# PYTHON PROGRAMMING

Presented By
POTU NARAYANA
Research Scholar
Osmania University
Dept. of CSE
B.Tech, M.Tech, M.A, PGDHR, MIAENG, MIS,(Ph.D)
Email : pnarayana@osmania.ac.in
Contact : +91 9704868721

# 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.
- Types of Files:

  There are 2 types of files

  1) Text Files: Usually we can use text files to store character data

  Eg: abc.txt
- 2) Binary Files:
- Usually we can use binary files to store binary data like images, video files, audio files etc...

# File Built in Functions

- **Opening a File**:

  Before performing any operation (like read or write) on the file, first we have to open that file. For this we should use Python's inbuilt function open()

  But at the time of open, we have to specify mode, which represents the purpose of opening file.

  - f = open(filename, mode)

- The allowed modes in Python are

  1) r → open an existing file for read operation. The file pointer is positioned at the beginning of the file. If the specified file does not exist then we will get FileNotFoundError.This is default mode.

  2) w → open an existing file for write operation. If the file already contains some data then it will be overridden. If the specified file is not already available then this mode will create that file.

  3) a → open an existing file for append operation. It won't override existing data. If the specified file is not already available then this mode will create a new file.

  4) r+ → To read and write data into the file. The previous data in the file will not be deleted. The file pointer is placed at the beginning of the file.

  5) w+ → To write and read data. It will override existing data.

  6) a+ → To append and read data from the file. It wont override existing data.

  7) x → To open a file in exclusive creation mode for write operation. If the file already exists then we will get FileExistsError.

- **Note**: All the above modes are applicable for text files. If the above modes suffixed with 'b' then these represents for binary files.

  Eg: rb,wb,ab,r+b,w+b,a+b,xb

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

  We are opening abc.txt file for writing data.

## The file() Factory Function

- The file() built-in function came into being in Python 2.2, during the types and classes unification. At this time, many built-in types that did not have associated built-in functions were given factory functions to create instances of those objects, i.e., dict(), bool(), file(), etc., to go along with those that did, i.e., list(), str(), etc.

- Both open() and file() do exactly the same thing and one can be used in place of the other. Anywhere you see references to open(), you can mentally substitute file() without any side effects whatsoever

- **Closing a File:**

  After completing our operations on the file, it is highly recommended to close the file. For this we have to use close() function.

  f.close()

- **Various Properties of File Object:**

  Once we opened a file and we got file object, we can get various details related to that file by using its properties.

  name → Name of opened file

  mode → Mode in which the file is opened

  closed → Returns boolean value indicates that file is closed or not

  readable() → Returns boolean value indicates that whether file is readable or not

  writable() → Returns boolean value indicates that whether file is writable or not.

```
f=open("abc.txt",'w')
 print("File Name: ",f.name)
 print("File Mode: ",f.mode)
 print("Is File Readable: ",f.readable())
 print("Is File Writable: ",f.writable())
 print("Is File Closed : ",f.closed)
 f.close()
 print("Is File Closed : ",f.closed)
```

**Output**

```
D:\Python_classes>py test.py
File Name: abc.txt
File Mode: w
Is File Readable: False
Is File Writable: True
Is File Closed: False
Is File Closed: True
```

# Writing Data to Text Files:

- We can write character data to the text files by using the following 2 methods.
    1) write(str)
    2) writelines(list of lines)

```
f=open(“xyz.txt",'w')
f.write("python\n")
f.write("programming\n")
f.write("language\n")
print("Data written to the file successfully")
f.close()
```

xyz.txt:

python
programming
language

- **Eg 2:**

```
f=open("abcd.txt",'w') list=["sunny\n","bunny\n","vinny\n","chinny"]
f.writelines(list)
print("List of lines written to the file successfully")
f.close()
```

abcd.txt:

sunny

bunny

vinny

chinny

**Reading Character Data from Text Files:**

- We can read character data from text file by using the following read methods.

  read() → To read total data from the file

  read(n) → To read 'n' characters from the file

  readline() → To read only one line

  readlines() → To read all lines into a list

- Eg 1: To read total data from the file

```
f=open("abc.txt",'r')
data=f.read()
print(data)
f.close()
```

Output

```
sunny
bunny
chinny
vinny
```

- Eg 2: To read only first 10 characters:

```
f=open("abc.txt",'r')
data=f.read(10)
print(data)
f.close()
```

Output:

```
sunny
bunn
```

- Eg 3: To read data line by line:

```
f=open("abc.txt",'r')
line1=f.readline()
print(line1,end='')
line2=f.readline()
print(line2,end='')
line3=f.readline()
print(line3,end='')
f.close()
```

- **Output**

```
sunny
bunny
chinny
```

- Eg 4: To read all lines into list:

```
f=open("abcd.txt",'r')
lines=f.readlines()
for line in lines:
print(line,end='')
f.close()
```

- Output

```
sunny
bunny
chinny
vinny
```

- Eg 5:

```
f=open("abcd.txt","r")
print(f.read(3))
print(f.readline())
print(f.read(4))
print("Remaining data")
print(f.read())
```

Output

```
sun
ny
bunn
Remaining data
y
chinny
vinny
```

- The seek() and tell() Methods:

tell():

- We can use tell() method to return current position of the cursor(file pointer) from beginning of the file. [ can you plese telll current cursor position]
- The position(index) of first character in files is zero just like string index.

```
f=open("abcd.txt","r")
print(f.tell())
print(f.read(2))
print(f.tell())
print(f.read(3))
print(f.tell())
```

**abc.txt:**

**sunny**
**bunny**
**chinny**
**vinny**

**Output:**

**0**
**su**
**2**
**nny**
**5**

- **seek():**
- We can use seek() method to move cursor (file pointer) to specified location.
- [Can you please seek the cursor to a particular location]
- f.seek(offset, fromwhere) → offset represents the number of positions
- The allowed Values for 2nd Attribute (from where) are
- 0 → From beginning of File (Default Value)
- 1 → From Current Position
- 2 → From end of the File

```
data="All Students are STUPIDS"
f=open("abc.txt","w")
f.write(data)
with open("abc.txt","r+") as f:
text=f.read()
print(text)
print("The Current Cursor Position: ",f.tell())
f.seek(17)
print("The Current Cursor Position: ",f.tell())
f.write("GEMS!!!")
f.seek(0)
text=f.read()
print("Data After Modification:")
print(text)
```

# How to check a particular File exists OR not?

- We can use os library to get information about files in our computer.
- os module has path sub module,which contains isFile() function to check whether a particular file exists or not?

  os.path.isfile(fname)
- **Write a Program to check whether the given File exists OR not. If it is available then print its content?**

```
import os,sys
fname=input("Enter File Name: ")
if os.path.isfile(fname):
print("File exists:",fname)
f=open(fname,"r")
else: print("File does not exist:",fname)
sys.exit(0)
print("The content of file is:")
data=f.read()
print(data)
```

**Note:**
**sys.exit(0) □ To exit system without executing rest of the program.**
**argument represents status code. 0 means normal termination and it is the default value**

# File built in Attributes

- **File Built-in Attributes**
- File objects also have data attributes in addition to methods. These attributes hold auxiliary data related to the file object they belong to, such as the file name (*file.name), the mode with which the file was*

  opened (*file.mode), whether the file is closed (file.closed), and a flag indicating whether an additional* space character needs to be displayed before successive data items when using the **print statement** (*file.softspace).*
- *file.closed*

  *TRue if file is closed and False otherwise*
- *file.encoding*

  Encoding that this file useswhen Unicode strings are written to file, they will be converted to byte strings using *file.encoding; a value of None indicates* that the system default encoding for converting Unicode strings should be used
- *file.mode*
- *Access mode with which file was opened*
- *file.name*
- *Name of file*
- *file.newlines*
- None if no line separators have been read, a string consisting of one type of
- line separator, or a tuple containing all types of line termination characters
- read so far
- *file.softspace*

  *0 if space explicitly required with print, 1 otherwise; rarely used by the* programmer generally for internal use only

# Standard Files

- Python system module is something cool if you are really interested in python. As of now, I am only showing how you can use **stdin**, **stdout** and **stderr** functions. You can do a lot of cool stuffs using system module. The stdin function can be used to get input data from the python shell. The stdout function is used for outputs and stderr is used to show error messages.

```
#Import System module
import sys
print("Enter a number")
#input is read using stdin
x=int(sys.stdin.readline())
print(x)
#output printed using stdout
sys.stdout.write('This is stdout text\n')
#Error message is shown using stderr
sys.stderr.write('This is a stderror\n')
```

```
Enter a number
 10
10
This is stdout text
This is a stderror
>>>
```

# Command Line Arguments

- The sys module also provides access to any *command-line arguments via sys.argv. Command-line* arguments are those arguments given to the program in addition to the script name on invocation.
- Historically, of course, these arguments are so named because they are given on the command line along with the program name in a text-based environment like a Unix- or DOS-shell. However, in an IDE or GUI environment, this would not be the case. Most IDEs provide a separate window with which to enter your "command-line arguments." These, in turn, will be passed into the program as if you started your application from the command line.
- Those of you familiar with C programming may ask, "Where is argc?" The names "argc" and "argv" stand for "argument count" and "argument vector," respectively. The argv variable contains an array of strings consisting of each argument from the command line while the argc variable contains the number of arguments entered. 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. Summary:
- ● sys.argv *is the list of command-line arguments*
- ● len(sys.argv) *is the number of command-line arguments(aka argc)*

Eg:

```
from sys import argv
print("the no of cmd line arguments are:",len(argv))
print("the list of cmd line arguments are:",argv)
print("The cmd line arguments one by one:")
for x in argv:print(x)
```

**Output**

```
C:\Python 3.7.4>py cmdl.py 10 20 30
the no of cmd line arguments are: 4
the list of cmd line arguments are: ['cmdl.py', '10', '20', '30']
The cmd line arguments one by one:
cmdl.py
10
20
30
```

# File System

- Access to your file system occurs mostly through the Python os module. 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, keeping all the underlying work hidden from sight. Depending on what your system supports, you may not have access to some of the attributes, which may be available in other operating system modules.

# os Module File/Directory Access Functions

| *Function* | *Description* |
|---|---|
| ***File Processing*** | |
| mkfifo()/mknod() | Create named pipe/create file system node |
| remove()/unlink() | Delete file |
| rename()/renames() | Rename file |
| *stat( ) | Return file statistics |
| symlink() | Create symbolic link |
| utime() | Update timestamp |
| tmpfile() | Create and open ('w+b') new temporary file |
| walk() | Generate filenames in a directory tree |
| ***Directories/Folders*** | |
| chdir()/fchdir() | Change working directory/via a file descriptor |
| chroot() | Change root directory of current process |
| listdir() | List files in directory |
| getcwd()/getcwdu() | Return current working directory/same but in Unicode |
| mkdir()/makedirs() | Create directory(ies) |
| rmdir()/removedirs() | Remove directory(ies) |

- ***Access/Permissions***

| | |
|---|---|
| access() | Verify permission modes |
| chmod() | Change permission modes |
| chown()/lchown() | Change owner and group ID/same, but do not follow links |
| umask() | Set default permission modes |

- ***File Descriptor Operations***

| | |
|---|---|
| open() | Low-level operating system open [for files, use the standard open() built-in functions |
| read()/write() | Read/write data to a file descriptor |
| dup()/dup2() | Duplicate file descriptor/same but to another FD |

- ***Device Numbers***

| | |
|---|---|
| makedev() | Generate raw device number from major and minor device numbers |
| major()/minor() | Extract major/minor device number from raw device number |

- second module that performs specific pathname operations is also available. The os.path module is accessible through the os module. Included with this module are functions to manage and manipulate file pathname components, obtain file or directory information, and make file path inquiries
- **os.path Module Pathname Access**

| **Functions** | ***Function Description*** |
|---|---|
| ***Separation*** | |
| basename() | Remove directory path and return leaf name |
| dirname() | Remove leaf name and return directory path |
| join() | Join separate components into single pathname |
| split() | Return (*dirname(), basename()) tuple* |
| splitdrive() | Return (*drivename, pathname) tuple* |
| splitext() | Return (*filename, extension) tuple* |

- ***Information***
- getatime() Return last file access time
- getctime() Return file creation time
- getmtime() Return last file modification time
- getsize() Return file size (in bytes)
- ***Inquiry***
- exists() Does pathname (file or directory) exist?
- isabs() Is pathname absolute?
- isdir() Does pathname exist and is a directory?
- isfile() Does pathname exist and is a file?
- islink() Does pathname exist and is a symbolic link?
- ismount() Does pathname exist and is a mount point?
- samefile() Do both pathnames point to the same file?

# Persistent Storage Modules

- In many of the exercises in this text, user input is required. After many iterations, it may be somewhat frustrating being required to enter the same data repeatedly. The same may occur if you are entering a significant amount of data for use in the future. This is where it becomes useful to have persistent storage, or a way to archive your data so that you may access them at a later time instead of having to re-enter all of that information. When simple disk files are no longer acceptable and full relational database management systems (RDBMSs) are overkill, simple persistent storage fills the gap. The majority of the persistent storage modules deals with storing strings of data, but there are ways to archive Python objects as well.

- **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.

- As we mentioned before, both marshal and pickle can flatten Python objects. These modules do not provide "persistent storage" per se, since they do not provide a namespace for the objects, nor can they provide concurrent write access to persistent objects. What they can do, however, is to pickle Python objects to allow them to be stored or transmitted. Storage, of course, 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 pickle 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.

# 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 any dbm module detects which DBM-compatible modules are installed on your system and uses the "best" one at its disposal. The dumbdbm module is the most limited one, and is the default used if none of the other packages is available. These modules do provide a namespace for your objects, using objects that behave similar to a combination of a dictionary object and a file object. The one limitation of these systems is that they can store only strings. In other words, they do not serialize Python objects
- **shelve *Module***
- Finally, we have a somewhat more complete solution, the shelve module. 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.This is about as close to persistent storage as you will find in the Python standard library. There may be other external extension modules that implement "true" persistent storage.

```
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

# Related Modules

| Module(s) | Contents |
| --- | --- |
| Base64 | Encoding/decoding of binary strings to/from text strings |
| binascii | Encoding/decoding of binary and ASCII-encoded binary strings |
| bz2 | Allows access to BZ2 compressed files |
| csv | Allows access to comma-separated value files |
| filecmp | Compares directories and files |
| fileinput | Iterates over lines of multiple input text files |
| getopt/optparse | Provides command-line argument parsing/manipulation |
| glob/fnmatch | Provides Unix-style wildcard character matching |
| gzip/zlib | Reads and writes GNU zip (gzip) files (needs zlib module for compression) |
| shutil | Offers high-level file access functionality |
| c/StringIO | Implements file-like interface on top of string objects |
| Tarfile | Reads and writes TAR archive files, even compressed ones |
| tempfile | Generates temporary file names or files |
| Zipfile | Tools and utilities to read and write ZIP archive files |

# Exceptions

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them:

- **Exception Handling:**
- **Assertions:**

**What is Exception?**

- An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- In general, when a Python script encounters a situation that it can't cope with, it raises an exception. An exception is a Python object that represents an error.
- When a Python script raises an exception, it must either handle the exception immediately otherwise it would terminate and come out.

- **NameError:** ***attempt to access an undeclared variable***

```
>>> foo
Traceback (innermost last):
File "<stdin>", line 1, in ?
NameError: name ‘foo' is not defined
```

- **ZeroDivisionError:** ***division by any numeric zero***

```
>>> 1/0
Traceback (innermost last):
File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

- **SyntaxError:** ***Python interpreter syntax error***

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

- **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.
- **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. In this case, a KeyErroris raised to indicate such an incident has occurred.
- **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.

# Detecting an Exception

- Exceptions can be detected by incorporating them as part of a **try statement.** Any code suite of a Try statement will be monitored for exceptions.
- There are two main forms of the **TRy statement: TRy-except and try-finally.** These statements are mutually exclusive, meaning that you pick only one of them. A **try statement can be accompanied by** one or more **except clauses, exactly one finally clause, or a hybrid try-except-finally combination.**
- **try-except statements allow one to detect and handle exceptions. There is even an optional else clause** for situations where code needs to run only when no exceptions are detected. Meanwhile, **TRy-finally**
- statements allow only for detection and processing of any obligatory cleanup (whether or not exceptions occur), but otherwise have no facility in dealing with exceptions. The combination, as you might imagine, does both.

# Handling an exception:

- If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

**Syntax:**

```
try:
   You do your operations here;
   ......................
except Exception I:
   If there is ExceptionI, then execute this block.
except Exception II:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Here are few important points about the above mentioned syntax:

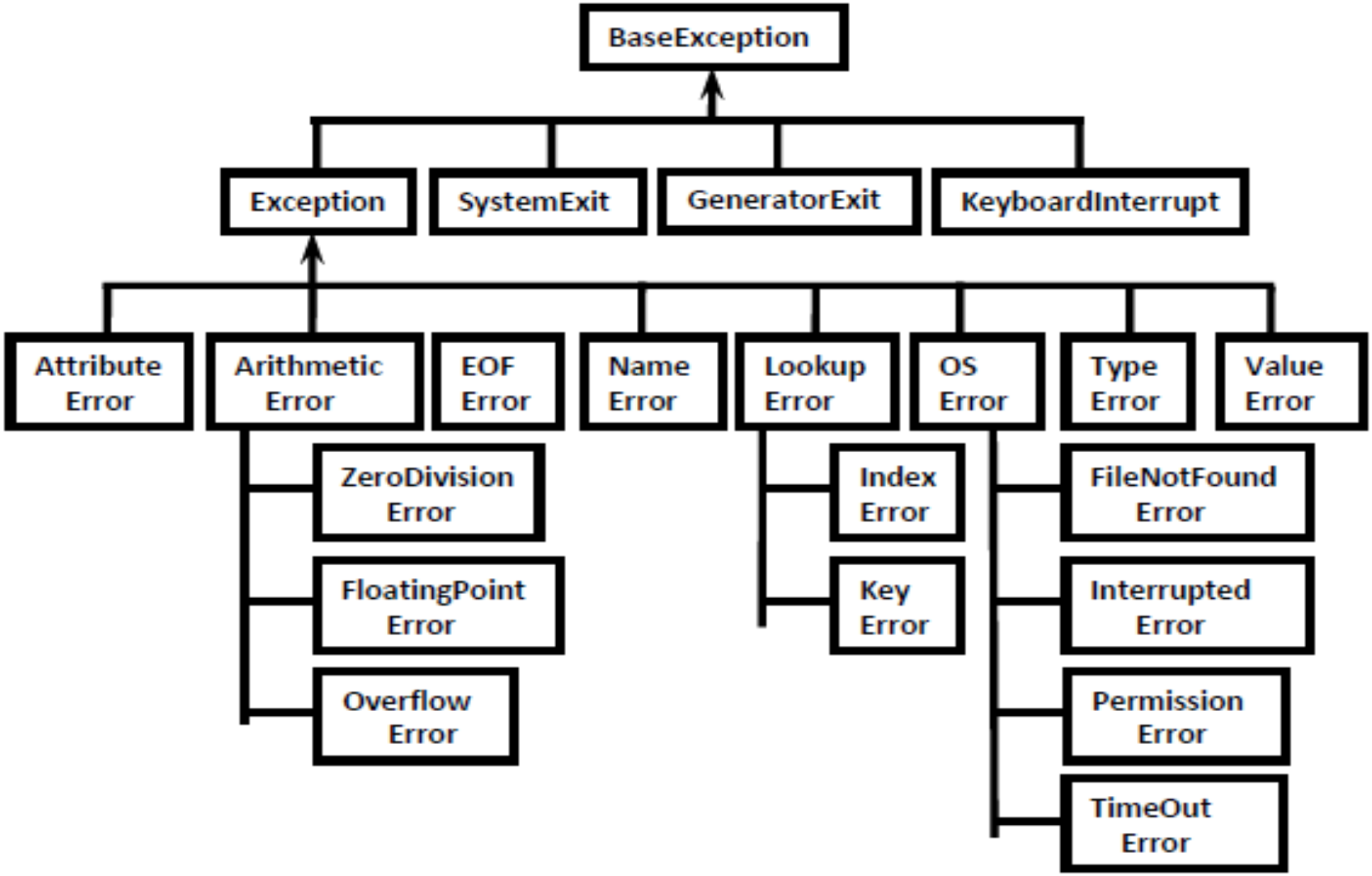
A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.

You can also provide a generic except clause, which handles any exception.

After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.

The else-block is a good place for code that does not need the try: block's protection.

# Python's Exception Hierarchy



- Every Exception in Python is a class.
- All exception classes are child classes of BaseException.i.e every exception class extends BaseException either directly or indirectly. Hence BaseException acts as root for Python Exception Hierarchy.
- Most of the times being a programmer we have to concentrate Exception and its child classes.
- **Customized Exception Handling by using try-except:**
- **It is highly recommended to handle exceptions.**
- **The code which may raise exception is called risky code and we have to take risky code inside try block. The corresponding handling code we have to take inside except block.**

```
try:
        Risky Code
except XXX:
        Handling code/Alternative Code
```

- **Without try-except:**
- **1) print("stmt-1")**
- **2) print(10/0)**
- **3) print("stmt-3")**

  **Output**

  **stmt-1**

  **ZeroDivisionError: division by zero**

  **Abnormal termination/Non-Graceful Termination**
- **With try-except:**
- **1) print("stmt-1")**
- **2) try:**
- **3) print(10/0)**
- **4) except ZeroDivisionError:**
- **5) print(10/2)**
- **6) print("stmt-3")**

  **Output**

  **stmt-1**

  **5.0**

  **stmt-3**

  **Normal termination/Graceful Termination**

Control Flow in try-except:

- try:
- stmt-1
- stmt-2
- stmt-3
- except XXX:
- stmt-4
- stmt-5

**Case-1: If there is no exception**
**1,2,3,5 and Normal Termination**
**Case-2: If an exception raised at stmt-2 and corresponding except block matched**
**1,4,5 Normal Termination**
**Case-3: If an exception rose at stmt-2 and corresponding except block not matched**
**1, Abnormal Termination**
**Case-4: If an exception rose at stmt-4 or at stmt-5 then it is always abnormal termination.**

# try with Multiple except Blocks:

- The way of handling exception is varied from exception to exception.Hence for every exception type a seperate except block we have to provide. i.e try with multiple except blocks is possible and recommended to use.
- Eg:
- try:
- -------
- -------
- -------
- except ZeroDivisionError:
- perform altern
- except FileNotFoundError:
- use local file instead of remote file
- If try with multiple except blocks available then based on raised exception the corresponding except block will be executed.
- 1) try:
- 2) x=int(input("Enter First Number: "))
- 3) y=int(input("Enter Second Number: "))
- 4) print(x/y)
- 5) except ZeroDivisionError :
- 6) print("Can't Divide with Zero")
- 7) except ValueError:
- 8) print("please provide int value only")

```
D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: 2
5.0

D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: 0
Can't Divide with Zero

D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: ten
please provide int value only ative arithmetic operations
```

# finally Block:

- It is not recommended to maintain clean up code(Resource Deallocating Code or Resource Releasing code) inside try block because there is no guarentee for the execution of every statement inside try block always.
- It is not recommended to maintain clean up code inside except block, because if there is no exception then except block won't be executed.
- Hence we required some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether exception handled or not handled. Such type of best place is nothing but finally block.
- Hence the main purpose of finally block is to maintain clean up code.
- try:
- Risky Code
- except:
- Handling Code
- **finally:**
- **Cleanup code**
- **The speciality of finally block is it will be executed always whether exception raised or not raised and whether exception handled or not handled.**
- **Case-1: If there is no exception**
- **1) try:**
- **2) print("try")**
- **3) except:**
- **4) print("except")**
- **5) finally:**
- **6) print("finally")**

- **Output**
- **try**
- **finally**

- **Case-2: If there is an exception raised but handled**
- **1) try:**
- **2) print("try")**
- **3) print(10/0)**
- **4) except ZeroDivisionError:**
- **5) print("except")**
- **6) finally:**
- **7) print("finally")**

**Output**

- **try**
- **except**
- **finally**
- **Case-3: If there is an exception raised but not handled**
- **1) try:**
- **2) print("try")**
- **3) print(10/0)**
- **4) except NameError:**
- **5) print("except")**
- **6) finally:**
- **7) print("finally")**

**Output**

- **try**
- **finally**
- **ZeroDivisionError: division by zero(Abnormal Termination)**

- Types of Exceptions:
- In Python there are 2 types of exceptions are possible.
- 1) Predefined Exceptions
- 2) User Definded Exceptions

- 1) Predefined Exceptions: Also known as inbuilt exceptions.
- The exceptions which are raised automatically by Python virtual machine whenver a particular event occurs are called pre defined exceptions.
- Eg 1: Whenever we are trying to perform Division by zero, automatically Python will raise ZeroDivisionError.
- print(10/0)

- User Defined Exceptions:  Also known as Customized Exceptions or Programatic Exceptions
- Some time we have to define and raise exceptions explicitly to indicate that something goes wrong, such type of exceptions are called User Defined Exceptions or Customized Exceptions
- Programmer is responsible to define these exceptions and Python not having any idea about these. Hence we have to raise explicitly based on our requirement by using "raise" keyword.
- Eg:
- InSufficientFundsException
- InvalidInputException
- TooYoungException
- TooOldException

# Assertions

- **Assertions**

  Assertions are diagnostic predicates that must evaluate to Boolean true; otherwise, an exception is raised to indicate that the expression is false. These work similarly to the assert macros, which are part of the C language preprocessor, but in Python these are runtime constructs (as opposed to precompile directives).
- If you are new to the concept of assertions, no problem. The easiest way to think of an assertion is to liken it to a **raise-if statement (or to be more accurate, a raise-if-not statement). An expression is** tested, and if the result comes up false, an exception is raised.
- Assertions are carried out by the **assert statement, introduced back in version 1.5.**

# • assert Statement

- The **assert statement evaluates a Python expression, taking no action if the assertion succeeds (similar** to a **pass statement), but otherwise raising an AssertionError exception. The syntax for assert is:**
- **assert *expression [, arguments]***
- Here are some examples of the use of the **assert statement:**

```
assert 1 == 1
assert 2 + 2 == 2 * 2
assert len(['my list', 12]) < 10
assert range(3) == [0, 1, 2]
```

- AssertionError exceptions can be caught and handled like any other exception using the **TRy-except**
- statement, but if not handled, they will terminate the program and produce a traceback similar to the following:

```
>>> assert 1 == 0
Traceback (innermost last):
File "<stdin>", line 1, in ?
AssertionError
```

# 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()
- What we get from sys.exc_info() in a tuple are:
    - exc_type: exception class object
    - exc_value: (this) exception class instance object
    - exc_traceback: traceback object
- The first two items we are familiar with: the actual exception class and this particular exception's instance (which is the same as the exception argument which we discussed in the previous section). The third item, a traceback object, is new. This object provides the execution context of where the exception occurred. It contains information such as the execution frame of the code that was running and the line number where the exception occurred :

# Related Modules

- **Exception-Related Standard Library Modules**

| ***Module*** | ***Description*** |
|---|---|
| exceptions | Built-in exceptions (never need to import this module) |
| Contextlib | Context object utilities for use **with the with statement** |
| sys | Contains various exception-related objects and functions (see sys.ex*) |

# Modules

- A *module allows you to logically organize your Python code. When code gets to be large enough, the* tendency is to break it up into organized pieces that can still interact with one another at a functioning level. These pieces generally have attributes that have some relation to one another, perhaps a singleclass with its member data variables and methods, or maybe a group of related, yet independently operating functions. These pieces should be shared, so Python allows a module the ability to "bring in”and use attributes from other modules to take advantage of work that has been done, maximizing code reusability. This process of associating attributes from other modules with your module is called *importing. In a nutshell, modules are self-contained and organized pieces of Python code that can be* shared.

# Modules and Files

- If modules represent a logical way to organize your Python code, then files are a way to physically organize modules. To that end, each file is considered an individual module, and vice versa. The filename of a module is the module name appended with the .py file extension. There are several aspects we need to discuss with regard to what the file structure means to modules. Unlike other languages in which you import classes, in Python you import modules or module attributes.
- **Namespaces**
- A *namespace is a mapping of names (identifiers) to objects. The process of adding a name to a* namespace consists of *binding the identifier to the object (and increasing the reference count to the* object by one). The Python Language Reference also includes the following definitions: "changing the mapping of a name is called *rebinding [, and] removing a name is unbinding."*

# Importing Modules

- **The import Statement**

  Importing a module requires the use of the **import statement,** whose syntax is:

  **import *module1***

  **import *module2***

  :

  **import *moduleN***

- It is also possible to import multiple modules on the same line like this ...

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

**The from-import Statement**

It is possible to import specific module elements into your own module. By this, we really mean importing specific names from the module into the current namespace. For this purpose, we can use the**from-import statement, whose syntax is:**

- **from *module import name1[, name2[,... nameN]]***

# Module Built-in Functions

- **__import__ ()**

  The __import__() function is new as of Python 1.5, and it is the function that actually does the importing, meaning that the import statement invokes the __import__() function to do its work.
- The purpose of making this a function is to allow for overriding it if the user is inclined to develop his or her own importation algorithm.
- The syntax of __import__() is:

  __import__(*module_name[, globals[, locals[, fromlist]]]*)
- **globals() and locals()**

  The globals() and locals() built-in functions return dictionaries of the global and local namespaces, respectively, of the caller. From within a function, the local namespace represents all names defined for execution of that function, which is what locals() will return. globals(), of course, will return those names globally accessible to that function.
- From the global namespace, however, globals() and locals() return the same dictionary because the global namespace is as local as you can get while executing there. Here is a little snippet of code that calls both functions from both namespaces:

- **reload()**
- The reload() built-in function performs another import on a previously imported module. The syntax of reload() is:
- reload(*module)*

# Packages

- Packages are a way of structuring many packages and modules which helps in a well-organized hierarchy of data set, making the directories and modules easy to access. Just like there are different drives and folders in an OS to help us store files, similarly packages help us in storing other sub-packages and modules, so that it can be used by the user when necessary.
- **Creating and Exploring Packages**
- To tell Python that a particular directory is a package, we create a file named __init__.py inside it and then it is considered as a package and we may create other modules and sub-packages within it. This __init__.py file can be left blank or can be coded with the initialization code for the package.
  **To create a package in Python, we need to follow these three simple steps:**
- First, we create a directory and give it a package name, preferably related to its operation.
- Then we put the classes and the required functions in it.
- Finally we create an __init__.py file inside the directory, to let Python know that the directory is a package.

- Let’s look at this example and see how a package is created. Let’s create a package named Cars and build three modules in it namely, Bmw, Audi and Nissan.
- **First we create a directory and name it Cars.**
- **Then we need to create modules**. To do this we need to create a file with the name Bmw.py and create its content by putting this code into it

```
# Python code to illustrate the Modules
class Bmw:
# First we create a constructor for this class
# and add members to it, here models
def __init__(self):
self.models = ['i8', 'x1', 'x5', 'x6']
# A normal print function
def outModels(self):
print('These are the available models for BMW')
for model in self.models:
print('\t%s ' % model)
```

Then we create another file with the name Audi.py and add the similar type of code to it with different members.

```
# Python code to illustrate the Module
class Audi:
# First we create a constructor for this class
# and add members to it, here models
def __init__(self):
self.models = ['q7', 'a6', 'a8', 'a3']
# A normal print function
def outModels(self):
print('These are the available models for Audi')
for model in self.models:
print('\t%s ' % model)
```

- Then we create another file with the name Nissan.py and add the similar type of code to it with different members.

```
# Python code to illustrate the Module
class Nissan:
# First we create a constructor for this class
# and add members to it, here models
def __init__(self):
        self.models = ['altima', '370z', 'cube', 'rogue']
# A normal print function
def outModels(self):
        print('These are the available models for Nissan')
        for model in self.models:
                print('\t%s ' % model)
```

- **Finally we create the __init__.py file.** This file will be placed inside Cars directory and can be left blank or we can put this initialisation code into it.

```
from Bmw import Bmw
from Audi import Audi
from Nissan import Nissan
```

- Now, let's use the package that we created. To do this make a sample.py file in the same directory where Cars package is located and add the following code to it:

```
# Import classes from your brand new package
from Cars import Bmw
from Cars import Audi
from Cars import Nissan
# Create an object of Bmw class & call its method
ModBMW = Bmw()
ModBMW.outModels()
# Create an object of Audi class & call its method
ModAudi = Audi()
ModAudi.outModels()
# Create an object of Nissan class & call its method
ModNissan = Nissan()
ModNissan.outModels()
```

# Other Features of Modules

- **Auto-Loaded Modules**

  When the Python interpreter starts up in standard mode, some modules are loaded by the interpreter for system use. The only one that affects you is the __builtin__ module, which normally gets loaded in as the __builtins__ module.

  The sys.modules variable consists of a dictionary of modules that the interpreter has currently loaded (in full and successfully) into the interpreter. The module names are the keys, and the location from which they were imported are the values.

  For example, in Windows, the sys.modules variable contains a large number of loaded modules, so we will shorten the list by requesting only the module names. This is accomplished by using the dictionary's

  keys() method:

- >>> **import sys**
- >>> sys.modules.keys()

  ['os.path', 'os', 'exceptions', '__main__', 'ntpath',

  'strop', 'nt', 'sys', '__builtin__', 'site',

  'signal', 'UserDict', 'string', 'stat']

# References


- Core Python Programming , Wesley J.chun, Second Edition, Pearson.
- Python Programming Using Problem Solving Approach, Reema Thareja.
- Core Python Programming, Dr. R.Nageswara Rao
- Core Python Material by Durga Software Solutions