Introduction to Python

Topics

- Python Data Types
- Manipulating string, list, dictionary & tuple
- Program flow control
- Functions, modules & packages
- Object Oriented Programming
  - Creating class and objects
  - __init__
  - Inheritance
  - Polymorphism
- Exception Handling
- Regular Exception

Data Types

Python is a dynamically typed language; therefore, we do not need to specify the variable’s type while declaring it. Whatever value we assign to the variable based on that data type will be automatically assigned.

- Text Type: str

  Eg.
  ```
  x = “Hello World!”
  ```
- Numeric Type: int, float, complex

  Eg.
  ```
  x = 20
  x = 20.5
  x = 1j
  ```
- Sequence Types: list, tuple, range

  Eg.
  ```
  x = [“apple”,  “banana”]
  x = (“apple”,  “banana”)
  x = range(6)
  ```
- Mapping Type: dict

  Eg.
  ```
  x = {"name" : "John", "age" : 36}
  ```

Data Types

- Set Types: set, frozenset

  Eg. x = {"apple", "banana", "cherry"}

  x = frozenset({"apple", "banana", "cherry"})

- Boolean Type: bool

  Eg. x = True

- Binary Types: bytes, bytesarray, memoryview

  Eg. x = b"Hello”

  x = bytearray(5)

  x = memoryview(bytes(5))

- None Type: NoneType

  Eg. x = None

Data Types

Int Data Type

- You can also store integer values other than base 10 such as
  - Binary (base 2): Eg. octal_num = 0o20 # decimal int 16 with base 8
  - Octal (base 8): Eg. hexadecimal_num = 0x10 # decimal int 16 with base 16
  - Hexadecimal numbers (base 16): Eg. binary_num = 0b10000 # decimal int 16 with base 2

Float Data Type

- Floating-point values can be represented using the exponential form, also called scientific notation. The benefit of using the exponential form to represent floating-point values is we can represent large values using less memory. Eg. num1 = 1.22e4

Complex Data Type

- The complex type is generally used in scientific applications and electrical engineering applications. If we want to declare a complex value, then we can use the a+bj form.

Boolean Data Type

- You can evaluate any expression in Python, and get one of two answers, True or False.
  - Eg. print(10 > 9), print(10 == 9), print(10 < 9)
- Most Values are True
- Some Values are False

Python Program Flow Controls

- The flow control statements are divided into three categories
  1. Conditional statements
  2. Iterative statements.
  3. Transfer statements
- Conditional statements: There are three types of conditional statements.
  1. if statement
  2. if-else
  3. if-elif-else
- Iterative statements: Python provides us the following two loop statement to perform some actions repeatedly
  1. for loop
  2. while loop
- Transfer statements: There are three types of transfer statements.
  1. break statement
  2. continue statement
  3. pass statements

Python Functions

- Python support two types of functions
  1. Built-in function
     - range(), id(), type(), input(), eval()
  2. User-defined function: Functions which are created by programmer explicitly according to the requirement are called a user-defined function.

Creating a Function:

- Use the def keyword with the function name to define a function

  -
    ```
    def my_function():
      print("Hello from a function")
    ```

Creating a function with parameters

-
  ```
  def my_function(fname):
    print(fname + " Refsnes")
  ```

Docstrings

- In Python, the documentation string is also called a docstring. It is a descriptive text (like a comment) written by a programmer to let others know what block of code does.

  -
    ```
    def factorial(x):
     """This function returns the factorial of a given number."""
    return x # access doc string
    print(factorial.__doc__)
    ```

Python Functions

Global & Location Variable in function

- There are four types of arguments allowed.
  1. Positional arguments: Positional arguments are arguments that are pass to function in proper positional order.
     - `def add(a, b): print(a - b)`
  2. keyword arguments: A keyword argument is an argument value, passed to function preceded by the variable name and an equals sign.
     - `def message(name, surname):`
  3. Default arguments: Default arguments take the default value during the function call if we do not pass them.
     - `def message(name="Guest")`
  4. Variable-length arguments: We can declare a variable-length argument with the * (asterisk) symbol.
     - `def fun(*var)`

Recursive Function

- A recursive function is a function that calls itself, again and again.
  - `def factorial(no)`

- Anonymous Function
  - `lambda: argument_list:expression`

Python Modules

- Consider a module to be the same as a code library.
- A file containing a set of functions you want to include in your application.

Create a Module

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

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

Use a Module

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

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

Python Strings

- The string is immutable, i.e., it can not be changed once defined. You need to create a copy of it if you want to modify it. This non-changeable behavior is called immutability.
- Multiline Strings
  - Eg. a = """Lorem ipsum dolor sit amet,
    consectetur adipiscing elit”””
- Strings are array
  - Eg. a = "Hello, World!"
    print(a[1])
- Looping Through a String. Eg. for loop
- Length. Eg. len()
- Check string. Eg. in keyword

Python Strings: Methods

- Slicing
  - Eg. b = "Hello, World!"

    print(b[2:5])
  - From the Start. Eg. print(b[:5])
  - To the End. Eg. print(b[2:])
  - Negative Indexing Eg. print(b[-5:-2])
- Upper Case. Eg. print(a.upper())
- Lower Case. Eg. print(a.lower())
- Remove Whitespace Eg. print(a.strip())
- Replace String Eg. print(a.replace("H", "J"))
- Split String Eg. print(a.split(","))

Python Strings: Concatenation & Formating

String Concatenation

-
  ```
  a = "Hello"
  b = "World"
  c = a + b
  print(c)
  ```

- Format - Strings
  - The format() method takes the passed arguments, formats them, and places them in the string where the placeholders {} are:

    -
      ```
      age = 36
      txt = "My name is John, and I am {}"
      print(txt.format(age))
      ```

    -
      ```
      quantity = 3
      itemno = 567
      price = 49.95
      myorder = "I want {} pieces of item {} for {} dollars."
      print(myorder.format(quantity, itemno, price))
      ```

    -
      ```
      quantity = 3
      itemno = 567
      price = 49.95
      myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
      print(myorder.format(quantity, itemno, price))
      ```

Python Lists

- List items are ordered, changeable, and allow duplicate values.
- List items are indexed, the first item has index [0], the second item has index [1] etc.

  -
    ```
    thislist = ["apple", "banana", "cherry", "apple", "cherry"]
    print(thislist)
    ```
- List Length Eg, len()
- List items can be of any data type:

  -
    ```
    list1 = ["abc", 34, True, 40, "male"]
    ```
- It is also possible to use the list() constructor when creating a new list.

  -
    ```
    thislist = list(("apple", "banana", "cherry"))
    print(thislist)
    ```

Python List

Access Items

- List items are indexed and you can access them by referring to the index number:
    - print(thislist[1])

Negative Indexing

- Negative indexing means start from the end
- -1 refers to the last item, -2 refers to the second last item etc.
    - print(thislist[-1])

Range of Indexes

- When specifying a range, the return value will be a new list with the specified items.
    - print(thislist[2:5])

Check if Item Exists

- To determine if a specified item is present in a list use the in keyword:
    - if "apple" in thislist

Python List

Change Item Value

- To change the value of a specific item, refer to the index number:

  -
    ```
    thislist = ["apple", "banana", "cherry"]
    thislist[1] = "blackcurrant"
    print(thislist)Negative Indexing
    ```

  -
    ```
    thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
    thislist[1:3] = ["blackcurrant", "watermelon"]
    print(thislist)
    ```

Insert Items

- To insert a new list item, without replacing any of the existing values, we can use the insert() method.

  -
    ```
    thislist.insert(2, "watermelon")
    print(thislist)
    ```

Append Items

- To add an item to the end of the list, use the append() method:

  -
    ```
    thislist.append("orange")
    print(thislist)
    ```

Extend List

- To append elements from another list to the current list, use the extend() method.

  -
    ```
    thislist = ["apple", "banana", "cherry"]
    tropical = ["mango", "pineapple", "papaya"]
    thislist.extend(tropical)
    print(thislist)
    ```

Python List

Remove Specified Item

- The remove() method removes the specified item
  - thislist.remove("banana")

Remove Specified Index

- The pop() method removes the specified index.
  - thislist.pop(1)
  - del thislist[0]

Clear the List

- The clear() method empties the list.
  - thislist.clear()

Loop Through a List

- You can loop through the list items by using a loop:
  - for x in thislist
  - while i < len(thislist)

Loop Through the Index Numbers

- You can also loop through the list items by referring to their index number.
- Use the range() and len() functions to create a suitable iterable
  - for i in range(len(thislist))

Python List

Sort List Alphanumerically

- List objects have a sort() method that will sort the list alphanumerically, ascending, by default:
  - thislist.sort()

Sort Descending

- To sort descending, use the keyword argument reverse = True:
  - thislist.sort(reverse = True)

Reverse Order

- The reverse() method reverses the current sorting order of the elements.
  - thislist.reverse()

Copy a List

- There are ways to make a copy, one way is to use the built-in List method copy()
  - mylist = thislist.copy()

Join Two Lists

- There are several ways to join, or concatenate, two or more lists in Python.
- One of the easiest ways are by using the + operator.
  - list3 = list1 + list2
  - list1.append(list2)
  - list1.extend(list2)

Python List Comprehension

- List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.
- Syntax:
  - newlist = [*expression* for *item* in *iterable* if *condition* == True]
- Eg.

  -
    ```
    fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
    newlist = []
    for x in fruits:
      if "a" in x:
        newlist.append(x)
    ```

  -
    ```
    fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
    newlist = [x for x in fruits if "a" in x]
    print(newlist)
    ```

Python Tuples

- Tuple items are ordered, unchangeable, and allow duplicate values.
- Tuple items are indexed, the first item has index [0], the second item has index [1] etc.
  - thistuple = ("apple", "banana", "cherry", "apple", "cherry")
- Tuple Length. Eg. len()
- Tuple items can be of any data type:
  - tuple1 = ("abc", 34, True, 40, "male")
- It is also possible to use the tuple() constructor when creating a new list.
  - thistuple = tuple(("apple", "banana", "cherry"))

Python Tuple

Access Items

- List items are indexed and you can access them by referring to the index number:
  - `print(thistuple[1])`

Negative Indexing

- Negative indexing means start from the end
- -1 refers to the last item, -2 refers to the second last item etc.
  - `print(thistuple[-1])`

Range of Indexes

- When specifying a range, the return value will be a new list with the specified items.
  - `print(thistuple[2:5])`

Check if Item Exists

- To determine if a specified item is present in a list use the in keyword:
  - `if "apple" in thistuple`

Python Tuple

Change Item Value

- Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.
- But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple:

  -
    ```
    x = ("apple", "banana", "cherry")
    y = list(x)
    y[1] = "kiwi"
    x = tuple(y)
    ```

Add & remove Items

- You can convert the tuple into a list

  -
    ```
    thistuple = ("apple", "banana", "cherry")
    y = list(thistuple)
    y.append("orange")
    thistuple = tuple(y)
    ```

  -
    ```
    thistuple = ("apple", "banana", "cherry")
    y = list(thistuple)
    y.remove("apple")
    thistuple = tuple(y)
    ```

Unpacking a Tuple

- When we create a tuple, we normally assign values to it. This is called "packing" a tuple:

  -
    ```
    fruits = ("apple", "banana", "cherry")
    (green, yellow, red) = fruits
    ```
- If the number of variables is less than the number of values, you can add an * to the variable name and the values will be assigned to the variable as a list

  -
    ```
    fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")
    (green, yellow, *red) = fruits
    ```

Python Tuple

Loop Through a Tuple

- You can loop through the tuple items by using loop
  - for x in thistuple
  - for i in range(len(thistuple))
  - while i < len(thistuple)

Join Two Tuples

- To join two or more tuples you can use the + operator:
  - tuple3 = tuple1 + tuple2

Multiply Tuples

- If you want to multiply the content of a tuple a given number of times, you can use the * operator:
  - mytuple = fruits * 2

Python Dictionaries

- Dictionary items are ordered, changeable, and does not allow duplicates.
- Dictionary items are presented in key:value pairs, and can be referred to by using the key name.
- Dictionaries cannot have two items with the same key
- Dictionary Length: len() function
- The values in dictionary items can be of any data type:

  -
    ```
    thisdict = {
      "brand": "Ford",
      "electric": False,
      "year": 1964,
      "colors": ["red", "white", "blue"]
    }
    ```
- It is also possible to use the dict() constructor to make a dictionary.
  - `thisdict = dict(name = "John", age = 36, country = "Norway")`

Python Dictionary

Accessing Items

- You can access the items of a dictionary by referring to its key name, inside square brackets:
  - thisdict["model"]
  - thisdict.get("model")
- The keys() method will return a list of all the keys in the dictionary.
  - thisdict.keys()
- The values() method will return a list of all the values in the dictionary.
  - thisdict.values()
- The items() method will return each item in a dictionary, as tuples in a list.
  - thisdict.items()

Check if Key Exists

- To determine if a specified key is present in a dictionary use the in keyword:
  - if "model" in thisdict:

Python Dictionary

Change Values

- You can change the value of a specific item by referring to its key name:
  - thisdict["year"] = 2018

Update Dictionary

- The update() method will update the dictionary with the items from the given argument.
  - thisdict.update({"year": 2020})

Removing Items

- There are several methods to remove items from a dictionary:
  - thisdict.pop("model")
  - thisdict.popitem()
  - del thisdict["model"]

Clear Dictionary

- The clear() method empties the dictionary:
  - thisdict.clear()

Python Dictionary

Loop Through a Dictionary

- When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well.
  - for x in thisdict
  - for x in thisdict.values()
  - for x in thisdict.keys()
  - for x, y in thisdict.items()

Copy a Dictionary

- There are ways to make a copy, one way is to use the built-in Dictionary method copy().
  - mydict = thisdict.copy()
  - mydict = dict(thisdict)

Nested Dictionaries

- A dictionary can contain dictionaries, this is called nested dictionaries.

  -
    ```
    myfamily = {
      "child1" : {
        "name" : "Emil",
        "year" : 2004
      }
    }
    ```

Python Operators

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Python Arithmetic Operators

- Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Python Assignment Operators

- Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Python Comparison Operators

- Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Python Logical Operators

- Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Python Identity Operators

- Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Python Membership Operators

- Membership operators are used to test if a sequence is presented in an object:

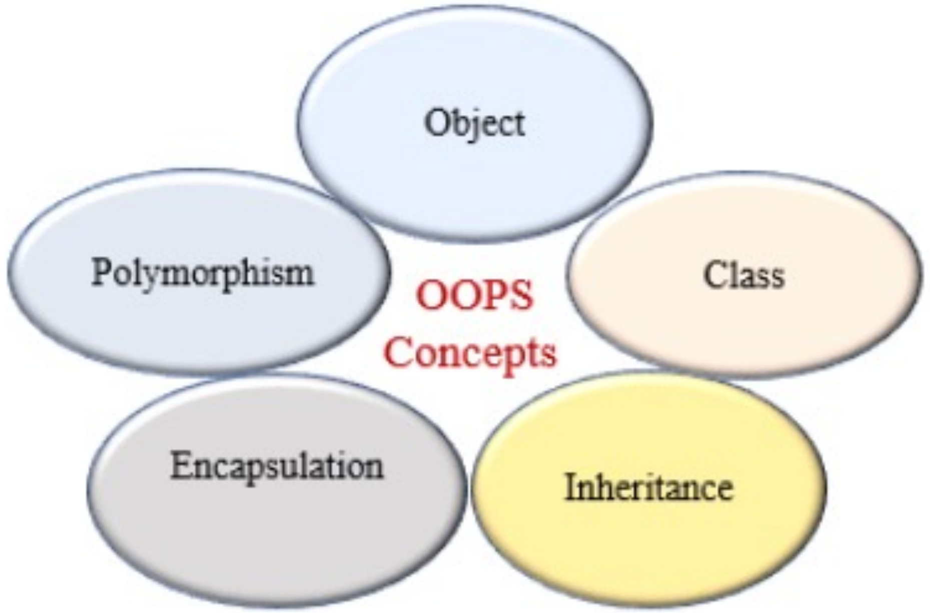
Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Python Bitwise Operators

- Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description	Example
&	AND	Sets each bit to 1 if both bits are 1	x & y
|	OR	Sets each bit to 1 if one of two bits is 1	x | y
^	XOR	Sets each bit to 1 if only one of two bits is 1	x ^ y
~	NOT	Inverts all the bits	~x
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off	x << 2
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	x >> 2

OOP: Object Oriented Programming



- **Object-oriented programming** (OOP) is a programming paradigm based on the concept of "**objects**”
- The object contains both data and code: Data in the form of properties (often known as attributes), and code, n the form of methods (actions object can perform).
- An object has the following two characteristics:
  - Attribute
  - Behavior

Class and Objects

- In Python, everything is an object. A **class is a blueprint for the object**. To create an object we require a model or plan or blueprint which is nothing but class.
- **Object is an instance of a class**. The physical existence of a class is nothing but an object. In other words, the object is an entity that has a state and behavior. It may be any real-world object like the mouse, keyboard, laptop, etc.
- **Creating Class and Objects**
  - In Python, Use the keyword class to define a Class. In the class definition, the first string is docstring which, is a brief description of the class.
  - The docstring is not mandatory but recommended to use. We can get docstring using __doc__ attribute. Use the following syntax to create a class.
  - Syntax.

```
class classname:
        '''documentation string'’
        class_suite  #contains class attributes and methods
```

- We can create any number of objects of a class. use the following syntax to create an object of a class.
  - eg. reference_variable = classname()

- **Constructors in Python**
  - In Python, a constructor is a special type of method used to initialize the object of a Class. The constructor will be executed automatically when the object is created. If we create three objects, the constructor is called three times and initialize each object.
  - The main purpose of the constructor is to declare and initialize instance variables. It can take at least one argument that is self. The __init()__ method is called the constructor in Python. In other words, the name of the constructor should be __init__(self).
  - eg.

```
class Employee:
    def __init__(self, name, salary):
        # public member
        self.name = name
        # private member
        # not accessible outside of a class
        self.__salary = salary

    def show(self):
        print("Name is ", self.name, "and salary is", self.__salary)

emp = Employee("Jessa", 40000)
emp.show()

# access salary from outside of a class
print(emp.__salary)
```

- Object Methods
  - Objects can also contain methods. Methods in objects are functions that belong to the object.
  - **eg.**

    -
      ```
      class Person:
        def __init__(self, name, age):
          self.name = name
          self.age = age

        def myfunc(self):
          print("Hello my name is " + self.name)

      p1 = Person("John", 36)
      p1.myfunc()
      ```

Inheritance

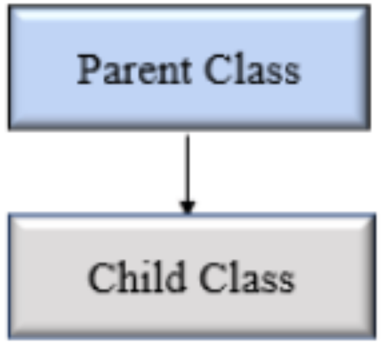
- In an Object-oriented programming language, inheritance is an important aspect. In Python, inheritance is the process of inheriting the properties of the parent class into a child class.
- The primary purpose of inheritance is the reusability of code. Using inheritance, we can use the existing class to create a new class instead of recreating it from scratch.
- Syntax:

```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```

- inheritance.py

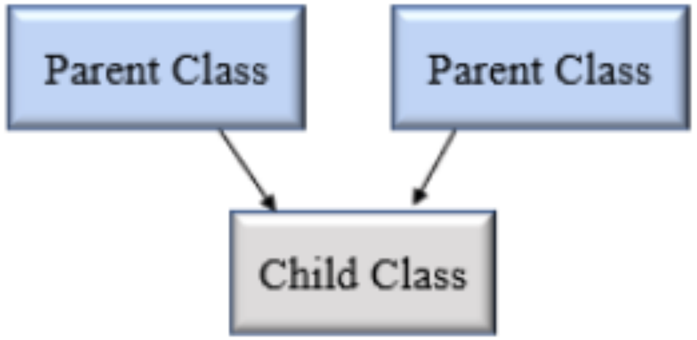
Types Of Inheritance

- In Python, based upon the number of child and parent classes involved, there are five types of inheritance. The type of inheritance are listed below:
  1. Single inheritance
  2. Multiple Inheritance
  3. Multilevel inheritance
  4. Hierarchical Inheritance
  5. Hybrid Inheritance
- In **single inheritance**, a child class inherits from a single-parent class. Here is one child class and one parent class.
  - eg. single_inheritance.py

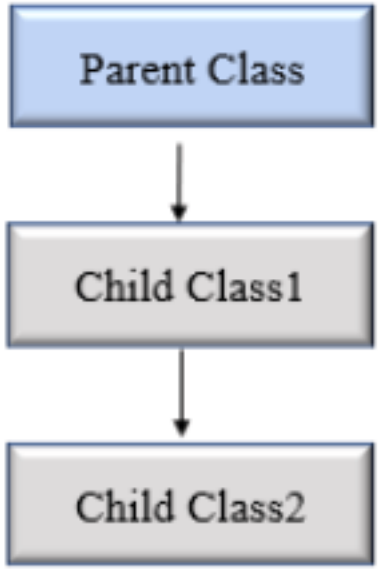


Types Of Inheritance

- In **multiple inheritance**, one child class can inherit from multiple parent classes. So here is one child class and multiple parent classes.
  - eg. multiple_inheritance.py



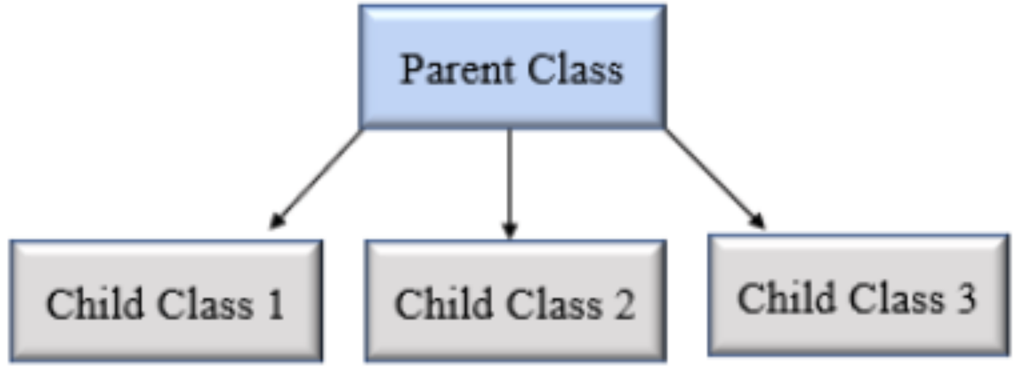
- In **multilevel inheritance**, a class inherits from a child class or derived class. Suppose three classes A, B, C. A is the superclass, B is the child class of A, C is the child class of B. In other words, we can say a **chain of classes** is **called multilevel inheritance.**
  - eg. multilevel_inheritance.py

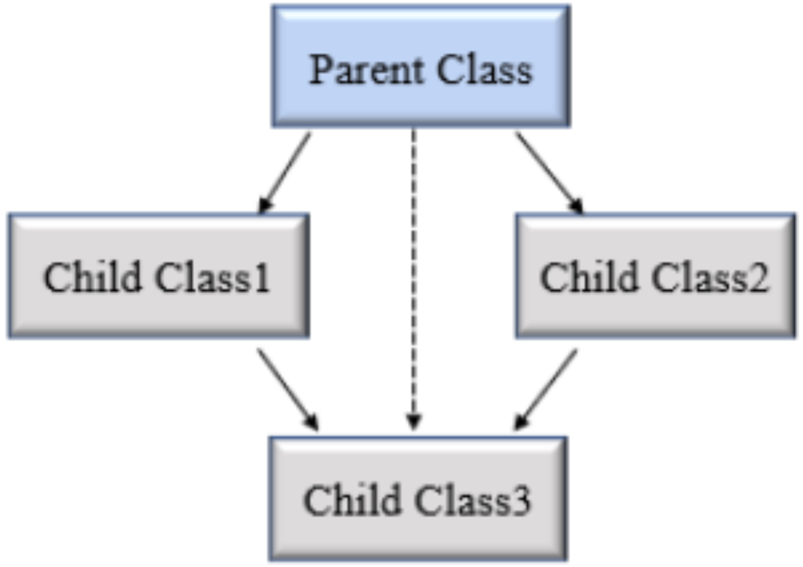


Types Of Inheritance

- In **Hierarchical inheritance**, more than one child class is derived from a single parent class. In other words, we can say one parent class and multiple child classes.
  - eg. hierarchical_inheritance.py

- When inheritance is consists of multiple types or a combination of different inheritance is called **hybrid inheritance.**
  - eg. hybrid_inheritance.py





Python **super()** function:

- In child class, we can refer to parent class by using the super() function. The super function returns a temporary object of the parent class that allows us to call a parent class method inside a child class method.

- Benefits of using the super() function.

1. We are not required to remember or specify the parent class name to access its methods.
2. We can use the super() function in both single and multiple inheritances.
3. The super() function support code reusability as there is no need to write the entire function

  - eg. super_function.py

issubclass():

- In Python, we can verify whether a particular class is a subclass of another class. For this purpose, we can use Python built-in function **issubclass()**.
- This function returns True if the given class is the subclass of the specified class. Otherwise, it returns False.
- Syntax: issubclass(class, classinfo)
  - classinfo: a class, type, or a tuple of classes or data types.

  - eg. is_subclass.py

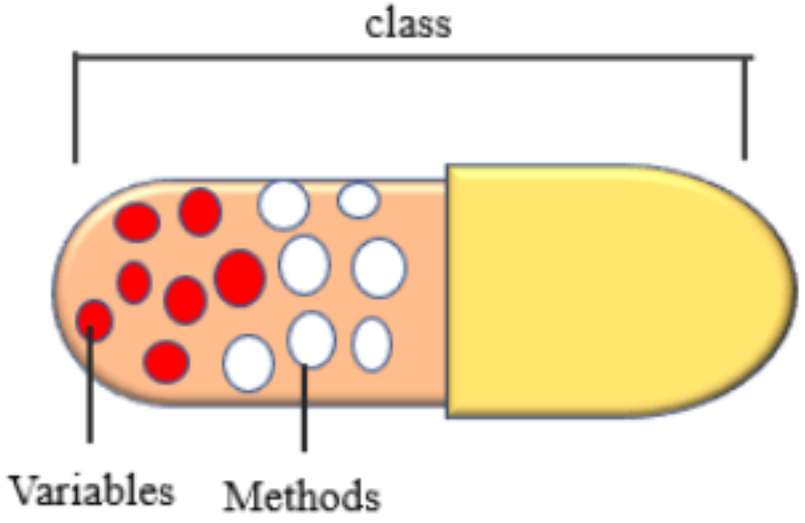
Method Overriding

- In inheritance, all members available in the parent class are by default available in the child class. If the child class does not satisfy with parent class implementation, then the child class is allowed to redefine that method by extending additional functions in the child class. This concept is called **method overriding**.
- When a child class method has the same name, same parameters, and same return type as a method in its superclass, then the method in the child is said to **override** the method in the parent class.

  - eg.
    - method_overriding.py
    - mro.py

Encapsulation in Python

- In Python, encapsulation is a method of wrapping data and functions into a single entity. For example, A class encapsulates all the data (methods and variables). Encapsulation means the internal representation of an object is generally hidden from outside of the object's definition.
  - eg. encapsulation.py



Access Modifiers in Python

- Encapsulation can be achieved by declaring the data members and methods of a class either as private or protected. But In Python, we don’t have direct access modifiers like public, private, and protected.
- We can achieve this by using single **underscore** and **double underscores**.
  - **Public Member**: Accessible anywhere from otside oclass.
  - **Private Member**: Accessible within the class
  - **Protected Member**: Accessible within the class and its sub-classes

Polymorphism

- Polymorphism in OOP is the **ability of an object to take many forms**. In simple words, polymorphism allows us to perform the same action in many different ways.
- Polymorphism is taken from the Greek words Poly (many) and morphism (forms). Polymorphism defines the ability to take different forms.
  - eg.

1. polymorphism.py
2. class_polymorphism.py
3. inheritance_class_polimorphism.py

Exception Handling

What are Exceptions?

- An exception is an **event that occurs during the execution of programs that disrupt the normal flow of execution** (e.g., KeyError Raised when a key is not found in a dictionary.) An exception is a Python object that represents an error..
- For example, bellow are the few standard exceptions
  - FileNotFoundException
  - ImportError
  - RuntimeError
  - NameError
  - TypeError

Why use Exception

- **Standardized error handling**: Using built-in exceptions or creating a custom exception with a more precise name and description, you can adequately define the error event, which helps you debug the error event.
- **Cleaner code**: Exceptions separate the error-handling code from regular code, which helps us to maintain large code easily.
- **Robust application**: With the help of exceptions, we can develop a solid application, which can handle error event efficiently
- **Exceptions propagation**: By default, the exception propagates the call stack if you don’t catch it. For example, if any error event occurred in a nested function, you do not have to explicitly catch-and-forward it; automatically, it gets forwarded to the calling function where you can handle it.
- **Different error types**: Either you can use built-in exception or create your custom exception and group them by their generalized parent class, or Differentiate errors by their actual class

Exception Handling

What are Errors?

- On the other hand, An **error** is an action that is incorrect or inaccurate. For example, syntax error. Due to which the program fails to execute.
- The errors can be broadly classified into two types:
    1. Syntax errors
    2. Logical errors

Syntax error

- The syntax error occurs when we are not following the proper structure or syntax of the language. A syntax error is also known as a **parsing error**.
- **Common Python Syntax errors:**
    - Incorrect indentation
    - Missing colon, comma, or brackets
    - Putting keywords in the wrong place.

Logical errors (Exception)

- Even if a statement or expression is syntactically correct, the error that occurs at the runtime is known as a **Logical error or Exception**. In other words, **Errors detected during execution are called exceptions**.
- **Common Python Logical errors:**
    - Indenting a block to the wrong level
    - using the wrong variable name
    - making a mistake in a boolean expression

Built-in Exceptions

- Python automatically generates many exceptions and errors. Runtime exceptions, generally a result of programming errors, such as:
  - Reading a file that is not present
  - Trying to read data outside the available index of a list
  - Dividing an integer value by zero

  eg. file_not_found.py

The try and except Block to Handling Exceptions

- When an exception occurs, Python stops the program execution and generates an exception message. It is highly recommended to handle exceptions. The doubtful code that may raise an exception is called risky code.
- To handle exceptions we need to use try and except block. Define risky code that can raise an exception inside the try block and corresponding handling code inside the except block.
- Syntax:

```
try :
   # statements in try block
except :
   # executed when exception occured in try block
```

eg. zero_division_error.py

Exception Handling

Catching Specific Exceptions

- We can also catch a specific exception. In the above example, we did not mention any specific exception in the except block. Catch all the exceptions and handle every exception is not good programming practice.
- It is good practice to **specify an exact exception that the except clause should catch**.
  - eg. specific_error.py

Handle multiple exceptions with a single except clause

- We can also handle multiple exceptions with a single except clause. For that, we can use an tuple of values to specify multiple exceptions in an except clause.
  - eg. specific_error.py

Exception Handling

Using try with finally

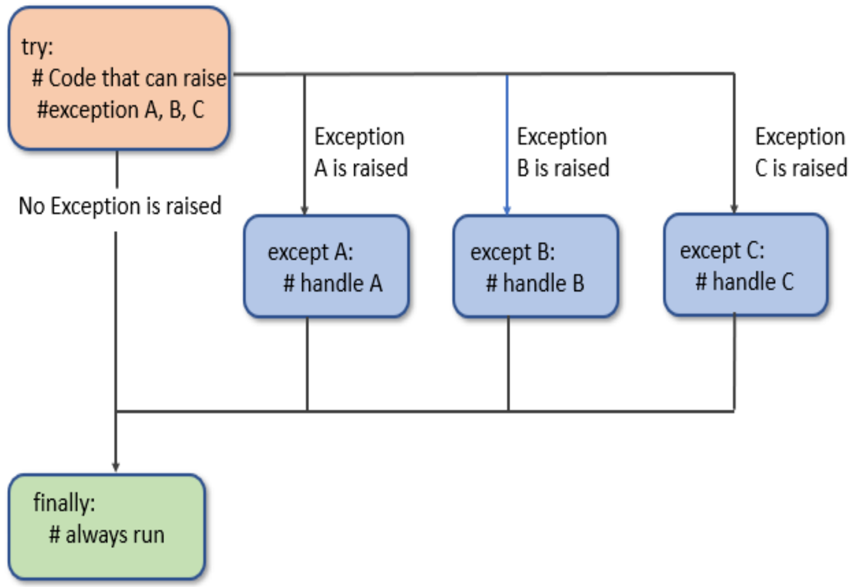
- Python provides the finally block, which is used with the try block statement. **The finally block is used to write a block of code that must execute, whether the try block raises an error or not**.

Clean-up actions using finally

- Sometimes we want to execute some action at any cost, even if an error occurred in a program. In Python, we can perform such actions using a finally statement with a try and except statement.
- **Syntax**

```
try:
    # block of code
    # this may throw an exception
finally:
    # block of code
    # this will always be executed
    # after the try and any except block
```

eg. try_with_finally.py



try-except-finally

Exception Handling

Using try with else clause

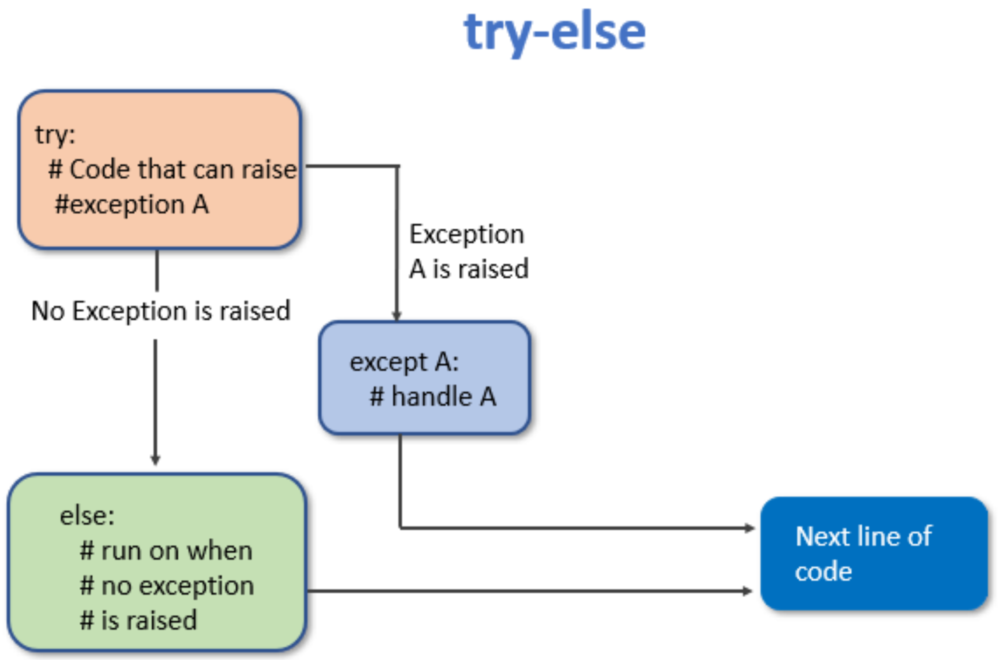
- Sometimes we might want to run a specific block of code. In that case, we can use else block with the try-except block. The else block will be executed if and only if there are no exception is the try block. For these cases, we can use the optional else statement with the try statement.

Why to use else block with try?

- Use else statemen with try block to check if try block executed without any exception or if you want to run a specific code only if an exception is not raised
- **Syntax**

```
try:
    # block of code
except Exception1:
    # block of code
else:
    # this code executes when exceptions not occured
```

eg. try_with_else.py



Exception Handling

Raising an Exceptions

- In Python, the raise statement allows us to throw an exception. The single arguments in the raise statement show an exception to be raised. This can be either an exception object or an Exception class that is derived from the Exception class.
- Follow the below steps to raise an exception:
    - Create an exception of the appropriate type. Use the existing built-in exceptions or create your won exception as per the requirement.
    - Pass the appropriate data while raising an exception.
    - Execute a raise statement, by providing the exception class.
- Syntax: raise Exception_class,<value>
    - eg. exception_chain.py

Exception Chaining

- The exception chaining is available only in Python 3. The raise statements allow us as optional from statement, which enables chaining exceptions. So we can implement exception chaining in python3 by using raise…from clause to **chain exception**.
    - eg. exception_chain.py

Exception Handling

Custom and User-defined Exceptions

- Sometimes we have to define and raise exceptions explicitly to indicate that something goes wrong. Such a type of exception is called a **user-defined exception** or **customized exception.**
  - eg. user_defined_exception.py

Customizing Exception Classes

- We can customize the classes by accepting arguments as per our requirements. Any custom exception class must be Extending from BaseException class or subclass of BaseException.
  - eg. user_defined_exception.py

Regular Expression

- A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.
- RegEx can be used to check if a string contains the specified search pattern.
  - import re

RegEx Functions

- The re module offers a set of functions that allows us to search a string for a match:

Function	Description
findall	Returns a list containing all matches
search	Returns a Match object if there is a match anywhere in the string
split	Returns a list where the string has been split at each match
sub	Replaces one or many matches with a string

Regular Expression

- Metacharacters

Character	Description	Example
[]	A set of characters	"[a-m]"
\	Signals a special sequence (can also be used to escape special characters)	"\d"
.	Any character (except newline character)	"he..o"
^	Starts with	"^hello"
$	Ends with	"planet$"
*	Zero or more occurrences	"he.*o"
+	One or more occurrences	"he.+o"
?	Zero or one occurrences	"he.?o"
{}	Exactly the specified number of occurrences	"he.{2}o"
|	Either or	"falls|stays"
()	Capture and group	

Regular Expression

- Special Sequences

Character	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"
\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"

Regular Expression

- Sets

Set	Description
[arn]	Returns a match where one of the specified characters (a, r, or n) is present
[a-n]	Returns a match for any lower case character, alphabetically between a and n
[^arn]	Returns a match for any character EXCEPT a, r, and n
[0123]	Returns a match where any of the specified digits (0, 1, 2, or 3) are present
[0-9]	Returns a match for any digit between 0 and 9
[0-5][0-9]	Returns a match for any two-digit numbers from 00 and 59
[a-zA-Z]	Returns a match for any character alphabetically between a and z, lower case OR upper case
[+]	In sets, +, *, ., |, (), $,{} has no special meaning, so [+] means: return a match for any + character in the string

Regular Expression

The findall() Function

- The findall() function returns a list containing all matches.

The search() Function

- The search() function searches the string for a match, and returns a match object if there is a match.

The split() Function

- The split() function returns a list where the string has been split at each match.
- You can control the number of occurrences by specifying the maxsplit parameter:\

The sub() Function

- The sub() function replaces the matches with the text of your choice.
- You can control the number of replacements by specifying the count parameter.

Match Object

- A Match Object is an object containing information about the search and the result.
- The Match object has properties and methods used to retrieve information about the search, and the result:
    - .span() returns a tuple containing the start-, and end positions of the match.
    - .string returns the string passed into the function
    - .group() returns the part of the string where there was a match

    eg. regex_exp.py