only ones that **do not occur at run-time.**

✓ They indicate an improperly constructed piece of Python code which cannot execute until corrected.

✓ **These errors are generated at compile-time**, when the interpreter loads and attempts to convert your script to Python bytecode

5. IndexError: *request for an out-of-range index for sequence*

```
>>> aList = []
>>> aList[0]
Traceback (innermost last):
File "<stdin>", line 1, in ?
IndexError: list index out of range
```

✓IndexError is raised when **attempting to access** an index that is **outside the valid range** of a sequence.

6. KeyError: *request for a non-existent dictionary key*

```
>>> aDict = {'host': 'earth', 'port': 80}
>>> print(aDict['server'])
Traceback (innermost last):
File "<stdin>", line 1, in ?
KeyError: server
```

✓Mapping types such as dictionaries depend on **keys to access data values**. Such values are not retrieved if an incorrect/nonexistent key is requested.

7.IOError: ***input/output error***

```
>>> f=open("blah")
Traceback (innermost last):
File "<stdin>", line 1, in ?
IOError: [Errno 2] No such file or directory: 'blah‘
```

✓**Attempting to open a nonexistent disk file** is one example of an operating system input/output (I/O) error. Any type of I/O error raises an IOError exception

Exception handling in python:

1. ZeroDivisionError:

Occurs when a number is divided by zero.

```
>>> a=eval(input("enter a value"))
enter a value4
>>> b=eval(input("enter b value"))
enter b value0
>>> c=a/b
```

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in
<module>
    c=a/b
ZeroDivisionError: division by
zero
```

✓If the python program contains suspicious code that may throw the exception, we must place that code in the try block.

✓The **try block** must be followed with the **except statement** which contains a block of code that will be executed if there is some exception in the try block.

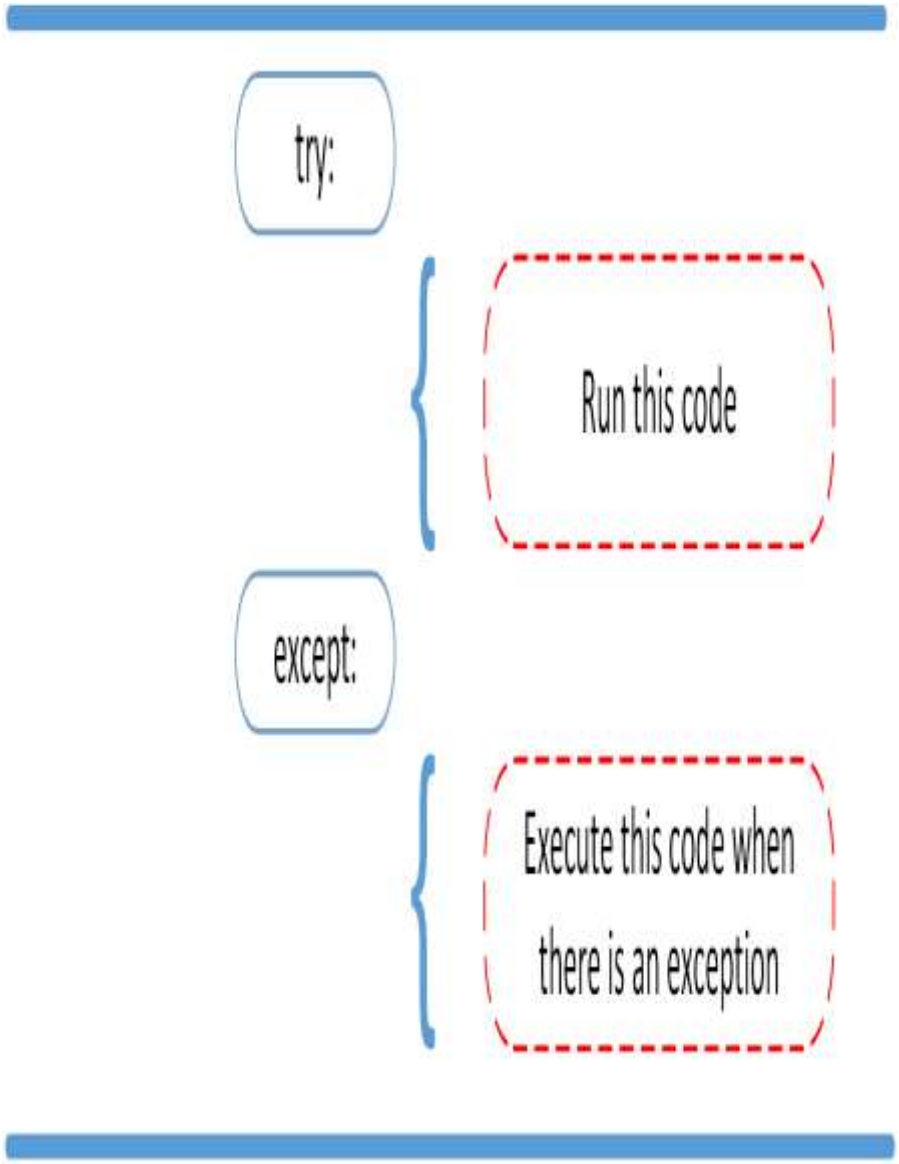
✓When an error occurs with in try block .python looks for matching **except block** to handle it ,if there is an error ,execution jumps there.

Ex:

```
a=int(input("enter a value"))
b=int(input("enter b value"))
try:
    c=a/b
    print("the result is",c)
except:
     print("error occured“)
```

```
C:\Users\ajee\Desktop\py>exception.py
enter a value4
enter b value0
error occured
```

```
C:\Users\ajee\Desktop\py>exception.py
enter a value4
enter b value2
The result is 2
```



2.Type error:

Ex:

```
>>> a=eval(input("enter a value"))
enter a value4
>>> b=eval(input("enter b value"))
enter b value5
>>> c=a/'b'
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    c=a/'b'
TypeError: unsupported operand type(s) for /: 'int' and 'str‘
```

```
a=int(input("enter a value"))
b=int(input("enter b value"))
try:
    c=a/'b'
    print("the result is",c)
except:
     print("error occured")
```

C:\Users\ajee\Desktop\py>typeerror.py
enter a value3
enter b value 4
error occured

3. NameError:

Ex:

```
>>> a=eval(input("enter a value"))
enter a value4
>>> b=eval(input("enter b value"))
enter b value6
>>> c=a/d
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    c=a/d
NameError: name 'd' is not defined
```

```
a=int(input("enter a value"))
b=int(input("enter b value"))
try:
    c=a/d
    print("the result is",c)
except:
     print("error occured")
```

C:\Users\ajee\Desktop\py>nameerror.py
enter a value2
enter b value3
error occured

except Statement with Multiple Exceptions:

✓The python allows us to declare the multiple exceptions with the except clause. Declaring multiple exceptions is useful in the cases where a try block throws multiple exceptions.

Ex:

```
a = int(input("Enter a:"))
b = int(input("Enter b:"))
try:
    c = a/b;
    print("a/b = %d"%c)
except ZeroDivisionError:
        print("please enter non-zero for denimoniator")
except NameError:
        print("please use defined names only")
except TypeError:
        print("please use proper types only")
```

```
C:\Users\ajee\Desktop\py>except.py
Enter a:4
Enter b:0
please enter non-zero for denimoniator
```

```
a = int(input("Enter a:"))
b = int(input("Enter b:"))
try:
        c = a/'b';
        print("a/b = %d"%c)
except ZeroDivisionError:
                print("please enter non-zero for denimoniator")
except NameError:
                print("please use defined names only")
except TypeError:
                print("please use proper types only")
```

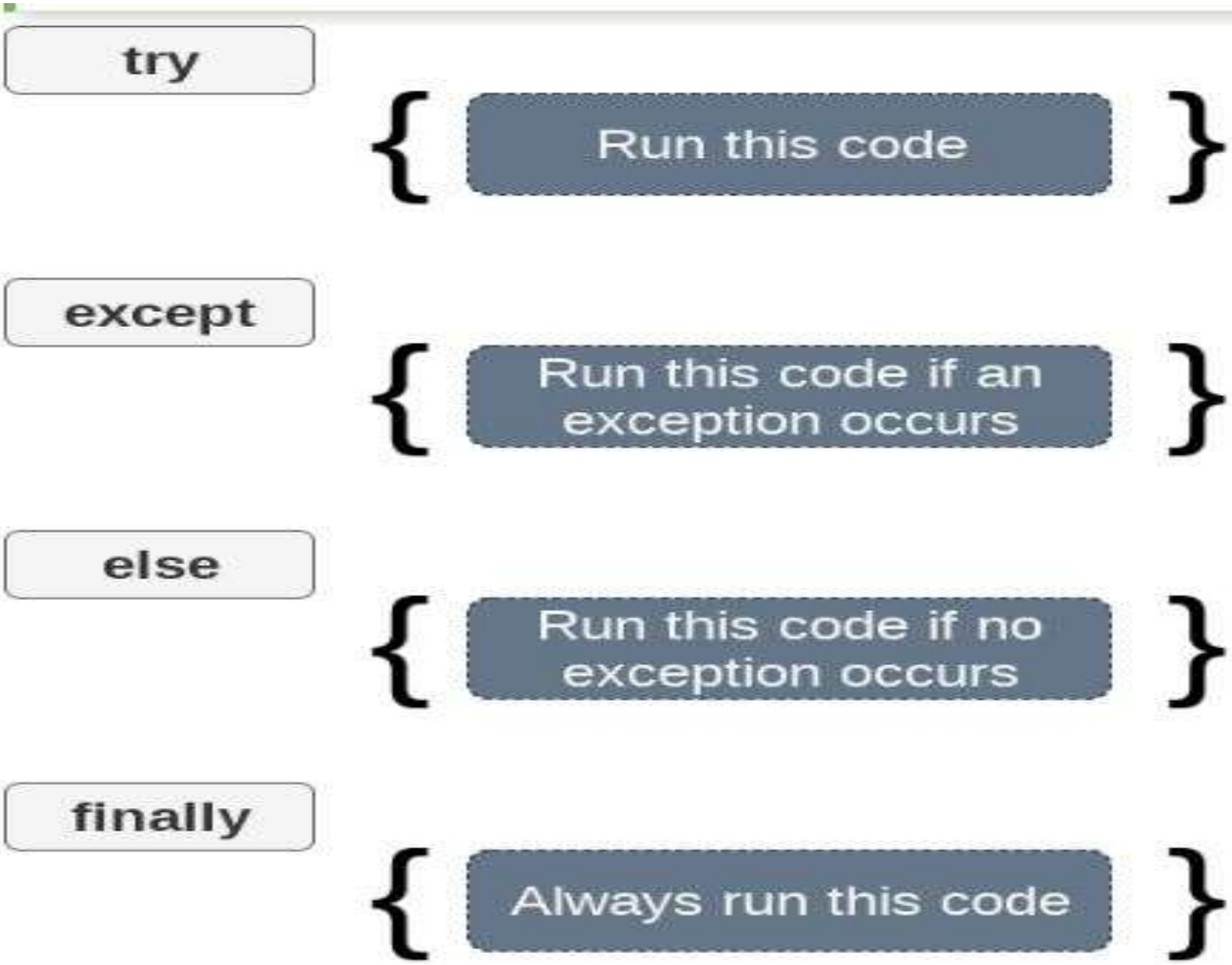
```
C:\Users\ajee\Desktop\py>except.py
Enter a:4
Enter b:2
please use proper types only
```

```
a = int(input("Enter a:"))
b = int(input("Enter b:"))
try:
        c = a/d;
        print("a/b = %d"%c)
except ZeroDivisionError:
                print("please enter non-zero for denimoniator")
except NameError:
                print("please use defined names only")
except TypeError:
                print("please use proper types only")
```

```
C:\Users\ajee\Desktop\py>except.py
Enter a:4
Enter b:6
please use defined names only
```

Some of the exception errors

```
except IOError:
    print('An error occurred trying to read the file.')
except ValueError:
    print('Non-numeric data found in the file.')
 except ImportError:
    print "NO module found"
except EOFError:
    print('Why did you do an EOF on me?')
 except KeyboardInterrupt:
    print('You cancelled the operation.')
except:
    print('An error occurred.')
```


try
{ Run this code }
except
{ Run this code if an exception occurs }
else
{ Run this code if no exception occurs }
finally
{ Always run this code }

else block:

✓We can also use the **else statement** with the **try-except statement** in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.

Ex:

```
a = int(input("Enter a:"))
b = int(input("Enter b:"))
try:
    c = a/b
except:
    print("can't divide by zero")
else:
    print("a/b = %d"%c)
```

```
C:\Users\ajee\Desktop\py>else.py
Enter a:4
Enter b:0
can't divide by zero
```

```
C:\Users\ajee\Desktop\py>else.py
Enter a:4
Enter b:2
a/b = 2
```

Declaring multiple exceptions (or)except Statement with Multiple Exceptions

✓The python allows **us to declare the multiple exceptions** with the except clause.

✓ Declaring multiple exceptions is useful in the cases where a try block throws multiple exceptions.

Syntax

try:

#block of code

except (<Exception 1>,<Exception 2>,<Exception 3>,...<Exception n>)

#block of code

else:

#block of code

Handling All Exceptions:

If you really want to handle all errors, you can still do that too, but use **BaseException**

try:

except BaseException, e:

handle all errors

or

try:

except Exception, e:

handle real errors

- BaseException

|- KeyboardInterrupt

|- SystemExit

|- **Exception**

|- (all other current built-in exceptions)

Finally

The finally block will be executed regardless **if the try block raises an error or not**.

```
a = int(input("Enter a:"))
b = int(input("Enter b:"))
try:
    c = a/b
      print("a/b = %d"%c)
except :
      print("please enter non-zero for denimoniator")
else:
      print("Hi I am else block")
finally:
       print("Thank you")
```

```
C:\Users\ajee\Desktop\py>except.py
Enter a:4
Enter b:3
a/b = 1
Hi I am else block
Thank you
```

```
C:\Users\ajee\Desktop\py>except.py
Enter a:4
Enter b:0
please enter non-zero for denimoniator
Thank you
```

Assert statement in python:

✓Python has built-in **assert statement** to use assertion condition in the program.

✓ The assert keyword is used when debugging code.

✓The assert keyword lets you test if a condition in your code returns True, if not, the program will raise an AssertionError.

Syntax:

assert <condition>,<error message>

Here error message is optional

Ex:

```
x = "hello"
#if condition returns True, then nothing happens:
assert x == "hello"
#if condition returns False, AssertionError is raised:
assert x == "Good Bye"
```

o/p:

```
hello
Traceback (most recent call last):
  File "main.py", line 17, in <module>
    assert  x == "goodbye", "x should be 'hello‘”
AssertionError: x should be 'hello'
```

2. User defined exception:

✓User can also define exceptions to indicate something is going wrong in your program ,those are called **customized exception or programmatic exception.**

✓The **raise keyword** is used to raise an exception.

✓You can define **what kind of error** to raise, and the text to print to the user.

ex: Raise an error and stop the program if x is lower than 0:

```
x = -1
if  x < 0:
  raise Exception("Sorry, no numbers below zero“)
```

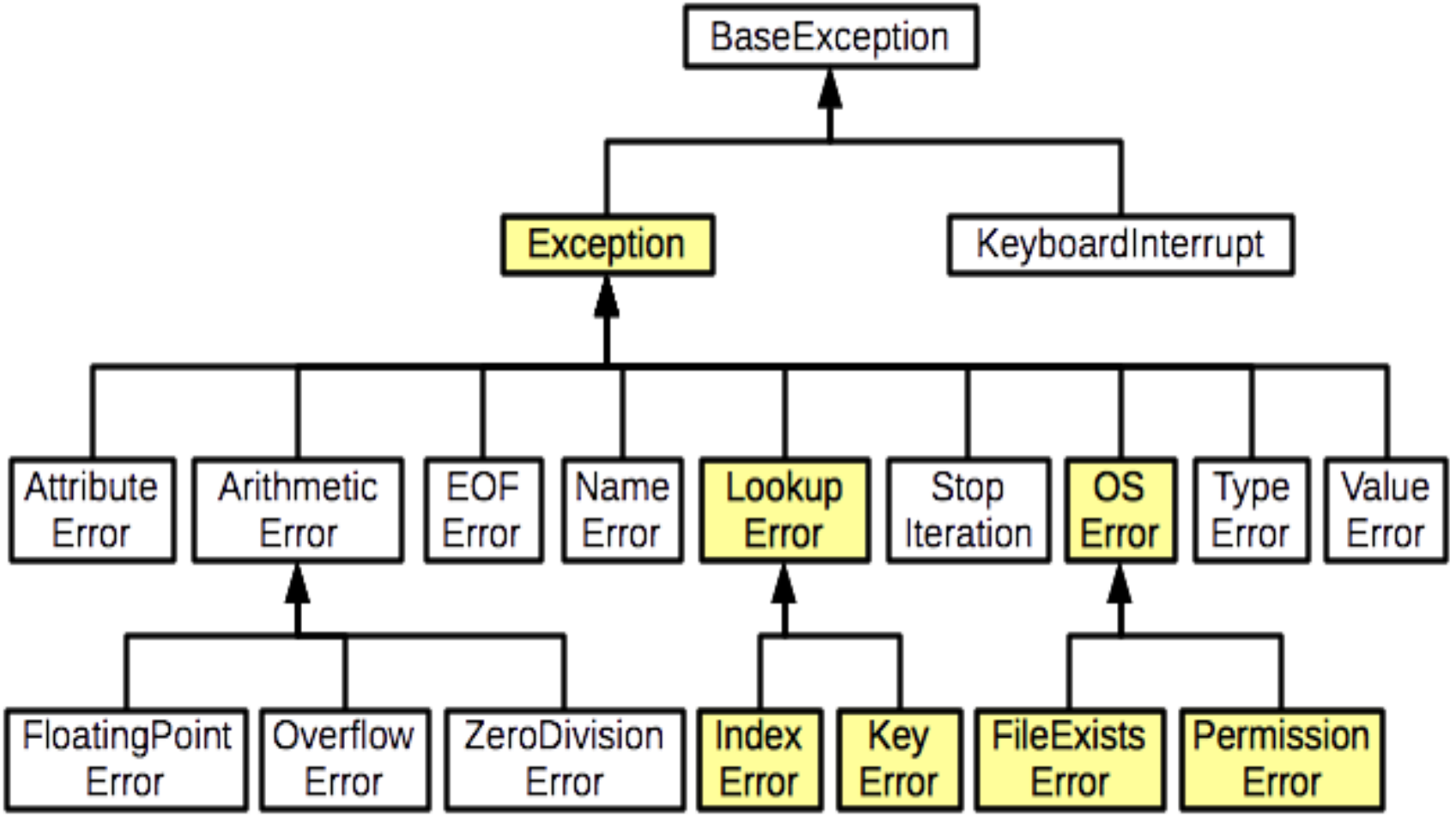
o/p:

```
Traceback (most recent call last):
  File "demo_ref_keyword_raise.py", line 4, in <module>
    raise Exception("Sorry, no numbers below zero")
Exception: Sorry, no numbers below zero
```

Ex: Raise a TypeError if x is not an integer:

```
x = "hello”
if not type(x) is int:
  raise TypeError("Only integers are allowed")
```

Python exception hierarchy or standard exceptions:



Exceptions and the sys Module:

✓An alternative way of **obtaining exception information** is by accessing the **exc_info()** function in the **sys** module. This function provides **a 3-tuple of information.**

1. **exc_type:** exception class object
2. **exc_value:** (this) exception class instance object
3. **exc_traceback**: traceback object

Ex:

```
>>> try:
        float('abc123')
    except:
        import sys
        exc_tuple = sys.exc_info()
>>> print exc_tuple
(  <class exceptions.ValueError at f9838>, <exceptions.ValueError instance at 122fa8>,
<traceback object at 10de18>   )
```

MODULES

What are modules in Python?

✓If we need to reuse the code then we can **define functions**, but if we need to reuse number **of functions** then we have to go for **modules**

✓Modules refer to a **file containing Python statements and definitions**.

✓A python module can be defined as a python program file which contains a python code **including python functions, class, or variables.**

✓for e.g.: **example.py**, is called a module and its module name would be **example.**

✓Modules provide **reusability of code.** Modules provides us the flexibility to organize the code in a logical way.

✓**To use the functionality of one module into another**, we must have to **import** the specific module.

Modules are three types:

1. Standard modules
2. User-defined modules
3. Third party modules

1.Standard modules:

- ✓ These modules are **already defined and kept in python software .**so, when we install python then automatically these standards modules will install in our machine

Ex: math, calender, os, sys,..etc

2. User-defined modules

- ✓ These modules are defined by **user** as per their requirement .so, here **.py file** contains methods, varaibles, and classes.

3. Third party modules:

- ✓ These modules are **already defined by some other people and kept in internet** . So that, we can download and install in our machines by using PIP(pyhton package installing)
- ✓ PIP is a package management system used to install and manage software packages written in python like..**pymysql, cx_oracle**

We can import module in three ways:

1. import <module_name>
2. from <module_name> import <method_name>
3. from <module_name> import *

1. import <module_name>:

- ✓ We can **import all methods** which is presented in that specified modules

  Ex: import math

- ✓ We can **import all methods** which are available in **math module**

2. from <module_name> import <method_name>:

- ✓ This import statement **will import a particular method** from the module which is specified in import statement.

  Ex: **from math import pi**

- ✓ **We cannot use other methods** which are available in that module.
- ✓ But, we need to **avoid this type of modules** as they may cause **name clashes** in the current python file

3. from <module_name> import * :

✓This import statement will **import all methods** from the specified module and also we can import all names(definitions) from a module

Ex: from math import *

Import with renaming

✓We can import a module by renaming it as follows.

```
>>> import math
>>> math.pi
3.141592653589793

>>> import math as m
>>> m.pi
3.141592653589793
```

dir() built-in function

✓We can use the **dir() function** to find out names that are defined inside a module.

>>> **dir(math)**

['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']

Math module:

```
# importing built-in module math
import math
    # using square root(sqrt) function contained
    # in math module
        print math.sqrt(25)

    # using pi function contained in math module
        print math.pi

    # 2 radians = 114.59 degreees
        print math.degrees(2)

    # 60 degrees = 1.04 radians
        print math.radians(60)

    # Sine of 2 radians
        print math.sin(2)

    # Cosine of 0.5 radians
        print math.cos(0.5)

    # Tangent of 0.23 radians
        print math.tan(0.23)
```

```
>>> math.ceil(10.9)
11
>>> math.floor(10.9)
10
>>> math.fabs(10)
10.0
>>> math.fabs(-10)
10.0
>>> math.fmod(10,2)
0.0
>>> math.fsum([1,2,4])
7.0
>>> math.pow(2,3)
8.0
# 1 * 2 * 3 * 4 = 24
        print math.factorial(4)
```

OS Module:

✓OS module provides functions for **interacting with the operating system**.

✓ OS, comes under Python's standard utility modules. **This module provides a portable way of using operating system dependent functionality**.

✓The *os* and *os.path* modules include many functions to interact with the file system.

1. **os.name:** This function gives the name of the operating system dependent module(real module) imported.

```
>>> import os
>>> os.name
'nt'
```

2.os.getcwd()
3.os.close():
4.os.rename()
5.os.mkdir("d:\\tempdir")
6.os.chdir("d:\\tempdir")
7.os.rmdir("d:\\tempdir")
8.os.listdir("c:\python37")

Sys module:

import sys

sys.modules # gives the names of the existing python modules current shell has imported.

sys.argv #collects the String arguments passed to the python script.

sys.path **#d**isplays the PYTHONPATH set in current system.

sys.stdin **# is used to take input**

sys.stdout **# is used to display message**

sys.stderr **# is used to display error message**

sys.copyright #displays the copyright information on currently installed Python version.

sys.exit #makes the Python interpretor exits the current flow of execution abruptly.

Random module:

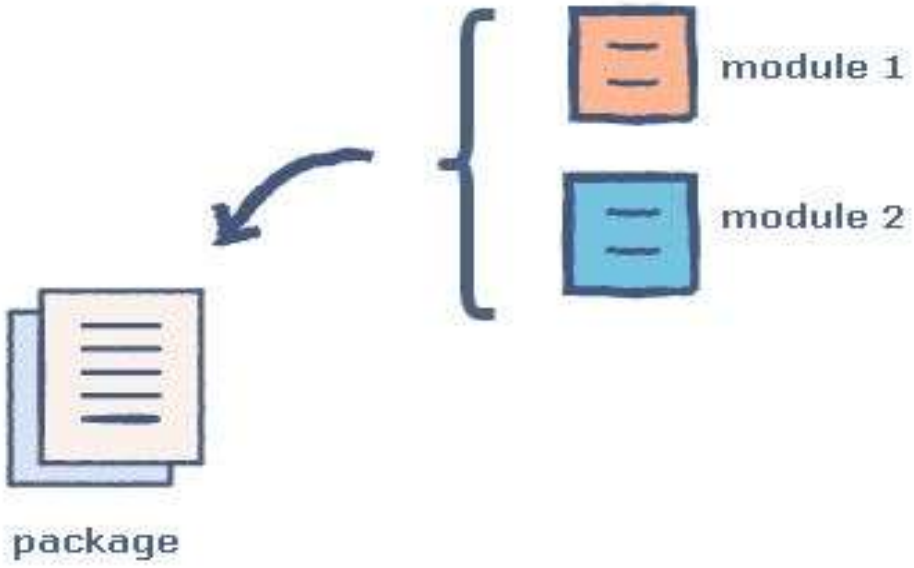
```
# importing built in module random
import random

    # printing random integer between 0 and 5
        print random.randint(0, 5)

    # print random floating point number between 0 and 1
        print random.random()

    # random number between 0 and 100
        print random.random() * 100
```

Platform module

```
    # to know your operting system

        >>> import platform

        >>> platform.system()

        'Windows'
```

What are Python packages?

- A **python package** is a collection of modules. Modules that are related to each other are mainly put in the same package. When a module from an external package is required in a program, that package can be imported and its modules can be put to use.
- Any Python file, whose name is the module's name property without the **.py** extension, is a **module**.



For example, let’s take the datetime module, which has a sub module called date. When datetime is imported.

```
>>> from datetime import date
>>> print(date.today())
```

OUTPUT:-

```
2020-10-15
```

Packages in Python

- Download a Package.
- Downloading a package is very easy.
- Open the command line interface and tell PIP to download the package you want.
- Navigate your command line to the location of Python's script directory, and type the following:

Example:-

- Download a package named "camelcase":
- C:\Users*Your Name*\AppData\Local\Programs\Python\Python36-32\Scripts>pip install camelcase.

Using a Package

- Once the package is installed, it is ready to use.
- Import the "camelcase" package into your project.
- **humps** 🐫 Convert strings (and dictionary keys) between snake case, camel case and pascal case in Python. Inspired by Humps for Node

Example

Import and use "camelcase":

```
import camelcase
c = camelcase.CamelCase()
txt =  "star star super star"
print(c.hump(txt))
#This method capitalizes the first letter of each word.
```

Output:-

```
C:\Users\My Name>python camel.py
                                    Star Star Super Star
```

Remove a Package

- Use the uninstall command to remove a package:
- Example
- Uninstall the package named "camelcase":
- **C:\Users*Your Name*\AppData\Local\Programs\Python\Python36-32\pip uninstall camelcase**
- The PIP Package Manager will ask you to confirm that you want to remove the camelcase package:
- Uninstalling camelcase-02.1:
  Would remove:
  c:\users*Your Name*\appdata\local\programs\python\python36-32\lib\site-packages\camecase-0.2-py3.6.egg-info
  c:\users*Your Name*\appdata\local\programs\python\python36-32\lib\site-packages\camecase*
  Proceed (y/n)?
- Press y and the package will be removed.

List Packages

- Use the list command to list all the packages installed on your system:

Example:-

- List installed packages:
- C:\Users*Your Name*\AppData\Local\Programs\Python\Python36-32\Scripts>pip list
- C:\Users\91898\Desktop>pip list

OUTPUT:-

Package Version

- ---------- --------
- asgiref 3.2.10
- cycler 0.10.0
- Django 3.0.8
- kiwisolver 1.2.0
- matplotlib 3.3.0
- mysql-connector 2.2.9