



PYTHON PROGRAMMING

(18IS5DEPYP)

Unit – 3

(EXCEPTION HANDLING, MODULES, OBJECT ORIENTED PROGRAMMING)

Mrs. Bhavani K
Assistant Professor
Dept. of ISE, DSCE

Unit - 3

- **EXCEPTION HANDLING:**
 - Catching and Raising Exceptions
 - Custom Functions:
 - Names and Docstrings
 - Argument and Parameter Unpacking
 - Accessing Variables in the global Scope
- **MODULES:**
 - Modules and Packages:
 - Packages
 - Custom Modules
- **OBJECT ORIENTED PROGRAMMING:**
 - The object Approach
 - The Object Oriented Concepts and Terminology
 - Custom classes:
 - Attribute and Methods
 - Inheritance and Polymorphism

Introduction

- There is a subtle difference between an error and an exception.
 - Errors cannot be handled, while Python exceptions can be handled at the run time.
 - An error can be a syntax (parsing) error, while there can be many types of exceptions that could occur during the execution and are not unconditionally inoperable.
 - An Error might indicate critical problems that a reasonable application should not try to catch, while an Exception might indicate conditions that an application should try to catch.
 - Errors are a form of an unchecked exception and are irrecoverable like an `OutOfMemoryError`, which a programmer should not try to handle.

Why Exception Handling?

- When an exception occurs, following things happen:
 - Program execution abruptly stops.
 - The complete exception message along with the file name and line number of code is printed on console.
 - All the calculations and operations performed till that point in code are lost.
- Exception handling is very important to handle errors gracefully and displaying appropriate message to inform the user about the malfunctioning.
- Exception handling makes the code more robust and helps prevent potential failures that would cause the program to stop in an uncontrolled manner.

Exception Handling

- Exception is unexpected errors which is caused as a result of faulty code during execution.
- Exception handling is a concept used to handle the exceptions that occur during the execution of program.
- Example:
 a = 10
 b = 0
 print("Result of Division: " + str(a/b))

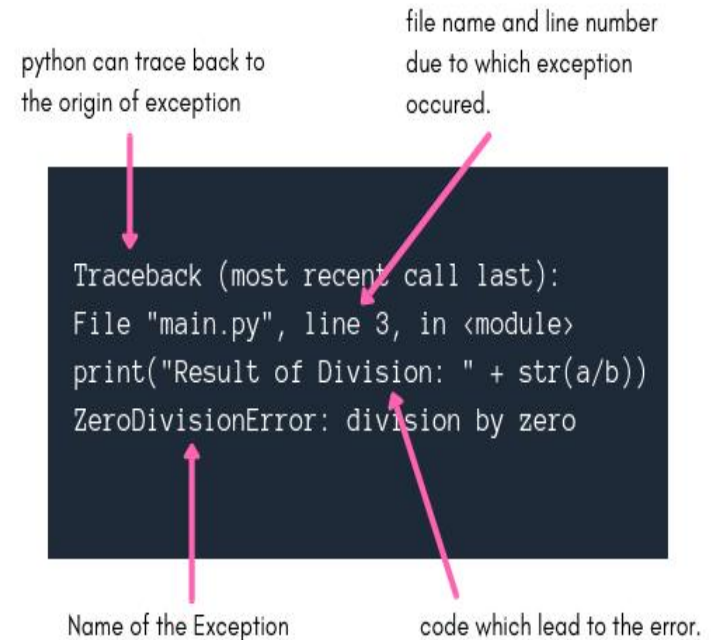
Output:

```
Traceback (most recent call last):  
File "main.py", line 3, in <module>  
print("Result of Division: " + str(a/b))  
ZeroDivisionError: division by zero
```

An exception object is created when a Python script raises an exception. If the script explicitly doesn't handle the exception, the program will be forced to terminate abruptly.

Decoding the Exception Message in Python

- The term **Traceback** in the exception message means that python has traced back the code to the point from where the exception occurred and will be showing the related messages after this line.
- The second line in the exception message, tells us the **name of the python file** and the exact **line number for the code** due to which exception was generated.
- If that is still not helpful for someone, in the third line of exception message the complete code statement which lead to the exception is printed.
- And then in the last line, python tells us which exception/error occurred, which in our example above is **ZeroDivisionError**.



The diagram shows a Python traceback message on a dark background. Four pink arrows point from text labels to specific parts of the message:

- An arrow from "python can trace back to the origin of exception" points to the word "Traceback".
- An arrow from "file name and line number due to which exception occurred." points to "File 'main.py', line 3".
- An arrow from "Name of the Exception" points to "ZeroDivisionError".
- An arrow from "code which lead to the error." points to the code line "print('Result of Division: ' + str(a/b))".

```
Traceback (most recent call last):  
File "main.py", line 3, in <module>  
    print("Result of Division: " + str(a/b))  
ZeroDivisionError: division by zero
```

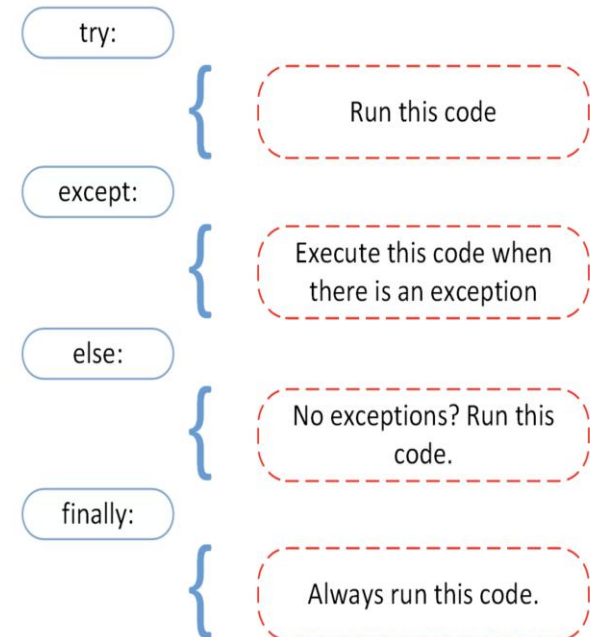
In-built Python Exception classes

Exception class	Description
AttributeError	This exception occurs when the attribute that we are trying to access(whether for assigning a value or getting a value) doesn't exists. For example: Trying to access a class member variable which is not defined in class.
ImportError	This exception occurs when the imported module is not found.
IndentationError	This exception occurs when there is some issue with the code indentation.
TypeError	When an operation is executed on a variable of incorrect type.
ValueError	When for a function, argument value is incorrect.
ZeroDivisionError	As discussed above, when we try to divide a number with zero.
TabError	When the indentation is not consistent throughout the code in terms of tabs and spaces used for indentation.
RuntimeError	When an error is not of any specific defined exception type, python calls it RuntimeError.
NameError	When a variable name we are trying to use is not defined.

Catching and Raising Exceptions

- Exceptions are caught using try ...except blocks, whose general syntax is:

```
try:  
    try_suite  
except exception_group1 as variable1:  
    except_suite1  
...  
except exception_groupN as variableN:  
    except_suiteN  
else:  
    else_suite  
finally:  
    finally_suite
```



- There must be at least one except block, but both the else and the finally blocks are optional.
 - The else block's suite is executed when the try block's suite has finished normally—but it is not executed if an exception occurs.
 - If there is a finally block, it is always executed at the end.

The try block

- The try block is used to put the whole code that is to be executed in the program(which can lead to exception)
 - if any exception occurs during execution of the code inside the try block, then it causes the execution of the code to be directed to the except block and the execution that was going on in the try block is interrupted.
 - But, if no exception occurs, then the whole try block is executed and the except block is never executed.

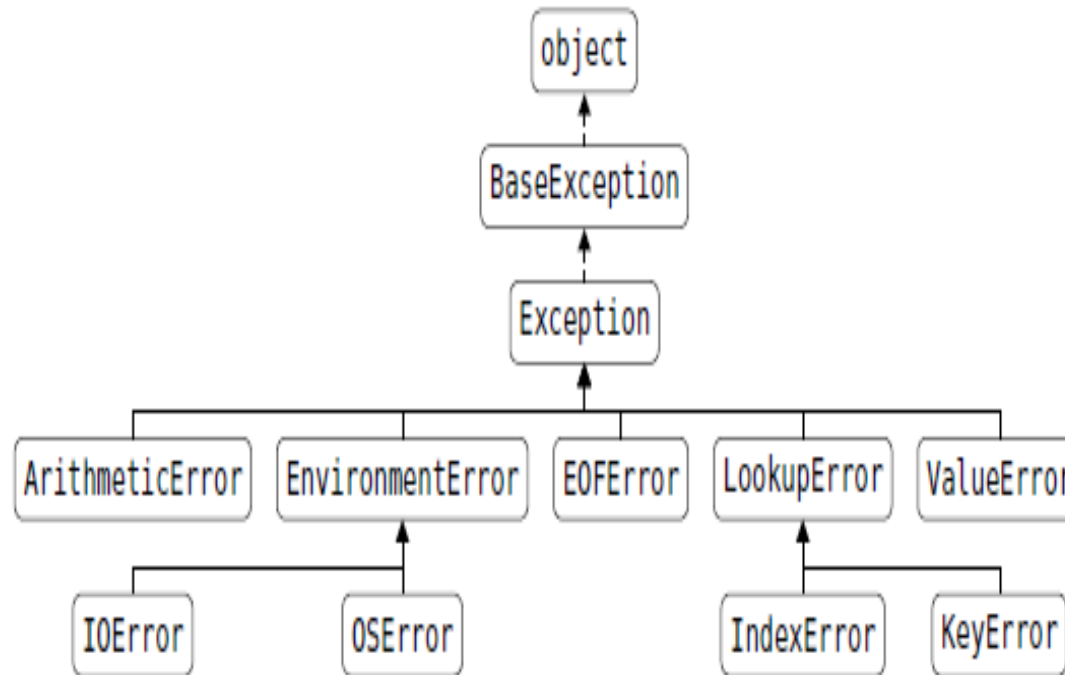
The except block

- The try block is generally followed by the except block which holds the exception cleanup code like some print statement to **print some message** or may be **trigger some event** or **store something in the database** etc.
- In the **except block**, along with the keyword except we can also provide the **name of exception class** which is expected to occur.
- In case there is no exception class name, it catches all the exceptions, otherwise it will only catch the exception of the type which is mentioned.
- Here is the **syntax**:

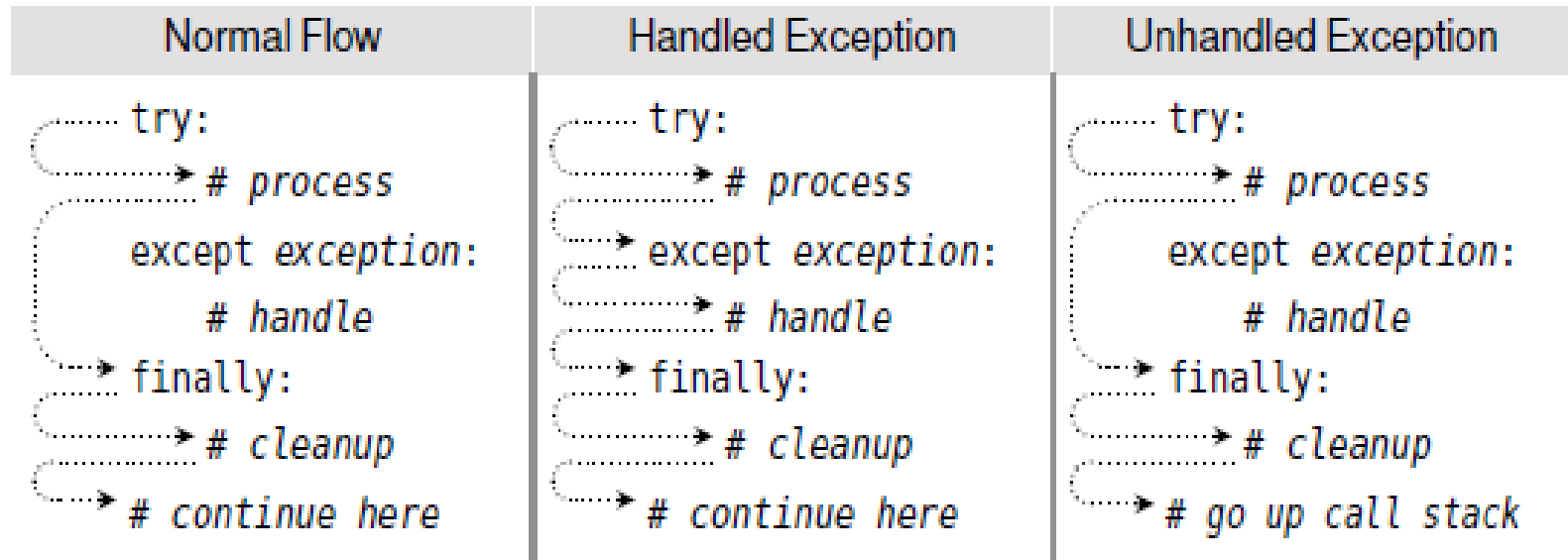
```
# except block
except(<Types of Exceptions to caught>):
    # except block starts
```

can even provide names of multiples exception classes separated by comma in the **except** statement.

Some of Python's exception hierarchy



The general try ...except ...finally block control flows



- Code execution is interrupted in the try block when an exception occurs, and the code statements inside the try block after the line which caused the exception are not executed.
 - The execution then jumps into the except block. And after the execution of the code statements inside the except block the code statements after it are executed, just like any other normal execution.

Examples

```
# try block
try:
    a = 10
    b = 0
    print("Result of Division: " + str(a/b))
    print("No! This line will not be executed.")
except:
    print("You have divided a number by zero,
which is not allowed.")

# outside the try-except blocks
print("Yo! This line will be executed.")
```

```
try:
    a = int(input("Enter numerator number: "))
    b = int(input("Enter denominator number: "))
    print("Result of Division: " + str(a/b))
# except block handling division by zero
except(ValueError, ZeroDivisionError):
    print("Please check the input value: It should
be an integer greater than 0")
```

```
# try block
try:
    a = int(input("Enter numerator number: "))
    b = int(input("Enter denominator number:
"))
    print("Result of Division: " + str(a/b))
# except block handling division by zero
except(ZeroDivisionError):
    print("You have divided a number by zero,
which is not allowed.")
# except block handling wrong value type
except(ValueError):
    print("You must enter integer value")
# generic except block
except:
    print("Oops! Something went wrong!")
```

finally block

- The finally block is always executed:
 - it is generally used for doing the concluding tasks like
 - closing file resources or
 - closing database connection or
 - may be ending the program execution with a delightful message

```
try:
    a = int(input("Enter numerator number: "))
    b = int(input("Enter denominator number: "))
    print("Result of Division: " + str(a/b))
# except block handling division by zero
except(ZeroDivisionError):
    print("You have divided a number by zero, which is
not allowed.")
finally:
    print("Code execution Wrap up!")

# outside the try-except block
print("Will this get printed?")
```

Output1:

```
Enter numerator number: 2
Enter denominator number: 2
Result of Division: 1.0
Code execution Wrap up!
Will this get printed?
```

Output2:

```
Enter numerator number: 2
Enter denominator number: 0
You have divided a number by zero, which is not allowed.
Code execution Wrap up!
Will this get printed?
```

Using Exception objects

- **Syntax:**

try:

this code is expected to throw exception **except**
 ExceptionType **as** ex:
code to handle exception

- **Example:**

try:

```
    number = eval(input("Enter a number: "))  
    print("The number entered is", number)  
except NameError as ex:  
    print("Exception:", ex)
```

Raising Exceptions

- While the try and except block are for handling exceptions, the raise keyword on the contrary is to **raise an exception**.
- To throw an exception if a condition occurs.
- There are three syntaxes for raising exceptions:
 - raise exception(args)*
 - raise exception(args) from original_exception*
 - raise*
 - When the first syntax is used the exception that is specified should be either one of the built-in exceptions, or a custom exception that is derived from Exception.
 - If we give the exception some text as its argument, this text will be output if the exception is printed when it is caught.
 - The second syntax is a variation of the exception is raised as a chained exception that includes the *original_exception* exception, so *this syntax* is used inside except suites.
 - When the third syntax is used, that is, when no exception is specified, raise will reraise the currently active exception—and if there isn't one it will raise a TypeError.

Raising Exceptions: Examples

```
a = 10
b = 0
try:
    print(a/b)
except ZeroDivisionError:
    print("Please enter valid integer value")
Output:
Please enter valid integer value
```

```
a = 10
b = 0
try:
    print(a/b)
except ZeroDivisionError:
    raise
Output:
ZeroDivisionError: division by zero
```

```
a = 10
b = 0
try:
    # condition for checking for negative values
    if a < 0 or b < 0:
        # raising exception using raise keyword
        raise ZeroDivisionError
    print(a/b)
except ZeroDivisionError:
    print("Please enter valid integer value")

Output:
Please enter valid integer value
```

Raising Exceptions: Examples

Example:

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")
```

Output:

Traceback (most recent call last):

```
File "D:\Python\python code\untitled1.py", line
10, in <module>
    raise Exception("Sorry, no numbers below zero")
```

Exception: Sorry, no numbers below zero

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

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

Output:

Traceback (most recent call last):

```
File "D:\Python\python code\untitled1.py",
line 12, in <module>
    raise TypeError("Only Integers are allowed")
```

TypeError: Only Integers are allowed

Functions

- Python provides many built-in functions, and the standard library and third-party libraries, so in many cases the function we want has already been written.
- It is always worth checking Python's " Online Documentation" to see what is already available.
- The general syntax for creating a (global or local) function is:
def *functionName(parameters)*:
***Suite* #Block of code or statements**
- The parameters are optional, and if there is more than one they are written as a sequence of comma-separated identifiers, or as a sequence of *identifier=value* pairs.
- Function block should always begin with the keyword `def`, followed by the function name and parentheses.
 - Any number of parameters or arguments can be passed inside the parentheses.
 - The block of a code of every function should begin with a colon (:)
 - An optional 'return' statement to return a value from the function.

Custom Functions

- Functions are a means by which we can package and parameterize functionality.
- Four kinds of functions can be created in Python:
 - global function
 - Local functions
 - lambda functions and
 - Methods
- **Global objects (including functions)** are accessible to any code in the same module (i.e., the same .py file) in which the object is created. Global objects can also be accessed from other modules.
- **Local functions** (also called nested functions) are functions that are defined inside other functions. These functions are visible only to the function where they are defined; they are especially useful for creating small helper functions that have no use elsewhere.
- **Lambda functions** are expressions, so they can be created at their point of use; however, they are much more limited than normal functions.
- **Methods** are functions that are associated with a particular data type and can be used only in conjunction with the data type

Function Arguments

- **In python, we can call a function using 4 types of arguments:**
 - Required argument
 - Keyworded argument
 - Default argument
 - Variable length arguments

Required Arguments

- Required arguments are the arguments which are passed to a function in a sequential order, the number of arguments defined in a function should match with the function definition.

- **Example:**

```
def addition(a, b):
```

```
    sum = a+b
```

```
    print("Sum of two numbers is:", sum)
```

```
addition(5, 6)
```

Output:

Sum of two numbers is: 11

Keyword Arguments

- When we use keyword arguments in a function call, the caller identifies the arguments by the argument name.

- **Example:**

```
def language(lname):  
    print("Current language is:", lname)
```

```
language(lname = "Python")
```

Output:

Current language is: Python

Default Arguments

- When a function is called without any arguments, then it uses the default argument.

- **Example:**

```
def country(cName = "India"):  
    print("Current country is:", cName)
```

```
country("New York")  
country("London")  
country()
```

Output:

```
Current country is: New York  
Current country is: London  
Current country is: India
```


Variable-length Arguments

- Sometimes, we do not know in advance the number of arguments that will be passed into a function. Python allows us to handle this kind of situation through function calls with an arbitrary number of arguments.
 - In the function definition, we use an asterisk (*) before the parameter name to denote this kind of argument.

- **Example 1:**

```
def greet(*names):  
    """This function greets all  
    the person in the names tuple."""  
  
    # names is a tuple with arguments  
    for name in names:  
        print("Hello", name)  
  
greet("ISE", "DSCE", "5th sem", "5B Section")
```

Output:

```
Hello ISE  
Hello DSCE  
Hello 5th sem  
Hello 5B Section
```

Variable-length Arguments : Examples (Argument and Parameter Unpacking)

```
#Non – Keyword argument
def add(*num):
    sum = 0
    for n in num:
        sum = n+sum
    print("Sum is:", sum)
```

```
add(2, 5)
add(5, 3, 5)
add(8, 78, 90)
```

Output:

```
Sum is: 7
Sum is: 13
Sum is: 176
```

Names and Docstrings

- Use a naming scheme, and use it consistently.
- Python documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes, and methods.
- It's specified in source code that is used, like a comment, to document a specific segment of code.
- Unlike conventional source code comments, the docstring should describe what the function does, not how.
- A string that comes immediately after the def line, and before the function's code begins.
- The docstrings are declared using `"""triple single quotes"""` or `"""triple double quotes"""`
- The docstrings can be accessed using the `__doc__` method of the object or using the help function.

Docstring Example

```
def my_function():  
    """Demonstrates triple double quotes  
    docstrings and does nothing really."""  
  
    return None  
  
print("Using __doc__:")  
print(my_function.__doc__)  
  
print("Using help:")  
help(my_function)
```

Output:

```
Using __doc__:  
Demonstrates triple double quotes  
    docstrings and does nothing really.  
Using help:  
Help on function my_function in module  
__main__:  
  
my_function()  
    Demonstrates triple double quotes  
    docstrings and does nothing really.
```

Docstring Example

One line docstrings fit in one line. They are used in obvious cases.

The closing quotes are on the same line as the opening quotes.

Example:

```
def power(a, b):  
    """Returns arg1 raised to power arg2."""  
  
    return a**b  
  
print(power.__doc__)
```

Output:

Returns arg1 raised to power arg2.

Docstring Example

- Multi-line docstrings consist of a summary line just like a one-line docstring, followed by a blank line, followed by a more elaborate description.
- The summary line may be on the same line as the opening quotes or on the next line.
- **Example:**

```
def my_function(arg1):  
    """
```

```
    Summary line.
```

```
  
    Extended description of function.
```

```
    Parameters:
```

```
    arg1 (int): Description of arg1
```

```
    Returns:
```

```
    int: Description of return value
```

```
    """
```

```
    return arg1
```

```
print(my_function.__doc__)
```

Output:

Summary line.

Extended description of function.

Parameters:

arg1 (int): Description of arg1

Returns:

int: Description of return value

Accessing Variables in the Global Scope

- A variable declared outside of the function or in global scope is known as a global variable.
- This means that a global variable can be accessed inside or outside of the function.
- A variable declared inside the function's body or in the local scope is known as a local variable.
- Example:

```
x = "global "
```

```
def foo():  
    global x  
    y = "local"  
    x = x * 2  
    print(x)  
    print(y)
```

```
foo()
```

Output:

```
global global  
local
```

Lambda Functions

- In Python, anonymous function means that a function is without a name.
- As we already know that *def* keyword is used to define the normal functions and the *lambda* keyword is used to create anonymous functions.
- It has the following syntax:
lambda arguments: expression
 - This function can have any number of arguments but only one expression, which is evaluated and returned.
 - One is free to use lambda functions wherever function objects are required.

Lambda Functions

- This example demonstrate the differences between a normal def defined function and lambda function.

- Example:

```
# Python code to illustrate cube of a number  
# showing difference between def() and lambda().
```

```
def cube(y):  
    return y*y*y;
```

```
g = lambda x: x*x*x  
print(g(5))
```

```
print(cube(5))
```

Output:
125
125

Lambda Functions

- Lambda functions can be used along with built-in functions like filter(), map() and reduce().
 - The filter() function in Python takes in a function and a list as arguments. This offers an elegant way to filter out all the elements of a sequence “sequence”, for which the function returns True.
 - The map() function in Python takes in a function and a list as argument. The function is called with a lambda function and a list and a new list is returned which contains all the lambda modified items returned by that function for each item.
 - The reduce() function in Python takes in a function and a list as argument. The function is called with a lambda function and a list and a new reduced result is returned. This performs a repetitive operation over the pairs of the list.

Lambda Functions : Examples

```
# Python code to illustrate
# filter() with lambda()
li = [5, 7, 22, 97, 54]
final_list = list(filter(lambda x: (x%2 != 0) , li))
print(final_list)
```

Output:

[5, 7, 97]

```
# Python code to illustrate
# map() with lambda()
# to get double of a list.
li = [5, 7, 22, 97, 54]
final_list = list(map(lambda x: x*2 , li))
print(final_list)
```

Output:

[10, 14, 44, 194, 108]

```
# Python code to illustrate
# reduce() with lambda()
# to get sum of a list
from functools import reduce
li = [5, 10, 20, 50, 100]
sum = reduce((lambda x, y: x + y), li)
print (sum)
```

Output:

185

Sort a list with a lambda expression

- Sorting a list with a lambda expression sorts the list with the lambda expression as the key for the sort comparison.

- **Example:**

```
data = [("Apples", 5, "20"), ("Pears", 1, "5"), ("Oranges", 6, "10")]
```

```
data.sort(key=lambda x:x[0])
```

```
print(data)
```

Output:

```
[('Apples', 5, '20'), ('Oranges', 6, '10'), ('Pears', 1, '5')]
```

```
data.sort(key=lambda x:x[1])
```

```
print(data)
```

Output:

```
[('Pears', 1, '5'), ('Apples', 5, '20'), ('Oranges', 6, '10')]
```

```
data.sort(key=lambda x: int(x[2]))
```

```
print(data)
```

Output:

```
[('Pears', 1, '5'), ('Oranges', 6, '10'), ('Apples', 5, '20')]
```

sorted method

- The sorted() method sorts the items of any iterable.
- You can optionally specify parameters for sort customization like sorting order and sorting criteria.

- **Syntax**

`sorted(iterable, key, reverse)`

The method has two optional arguments, which must be specified as keyword arguments.

Iterable - Required - Any iterable (list, tuple, dictionary, set etc.) to sort.

Key - Optional - A function to specify the sorting criteria. Default value is None.

Reverse – Optional - Setting it to True sorts the list in reverse order. Default value is False.

Sort with lambda

- Example:

```
# Sort by the age of students  
L = [('Sam', 35), ('Max', 25), ('Bob', 30)]  
x = sorted(L, key=lambda student: student[1]) print(x)
```

Output:

```
[('Max', 25), ('Bob', 30), ('Sam', 35)]
```

- Example

```
# with a list of strings, specifying key=len (the built-in len() function)  
#sorts the strings by length, from shortest to longest  
L = ['orange', 'red', 'green', 'blue']  
x = sorted(L, key=len)  
print(x)
```

Output:

```
['red', 'blue', 'green', 'orange']
```

sort vs sorted

- **sort()** can only be used for lists as it is a method of list class.
 - it modifies the values and returns none.
- **sorted()** does not change the original variable. The sorted() provides the list in ascending order without changing the original value.
 - When sorted() function is called, it returns the ordered list.
 - This means that you can assign a variable to the output of sorted().

- **Example:**

```
str = "to be or not to be".split()  
print("Str = ",str)
```

```
print("sorted(str) = ", sorted(str))  
print("Str = ",str)
```

```
print("str.sort()= ",str.sort())  
print("Str = ",str)
```

Output:

```
Str = ['to', 'be', 'or', 'not', 'to', 'be']
```

```
sorted(str) = ['be', 'be', 'not', 'or', 'to', 'to']  
Str = ['to', 'be', 'or', 'not', 'to', 'be']
```

```
str.sort()= None  
Str = ['be', 'be', 'not', 'or', 'to', 'to']
```

Modules and Packages

Modules and Packages

- When the Python code grows in size, most probably it becomes unorganized over time.
 - Keeping the code in the same file as it grows makes the code difficult to maintain.
- At this point, Python **modules** and **packages** help us to organize and group the contents by using files and folders.

Modules and Packages

- **Modules** are files with “.py” extension containing Python code. They help to organize related functions, classes or any code block in the same file.
- It is considered as a best practice to split the large Python code blocks into **modules** containing up to 300–400 lines of code.
- **Packages** group similar modules in a separate directory. They are folders containing related modules and an **__init__.py** file which is used for optional package-level initialisation.
- Depending on Python application, you can consider to group modules in sub-packages such as **doc, core, utils, data, examples, test**.

Modules and Packages

- Modules refer to a file containing Python statements and definitions. The name of the module will be the name of the file.
 - Example: A file containing Python code, `example.py`, is called a module, and its module name is `example`.
- Programs can be written in a single `.py` file, and so they are modules as well as programs.
 - The key difference is that programs are designed to be run, whereas modules are designed to be imported and used by programs.
- Modules are used to break down large programs into small manageable and organized files.
- Modules provide reusability of code.
 - Define the most used functions in a module and import it, instead of copying their definitions into different programs.

Modules and Packages

- There are actually three different ways to define a **module** in Python:
 - A module can be written in Python itself.
 - A module can be written in **C** and loaded dynamically at run-time, like the re (**regular expression**) module.
 - A **built-in** module is intrinsically contained in the interpreter, like the itertools module.
- A module's contents are accessed the same way in all three cases: with the import statement.

Modules and Packages

- Each file in Python is considered a module. Everything within the file is encapsulated within a namespace (which is the name of the file)
- To access module code in another module (file), import that file, and then access the functions or data of that module by prefixing with the name of the module, followed by a period
- To import a module:
 import sys
(note: no file suffix)
- Can import user-defined modules or some “standard” modules like sys and random
- Any python program needs one “top level” file which imports any other needed modules

Creating a module

- # Python Module example (example.py)

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

Importing Modules

- To use a module import that specific module into our code structure.

- Syntax

import <module_name>

- Module name here refers to the Python file name without “.py” extension.
- Once the module is imported, use the dot notation, “.”, to access the elements inside the module.

- To import our previously defined module example, type the following:

```
import example
```

```
#Using the module name we can now access the function using the dot . Operator
```

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

Output:

9.5

Python import statements

- **Python import statement**

- We can import a module using the import statement and access the definitions inside it using the dot operator .

```
# import statement example  
# to import standard module math  
import math  
print("The value of pi is", math.pi)
```

Output:

The value of pi is 3.141592653589793

- **Import with renaming**

- We can import a module by renaming it as follows:

```
# import module by renaming it  
import math as m  
print("The value of pi is", m.pi)
```

Output:

The value of pi is 3.141592653589793

Python import statements

- **Python from...import statement**

- We can import specific names from a module without importing the module as a whole. Here is an example.

```
# import only pi from math module  
from math import pi  
print("The value of pi is", pi)
```

Output:

The value of pi is 3.141592653589793

- **Import all names**

- We can import all names(definitions) from a module using the following construct:

```
# import all names from the standard module math  
from math import *  
print("The value of pi is", pi)
```

Output:

The value of pi is 3.141592653589793

- This includes all names visible in our scope except those beginning with an underscore(private definitions).
- Importing everything with the asterisk (*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier.

Python Module Search Path

- While importing a module, Python looks at several places. Interpreter first looks for a built-in module. Then(if built-in module not found), Python looks into a list of directories defined in sys.path.
- The search is in this order:
 - The current directory.
 - PYTHONPATH (an environment variable with a list of directories).
 - The installation-dependent default directory.

```
import sys  
print(sys.path)
```

Output:

```
['', 'C:\\Python34\\Lib\\idlelib', 'C:\\Windows\\SYSTEM32\\python34.zip',  
 'C:\\Python34\\DLLs', 'C:\\Python34\\lib', 'C:\\Python34', 'C:\\Python34\\lib\\site-  
packages']
```

Why packages?

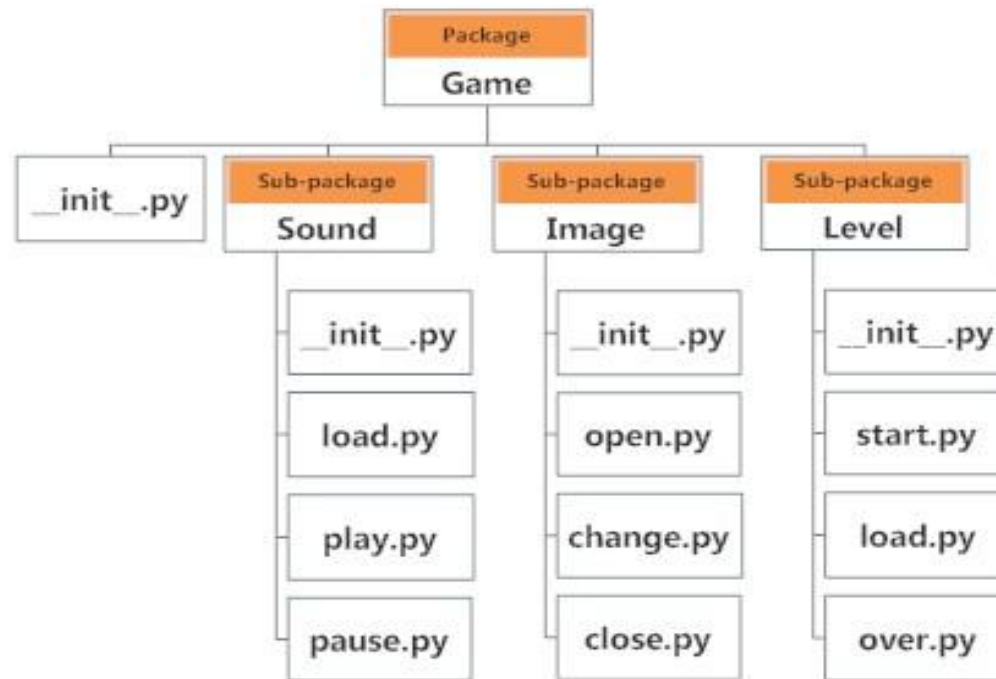
- A well-organized hierarchy of directories are used for easier access.
- Similar files are usually kept in the same directory, for example, keeping all the songs in the "**music**" directory. Analogous to this, Python has packages for directories and modules for files.
- As the application program grows larger in size with a lot of modules, place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear.

What are packages?

- A package is simply a directory that contains a set of modules and a file called `__init__.py`.
- As a directory can contain subdirectories and files, a Python package can also have sub-packages and modules.
- A directory must contain a file named `__init__.py` in order for Python to consider it as a package. This file can be left empty but generally initialization code can be placed for that package in this file.

Packages : Example

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



Packages

- Let's create a package named mypackage, using the following steps:
 - Create a new folder named D:\MyApp.
 - Inside MyApp, create a subfolder with the name 'mypackage'.
 - Create an empty `__init__.py` file in the mypackage folder.
 - Using a Python-aware editor like IDLE, create modules `greet.py` and `functions.py` with code.

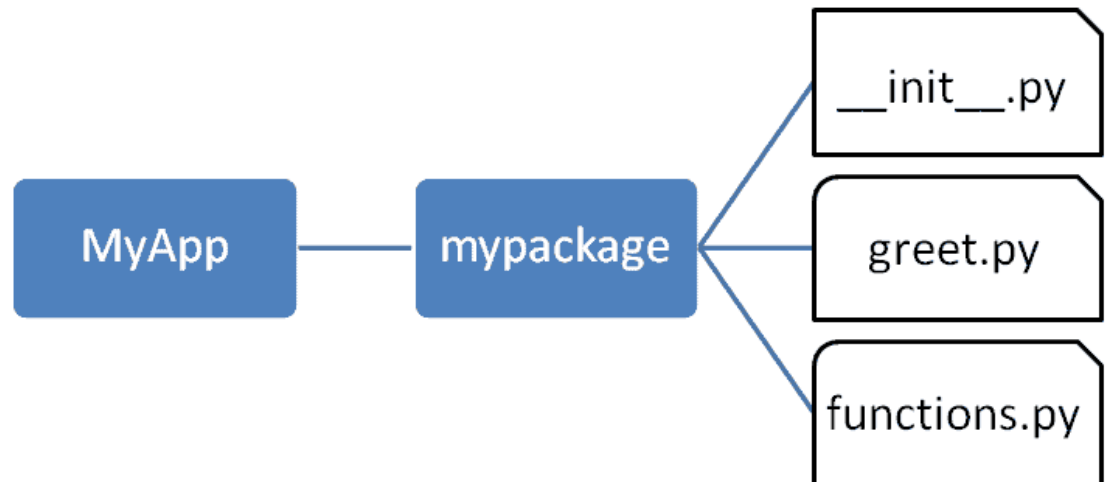
Packages

```
# greet.py
def SayHello(name):
    print("Hello " + name)
    return
```

```
# functions.py
def sum(x,y):
    return x+y

def average(x,y):
    return (x+y)/2

def power(x,y):
    return x**y
```



Package Folder Structure

Importing a Module from a Package

- Now, to test the package, invoke the Python prompt from the MyApp folder.
D:\MyApp>python
- Import the functions module from the mypackage package and call its power() function.

```
>>> from mypackage import functions  
>>> functions.power(3,2)  
9
```
- It is also possible to import specific functions from a module in the package

```
>>> from mypackage.functions import sum  
>>> sum(10,20)  
30  
  
>>> average(10,12)  
Traceback (most recent call last):  
File "<pyshell#13>", line 1, in <module>  
NameError: name 'average' is not defined
```


Packages

- `__init__.py`
 - The package folder contains a special file called `__init__.py`, which stores the package's content.
 - It serves two purposes:
 - The Python interpreter recognizes a folder as the package if it contains `__init__.py` file.
 - `__init__.py` exposes specified resources from its modules to be imported.
 - An empty `__init__.py` file makes all functions from above modules available when this package is imported.

Packages

- The `__init__.py` file is normally kept empty. However, it can also be used to choose specific functions from modules in the package folder and make them available for import. Modify `__init__.py` as below:

```
# __init__.py
from .functions import average, power
from .greet import SayHello
```

- The specified functions can now be imported in the interpreter session or another executable script.
- Create `test.py` in the MyApp folder to test mypackage.

```
# test.py
from mypackage import power, average, SayHello
SayHello()
x=power(3,2)
print("power(3,2) : ", x)
```

- Note that functions `power()` and `SayHello()` are imported from the package and not from their respective modules, as done earlier.
- The output of above script is:
D:\MyApp>python test.py
Hello world
power(3,2) : 9

Packages

- In some situations it is convenient to load in all of a package's modules using a single statement.
- To do this we must edit the package's `__init__.py` file to contain a statement which specifies which modules we want loaded. This statement must assign a list of module names to the special variable `__all__`.
- For example, here is the necessary line for the `__init__.py` file:
`__all__ = ["sum", "average", "power"]`
- Now we can write a different kind of import statement:

```
from functions import *  
x=power(3,2)  
print("power(3,2) : ", x)
```
- The *from package import ** syntax directly imports all the modules named in the `__all__` list.
- While importing packages, Python looks in the list of directories defined in `sys.path`, similar as for module search path.

Subpackages

- Packages can contain nested subpackages to arbitrary depth.



- The four modules (mod1.py, mod2.py, mod3.py, and mod4.py) are split out into two **sub package directories** : sub_pkg1 and sub_pkg2.
- Additional **dot notation** is used to separate the **package** name from the **subpackage** name

Packages : Path

- **Absolute Imports**

- Absolute imports include the entire path to the script, starting with the program's root folder. Separate each folder by a period.

- The following are examples of absolute imports:

```
from package1.firstmodule import firstmodule  
import package1.secondmodule.myfunction
```

- **Absolute Import Advantages and Disadvantages**

- Unlike other languages, most Python developers prefer using absolute imports over their relative cousins.
- This is because absolute imports make it really clear what you are trying to do.
- The actual location of the files is right there in the code. In fact, you can use them anywhere in your code. They will just work.
- If the project has sub packages in sub packages in sub packages, the import statements can expand beyond a single line of code. When that happens, its much better to use relative imports instead.

Packages : Path

- **Relative Imports**

- With relative imports, specify where the resources are relative to the current code file.
- As for the syntax, relative imports make use of *dot notation*. *Single dots* represent the directory of the current script. *Two dots* represent the parent folder. *Three dots* mean the grandparent, and so forth.
- The following are examples of relative imports:

```
import other  
from . import some_class  
from .subA import sa1
```

- **Relative Imports and their Advantages and Disadvantages**

- Relative imports rarely grow as long as absolute imports. They can even turn a ridiculously long absolute statement into something as simple as:

```
from ..my_sub_package.my_module import my_function
```
- However, they also hide the paths to modules. This might be okay if you are the only developer, but it gets messy if you are a part of a development team where the actual directory structure can change.

Which Import to Use?

- Unless you are working on a large project with several layers of sub-packages, you should always use absolute imports.
 - That way, the code will be easily understood by anyone that looks at it, including yourself if you got back to it to update it later.
- Even if there are no long paths, still try to write the program to only use absolute statements to simplify the code and its understandability .

Custom Modules

- A module is a file containing definition of functions, classes, variables, constants or any other Python object, even some executable code.
- Standard distribution of Python contains a large number of modules, generally known as built-in modules. These built-in modules may be written in C and integrated with Python interpreter, or a Python script (with .py extension).
- Since modules are just .py files , custom modules can be created without formality.
 - Just as in built-in modules, different Python objects in a custom module can be imported in interpreter session or another script.
- If the program has a large number of functions and classes, they are logically organized in separate modules. Such a modular design of the code is easy to understand, use and maintain.

The TextUtil Module

- The first module, TextUtil (in file TextUtil.py), contains just three functions:
 - `Is_balanced()` which returns True if the string it is passed has balanced parentheses of various kinds,
 - `shorten()` and
 - `simplify()`, a function that can strip spurious whitespace and other characters from a string.

```
import TextUtil
text = " ISE          DSCE  "
text = TextUtil.simplify(text) # text == ' ISE          DSCE  '
print(text)
```

Output:
ISE DSCE

The TextUtil Module

- Execute the examples in the docstrings and make sure that they produce the expected results. This can be done by adding just three lines at the end of the module's .py file:

```
if __name__ == "__main__":  
    import doctest  
    doctest.testmod()
```
- Whenever a module is imported Python creates a variable for the module called `__name__` and stores the module's name in this variable. A module's name is simply the name of its .py file but without the extension.
- Example, when the module is imported `__name__` will have the value "TextUtil", and the if condition will not be met, so the last two lines will not be executed.
- If we were to *run TextUtil.py* as though it were a program, Python will set `__name__` to "`__main__`" and the if condition will evaluate to True and the last two lines will be executed.
 - The `doctest.testmod()` function uses Python's introspection features to discover all the functions in the module and their docstrings, and attempts to execute all the docstring code snippets it finds. Running a module like this produces output only if there are errors.

Object-Oriented Programming

Object-oriented vs. Procedure-oriented Programming languages

Index	Object-oriented Programming	Procedural Programming
1.	Object-oriented programming is the problem-solving approach and used where computation is done by using objects.	Procedural programming uses a list of instructions to do computation step by step.
2.	It makes the development and maintenance easier.	In procedural programming, It is not easy to maintain the codes when the project becomes lengthy.
3.	It simulates the real world entity. So real-world problems can be easily solved through oops.	It doesn't simulate the real world. It works on step by step instructions divided into small parts called functions.
4.	It provides data hiding. So it is more secure than procedural languages. You cannot access private data from anywhere.	Procedural language doesn't provide any proper way for data binding, so it is less secure.
5.	Example of object-oriented programming languages is C++, Java, .Net, Python, C#, etc.	Example of procedural languages are: C, Fortran, Pascal, VB etc.

Encapsulation and Data Abstraction

- Encapsulation is also an essential aspect of object-oriented programming.
- It is used to restrict access to methods and variables.
- In encapsulation, code and data are wrapped together within a single unit.
- Abstraction is used to hide internal details and show only functionalities.
 - Abstracting something means to give names to things so that the name captures the core of what a function does.

Public, Protected and Private Data

- If an identifier doesn't start with an underscore character "_" it can be accessed from outside, i.e. the value can be read and changed
- Data can be protected by making members private or protected. Instance variable names starting with two underscore characters cannot be accessed from outside of the class.
- At least not directly, but they can be accessed through private name mangling.
- That means, private data `__A` can be accessed by the following name construct: ***instance_name._classname__A***

Public, Protected and Private Data

- If an identifier is only preceded by one underscore character, it is a protected member.
- Protected members can be accessed like public members from outside of class
- Example:

```
class Encapsulation(object):
```

```
    def __init__(self, a, b, c):
```

```
        self.public = a
```

```
        self._protected = b
```

```
        self.__private = c
```


Public, Protected and Private Data

- The following interactive sessions shows the behavior of public, protected and private members:

```
>>> x = Encapsulation(11,13,17)
>>> x.public
11
>>> x._protected
13
>>> x._protected = 23
>>> x._protected
23
>>> x.__private
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Encapsulation' object has no attribute '__private'
>>>
```

Public, Protected and Private Data

- The following table shows the different behavior Public, Protected and Private Data

Name	Notation	Behavior
name	Public	Can be accessed from inside and outside
_name	Protected	Like a public member, but they shouldn't be directly accessed from outside
__name	Private	Can't be seen and accessed from outside

Inheritance

- Inheritance is one of the most powerful concepts of OOPs.
- The mechanism by which a class which inherits the properties of another class is called Inheritance.
- The class which inherits the properties is called child class/subclass and the class from which properties are inherited is called parent class/base class.
- It provides the re-usability of the code.

Inheritance

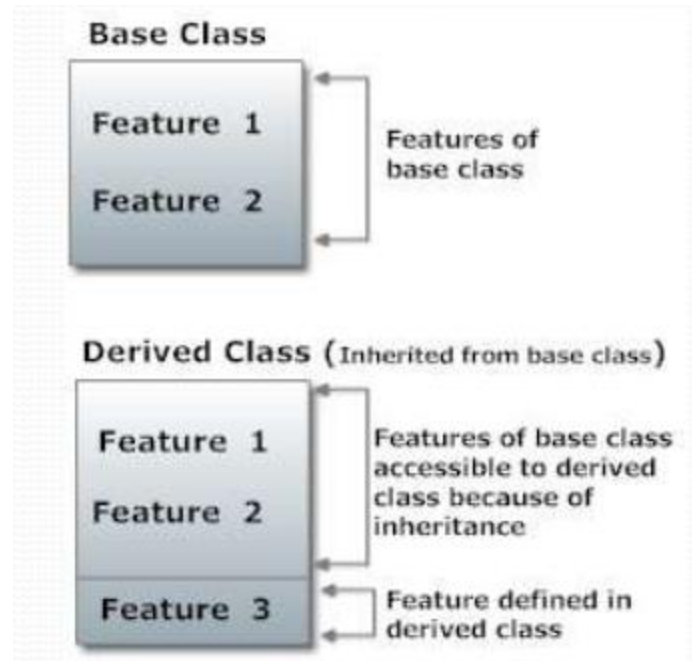
- Syntax:

```
class Baseclass(Object):
```

```
    body_of_base_class
```

```
class DerivedClass(BaseClass):
```

```
    body_of_derived_clas
```



Inheritance

- Two built-in functions *isinstance()* and *issubclass()* are used to check inheritances.
- *isinstance()* is used to check if an object is an instance of a certain class or any of its subclass.
 - Function *isinstance()* returns True if the object is an instance of the class or other classes derived from it.
- Each and every class in Python inherits from the base class object.
- Python *issubclass()* is built-in function used to check if a class is a subclass of another class or not.
 - This function returns True if the given class is the subclass of given class else it returns False.

Example

```
class Parent():  
    def first(self):  
        print('first function')  
  
class Child(Parent):  
    def second(self):  
        print('second function')  
  
ob = Child()  
ob.first()  
ob.second()
```

Output:

first function
second function

Sub-classing

- Calling a constructor of the parent class by mentioning the parent class name in the declaration of the child class is known as sub-classing.
 - A child class identifies its parent class by sub-classing.
- **`__init__()` Function**

The `__init__()` function is called every time a class is being used to make an object.

When we add the `__init__()` function in a parent class, the child class will no longer be able to inherit the parent class's `__init__()` function.

The child's class `__init__()` function overrides the parent class's `__init__()` function.

Example

```
class Parent:
```

```
    def __init__(self , fname, fage):
```

```
        self.firstname = fname
```

```
        self.age = fage
```

```
    def view(self):
```

```
        print(self.firstname , self.age)
```

Output:

course name Python first came 28 years ago. ISE
has courses to master python

```
class Child(Parent):
```

```
    def __init__(self , fname , fage):
```

```
        Parent.__init__(self, fname, fage)
```

```
        self.lastname = "ISE"
```

```
    def view(self):
```

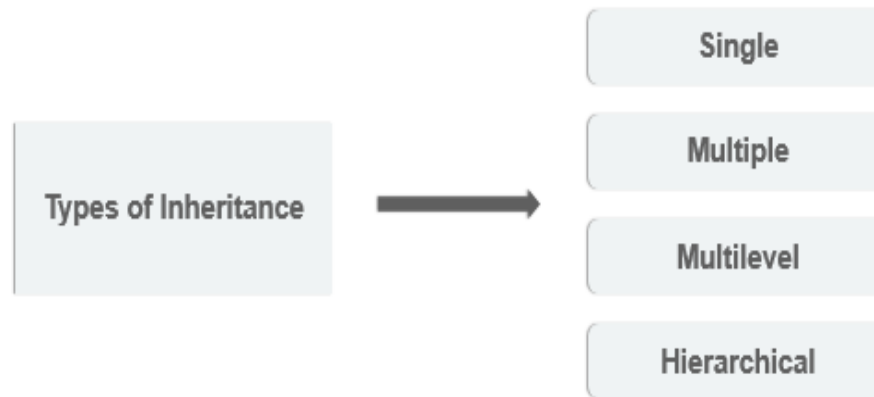
```
        print("course name" , self.firstname , "first came" , self.age , " years ago." ,  
              self.lastname , " has courses to master python")
```

```
ob = Child("Python" , '28')
```

```
ob.view()
```


Types Of Inheritance

- Depending upon the number of child and parent classes involved, there are four types of inheritance in python.



Single Inheritance

- a child class inherits only a single parent class.
- Example:

```
class Parent:
    def func1(self):
        print("this is function one")
class Child(Parent):
    def func2(self):
        print(" this is function 2 ")
ob = Child()
ob.func1()
ob.func2()
```

Output:

this is function one
this is function 2

Multiple Inheritance

- a child class inherits from more than one parent class.
- Example:

```
class Parent:
    def func1(self):
        print("this is function 1")
class Parent2:
    def func2(self):
        print("this is function 2")
class Child(Parent , Parent2):
    def func3(self):
        print("this is function 3")
```

```
ob = Child()
ob.func1()
ob.func2()
ob.func3()
```

Output:

```
this is function 1
this is function 2
this is function 3
```

Multilevel Inheritance

- a derived class inherits another derived class.
- Example:

```
class Parent:
    def func1(self):
        print("this is function 1")
class Child(Parent):
    def func2(self):
        print("this is function 2")
class Child2(Child):
    def func3(self):
        print("this is function 3")
ob = Child2()
ob.func1()
ob.func2()
ob.func3()
```

Output:

```
this is function 1
this is function 2
this is function 3
```

Hierarchical Inheritance

- Hierarchical inheritance involves multiple inheritance from the same base or parent class.
- Example:

```
class Parent:
    def func1(self):
        print("this is function 1")
class Child(Parent):
    def func2(self):
        print("this is function 2")
class Child2(Parent):
    def func3(self):
        print("this is function 3")
```

```
ob = Child()
ob1 = Child2()
ob.func1()
ob.func2()
ob1.func1()
ob1.func3()
```

Output:

```
this is function 1
this is function 2
this is function 1
this is function 3
```

Python Super() Function

- Super function allows us to call a method from the parent class.

- Example:

```
class Parent:
    def func1(self):
        print("this is function 1")
class Child(Parent):
    def func2(self):
        super().func1()
        print("this is function 2")
```

```
ob = Child()
ob.func2()
```

Output:

```
this is function 1
this is function 2
```

Polymorphism

- All of us have used GPS for navigating the route, Isn't it amazing how many different routes you come across for the same destination depending on the traffic, from a programming point of view this is called 'polymorphism'.
- It is one such OOP methodology where one task can be performed in several different ways.
- To put it in simple words, *it is a property of an object which allows it to take multiple forms.*
- Polymorphism defines the ability to take different forms.
- Polymorphism is of two types:
 - *Compile-time Polymorphism* : A compile-time polymorphism also called as static polymorphism gets resolved during the compilation time of the program. One common example is "method overloading".
 - *Run-time Polymorphism* : A run-time Polymorphism also, called as dynamic polymorphism gets resolved at run time. One common example of Run-time polymorphism is "method overriding".

Example: Compile-time Polymorphism

- Multiple methods with the same name but with a different type of parameter or a different number of parameters is called Method overloading

- Example:

```
class Person:
    def Hello(self, name=None):
        if name is not None:
            print('Hello ' + name)
        else:
            print('Hello ')

# Create instance
obj = Person()
# Call the method
obj.Hello()
# Call the method with a parameter
obj.Hello('ISE')
```

Output:

Hello
Hello ISE

Example : Compile-time Polymorphism

- Example:

```
class Compute:
    # area method
    def area(self, x = None, y = None):
        if x != None and y != None:
            return x * y
        elif x != None:
            return x * x
        else:
            return 0

# object
obj = Compute()
# zero argument
print("Area Value:", obj.area())
# one argument
print("Area Value:", obj.area(4))
# two argument
print("Area Value:", obj.area(3, 5))
```

Output:

Area Value: 0
Area Value: 16
Area Value: 15

Run-time Polymorphism :

Method Overriding

- In Python, to override a method points to remember:
 - One can't override a method within the same class. It means you have to do it in the child class using the **Inheritance** concept.
 - To override the Parent **Class** method, you have to create a method in the Child class with the same name and the same number of parameters.

Example: Run-time Polymorphism

```
class Employee:
```

```
    def message(self):
```

```
        print('This message is from Employee Class')
```

```
class Department(Employee):
```

```
    def message(self):
```

```
        print('This message is from Department class inherited from Employee')
```

```
emp = Employee()
```

```
emp.message()
```

```
print('-----')
```

```
dept = Department()
```

```
dept.message()
```

Output:

```
This message is from Employee Class
```

```
-----
```

```
This message is from Department class inherited from Employee
```

Example: Run-time Polymorphism

```
class Employee:
```

```
    def add(self, a, b):  
        print('The Sum of Two = ', a + b)
```

```
class Department(Employee):
```

```
    def add(self, a, b, c):  
        print('The Sum of Three = ', a + b + c)
```

```
emp = Employee()  
emp.add(10, 20)
```

```
print('-----')  
dept = Department()  
dept.add(50, 30, 10)
```

Output:

The Sum of Two = 30

The Sum of Three = 90

Example: Run-time Polymorphism

```
class Employee:
```

```
    def add(self, a, b):  
        print('The Sum of Two = ', a + b)
```

```
class Department(Employee):
```

```
    def add(self, a, b, c):  
        print('The Sum of Three = ', a + b + c)
```

```
emp = Employee()  
emp.add(10, 20)
```

```
print('-----')  
dept = Department()  
dept.add(50, 30, 10)
```

Output:

The Sum of Two = 30

The Sum of Three = 90

Copying an Object in Python

- use = operator to create a copy of an object.
- It only creates a new variable that shares the reference of the original object.
- **Example : Copy using = operator**

```
ld_list = [[1, 2, 3], [4, 5, 6], [7, 8, 'a']]
new_list = old_list
new_list[2][2] = 9
print('Old List:', old_list)
print('ID of Old List:', id(old_list))
print('New List:', new_list)
print('ID of New List:', id(new_list))
```

Output:

Old List: [[1, 2, 3], [4, 5, 6], [7, 8, 9]] ID of
Old List: 140673303268168

New List: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
ID of New List: 140673303268168

Copying an Object in Python :

Shallow Copy

- A shallow copy creates a new object which stores the reference of the original elements.
- So, a shallow copy doesn't create a copy of nested objects, instead it just copies the reference of nested objects. This means, a copy process does not recurse or create copies of nested objects itself.

- Syntax

```
import copy  
copy.copy(object_name)
```

- Example:

```
import copy  
old_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
new_list = copy.copy(old_list)  
print("Old list:", old_list)  
print("New list:", new_list)
```

Output:

Old list: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

New list: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

Shallow Copy

- **Adding [4, 4, 4] to old_list, using shallow copy**

```
import copy
old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.copy(old_list)
old_list.append([4, 4, 4])
print("Old list:", old_list)
print("New list:", new_list)
```

Output:

Old list: [[1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4]]
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]

- **Adding new nested object using Shallow copy**

```
import copy
old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.copy(old_list)
old_list[1][1] = 'AA'
print("Old list:", old_list)
print("New list:", new_list)
```

Output:

Old list: [[1, 1, 1], [2, 'AA', 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 'AA', 2], [3, 3, 3]]

Copying an Object in Python :

Deep Copy

- A deep copy creates a new object and recursively adds the copies of nested objects present in the original elements.
- The deep copy creates independent copy of original object and all its nested objects.

- Syntax:

```
import copy  
copy.deepcopy(object_name)
```

- Example:

```
import copy  
old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]  
new_list = copy.deepcopy(old_list)  
print("Old list:", old_list)  
print("New list:", new_list)
```

Output:

```
Old list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]  
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

Deep Copy

- **Adding a new nested object in the list using Deep copy**

```
import copy
old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.deepcopy(old_list)
old_list[1][0] = 'BB'
print("Old list:", old_list)
print("New list:", new_list)
```

Output:

```
Old list: [[1, 1, 1], ['BB', 2, 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

shallow and deep copy

```
import copy
```

```
a = [ [1, 2, 3], [4, 5, 6] ]
```

```
b = copy.copy(a)
```

```
c = [ [7, 8, 9], [10, 11, 12] ]
```

```
d = copy.deepcopy(c)
```

```
print("---- Shallow copy ----")
```

```
print(a)
```

```
print(b)
```

```
a[1][2] = 23
```

```
b[0][0] = 98
```

```
print(a)
```

```
print(b)
```

```
print("\n")
```

```
print("---- Deep copy ----")
```

```
print(c)
```

```
print(d)
```

```
c[1][2] = 23
```

```
d[0][0] = 98
```

```
print(c)
```

```
print(d)
```

Output:

```
---- Shallow copy ----
```

```
[[1, 2, 3], [4, 5, 6]]
```

```
[[1, 2, 3], [4, 5, 6]]
```

```
[[98, 2, 3], [4, 5, 23]]
```

```
[[98, 2, 3], [4, 5, 23]]
```

```
---- Deep copy ----
```

```
[[7, 8, 9], [10, 11, 12]]
```

```
[[7, 8, 9], [10, 11, 12]]
```

```
[[7, 8, 9], [10, 11, 23]]
```

```
[[98, 8, 9], [10, 11, 12]]
```

shallow and deep copy

- A shallow copy is one which makes a new object stores the reference of another object. While, in deep copy, a new object stores the copy of all references of another object making it another list separate from the original one.
- Thus, when you make a change to the deep copy of a list, the old list doesn't get affected and vice-versa. But shallow copying causes changes in both the new as well as in the old list.
- This copy method is applicable in compound objects such as a list containing another list.

Thank you