UNIT-IV

Files in Python: File Objects, File Built-in Function [open()], File Built-in Methods, File Built-in Attributes, Standard Files, Command-line Arguments, File System, File Execution, Persistent Storage Modules, Related Modules.

Exceptions: Exceptions in Python, Detecting and Handling Exceptions, Context Management, *Exceptions as Strings, Raising Exceptions, Assertions, Standard Exceptions, *Creating Exceptions, Why Exceptions (Now)?, Why Exceptions at All?, Exceptions and the sys Module, Related Modules.

Modules: Modules and Files, Namespaces, Importing Modules, Importing Module Attributes, Module Built-in Functions, Packages, Other Features of Modules.

File Objects

A file object allows us to use, access and manipulate all the user accessible files. One can read and write any such files.

When a file operation fails for an I/O-related reason, the exception IOError is raised. This includes situations where the operation is not defined for some reason, like seek() on a tty device or writing a file opened for reading.

Files have the following methods:

Built-in Methods

open():

Opens a file in given access mode.

Syntax: open(file_address, access_mode)

Examples of accessing a file: A file can be opened with a built-in function called open(). This function takes in the file's address and the access_mode and returns a file object.

There are different types of access_modes:

- **r** : Opens a file for reading only
- **r+**: Opens a file for both reading and writing
- **w** : Opens a file for writing only
- **w+**: Open a file for writing and reading.
- **a** : Opens a file for appending
- **a+**: Opens a file for both appending and reading

When you add 'b' to the access modes you can read the file in binary format rather than the

read([size]):

It reads the entire file and returns it contents in the form of a string. Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes). If the size argument is negative or omitted, read all data until EOF is reached.

In [ ]:
```
# Reading a file
f = open("test.txt", 'r')

#read()
text = f.read(10)

print(text)
f.close()
```

In [ ]:
```
# Reading a file
f = open("D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks/test.txt",

#read()
text = f.read(10)

print(text)
f.close()
```

readline([size]):

It reads the first line of the file i.e till a newline character or an EOF in case of a file having a single line and returns a string. If the size argument is present and non-negative, it is a maximum byte count (including the trailing newline) and an incomplete line may be returned. An empty string is returned only when EOF is encountered immediately.

In [ ]:
```
# Reading a line in a file
f = open("test.txt", 'r')

#readline()
text = f.readline(20)
print(text)
f.close()
```

readlines([sizehint]):

It reads the entire file line by line and updates each line to a list which is returned.Read until EOF using readline() and return a list containing the lines thus read. If the optional sizehint argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read.

In [ ]:
```
# Reading a file
f = open("test.txt", 'r')

#readline()
text = f.readlines(25)
print(text)
f.close()
```

In [ ]:
```
# Reading a file
f = open("test.txt", 'r')

#readline()
text = f.readlines()
print(text)
f.close()
```

write(string):

It writes the contents of string to the file. It has no return value. Due to buffering, the string may not actually show up in the file until the **flush()** or **close()** method is called.

In [ ]:
```
# Writing a file
f = open("test1.txt", 'w')
line = 'Welcome to CMR Technical Campus\n'

#write()
f.write(line)
f.close()
```

In [ ]:
```
# Reading a file
f = open("test1.txt", 'r')

#read()
text = f.read()

print(text)
f.close()
```

In [ ]:
```
# Reading and Writing a file
f = open("test1.txt", 'r+')
lines = f.read()
f.write(lines)
f.close()
```

In [ ]:
```
# Writing and Reading a file
f = open("test1.txt", 'w+')
lines = f.read()
f.write(lines)
f.close()
```

In [ ]:
```
# Appending a file
f = open("test1.txt", 'a')
lines = 'Welcome to CSE Department\n'
f.write(lines)
f.close()
```

writelines(sequence):

It is a sequence of strings to the file usually a list of strings or any other iterable data type. It has no return value.

In [ ]:
```
# Writing a file
f = open("test1.txt", 'a+')
lines = f.readlines()


#writelines()
lines="Department of CMR TC"
f.writelines(lines)
print(lines)
f.close()
```

tell():

It returns an integer that tells us the file object's position from the beginning of the file in the form of bytes

In [ ]:
```
# Telling the file object position
f = open("test1.txt", 'r')
lines = f.read()

#tell()
print(f.tell())
f.close()
```

seek(offset, from_where):

It is used to change the file object's position. Offset indicates the number of bytes to be moved. from_where indicates from where the bytes are to be moved.

from_where defines your point of reference:

- **0**: means your reference point is the beginning of the file
- **1**: means your reference point is the current file position
- **2**: means your reference point is the end of the file

if omitted, from_whare defaults to 0.

Never forget that when managing files, there'll always be a position inside that file where you are currently working on. When just open, that position is the beginning of the file, but as you work with it, you may advance. seek will be useful to you when you need to walk along that open file, just as a path you are traveling into.

In [ ]:
```
# Setting the file object position
f = open("test.txt", 'rb')
lines = f.read(10)
print(lines)

#seek()
print(f.seek(2,1))
lines = f.read(10)
print(lines)
f.close()
```

In [ ]:
```
#ascci file example
f = open("simple.txt", 'r')
f.seek(1)
print(f.readline())
```

flush():

Flush the internal buffer, like stdio‘s fflush(). It has no return value. close() automatically flushes the data but if you want to flush the data before closing the file then you can use this method.

In [ ]:
```
# Clearing the internal buffer before closing the file
f = open("test.txt", 'r')
lines = f.read(10)

#flush()
f.flush()
print(f.read())
f.close()
```

fileno():

Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system.

In [ ]:
```
# Getting the integer file descriptor
f = open("test.txt", 'r')

#fileno()
print(f.fileno())
f.close()
```

isatty():

Returns True if the file is connected to a tty(-like) device and False if not.

The isatty() method returns True if the file stream is interactive, example: connected to a terminal device.

In [2]:
```
# Checks if file is connected to a tty(-like) device
f = open("test.txt", 'r')

#isatty()
print(f.isatty())
f.close()
```

```
False
```

next():

It is used when a file is used as an iterator. The method is called repeatedly. This method returns the next input line or raises StopIteration at EOF when the file is open for reading(behaviour is undefined when opened for writing).

In [3]:
```
marks = [65, 71, 68, 74, 61]

# convert list to iterator
iterator_marks = iter(marks)

# the next element is the first element
marks_1 = next(iterator_marks)
print(marks_1)

# find the next element which is the second element
marks_2 = next(iterator_marks)
print(marks_2)
```

```
65
71
```

In [4]:
```
# Iterates over the file
f = open("test.txt", 'r')

#next()
try:
    while f.next():
        print(f.next())
except:
    f.close()
```

truncate([size]):

Truncate 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. The current file position is not changed. Note that if a specified size exceeds the file's current size, the result is platform-dependent: possibilities include that the file may remain unchanged, increase to the specified size as if zero-filled, or increase to the specified size with undefined new content.

In [5]:
```
# Truncates the file
f = open("test.txt", 'w')

#truncate()
f.truncate(100)
f.close()
```

In [7]:
```
f = open("test.txt", 'r')

l=f.read()
print(l)
f.close()
```

close():

Used to close an open file. A closed file cannot be read or written any more

In [ ]:
```
# Opening and closing a file
# Reading a file
f = open("test1.txt", 'r')

#readline()
text = f.readline(20)
print(text)
f.close()
text = f.readline(10)
print(text)
```

Built-in Attributes

- **closed**: returns a boolean indicating the current state of the file object. It returns true if the file is closed and false when the file is open.
- **encoding**: The encoding that this file uses. When Unicode strings are written to a file, they will be converted to byte strings using this encoding.
- **mode**: The I/O mode for the file. If the file was created using the open() built-in function, this will be the value of the mode parameter.
- **name**: If the file object was created using open(), the name of the file.
- **newlines**: A file object that has been opened in universal newline mode have this attribute which reflects the newline convention used in the file. The value for this attribute are “\r”, “\n”, “\r\n”, None or a tuple containing all the newline types seen.
- **softspace**: It is a boolean that indicates whether a space character needs to be printed before another value when using the print statement.

In [8]:
```
f = open("test1.txt", 'a+')
print(f.closed)
print(f.encoding)
print(f.mode)
print(f.newlines)
```

```
False
cp1252
a+
None
```

In [9]:
```
f = open ("test1.txt", "r")
print(f.name)
print(f.mode)
print(f.closed)
f.close()
print(f.closed)
```

```
test1.txt
r
False
True
```

In [11]:
```
fname = input("Enter file name: ")
num_lines = 0
num_words = 0
num_chars = 0
try:
    fp=open(fname,"r")
    for i in fp:
        # i contains each line of the file
        words = i.split()
        num_lines += 1
        num_words += len(words)
        num_chars += len(i)
    print("Lines = ",num_lines)
    print("Words = ",num_words)
    print("Characters = ",num_chars)
    fp.close()
except Exception:
    print("Enter valid filename")
```

```
Enter file name: test1.txt
Lines =  1
Words =  4
Characters =  20
```

File System Methods

programming in any language, interaction between the programs and the operating system (Windows, Linux, macOS) can become important at some point in any developer's life. This interaction may include moving files from one location to another, creating a new file, deleting a file, etc.

os.getcwd()

os.getcwd() method tells us the location of the current working directory (CWD).

In [16]:
```
# Python program to explain os.getcwd() method

# importing os module
import os

# Get the current working
# directory (CWD)
cwd = os.getcwd()

# Print the current working
# directory (CWD)
print("Current working directory:", cwd)
```

```
Current working directory: C:\Users\NUTHANAKANTI BHASKAR\Python Programming
```

os.chdir()

os.chdir() method in Python used to change the current working directory to a specified path. It takes only a single argument as a new directory path.

In [15]:
```
# Python3 program to change the
# directory of file using os.chdir() method

# import os library
import os

# change the current directory
# to specified directory
os.chdir(r"C:\Users\NUTHANAKANTI BHASKAR\Python Programming")

print("Directory changed")
```

```
Directory changed
```

os.listdir()

os.listdir() method in python is used to get the list of all files and directories in the specified directory. If we don't specify any directory, then list of files and directories in the current working directory will be returned.

In [19]:
```
# Python program to explain os.listdir() method

# importing os module
import os

# Get the path of current working directory
path = 'D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks/14 exp'

# Get the list of all files and directories
# in current working directory
dir_list = os.listdir(path)

print("Files and directories in '", path, "' :")
print(dir_list)
```

```
Files and directories in ' D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise No
tebooks/14 exp ' :
['fibonacci.py', 'using_fibonacci.py', '__pycache__']
```

os.walk()

os.walk() generate the file names in a directory tree by walking the tree either top-down or bottom-up. For each directory in the tree rooted at directory top (including top itself), it yields a 3-tuple (dirpath, dirnames, filenames).

In [22]:
```
# Driver function
import os

for (root,dirs,files) in os.walk('D:/CMR TC/A Y 2022-2023/II Sem/Python Unit w
    print (root)
    print (dirs)
    print (files)
    print ('--------------------------------')
```

```
D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks\14 exp\__pycache__
[]
['fibonacci.cpython-39.pyc']
--------------------------------
D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks\14 exp
['__pycache__']
['fibonacci.py', 'using_fibonacci.py']
--------------------------------
D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks\15 exp\__pycache__
[]
['arth.cpython-39.pyc']
--------------------------------
D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks\15 exp
['__pycache__']
['arth.py', 'week15.py']
--------------------------------
D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks
['14 exp', '15 exp']
['Arithematic operators.PNG', 'Assert Statement.png', 'Assignment Operators.P
NG', 'Bitwise Operators.PNG', 'Comparison Operators.PNG', 'Data Persistence.P
NG', 'dbm.PNG', 'elif statement.png', 'Guido_Van_Rossum.jpg', 'Identity Opera
tors.PNG', 'if statement.png', 'if-else statement.png', 'json encoder method.
PNG', 'JSONEncoder class.PNG', 'Logical Operators.PNG', 'Membership Operator
s.PNG', 'Pickle Module.PNG', 'Python Programming -LAB-MANUAL-II-CSE-II-SEM.pd
f', 'python-applications.png', 'python-data-types.png', 'python-variables.pn
g', 'python-variables2.png', 'python-variables3.png', 'shelve methods.PNG',
'Shelve Module.PNG', 'string.png', 'Unit-I.ipynb', 'Unit-II.ipynb', 'Unit-II.
pdf', 'Unit-III.ipynb', 'Unit-III.pdf', 'Unit-IV.ipynb', 'while statement.pn
g']
--------------------------------
```

os.path.join()

os.path.join() method in Python join one or more path components intelligently. This method concatenates various path components with exactly one directory separator (‘/’) following each non-empty part except the last path component. If the last path component to be joined is empty then a directory separator (‘/’) is put at the end.

In [30]:
```
# Python program to explain os.path.join() method

# importing os module
import os

# Path
path = "D:/CMR TC/"

# Join various path components
print(os.path.join(path, "A Y 2022-2023/II Sem/Python Unit wise Notebooks/", "
```

```
D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks/test.txt
```

In [ ]:
```
# Path
path = "D:/CMR TC/"

# Join various path components
print(os.path.join(path, "A Y 2022-2023/II Sem/", "Python Unit wise Notebooks/
```

os.makedirs()

os.makedirs() method in Python is used to create a directory recursively. That means while making leaf directory if any intermediate-level directory is missing, os.makedirs() method will create them all.

In [31]:
```
# Python program to explain os.makedirs() method

# importing os module
import os

# Leaf directory
directory = "files"

# Parent Directories
parent_dir = "D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks"

# Path
path = os.path.join(parent_dir, directory)

# Create the directory
# 'ihritik'
os.makedirs(path)
print("Directory '%s' created" %directory)
```

```
Directory 'files' created
```

shutil.copy2()

shutil.copy2() method in Python is used to copy the content of the source file to the destination file or directory. This method is identical to shutil.copy() method but it also tries to preserve the file’s metadata.

In [33]:
```
# Python program to explain shutil.copy2() method

# importing os module
import os

# importing shutil module
import shutil

# path
path = 'D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks'

# List files and directories
print("Before copying file:")
print(os.listdir(path))


# Source path
source = "D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks/Unit-II.pd

# Print the metadeta of source file
metadata = os.stat(source)
print("Metadata:", metadata, "\n")

# Destination path
destination = "D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks/files

# Copy the content of source to destination
dest = shutil.copy2(source, destination)

# List files and directories

print("After copying file:")
print(os.listdir(path))

# Print the metadata of the destination file
matadata = os.stat(destination)
print("Metadata:", metadata)

# Print path of newly created file
print("Destination path:", dest)
```

```
Before copying file:
['14 exp', '15 exp', 'Arithematic operators.PNG', 'Assert Statement.png', 'as
sertion.png', 'Assignment Operators.PNG', 'Bitwise Operators.PNG', 'Compariso
n Operators.PNG', 'Data Persistence.PNG', 'dbm.PNG', 'elif statement.png', 'f
iles', 'Guido_Van_Rossum.jpg', 'Identity Operators.PNG', 'if statement.png',
'if-else statement.png', 'json encoder method.PNG', 'JSONEncoder class.PNG',
'Logical Operators.PNG', 'Membership Operators.PNG', 'package.png', 'Pickle M
odule.PNG', 'Python Programming -LAB-MANUAL-II-CSE-II-SEM.pdf', 'python-appli
cations.png', 'python-data-types.png', 'python-variables.png', 'python-variab
les2.png', 'python-variables3.png', 'shelve methods.PNG', 'Shelve Module.PN
G', 'string.png', 'Unit-I.ipynb', 'Unit-II.ipynb', 'Unit-II.pdf', 'Unit-III.i
pynb', 'Unit-III.pdf', 'Unit-IV.ipynb', 'while statement.png']
Metadata: os.stat_result(st_mode=33206, st_ino=3659174697677430, st_dev=17860
70277, st_nlink=1, st_uid=0, st_gid=0, st_size=3333301, st_atime=1687580300,
st_mtime=1682755412, st_ctime=1682755412)

After copying file:
['14 exp', '15 exp', 'Arithematic operators.PNG', 'Assert Statement.png', 'as
sertion.png', 'Assignment Operators.PNG', 'Bitwise Operators.PNG', 'Compariso
n Operators.PNG', 'Data Persistence.PNG', 'dbm.PNG', 'elif statement.png', 'f
iles', 'Guido_Van_Rossum.jpg', 'Identity Operators.PNG', 'if statement.png',
'if-else statement.png', 'json encoder method.PNG', 'JSONEncoder class.PNG',
'Logical Operators.PNG', 'Membership Operators.PNG', 'package.png', 'Pickle M
odule.PNG', 'Python Programming -LAB-MANUAL-II-CSE-II-SEM.pdf', 'python-appli
cations.png', 'python-data-types.png', 'python-variables.png', 'python-variab
les2.png', 'python-variables3.png', 'shelve methods.PNG', 'Shelve Module.PN
G', 'string.png', 'Unit-I.ipynb', 'Unit-II.ipynb', 'Unit-II.pdf', 'Unit-III.i
pynb', 'Unit-III.pdf', 'Unit-IV.ipynb', 'while statement.png']
Metadata: os.stat_result(st_mode=33206, st_ino=3659174697677430, st_dev=17860
70277, st_nlink=1, st_uid=0, st_gid=0, st_size=3333301, st_atime=1687580300,
st_mtime=1682755412, st_ctime=1682755412)
Destination path: D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks/f
iles/II.pdf
```

shutil.move()

shutil.move() method Recursively moves a file or directory (source) to another location (destination) and returns the destination. If the destination directory already exists then src is moved inside that directory. If the destination already exists but is not a directory then it may be overwritten depending on os.rename() semantics.

In [34]:
```
# Python program to explain shutil.move() method

# importing os module
import os

# importing shutil module
import shutil

# path
path = 'D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks/'

# List files and directories
print("Before moving file:")
print(os.listdir(path))


# Source path
source = 'D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks/files'

# Destination path
destination = 'D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks/files

# Move the content of source to destination
dest = shutil.move(source, destination)

# List files and directories
print("After moving file:")
print(os.listdir(path))

# Print path of newly created file
print("Destination path:", dest)
```

```
Before moving file:
['14 exp', '15 exp', 'Arithematic operators.PNG', 'Assert Statement.png', 'as
sertion.png', 'Assignment Operators.PNG', 'Bitwise Operators.PNG', 'Compariso
n Operators.PNG', 'Data Persistence.PNG', 'dbm.PNG', 'elif statement.png', 'f
iles', 'Guido_Van_Rossum.jpg', 'Identity Operators.PNG', 'if statement.png',
'if-else statement.png', 'json encoder method.PNG', 'JSONEncoder class.PNG',
'Logical Operators.PNG', 'Membership Operators.PNG', 'package.png', 'Pickle M
odule.PNG', 'Python Programming -LAB-MANUAL-II-CSE-II-SEM.pdf', 'python-appli
cations.png', 'python-data-types.png', 'python-variables.png', 'python-variab
les2.png', 'python-variables3.png', 'shelve methods.PNG', 'Shelve Module.PN
G', 'string.png', 'Unit-I.ipynb', 'Unit-II.ipynb', 'Unit-II.pdf', 'Unit-III.i
pynb', 'Unit-III.pdf', 'Unit-IV.ipynb', 'while statement.png']
After moving file:
['14 exp', '15 exp', 'Arithematic operators.PNG', 'Assert Statement.png', 'as
sertion.png', 'Assignment Operators.PNG', 'Bitwise Operators.PNG', 'Compariso
n Operators.PNG', 'Data Persistence.PNG', 'dbm.PNG', 'elif statement.png', 'f
iles2', 'Guido_Van_Rossum.jpg', 'Identity Operators.PNG', 'if statement.png',
'if-else statement.png', 'json encoder method.PNG', 'JSONEncoder class.PNG',
'Logical Operators.PNG', 'Membership Operators.PNG', 'package.png', 'Pickle M
odule.PNG', 'Python Programming -LAB-MANUAL-II-CSE-II-SEM.pdf', 'python-appli
cations.png', 'python-data-types.png', 'python-variables.png', 'python-variab
les2.png', 'python-variables3.png', 'shelve methods.PNG', 'Shelve Module.PN
G', 'string.png', 'Unit-I.ipynb', 'Unit-II.ipynb', 'Unit-II.pdf', 'Unit-III.i
pynb', 'Unit-III.pdf', 'Unit-IV.ipynb', 'while statement.png']
Destination path: D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks/f
iles2
```

os.remove()

os.remove() method in Python is used to remove or delete a file path. This method can not remove or delete a directory.

In [36]:
```
# Python program to explain os.remove() method

# importing os module
import os

# File name
file = 'II.pdf'

# File location
location = "D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks/files2"

# Path
path = os.path.join(location, file)

# Remove the file
os.remove(path)
print("%s has been removed successfully" %file)
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In[36], line 16
     13 path = os.path.join(location, file)
     15 # Remove the file
---> 16 os.remove(path)
     17 print("%s has been removed successfully" %file)

FileNotFoundError: [WinError 2] The system cannot find the file specified:
'D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks/files2\\II.pdf'
```

shutil.rmtree()

shutil.rmtree() is used to delete an entire directory tree, path must point to a directory.

In [37]:
```
# Python program to demonstrate shutil.rmtree()

import shutil
import os

# location
location = "D:/CMR TC/A Y 2022-2023/II Sem/Python Unit wise Notebooks/"

# directory
dir = "files2"

# path
path = os.path.join(location, dir)

# removing directory
shutil.rmtree(path)
```

Persistent Storage Modules

During the course of using any software application, user provides some data to be processed. The data may be input, using a standard input device (keyboard) or other devices such as disk file, scanner, camera, network cable, WiFi connection, etc.

Data so received, is stored in computer's main memory (RAM) in the form of various data structures such as, variables and objects until the application is running. Thereafter, memory contents from RAM are erased.

However, more often than not, it is desired that the values of variables and/or objects be stored in such a manner, that it can be retrieved whenever required, instead of again inputting the same data.

The word 'persistence' means "the continuance of an effect after its cause is removed". The term data persistence means it continues to exist even after the application has ended. Thus, data stored in a non-volatile storage medium such as, a disk file is a persistent data storage.

For this various built-in and third party Python modules to store and retrieve data to/from various formats such as text file, CSV, JSON and XML files as well as relational and non-relational databases.

Using Python's built-in File object, it is possible to write string data to a disk file and read from it. Python's standard library, provides modules to store and retrieve serialized data in various data structures such as JSON and XML.

Python's DB-API provides a standard way of interacting with relational databases. Other third party Python packages, present interfacing functionality with NOSQL databases such as MongoDB and Cassandra.

ZODB database which is a persistence API for Python objects. Microsoft Excel format is a very popular data file format.

Python Data Persistence - Object Serialization

Python's built-in file object returned by Python's built-in open() function has one important shortcoming. When opened with 'w' mode, the write() method accepts only the string object.

That means, if you have data represented in any non-string form, the object of either in built-in classes (numbers, dictionary, lists or tuples) or other user-defined classes, it cannot be written to file directly. Before writing, you need to convert it in its string representation.

In [38]:
```
numbers=[10,20,30,40]
file=open('numbers.txt','w')
file.write(str(numbers))
file.close()
```

- For a binary file, argument to write() method must be a byte object. For example, the list of integers is converted to bytes by bytearray() function and then written to file.

In [39]:
```
numbers=[10,20,30,40]
file=open('numbers.txt','w')
data=bytearray(numbers)
file.write(str(data))
file.close()
```

- To read back data from the file in the respective data type, reverse conversion needs to be done.

In [40]:
```
file=open('numbers.txt','rb')
data=file.read()
print (list(data))
```

```
[98, 121, 116, 101, 97, 114, 114, 97, 121, 40, 98, 39, 92, 110, 92, 120, 49,
52, 92, 120, 49, 101, 40, 39, 41]
```

- This type of manual conversion, of an object to string or byte format (and vice versa) is very cumbersome and tedious. It is possible to store the state of a Python object in the form of byte stream directly to a file, or memory stream and retrieve to its original state. This process is called serialization and de-serialization.

Python's built in library contains various modules for serialization and deserialization process.

Sr.No.	Name & Description
1	

Python Data Persistence - Pickle Module

Python's terminology for serialization and deserialization is pickling and unpickling respectively. The pickle module in Python library, uses very Python specific data format. Hence, non-Python applications may not be able to deserialize pickled data properly. It is also advised not to unpickle data from un-authenticated source.

The serialized (pickled) data can be stored in a byte string or a binary file. This module defines **dumps()** and **loads()** functions to pickle and unpickle data using byte string. For file based process, the module has dump() and load() function.

Python's pickle protocols are the conventions used in constructing and deconstructing Python objects to/from binary data. Currently, pickle module defines 5 different protocols as listed below −

Sr.No.	Names & Description
1	**Protocol version 0** Original "human-readable" protocol backwards compatible with earlier versions.
2	**Protocol version 1** Old binary format also compatible with earlier versions of Python.
3	**Protocol version 2** Introduced in Python 2.3 provides efficient pickling of new-style classes.
4	**Protocol version 3** Added in Python 3.0. recommended when compatibility with other Python 3 versions is required.
5	**Protocol version 4** was added in Python 3.4. It adds support for very large objects

In [41]:
```
import pickle
dct={"name":"Ravi", "age":23, "Gender":"M","marks":75}
dctstring=pickle.dumps(dct)
print(dctstring)
```

```
b'\x80\x04\x952\x00\x00\x00\x00\x00\x00\x00}\x94(\x8c\x04name\x94\x8c\x04Ravi
\x94\x8c\x03age\x94K\x17\x8c\x06Gender\x94\x8c\x01M\x94\x8c\x05marks\x94KKu.'
```

- **Use loads() function, to unpickle the string and obtain original dictionary object**.

In [42]:
```
import pickle
dct=pickle.loads(dctstring)
print (dct)
```

```
{'name': 'Ravi', 'age': 23, 'Gender': 'M', 'marks': 75}
```

- **Pickled objects can also be persistently stored in a disk file, using dump() function and retrieved using load() function**.

In [ ]:
```
import pickle
f=open("data.txt","wb")
dct={"name":"Ravi", "age":23, "Gender":"M","marks":75}
pickle.dump(dct,f)
f.close()

#to read
import pickle
f=open("data.txt","rb")
d=pickle.load(f)
print (d)
f.close()
```

Python Data Persistence - Marshal Module

Object serialization features of marshal module in Python's standard library are similar to pickle module. However, this module is not used for general purpose data. On the other hand, it is used by Python itself for Python's internal object serialization to support read/write operations on compiled versions of Python modules (.pyc files).

The data format used by marshal module is not compatible across Python versions. Therefore, a compiled Python script (.pyc file) of one version most probably won't execute on another.

Just as pickle module, marshal module also defined load() and dump() functions for reading and writing marshalled objects from / to file.

dump() This function writes byte representation of supported Python object to a file. The file itself be a binary file with write permission

load() This function reads the byte data from a binary file and converts it to Python object.

The code uses built-in **compile()** function to build a code object out of a source string which embeds Python instructions.

compile(source, file, mode)

The file parameter should be the file from which the code was read. If it wasn't read from a file pass any arbitrary string.

The mode parameter is 'exec' if the source contains sequence of statements, 'eval' if there is a single expression or 'single' if it contains a single interactive statement.

The compile code object is then stored in a .pyc file using dump() function.

In [7]:
```
import marshal
script = """
a=10
b=20
print ('addition=',a+b)
"""
code = compile(script, "script", "exec")
f=open("a.pyc","wb")
marshal.dump(code, f)
f.close()
```

- To deserialize, the object from .pyc file use load() function. Since, it returns a code object, it can be run using exec(), another built-in function.

In [8]:
```
import marshal
f=open("a.pyc","rb")
data=marshal.load(f)
exec (data)
```

```
addition= 30
```

Python Data Persistence - Shelve Module

The shelve module in Python's standard library provides simple yet effective object persistence mechanism. The shelf object defined in this module is dictionary-like object which is persistently stored in a disk file. This creates a file similar to dbm database on UNIX like systems.

The shelf dictionary has certain restrictions. Only string data type can be used as key in this special dictionary object, whereas any picklable Python object can be used as value.

The shelve module defines three classes as follows –
 Shelve%20Module.PNG

The **open()** function defined in shelve module which return a DbfilenameShelf object.

```
open(filename, flag='c', protocol=None, writeback=False)
```

The filename parameter is assigned to the database created. Default value for flag parameter is ‘c’ for read/write access. Other flags are ‘w’ (write only) ‘r’ (read only) and ‘n’ (new with read/write).

The serialization itself is governed by pickle protocol, default is none. Last parameter writeback parameter by default is false. If set to true, the accessed entries are cached. Every access calls sync() and close() operations, hence process may be slow.

Following code creates a database and stores dictionary entries in it.

In [3]:
```
import shelve
s=shelve.open("test")
s['name']="Ajay"
s['age']=23
s['marks']=75
s.close()
```

This will create test.dir file in current directory and store key-value data in hashed form. The Shelf object has following methods available −

Sr.No.	Methods & Description
1	**close()** synchronise and close persistent dict object.
2	**sync()** Write back all entries in the cache if shelf was opened with writeback set to True.
3	**get()** returns value associated with key
4	**items()** list of tuples – each tuple is key value pair
5	**keys()** list of shelf keys
6	**pop()** remove specified key and return the corresponding value.
7	**update()** Update shelf from another dict/iterable
8	**values()** list of shelf values

In [4]:
```
#To access value of a particular key in shelf –
s=shelve.open('test')
print (s['age']) #this will print 23
s['age']=25
print (s.get('age')) #this will print 25
s.pop('marks') #this will remove corresponding k-v pair
```

```
23
25
```

Out[4]:
```
75
```

In [5]:
```
#As in a built-in dictionary object, the items(), keys() and values() methods
#return view objects.
print (list(s.items()))
[('name', 'Ajay'), ('age', 25), ('marks', 75)]

print (list(s.keys()))
['name', 'age', 'marks']

print (list(s.values()))
['Ajay', 25, 75]
```

```
[('name', 'Ajay'), ('age', 25)]
['name', 'age']
['Ajay', 25]
```

Out[5]:
```
['Ajay', 25, 75]
```

In [6]:
```
#To merge items of another dictionary with shelf use update() method.
d={'salary':10000, 'designation':'manager'}
s.update(d)
print (list(s.items()))

[('name', 'Ajay'), ('age', 25), ('salary', 10000), ('designation', 'manager')]
```

```
[('name', 'Ajay'), ('age', 25), ('salary', 10000), ('designation', 'manage
r')]
```

Out[6]:
```
[('name', 'Ajay'), ('age', 25), ('salary', 10000), ('designation', 'manage
r')]
```

Python Data Persistence - dbm Package

The dbm package presents a dictionary like interface DBM style databases. **DBM stands for DataBase Manager**. This is used by UNIX (and UNIX like) operating system. The dbbm library is a simple database engine written by Ken Thompson. These databases use binary encoded string objects as key, as well as value.

The database stores data by use of a single key (a primary key) in fixed-size buckets and uses hashing techniques to enable fast retrieval of the data by key.

The dbm package contains following modules −

- **dbm.gnu** module is an interface to the DBM library version as implemented by the GNU project.
- **dbm.ndbm** module provides an interface to UNIX nbdm implementation.
- **dbm.dumb** is used as a fallback option in the event, other dbm implementations are not found. This requires no external dependencies but is slower than others.

In [9]:
```
import dbm
db=dbm.open('mydbm.db','n')
db['name']='Raj Deshmane'
db['address']='Kirtinagar Pune'
db['PIN']='431101'
db.close()
```

The open() function allows mode these flags −

Sr.No.	Value & Meaning
1	**'r'** Open existing database for reading only (default)
2	**'w'** Open existing database for reading and writing
3	**'c'** Open database for reading and writing, creating it if it doesn't exist
4	**'n'** Always create a new, empty database, open for reading and writing

The dbm object is a dictionary like object, just as shelf object. Hence, all dictionary operations can be performed. The dbm object can invoke get(), pop(), append() and update() methods.

Following code opens 'mydbm.db' with 'r' flag and iterates over collection of key-value pairs.

In [10]:
```
db=dbm.open('mydbm.db','r')
for k,v in db.items():
    print (k,v)
```

```
b'name' b'Raj Deshmane'
b'address' b'Kirtinagar Pune'
b'PIN' b'431101'
```

Python Data Persistence - CSV Module

CSV stands for comma separated values. This file format is a commonly used data format while exporting/importing data to/from spreadsheets and data tables in databases. The csv module was incorporated in Python's standard library as a result of PEP 305. It presents classes and methods to perform read/write operations on CSV file as per recommendations of PEP 305.

CSV is a preferred export data format by Microsoft's Excel spreadsheet software. However, csv module can handle data represented by other dialects also.

The CSV API interface consists of following writer and reader classes −

writer()

This function in csv module returns a writer object that converts data into a delimited string and stores in a file object. The function needs a file object with write permission as a parameter. Every row written in the file issues a newline character. To prevent additional space between lines, newline parameter is set to ''.

The writer class has following methods −

writerow() This method writes items in an iterable (list, tuple or string), separating them by comma character.

writerows() This method takes a list of iterables, as parameter and writes each item as a comma separated line of items in the file.

In [11]:
```
import csv
persons=[('Lata',22,45),('Anil',21,56),('John',20,60)]
csvfile=open('persons.csv','w', newline='')
obj=csv.writer(csvfile)
for person in persons:
    obj.writerow(person)
csvfile.close()
```

- Instead of iterating over the list to write each row individually, we can use writerows() method.

In [12]:
```
import csv
csvfile=open('persons1.csv','w', newline='')
persons=[('ram',22,45),('raj',21,56),('krish',20,60)]
obj=csv.writer(csvfile)
obj.writerows(persons)
csvfile.close()
```

reader()

This function returns a reader object which returns an iterator of lines in the csv file. Using the regular for loop, all lines in the file are displayed in following example −

In [13]:
```
csvfile=open('persons.csv','r', newline='')
obj=csv.reader(csvfile)
for row in obj:
    print (row)
```

```
['Lata', '22', '45']
['Anil', '21', '56']
['John', '20', '60']
```

- The reader object is an iterator. Hence, it supports next() function which can also be used to display all lines in csv file instead of a **for loop**.

In [14]:
```
csvfile=open('persons.csv','r', newline='')
obj=csv.reader(csvfile)
while True:
   try:
      row=next(obj)
      print (row)
   except StopIteration:
        break
```

```
['Lata', '22', '45']
['Anil', '21', '56']
['John', '20', '60']
```

As mentioned earlier, csv module uses Excel as its default dialect. The csv module also defines a dialect class. Dialect is set of standards used to implement CSV protocol. The list of dialects available can be obtained by list_dialects() function.

In [ ]:
```
csv.list_dialects()
```

In addition to iterables, csv module can export a dictionary object to CSV file and read it to populate Python dictionary object. For this purpose, this module defines following classes −

DictWriter() This function returns a DictWriter object. It is similar to writer object, but the rows are mapped to dictionary object. The function needs a file object with write permission and a list of keys used in dictionary as fieldnames parameter. This is used to write first line in the file as header.

writeheader() This method writes list of keys in dictionary as a comma separated line as first line in the file.

In following example, a list of dictionary items is defined. Each item in the list is a dictionary.

In [ ]:
```
persons=[
   {'name':'Lata', 'age':22, 'marks':45},
   {'name':'Anil', 'age':21, 'marks':56},
   {'name':'John', 'age':20, 'marks':60}
]
csvfile=open('persons.csv','w', newline='')
fields=list(persons[0].keys())
obj=csv.DictWriter(csvfile, fieldnames=fields)
obj.writeheader()
obj.writerows(persons)
csvfile.close()
```

DictReader()

This function returns a DictReader object from the underlying CSV file. As, in case of, reader object, this one is also an iterator, using which contents of the file are retrieved.

In [ ]:
```
csvfile=open('persons.csv','r', newline='')
obj=csv.DictReader(csvfile)
#The class provides fieldnames attribute, returning the dictionary keys used a
print (obj.fieldnames)
```

In [ ]:
```
#Use loop over the DictReader object to fetch individual dictionary objects.
for row in obj:
    print (row)
```

- To convert OrderedDict object to normal dictionary, we have to first import OrderedDict from collections module.

In [ ]:
```
from collections import OrderedDict
r=OrderedDict([('name', 'Lata'), ('age', '22'), ('marks', '45')])
dict(r)
```

Python Data Persistence - JSON Module

JSON stands for JavaScript Object Notation. It is a lightweight data interchange format. It is a language-independent and cross platform text format, supported by many programming languages. This format is used for data exchange between the web server and clients.

JSON format is similar to pickle. However, pickle serialization is Python specific whereas JSON format is implemented by many languages hence has become universal standard. Functionality and interface of json module in Python's standard library is similar to pickle and marshal modules.

Just as in pickle module, the json module also provides dumps() and loads() function for serialization of Python object into JSON encoded string, and dump() and load() functions write and read serialized Python objects to/from file.

- **dumps() −** This function converts the object into JSON format.
- **loads() −** This function converts a JSON string back to Python object.

In [15]:
```
import json
data=['Rakesh',{'marks':(50,60,70)}]
s=json.dumps(data)
json.loads(s)
```

Out[15]:
```
['Rakesh', {'marks': [50, 60, 70]}]
```

The dumps() function can take optional sort_keys argument. By default, it is False. If set to True, the dictionary keys appear in sorted order in the JSON string.

The dumps() function has another optional parameter called indent which takes a number as value. It decides length of each segment of formatted representation of json string, similar to print output.

The json module also has object oriented API corresponding to above functions. There are two classes defined in the module – JSONEncoder and JSONDecoder.

JSONEncoder class

Object of this class is encoder for Python data structures. Each Python data type is converted in corresponding JSON type as shown in following table −

Python	JSON
Dict	object
list, tuple	array
Str	string
int, float, int- & float-derived Enums	number
True	true
False	false
None	null

The JSONEncoder class is instantiated by JSONEncoder() constructor. Following important methods are defined in encoder class −

Sr.No.	Methods & Description
1	**encode()** serializes Python object into JSON format
2	**iterencode()** Encodes the object and returns an iterator yielding encoded form of each item in the object.
3	**indent** Determines indent level of encoded string
4	**sort_keys**

In [ ]:
```
e=json.JSONEncoder()
e.encode(data)
```

JSONDecoder class

Object of this class helps in decoded in json string back to Python data structure. Main method in this class is decode().

Following example code retrieves Python list object from encoded string in earlier step.

In [ ]:
```
d=json.JSONDecoder()
d.decode(s)
```

The json module defines load() and dump() functions to write JSON data to a file like object – which may be a disk file or a byte stream and read data back from them.

dump() This function writes JSONed Python object data to a file. The file must be opened with 'w' mode.

In [ ]:
```
import json
data=['Rakesh', {'marks': (50, 60, 70)}]
fp=open('json.txt','w')
json.dump(data,fp)
fp.close()
```

load() This function loads JSON data from the file and returns Python object from it. The file must be opened with read permission (should have 'r' mode).

In [ ]:
```
fp=open('json.txt','r')
ret=json.load(fp)
print (ret)
fp.close()
```

Exceptions

Errors that occur at runtime (after passing the syntax test) are called **exceptions** or **logical errors**.

For instance, they occur when we

- try to open a file(for reading) that does not exist (FileNotFoundError)
- try to divide a number by zero (ZeroDivisionError)
- try to import a module that does not exist (ImportError) and so on.

Whenever these types of runtime errors occur, Python creates an exception object.

If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

When a Python program meets an error, it stops the execution of the rest of the program. An error in Python might be either an error in the syntax of an expression or a Python exception.

An exception in Python is an incident that happens while executing a program that causes the regular course of the program's commands to be disrupted. When a Python code comes across a condition it can't handle, it raises an exception. An object in Python that describes an error is called an exception.

When a Python code throws an exception, it has two options: handle the exception immediately or stop and quit.

Note: Exception is the base class for all the exceptions in Python.

Exception Handling

Different types of exceptions in python:

In Python, there are several built-in exceptions that can be raised when an error occurs during the execution of a program. Here are some of the most common types of exceptions in Python:

- **SyntaxError**: This exception is raised when the interpreter encounters a syntax error in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.
- **TypeError**: This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.
- **NameError**: This exception is raised when a variable or function name is not found in the current scope.

- **IndexError**: This exception is raised when an index is out of range for a list, tuple, or other sequence types.
- **KeyError**: This exception is raised when a key is not found in a dictionary.
- **ValueError**: This exception is raised when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer when the string does not represent a valid integer.
- **AttributeError**: This exception is raised when an attribute or method is not found on an object, such as trying to access a non-existent attribute of a class instance.
- **IOError**: This exception is raised when an I/O operation, such as reading or writing a file, fails due to an input/output error.
- **ZeroDivisionError**: This exception is raised when an attempt is made to divide a number by zero.
- **ImportError**: This exception is raised when an import statement fails to find or load a module.

These are just a few examples of the many types of exceptions that can occur in Python. It's important to handle exceptions properly in your code using try-except blocks or other error-handling techniques, in order to gracefully handle errors and prevent the program from

Difference between Syntax Error and Exceptions

Syntax Error: As the name suggests this error is caused by the wrong syntax in the code. It leads to the termination of the program.

In [ ]:
```
# initialize the amount variable
amount = 10000

# check that You are eligible to purchase Dsa Self Paced or not
if(amount > 2999)
print("You are eligible to purchase Dsa Self Paced")
```

Exceptions: Exceptions are raised when the program is syntactically correct, but the code results in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

In [ ]:
```
# initialize the amount variable
marks = 10000

# perform division with 0
a = marks / 0
print(a)
```

TypeError:

This exception is raised when an operation or function is applied to an object of the wrong type.

x = 5 y = "hello" z = x + y # Raises a TypeError: unsupported operand type(s) for +: 'int' and 'str'

try catch block to resolve it:

In [ ]:
```
x = 5
y = "hello"
try:
    z = x + y
except TypeError:
    print("Error: cannot add an int and a str")
```

Try and Except Statement – Catching Exceptions

Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

In [ ]:
```
# Python program to handle simple runtime error

a = [1, 2, 3]
try:
    print ("Second element = %d" %(a[1]))

# Throws error since there are only 3 elements in array
    print ("Fourth element = %d" %(a[3]))

except:
    print ("An error occurred")
```

In the above example, the statements that can cause the error are placed inside the try statement (second print statement in our case). The second print statement tries to access the fourth element of the list which is not there and this throws an exception. This exception is then caught by the except statement.

Catching Specific Exception

A try statement can have more than one except clause, to specify handlers for different exceptions. Please note that at most one handler will be executed. For example, we can add IndexError in the above code.

The general syntax for adding specific exceptions are –

try:

```
    # statement(s)
```

except IndexError:

```
    # statement(s)
```

except ValueError:

```
    # statement(s)
```

In [ ]:
```
# Program to handle multiple errors with one except statement

def fun(a):
    if a < 4:

# throws ZeroDivisionError for a = 3
        b = a/(a-3)

# throws NameError if a >= 4
    print("Value of b = ", b)

try:
    fun(3)
    fun(5)

# note that braces () are necessary here for multiple exceptions
except ZeroDivisionError:
    print("ZeroDivisionError Occurred and Handled")
except NameError:
    print("NameError Occurred and Handled")
```

Try with Else Clause

In Python, you can also use the else clause on the try-except block which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.

In [ ]:
```
# Program to depict else clause with try-except
# Function which returns a/b
def AbyB(a , b):
    try:
        c = ((a+b) / (a-b))
    except ZeroDivisionError:
        print ("a/b result in 0")
    else:
        print (c)

# Driver program to test above function
AbyB(2.0, 3.0)
AbyB(3.0, 3.0)
```

Finally Keyword in Python

Python provides a keyword finally, which is always executed after the try and except blocks. The final block always executes after the normal termination of the try block or after the try block terminates due to some exception.

Syntax:

try:

```
    # Some Code....
```

except:

```
    # optional block
    # Handling of exception (if required)
```

else:

```
    # execute if no exception
```

finally:

```
    # Some code .....(always executed)
```

In [ ]:
```
# Python program to demonstrate finally No exception Exception raised in try b
try:
    k = 5//0 # raises divide by zero exception.
    print(k)

# handles zerodivision exception
except ZeroDivisionError:
    print("Can't divide by zero")

finally:
# this block is always executed regardless of exception generation.
    print('This is always executed')
```

Raising Exception

The raise statement allows the programmer to force a specific exception to occur. The sole argument in raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).

In [ ]:
```
# Program to depict Raising Exception

try:
    raise NameError("Hi there") # Raise Error
except NameError:
    print ("An exception")
    raise # To determine whether the exception was raised or not
```

Advantages of Exception Handling:

- **Improved program reliability**: By handling exceptions properly, you can prevent your program from crashing or producing incorrect results due to unexpected errors or input.
- **Simplified error handling**: Exception handling allows you to separate error handling code from the main program logic, making it easier to read and maintain your code.

- **Cleaner code**: With exception handling, you can avoid using complex conditional statements to check for errors, leading to cleaner and more readable code.
- **Easier debugging**: When an exception is raised, the Python interpreter prints a traceback that shows the exact location where the exception occurred, making it easier to debug your code.

Disadvantages of Exception Handling:

- **Performance overhead**: Exception handling can be slower than using conditional statements to check for errors, as the interpreter has to perform additional work to catch and handle the exception.
- **Increased code complexity**: Exception handling can make your code more complex, especially if you have to handle multiple types of exceptions or implement complex error handling logic.
- **Possible security risks**: Improperly handled exceptions can potentially reveal sensitive information or create security vulnerabilities in your code, so it's important to handle exceptions carefully and avoid exposing too much information about your program.

Context Manager

In any programming language, the usage of resources like file operations or database connections is very common. But these resources are limited in supply. Therefore, the main problem lies in making sure to release these resources after usage. If they are not released then it will lead to resource leakage and may cause the system to either slow down or crash. It would be very helpful if users have a mechanism for the automatic setup and teardown of resources. In Python, it can be achieved by the usage of context managers which facilitate the proper handling of resources.

The most common way of performing file operations is by using the keyword as shown below:

In [ ]:
```
# Python program showing a use of with keyword

with open("test.txt") as f:
    data = f.read()
```

- Let's take the example of file management. When a file is opened, a file descriptor is consumed which is a limited resource. Only a certain number of files can be opened by a process at a time. The following program demonstrates it.

In [ ]:
```
file_descriptors = []
for x in range(100000):
    file_descriptors.append(open('test.txt', 'w'))
```

- **An error message saying that too many files are open. The above example is a case of file descriptor leakage. It happens because there are too many open files and they**

are not closed. There might be chances where a programmer may forget to close an opened file.

Managing Resources using context manager: Suppose a block of code raises an exception or if it has a complex algorithm with multiple return paths, it becomes cumbersome to close a file in all the places. Generally in other languages when working with files try-except-finally is used to ensure that the file resource is closed after usage even if there is an exception.

Python provides an easy way to manage resources: **Context Managers.** The with keyword is used. When it gets evaluated it should result in an object that performs context management. Context managers can be written using classes or functions(with decorators).

Creating a Context Manager: When creating context managers using classes, user need to ensure that the class has the methods: **enter**() and **exit**().

The **enter**() returns the resource that needs to be managed and the **exit**() does not return anything but performs the cleanup operations.

First, let us create a simple class called ContextManager to understand the basic structure of creating context managers using classes, as shown below:

In [ ]:
```
# Python program creating a context manager

class ContextManager():
    def __init__(self):
        print('init method called')

    def __enter__(self):
        print('enter method called')
        return self

    def __exit__(self, exc_type, exc_value, exc_traceback):
        print('exit method called')

with ContextManager() as manager:
    print('with statement block')
```

In this case, a ContextManager object is created. This is assigned to the variable after the keyword i.e manager. On running the above program, the following get executed in sequence:

- **init**()
- **enter**()
- statement body (code inside the with block)
- **exit**()[the parameters in this method are used to manage exceptions]

File management using context manager:

Let's apply the above concept to create a class that helps in file resource management. The FileManager class helps in opening a file, writing/reading contents, and then closing it.

In [ ]:
```
# Python program showing file management using context manager

class FileManager():
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        self.file = None

    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_value, exc_traceback):
        self.file.close()

# loading a file
with FileManager('simple.txt', 'w') as f:
    f.write('Test')

print(f.closed)
```

*Exceptions as Strings

Prior to Python 1.5, standard exceptions were implemented as strings. However, this became limiting in that it did not allow for exceptions to have relationships to each other. With the advent of exception classes, this is no longer the case. As of 1.5, all standard exceptions are now classes. It is still possible for programmers to generate their own exceptions as strings, but we recommend using exception classes from now on.

For backward compatibility, it is possible to revert to string-based exceptions. Starting the Python interpreter with the command-line option -Xwill provide you with the standard exceptions as strings. This feature will be obsolete beginning with Python 1.6.

Python 2.5 begins the process of deprecating string exceptions from Python forever. In 2.5, raise of string exceptions generates a warning.

In 2.6, the catching of string exceptions results in a warning. Since they are rarely used and are being deprecated, we will no longer consider string exceptions within the scope of this book and have removed it. (You may find the original text in prior editions of this book.) The only point of relevance and the final thought is a caution: You may use an external or third-party module, which may still have string exceptions. String exceptions are a bad idea anyway. One reader vividly recalls seeing Linux RPM exceptions with spelling errors in the exception text.

Raising Exceptions

As a Python developer you can choose to throw an exception if a condition occurs.

To throw (or raise) an exception, use the **raise** keyword.

In [ ]:
```
#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")
```

- 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.

In [ ]:
```
x = "hello"

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

In [ ]:
```
def division(a, b):
    try:
        return a / b
    except ZeroDivisionError as ex:
        print('Logging exception:', str(ex))
        raise
```

In [ ]:
```
division(1, 0)
```

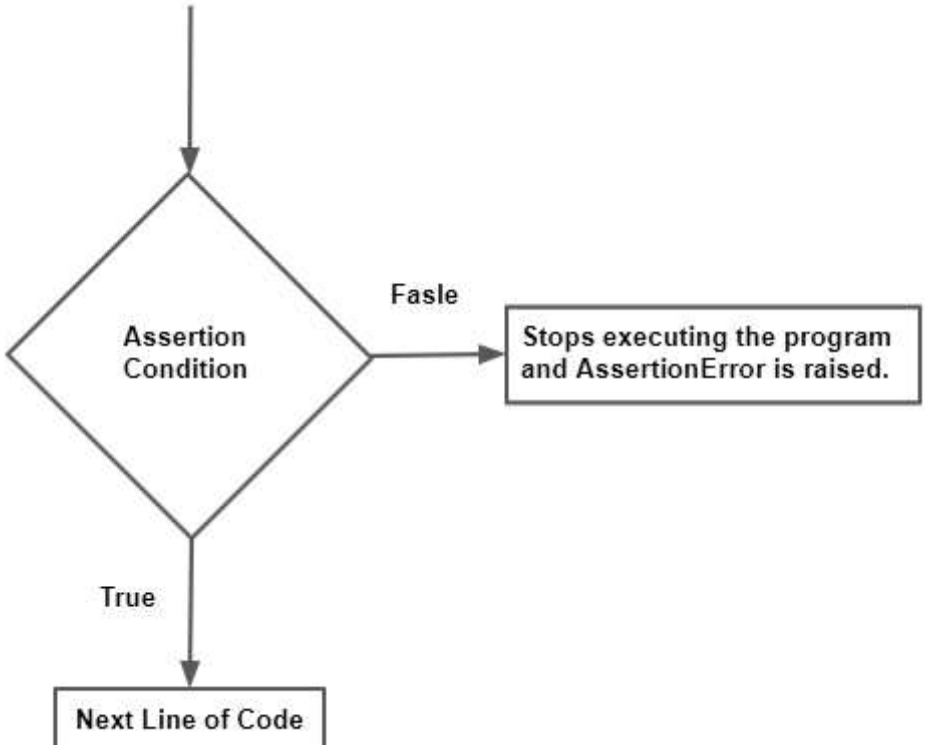
In [ ]:
```
def division(a, b):
    try:
        return a / b
    except ZeroDivisionError as ex:
        raise ValueError('b must not zero')
```

In [ ]:
```
division(1, 0)
```

Assertion

Assertion is a programming concept used while writing a code where the user declares a condition to be true using assert statement prior to running the module. If the condition is True, the control simply moves to the next line of code. In case if it is False the program stops running and returns AssertionError Exception.

The function of assert statement is the same irrespective of the language in which it is implemented, it is a language-independent concept, only the syntax varies with the programming language.



In [1]:
```
# AssertionError with error_message.
x = 1
y = 0
assert y != 0, "Invalid Operation" # denominator can't be 0
print(x / y)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[1], line 4
      2 x = 1
      3 y = 0
----> 4 assert y != 0, "Invalid Operation" # denominator can't be 0
      5 print(x / y)

AssertionError: Invalid Operation
```

The default exception handler in python will print the error_message written by the programmer, or else will just handle the error without any message. Both of the ways are valid.

Handling AssertionError exception: AssertionError is inherited from Exception class, when this exception occurs and raises AssertionError there are two ways to handle, either the user handles it or the default exception handler.

In [2]:
```
# Handling it manually
try:
    x = 1
    y = 0
    assert y != 0, "Invalid Operation"
    print(x / y)

# the errror_message provided by the user gets printed
except AssertionError as msg:
    print(msg)
```

```
Invalid Operation
```

In [3]:
```
# Roots of a quadratic equation
import math
def cmrtc(a, b, c):
    try:
        assert a != 0, "Not a quadratic equation as coefficient of x ^ 2 can't
        D = (b * b - 4 * a*c)
        assert D>= 0, "Roots are imaginary"
        r1 = (-b + math.sqrt(D))/(2 * a)
        r2 = (-b - math.sqrt(D))/(2 * a)
        print("Roots of the quadratic equation are :", r1, "", r2)
    except AssertionError as msg:
        print(msg)
cmrtc(-1, 5, -6)
cmrtc(1, 1, 6)
cmrtc(2, 12, 18)
```

```
Roots of the quadratic equation are : 2.0  3.0
Roots are imaginary
Roots of the quadratic equation are : -3.0  -3.0
```

Standard Exceptions

- **StopIteration** : It raised when the next() method in iterator does not point to any object or we can say when the number of items is not present in the container for the next() method.
- **SystemExit** : This is raised when the sys.exit() function does not use carefully or fails to work.
- **FloatingPointError** : It raised when the calculation involves floating-point fails.
- **ZeroDivisonError** : This raised when we divide the number by zero.
- **AssertionError** : raised when the Assert statement fails.
- **ImportError** : Raised if the import statement fails.
- **IndexError**: Raised if the index is not found in the given sequence.
- **KeyError** : Raised if the specified key appears to be not found in the given dictionary.
- **IndentationError** : When we don't provide the proper indentation.
- **TypeError** : Raised when we give the wrong input to the program like an integer in place of a name.
- **ValueError** : Raised when we provide the invalid values to the argument that specified to the built-in function.

- **RuntimeError**: This error raised when the generated error does not fall into any type of category.
- **KeyboardInterrupt** : When we run our code and by mistake, we press the wrong key then this error is raised.
- **UnboundLocalError** : This error is raised when we want to access the local variable that we have created but we have to forget to assign value to it.
- **SyntaxError** : When the written code does not according to the python syntax then this error is raised.
- **SystemError**: This error is raised when the interpreter finds some system-related problems.
- **ModuleNotFoundError**: It raised when the import statements unable to load the specified module.
- **RecursionError**: It raised when the recursion depth exceeds its limiting value.
- **ReferenceError**: This raised when the generated error does not fall in any of the categories.
- **TabError** : Raised when the given indentation contains some extra or unwanted tabs and spaces.

Creating Exceptions

Catching all exceptions is sometimes used as a crutch by programmers who can't remember all of the possible exceptions that might occur in complicated operations. As such, it is also a very good way to write undebuggable code. Because of this, if one catches all exceptions, it is absolutely critical to log or reports the actual reason for the exception somewhere (e.g., log file, error message printed to screen, etc.).

Problem – Code that catches all the exceptions

In [4]:
```
try:
    ...
except Exception as e:
    ...
# Important
    log('Reason:', e)
```

- This will catch all exceptions save **SystemExit**, **KeyboardInterrupt**, and **GeneratorExit**.

In [5]:
```
def parse_int(s):
    try:
        n = int(v)
    except Exception:
        print("Couldn't parse")
```

In [6]:
```
print (parse_int('n / a'), "\n")

print (parse_int('42'))
```

```
Couldn't parse
None

Couldn't parse
None
```

In [7]:
```
def parse_int(s):
    try:
        n = int(v)
    except Exception as e:
        print("Couldn't parse")
        print('Reason:', e)
```

In [8]:
```
parse_int('42')
```

```
Couldn't parse
Reason: name 'v' is not defined
```

Problem – To wrap lower-level exceptions with custom ones that have more meaning in the context of the application (one is working on).

To create new exceptions just define them as classes that inherit from Exception (or one of the other existing exception types if it makes more sense).

In [9]:
```
#Defining some custom exceptions
class NetworkError(Exception):
    pass
class HostnameError(NetworkError):
    pass
class TimeoutError(NetworkError):
    pass
class ProtocolError(NetworkError):
    pass
```

In [10]:
```
#Using these exceptions in the normal way.
try:
    msg = s.recv()
except TimeoutError as e:
    ...
except ProtocolError as e:
    ...
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[10], line 3
      1 #Using these exceptions in the normal way.
      2 try:
----> 3     msg = s.recv()
      4 except TimeoutError as e:
      5     ...

NameError: name 's' is not defined
```

- Custom exception classes should almost always inherit from the built-in Exception class, or inherit from some locally defined base exception that itself inherits from Exception.
- BaseException is reserved for system-exiting exceptions, such as KeyboardInterrupt or SystemExit, and other exceptions that should signal the application to exit. Therefore, catching these exceptions is not the intended use case.

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, more than what we can achieve by simply using only the exception argument.

Let us see what we get using sys.exc_info():

In [11]:
```
try:
    float('abc123')
except:
    import sys
    exc_tuple = sys.exc_info()
```

In [13]:
```
print(exc_tuple)
```

```
(<class 'ValueError'>, ValueError("could not convert string to float: 'abc12
3'"), <traceback object at 0x000001EDFE907240>)
```

In [15]:
```
for eachItem in exc_tuple:
    print(eachItem)
```

```
<class 'ValueError'>
could not convert string to float: 'abc123'
<traceback object at 0x000001EDFE907240>
```

Modules

A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code. Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

Create a Python Module

Let's create a simple calc.py in which we define two functions, one add and another subtract.

In [16]:
```
# A simple module, calc.py
def add(x, y):
    return (x+y)

def subtract(x, y):
    return (x-y)
```

Import module in Python

We can import the functions, and classes defined in a module to another module using the import statement in some other Python source file.

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 for importing a module. For example, to import the module calc.py, we need to put the following command at the top of the script.

Syntax:

import module

Note: This does not import the functions or classes directly instead imports the module only. To access the functions inside the module the dot(.) operator is used.

In [17]:
```
# importing module calc.py
import calc

print(calc.add(10, 2))
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[17], line 2
      1 # importing module calc.py
----> 2 import calc
      4 print(calc.add(10, 2))

ModuleNotFoundError: No module named 'calc'
```

Python Import From Module

Python's from statement lets you import specific attributes from a module without importing the module as a whole.

Import Specific Attributes from a Python module Here, we are importing specific sqrt and factorial attributes from the math module.

In [18]:
```
# importing sqrt() and factorial from the module math
from math import sqrt, factorial

# if we simply do "import math", then math.sqrt(16) and math.factorial()are re
print(sqrt(16))
print(factorial(6))
```

```
4.0
720
```

Importing Module Attributes

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

In [23]:
```
#Save this code in the file mymodule.py
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

In [24]:
```
#Import the module named mymodule, and access the person1 dictionary:
import mymodule

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

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[24], line 2
      1 #Import the module named mymodule, and access the person1 dictionary:
----> 2 import mymodule
      4 a = mymodule.person1["age"]
      5 print(a)

ModuleNotFoundError: No module named 'mymodule'
```

Import all Names

The * symbol used with the import statement is used to import all the names from a module to a current namespace.

Syntax:

from module_name import *

What does import * do in Python

The use of * has its advantages and disadvantages. If you know exactly what you will be needing from the module, it is not recommended to use *, else do so.

In [19]:
```
# importing sqrt() and factorial from the
# module math
from math import *

# if we simply do "import math", then
# math.sqrt(16) and math.factorial()
# are required.
print(sqrt(16))
print(factorial(6))
```

```
4.0
720
```

Locating Python Modules

Whenever a module is imported in Python the interpreter looks for several locations. First, it will check for the built-in module, if not found then it looks for a list of directories defined in the sys.path.

Python interpreter searches for the module in the following manner –

- First, it searches for the module in the current directory.
- If the module isn't found in the current directory, Python then searches each directory in the shell variable PYTHONPATH. The PYTHONPATH is an environment variable, consisting of a list of directories.
- If that also fails python checks the installation-dependent list of directories configured at the time Python is installed.

Directories List for Modules Here, sys.path is a built-in variable within the sys module. It contains a list of directories that the interpreter will search for the required module.

In [20]:
```
# importing sys module
import sys

# importing sys.path
print(sys.path)
```

```
['C:\\Users\\NUTHANAKANTI BHASKAR\\Python Programming', 'C:\\Users\\NUTHANAKA
NTI BHASKAR\\anaconda3\\python39.zip', 'C:\\Users\\NUTHANAKANTI BHASKAR\\anac
onda3\\DLLs', 'C:\\Users\\NUTHANAKANTI BHASKAR\\anaconda3\\lib', 'C:\\Users
\\NUTHANAKANTI BHASKAR\\anaconda3', '', 'C:\\Users\\NUTHANAKANTI BHASKAR\\ana
conda3\\lib\\site-packages', 'C:\\Users\\NUTHANAKANTI BHASKAR\\anaconda3\\lib
\\site-packages\\win32', 'C:\\Users\\NUTHANAKANTI BHASKAR\\anaconda3\\lib\\si
te-packages\\win32\\lib', 'C:\\Users\\NUTHANAKANTI BHASKAR\\anaconda3\\lib\\s
ite-packages\\Pythonwin']
```

Renaming the Python module

We can rename the module while importing it using the keyword.

Syntax:

Import Module_name as Alias_name

In [21]:
```
# importing sqrt() and factorial from the module math
import math as mt

# if we simply do "import math", then math.sqrt(16) and math.factorial() are r
print(mt.sqrt(16))
print(mt.factorial(6))
```

```
4.0
720
```

Python Built-in modules

There are several built-in modules in Python, which you can import whenever you like.

In [22]:
```
# 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 degrees
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))

# 1 * 2 * 3 * 4 = 24
print(math.factorial(4))

# 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)

List = [1, 4, True, 800, "python", 27, "hello"]

# using choice function in random module for choosing
# a random element from a set such as a list
print(random.choice(List))


# importing built in module datetime
import datetime
from datetime import date
import time

# Returns the number of seconds since the Unix Epoch, January 1st 1970
print(time.time())

# Converts a number of seconds to a date object
```

```
print(date.fromtimestamp(454554))
```

```
5.0
3.141592653589793
114.59155902616465
1.0471975511965976
0.9092974268256817
0.8775825618903728
0.23414336235146527
24
3
0.49009830763434903
85.7164694347192
4
1687530998.755913
1970-01-06
```

Packages

Python modules may contain several classes, functions, variables, etc. whereas Python packages contain several modules. In simpler terms, Package in Python is a folder that contains various modules as files.

Creating Package

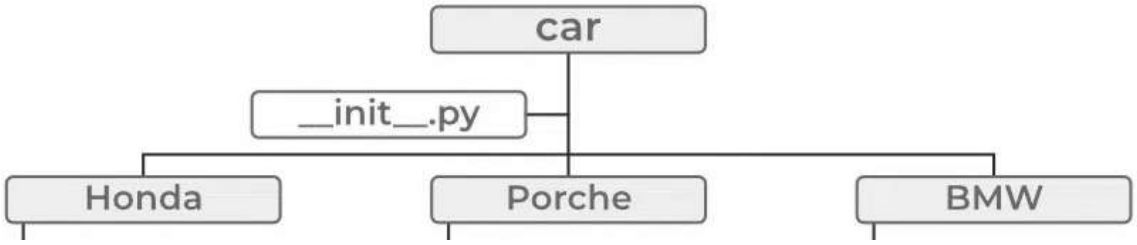
Let's create a package in Python named mypckg that will contain two modules mod1 and mod2. To create this module follow the below steps:

- Create a folder named mypckg.
- Inside this folder create an empty Python file i.e. **init**.py
- Then create two modules mod1 and mod2 in this folder.

In [25]:
```
#Mod1.py
def gfg():
    print("Welcome to GFG")
```

In [26]:
```
#Mod2.py
def sum(a, b):
    return a+b
```

The Hierarchy of our Python package looks like this:



Understanding init.py

init.py helps the Python interpreter recognize the folder as a package. It also specifies the resources to be imported from the modules. If the **init**.py is empty this means that all the functions of the modules will be imported. We can also specify the functions from each module to be made available.

For example, we can also create the **init**.py file for the above module as:

init.py

```
In [27]: from .mod1 import gfg
         from .mod2 import sum
```

```
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
Cell In[27], line 1
----> 1 from .mod1 import gfg
      2 from .mod2 import sum

ImportError: attempted relative import with no known parent package
```

Import Modules from a Package

We can import these Python modules using the from…import statement and the dot(.) operator.

Syntax:

import package_name.module_name

In [28]:
```
#We will import the modules from the above-created package and will use the fu
from mypckg import mod1
from mypckg import mod2

mod1.gfg()
res = mod2.sum(1, 2)
print(res)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[28], line 2
      1 #We will import the modules from the above-created package and will u
se the functions inside those modules.
----> 2 from mypckg import mod1
      3 from mypckg import mod2
      5 mod1.gfg()

ModuleNotFoundError: No module named 'mypckg'
```

In [29]:
```
#We can also import the specific function also using the same syntax.
from mypckg.mod1 import gfg
from mypckg.mod2 import sum

gfg()
res = sum(1, 2)
print(res)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[29], line 2
      1 #We can also import the specific function also using the same syntax.
----> 2 from mypckg.mod1 import gfg
      3 from mypckg.mod2 import sum
      5 gfg()

ModuleNotFoundError: No module named 'mypckg'
```

In [ ]:
```

```