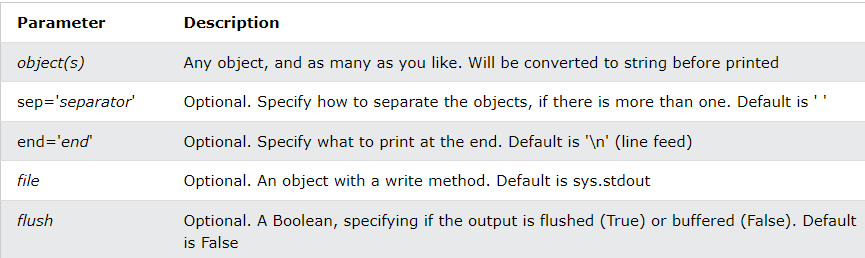
# **DevOps- Python Handbook**

|  |
| --- |
| Operators |
| String |
| Conditional logic |
| Functions |
| Modules |
| List |
| Tuple |
| Set |
| Dictionary |
| Exception Handling |
| Lambda, Map, Filters, ZIP |
| OOPS |
| Exception Handling |

Print(‘…………..’)- print on console

Print on next line \n





**Arithmetic Operators**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| **+ (Addition)** | It is used to add two operands. For example, if a = 20, b = 10 => a+b = 30 |
| **- (Subtraction)** | It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value results negative. For example, if a = 20, b = 10 => a - b = 10 |
| **/ (divide)** | It returns the quotient after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a/b = 2.0 |
| **\* (Multiplication)** | It is used to multiply one operand with the other. For example, if a = 20, b = 10 => a \* b = 200 |
| **% (reminder)** | It returns the reminder after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a%b = 0 |
| **\*\* (Exponent)** | It is an exponent operator represented as it calculates the first operand power to the second operand. |
| **// (Floor division)** | It gives the floor value of the quotient produced by dividing the two operands. |

**Comparison operator**

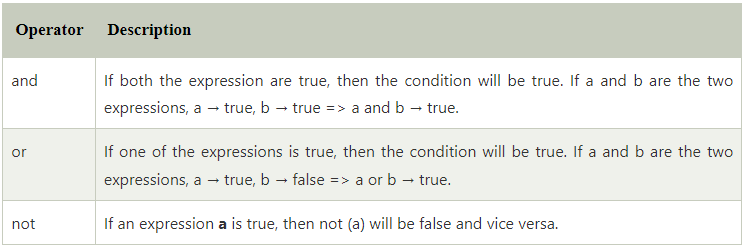
|  |  |
| --- | --- |
| **Operator** | **Description** |
| == | If the value of two operands is equal, then the condition becomes true. |
| != | If the value of two operands is not equal, then the condition becomes true. |
| <= | If the first operand is less than or equal to the second operand, then the condition becomes true. |
| >= | If the first operand is greater than or equal to the second operand, then the condition becomes true. |
| > | If the first operand is greater than the second operand, then the condition becomes true. |
| **<** | If the first operand is less than the second operand, then the condition becomes true. |

**Assignment Operators**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| = | It assigns the value of the right expression to the left operand. |
| += | It increases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 10, b = 20 => a+ = b will be equal to a = a+ b and therefore, a = 30. |
| -= | It decreases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 20, b = 10 => a- = b will be equal to a = a- b and therefore, a = 10. |
| \*= | It multiplies the value of the left operand by the value of the right operand and assigns the modified value back to then the left operand. For example, if a = 10, b = 20 => a\* = b will be equal to a = a\* b and therefore, a = 200. |
| %= | It divides the value of the left operand by the value of the right operand and assigns the reminder back to the left operand. For example, if a = 20, b = 10 => a % = b will be equal to a = a % b and therefore, a = 0. |
| \*\*= | a\*\*=b will be equal to a=a\*\*b, for example, if a = 4, b =2, a\*\*=b will assign 4\*\*2 = 16 to a. |
| //= | A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign 4//3 = 1 to a. |

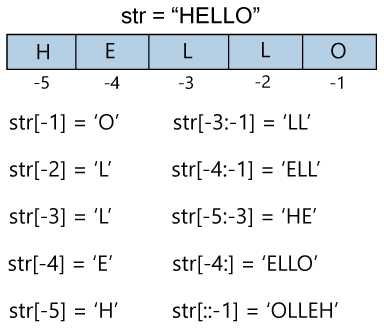
|  |  |
| --- | --- |
| **Operator** | **Description** |
| in | It is evaluated to be true if the first operand is found in the second operand (list, tuple, or dictionary). |
| not in | It is evaluated to be true if the first operand is not found in the second operand (list, tuple, or dictionary). |

|  |  |
| --- | --- |
| **Operator** | **Description** |
| is | It is evaluated to be true if the reference present at both sides point to the same object. |
| is not | It is evaluated to be true if the reference present at both sides do not point to the same object. |



**Strings:**

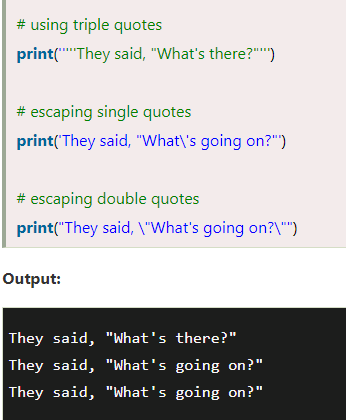
* Passed under single or double quotes ( ‘…….’ / “……..”)
* Multiline string is passed in triple quotes ( “”” …………………… “””)
* Indexing starts at 0
* Printing specific index, pass the index in square brackets **string[index]**
* Slicing: **string[startIndex : endIndex]**
* Indexing from right of string i.e. in reverse direction starts with -1



* Strings are immutable i.e. cannot update or delete any character from string
* To delete a string use **del string**
* ***String Operators:***

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + | It is known as concatenation operator used to join the strings given either side of the operator. |
| \* | It is known as repetition operator. It concatenates the multiple copies of the same string. |
| [] | It is known as slice operator. It is used to access the sub-strings of a particular string. |
| [:] | It is known as range slice operator. It is used to access the characters from the specified range. |
| in | It is known as membership operator. It returns if a particular sub-string is present in the specified string. |
| not in | It is also a membership operator and does the exact reverse of in. It returns true if a particular substring is not present in the specified string. |

* ***String Formatting:***
* The backslash(/) symbol denotes the escape sequence. The backslash can be followed by a special character and it interpreted differently. The single quotes inside the string must be escaped. We can apply the same as in the double quotes.

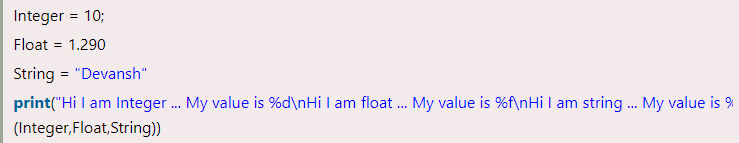


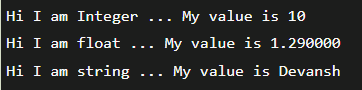
|  |  |  |
| --- | --- | --- |
| **Escape Sequence** | **Description** | **Example** |
| \newline | It ignores the new line. | print("Python1 \  Python2 \  Python3")  **Output:**  Python1 Python2 Python3 |
| \\ | Backslash | print("\\")  **Output:**  \ |
| \' | Single Quotes | print('\'')  **Output:**  ' |
| \\'' | Double Quotes | print("\"")  **Output:**  " |

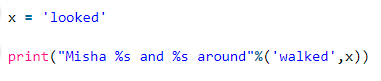
* Format method: curly braces {} are used as the placeholder in the string and replaced by the **format()** method argument. Argument is mapped to values in 3 ways: using curly braces, positional arguments and keyword argument



* % method: it binds the format specifiers to the values.

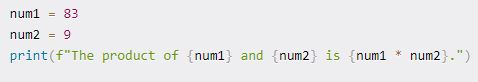








* Using f method: Statement starts with f followed by “….” And { } is used as placeholder also it can be used to hold expressions





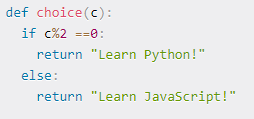




Calling a method with f string:

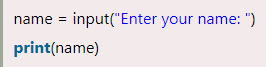


Calling a function inside f string:





* Accepting user input: using input() method



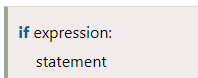
**In Place operator**: var +=10 i.e. var=var+10; var\*=2 i.e. var=var\*2

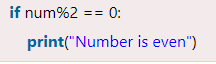
Str= “hi”

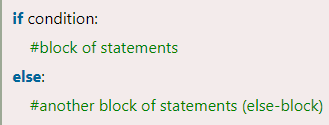
Str+=” there”

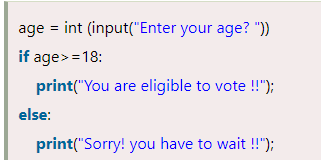
Print(Str) === hi there

**Conditional Statements**

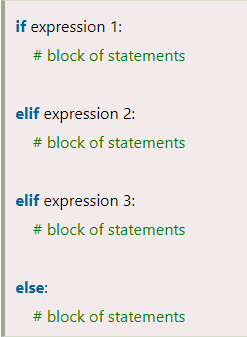


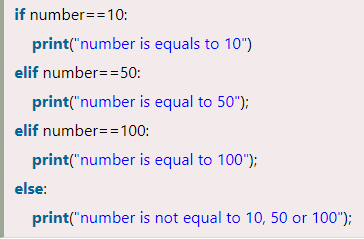






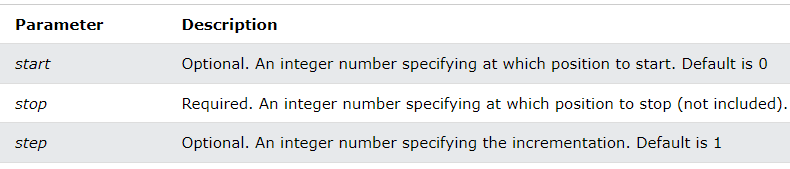
The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them



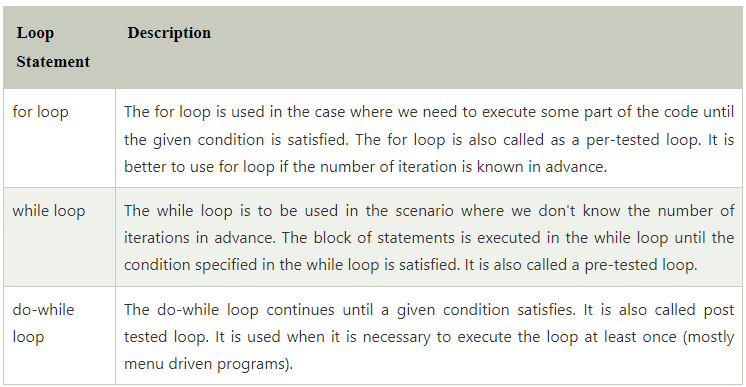


* The **range()** function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

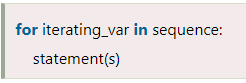




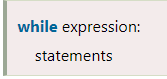
**Loops**



***For Loop***

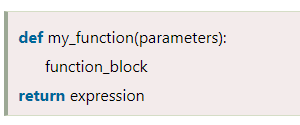


***While Loop***



1. **Continue Statement -**When the continue statement is encountered, the control transfer to the beginning of the loop
2. **Break Statement -**When the break statement is encountered, it brings control out of the loop.
3. The pass statement is used to declare the empty loop. It is also used to define empty class, function, and control statement

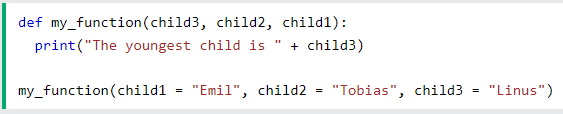
**Functions in Python**



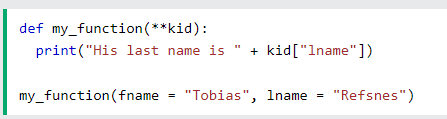
***Types of arguments***

There may be several types of arguments which can be passed at the time of function call.

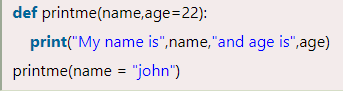
1. Required arguments
2. Keyword arguments: send arguments with the key = value syntax



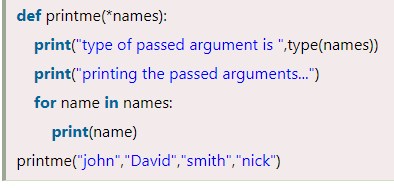
If you do not know how many keyword arguments that will be passed into your function, add two asterisk: **\*\*** before the parameter name in the function definition.



1. Default arguments: Python allows us to initialize the arguments at the function definition. If the value of any of the arguments is not provided at the time of function call, then that argument can be initialized with the value given in the definition even if the argument is not specified at the function call.



1. Variable-length arguments: If you do not know how many arguments that will be passed into your function, add a **\*** before the parameter name in the function definition.



**Variable Scope:**

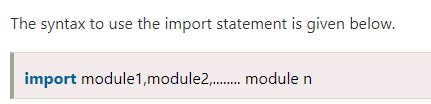
In python, the variables are defined with the two types of scopes.

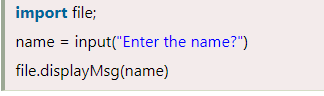
1. Global variables
2. Local variables

**Modules:**

We need to load the module in our python code to use its functionality. Python provides two types of statements as defined below.

1. The import statement
2. The from-import statement- Instead of importing the whole module into the namespace, python provides the flexibility to import only the specific attributes of a module.

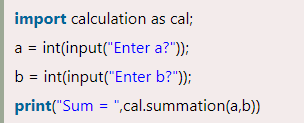






Python provides us the flexibility to import some module with a specific name so that we can use this name to use that module in our python source file.

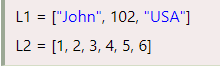




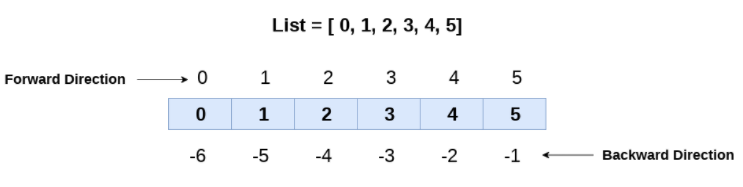
The **dir()** function returns a sorted list of names defined in the passed module. This list contains all the sub-modules, variables and functions defined in this module.

**Lists**

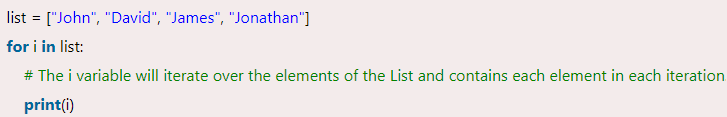
* A list can be defined as a collection of values or items of different types. The items in the list are separated with the comma (,) and enclosed with the square brackets []
* Python lists are mutable type it means we can modify its element after it created



* The list has the following characteristics:
  + The lists are ordered.
  + The element of the list can access by index.
  + The lists are the mutable type.
  + A list can store the number of various elements.



* **Iterating a list:**
  + A list can be iterated by using a for - in loop



* **Adding element:**
  + The append() method appends an element to the end of the list.

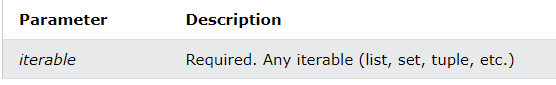


* + The insert() method inserts the specified value at the specified position.



* + The extend() method adds the specified list elements (or any iterable) to the end of the current list.





* **Removing element:**
  + The remove() method removes the first occurrence of the element with the specified value.

****

* + The pop() method removes the element at the specified position.

****

* + The clear() method removes all the elements from a list.

****

* The **copy()** method returns a copy of the specified list.

****

* The **count()** method returns the number of elements with the specified value.

****

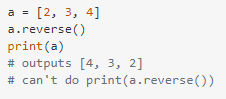
* The **index()** method returns the position at the first occurrence of the specified value.

****

* The **reverse()** method reverses the sorting order of the elements.

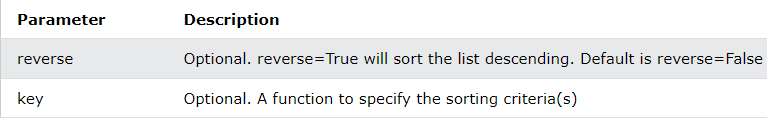
****

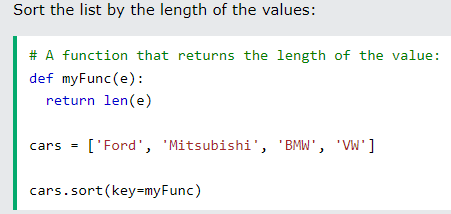
In python, **a.reverse() reverses in place, but does not return the array**

****

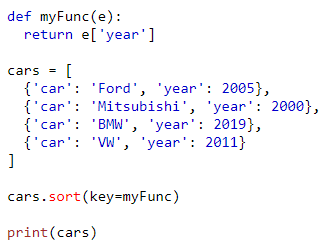
* The **sort()** method sorts the list ascending by default. You can also make a function to decide the sorting criteria(s).

****

****

****

****

****

****

* **List Comprehension:** List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

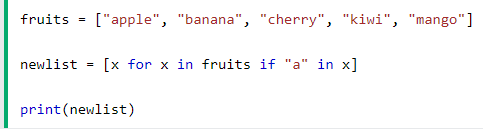
****

****

The return value is a new list, leaving the old list unchanged.

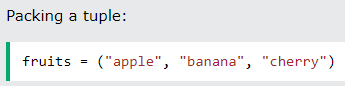
The condition is like a filter that only accepts the items that valuate to ***True.***

The expression is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list. The expression can also contain conditions, not like a filter, but as a way to manipulate the outcome

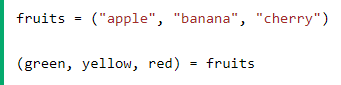


* **Unpacking / Packing:**

When we create a list or tuple variable and assign values to it, this is called packing

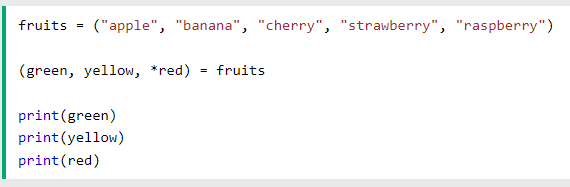


We can also extract the values back into variables, this is called unpacking



The number of variables must match the number of values in the tuple, if not, you must use an asterisk to collect the remaining values as a list.

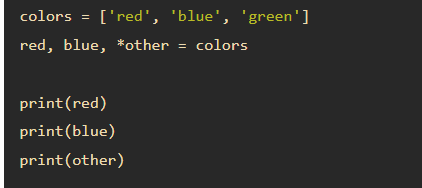
If the number of variables is less than the number of values, you can add an**\*** to the variable name and the values will be assigned to the variable as a list:

****

If you want to unpack the first few elements of a list and don’t care about the other elements, you can:

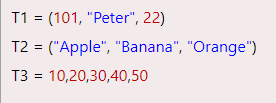
* First, unpack the needed elements to variables.
* Second, pack the leftover elements into a new list and assign it to another variable.

By putting the asterisk (\*) in front of a variable name, you’ll pack the leftover elements into a list and assign it to a variable. For example:

****

**Tuples:**

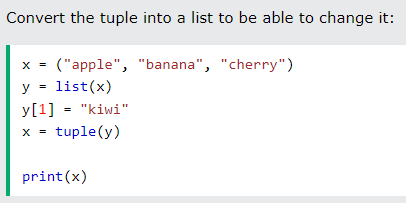
* Tuple is used to store the sequence of immutable Python objects
* A tuple is a collection which is ordered and **unchangeable**.
* Tuples are written with round brackets. ( …… )

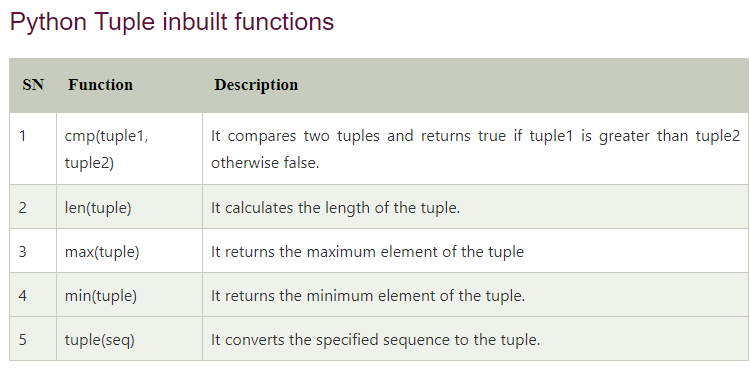
****

**Adding Elements:**

* Same as lists
* As tuple is immutable, to add / remove / update: first convert it into a list and then back to tuple
* Convert tuple to list:

Var = list(tuple\_var)

****

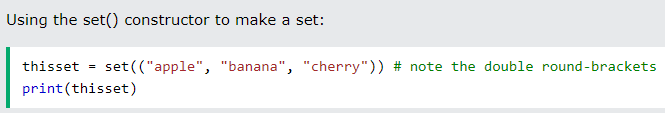
****

**Set**

* Set is the collection of the unordered items
* Each element in the set must be unique, the sets remove the duplicate elements.
* Set items are unordered, unchangeable, and do not allow duplicate values
* There is no index attached to the elements of the set, i.e., we cannot directly access any element of the set by the index. However, we can print them all together, or we can get the list of elements by looping through the set.
* Declared using curly braces { …… }

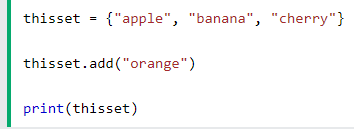
****

* It is also possible to use the set() constructor to make a set.

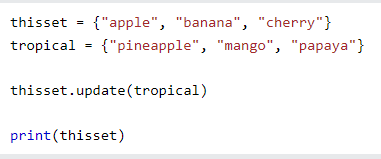


**Adding element:**

* To add one item to a set use the add() method.

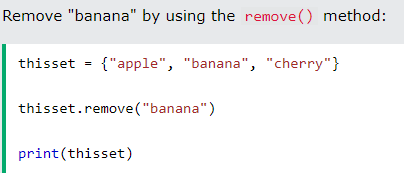


* To add items from another set into the current set, use the update() method.

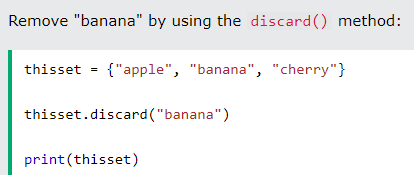


**Removing element:**

* To remove an item in a set, use the remove(), or the discard() method.

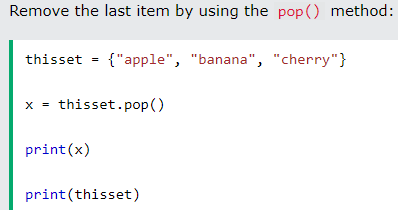


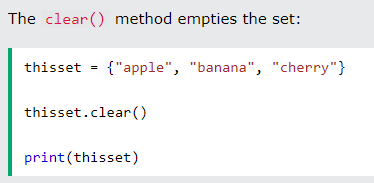
**Note:** If the item to remove does not exist, remove() will raise an error.

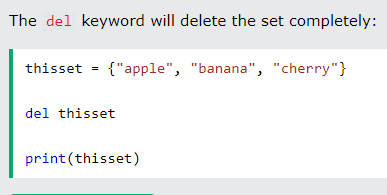


**Note:** If the item to remove does not exist, discard() will **NOT** raise an error.

You can also use the pop() method to remove an item, but this method will remove the *last* item. Remember that sets are unordered, so you will not know what item that gets removed. The return value of the pop() method is the removed item.

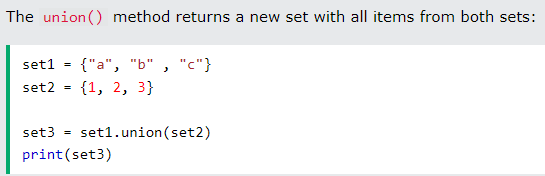


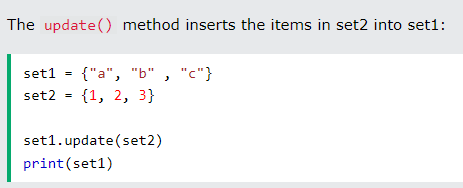




**Joining set:**

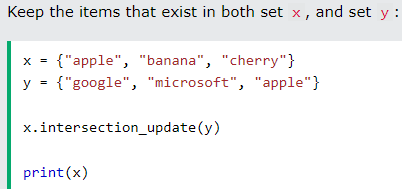
* You can use the union() method that returns a new set containing all items from both sets, or the update() method that inserts all the items from one set into another:



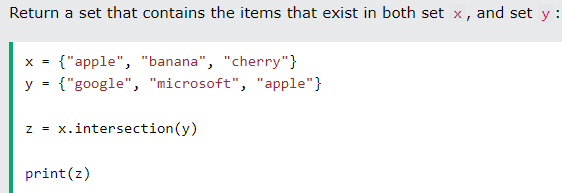




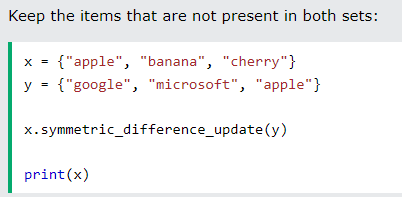
The intersection\_update() method will keep only the items that are present in both sets.



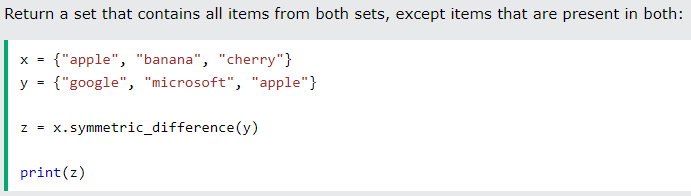
The intersection() method will return a new set, that only contains the items that are present in both sets.



The symmetric\_difference\_update() method will keep only the elements that are NOT present in both sets.

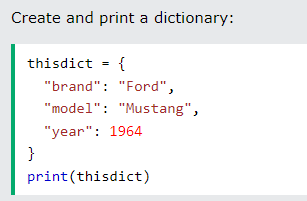


The symmetric\_difference() method will return a new set, that contains only the elements that are NOT present in both sets.

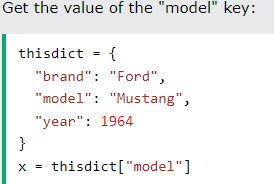


**Dictionary:**

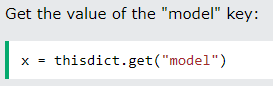
* Dictionaries are used to store data values in key:value pairs
* A dictionary is a collection which is ordered\*, changeable and do not allow duplicates.
* Dictionaries are written with curly brackets, and have keys and values:



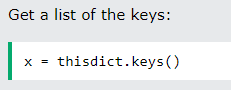
* Python provides the built-in function **dict()** method which is also used to create dictionary. The empty curly braces {} is used to create empty dictionary.
* ***Accessing Items***: You can access the items of a dictionary by referring to its key name, inside square brackets:



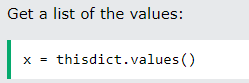
There is also a method called get() that will give you the same result:



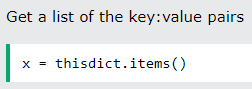
* The keys() method will return a list of all the keys in the dictionary.



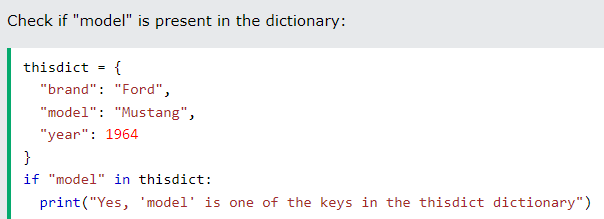
* The values() method will return a list of all the values in the dictionary.



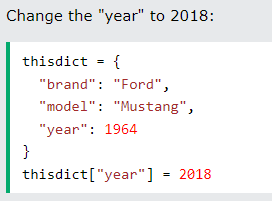
* The items() method will return each item in a dictionary, as tuples in a list.



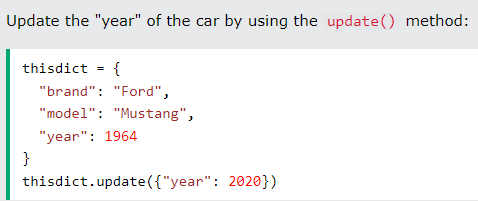
* To determine if a specified key is present in a dictionary use the in keyword:



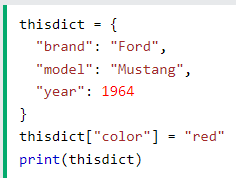
* ***Accessing Items***: You can change the value of a specific item by referring to its key name:



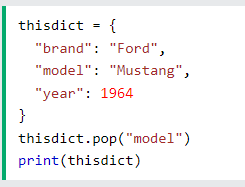
* The update() method will update the dictionary with the items from the given argument. The argument must be a dictionary, or an iterable object with key:value pairs.



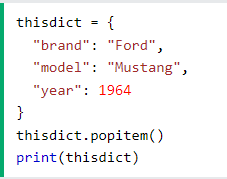
* ***Adding Items:*** Adding an item to the dictionary is done by using a new index key and assigning a value to it:

******

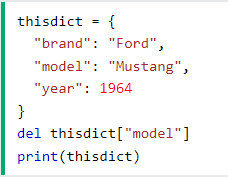
* ***Removing Items:*** The pop() method removes the item with the specified key name:



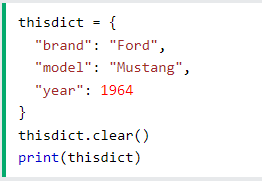
* The popitem() method removes the last inserted item



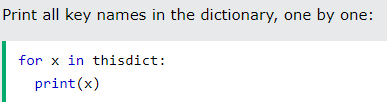
* The del keyword removes the item with the specified key name:



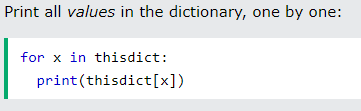
* The clear() method empties the dictionary:



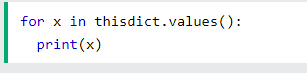
* ***Iteration***: When looping through a dictionary, the return value are the keys of the dictionary



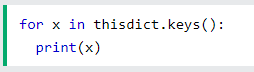
Values:



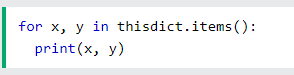
* You can also use the values() method to return values of a dictionary:



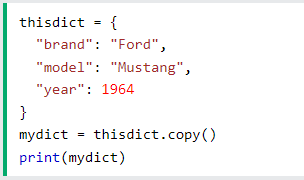
* You can use the keys() method to return the keys of a dictionary:



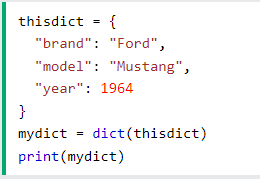
* Loop through both keys and values, by using the items() method:



* Make a copy of a dictionary with the copy() method:



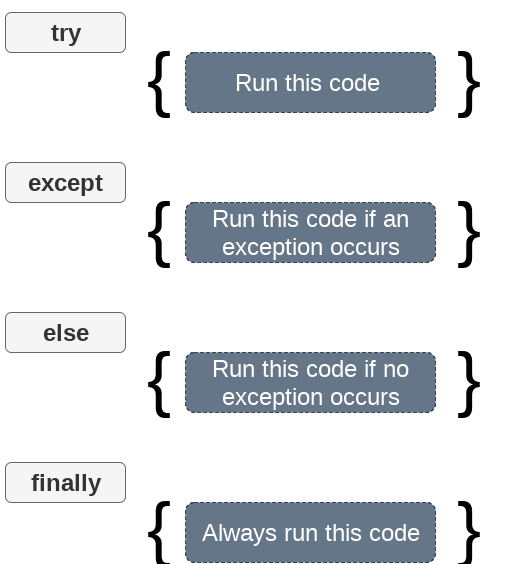
Another way to make a copy is to use the built-in function dict().



**Exception Handling**

A list of common exceptions that can be thrown from a standard Python program is given below.

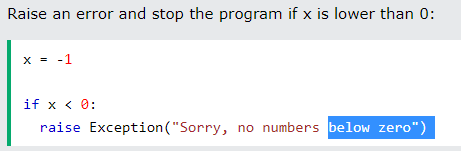
1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.



You can use the else keyword to define a block of code to be executed if no errors were raised:



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

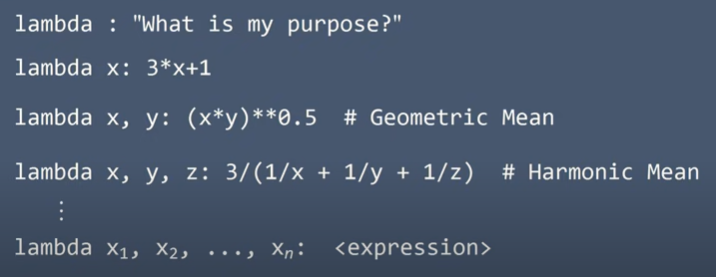


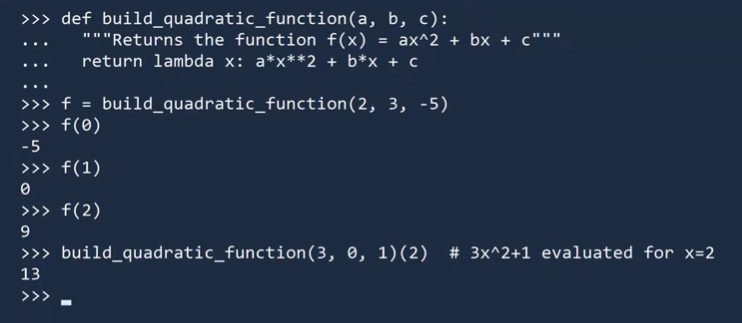
**Lambda, Map, Filters, Generators,ZIP**

Lambda:

* Python Lambda function is known as the anonymous function that is defined without a name
* Lambda functions can accept any number of arguments, but they can return only one value in the form of expression.



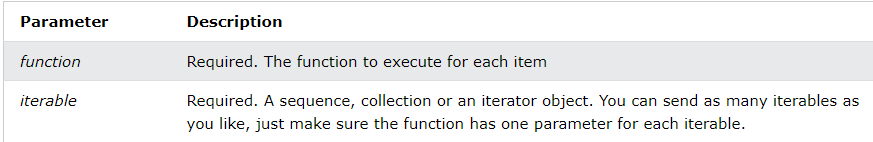


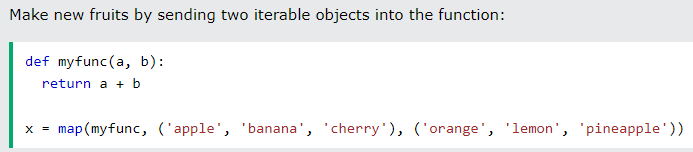




* The map() function executes a specified function for each item in an iterable. The item is sent to the function as a parameter.

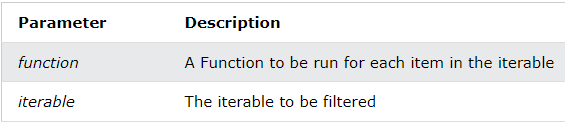






* The filter() function returns an iterator were the items are filtered through a function to test if the item is accepted or not.

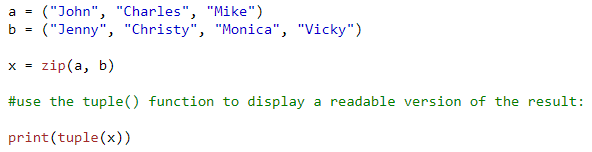




* The zip() function returns a zip object, which is an iterator of tuples where the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc.



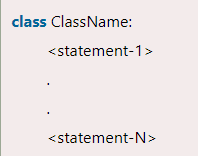
If the passed iterators have different lengths, the iterator with the least items decides the length of the new iterator.





**OOPS**

**Class**: The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods



**Object**: The object is an entity that has state and behavior. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.

The syntax to create the instance of the class is given below.

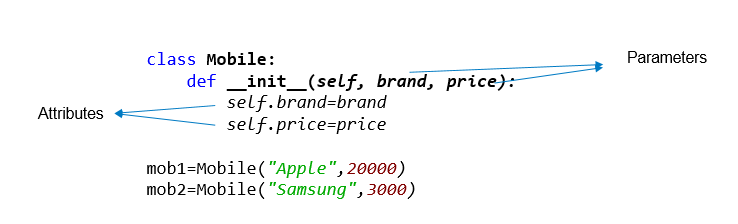
<object-name> = <**class**-name>(<arguments>)

 The syntax for creating attribute and value for that is as below:

reference\_variable.attribute\_name=value.

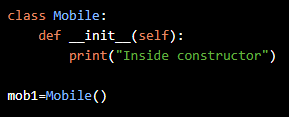
If we try to access a non-existing attribute, we will get an Attribute Error.

**Attribute in a class:**

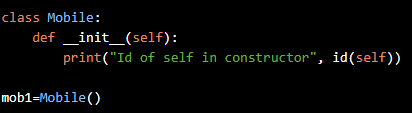


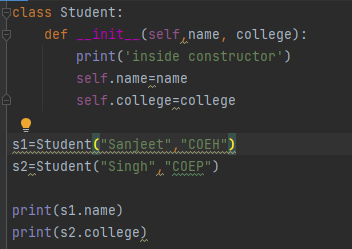
**Constructor & Self:**

When we create an object, the special \_\_init\_\_() method inside the class of that object is invoked automatically. This special function is called as a **constructor**.



**self** is not a keyword. self refers to the current object being executed. Like we have this in java

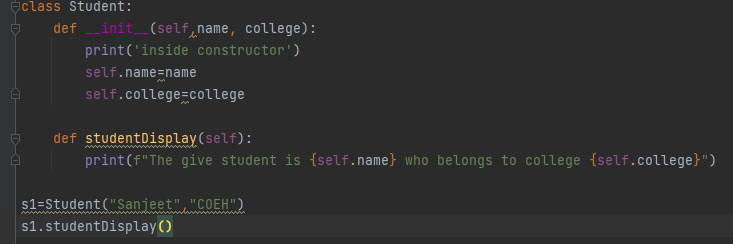




Attributes can be created only by using the self variable and the dot operator. Without self we are only creating a local variable and not an attribute.

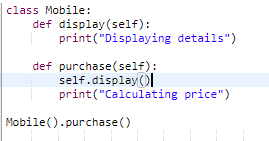
We can create behavior in a class by adding functions in a class. However, such functions should have a special parameter called self as the first parameter.

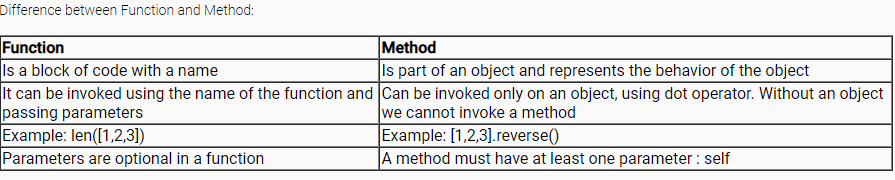
Such functions which describe the behavior are also called as methods. We can invoke the methods using the dot operator as shown.





We can also invoke one method from another using self.



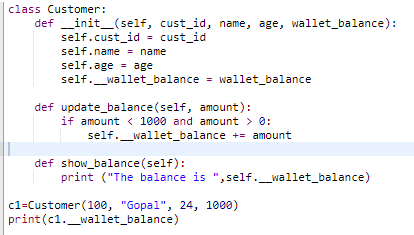


For a more readable output when printing an object we can use the inbuilt special \_\_str\_\_ method. This method MUST return a string and this string will be used when the object is printed. This is useful in debugging as we can print the values of the attributes

**Abstraction**: ability to use something without having to know the details of how it is working is called as **abstraction.**

**Encapsulation**: wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object’s variable can only be changed by an object’s method. Those types of variables are known as **private**

Adding a double underscore makes the attribute a private attribute. Private attributes are those which are accessible only inside the class. This method of restricting access to our data is called **encapsulation**



When we put a double underscore in front of the attribute name, python will internally change its name to \_Classname\_\_attribute.

This is why we get an error when we try to access a private attribute.

**Note**:

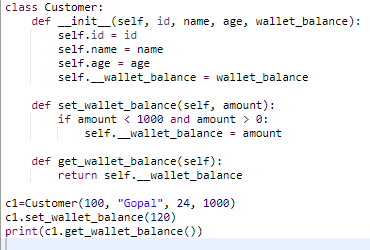
If we try to assign a value to a private variable, we end up creating a new attribute in python. Thus this code does not give an error, but it is logically flawed and does not produce the intended result.

**Getters & Setters:**

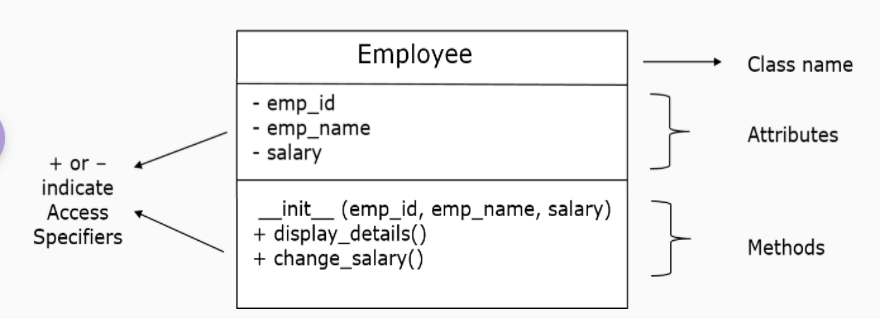
To have a error free way of accessing and updating private variables, we create specific methods for this.

* The methods which are meant to set a value to a private variable are called **setter methods**
* The methods meant to access private variable values are called **getter methods**

The below code is an example of getter and setter methods:



Class diagram:

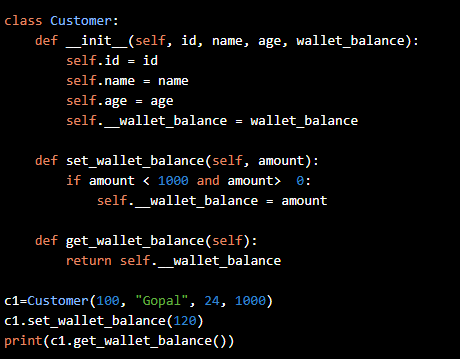


In a class diagram, a – sign indicates private access and + indicates public access.

**Note:**We can create private methods by adding a double underscore in front of it, just like private variables. Also, if a method has both leading and trailing double underscores ( like \_\_init\_\_, \_\_str\_\_, etc) it indicates that it is a special built-in method. As per coding convention, we are not supposed to create our own methods with both leading and trailing underscores.

All setter methods must accept the value to be updated as a parameter and all getter methods must not have any parameter and they must return the value.

Setter methods are called as mutator methods ( as they change or mutate the value ) and the getter methods are called accessor methods ( as they access the values )



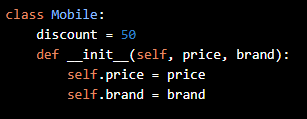
# A student is identified by student id, age and marks in qualifying exam.  
  
class Student:  
 def \_\_init\_\_(self):  
 self.\_\_student\_id = None  
 self.\_\_marks = None  
 self.\_\_age = None  
 self.\_\_course\_id = None  
 self.\_\_fees = None  
  
 def set\_course\_id(self, course\_id):  
 self.\_\_course\_id = course\_id  
  
 def get\_course\_id(self):  
 return self.\_\_course\_id  
  
 def set\_fees(self, fees):  
 self.\_\_fees = fees  
  
 def get\_fees(self):  
 return self.\_\_fees  
  
 def set\_student\_id(self, studentId):  
 self.\_\_student\_id = studentId  
  
 def get\_student\_id(self):  
 return self.\_\_student\_id  
  
 def set\_marks(self,marks):  
 self.\_\_marks = marks  
  
 def get\_marks(self):  
 return self.\_\_marks  
  
 def set\_age(self,age):  
 self.\_\_age = age  
  
 def get\_age(self):  
 return self.\_\_age  
  
 def validate\_marks(self):  
 marks = self.get\_marks()  
 if marks >=0 and marks <=100:  
 return True  
 else:  
 return False  
  
 def validate\_age(self):  
 age = self.get\_age()  
 if age > 20:  
 return True  
 else:  
 return False  
  
 def check\_qualification(self):  
 marks\_flag = self.validate\_marks()  
 age\_flag = self.validate\_age()  
 if marks\_flag == True and age\_flag == True :  
 if self.get\_marks() >= 65:  
 return True  
 else:  
 return False  
 else:  
 return False  
  
 def choose\_course(self,course\_id):  
 self.set\_course\_id(course\_id)  
 if course\_id == 1001:  
 if self.get\_marks() > 85:  
 total\_fee = 25575 - (.25 \* 25575)  
 else:  
 total\_fee = 25575  
 self.set\_fees(total\_fee)  
 return True  
 elif course\_id == 1002:  
 if self.get\_marks() > 85:  
 total\_fee = 15500 - (.25 \* 15500)  
 else:  
 total\_fee = 15500  
 self.set\_fees(total\_fee)  
 return True  
 else:  
 return False  
  
  
maddy=Student()  
maddy.set\_student\_id(1004)  
maddy.set\_age(21)  
maddy.set\_marks(65)  
if(maddy.check\_qualification()):  
 print("Student has qualified")  
 if(maddy.choose\_course(1002)):  
 print("Course allocated")  
 else:  
 print("Invalid course id")  
else:  
 print("Student has not qualified")

We have a list of customer objects. Complete the code so that we have a dictionary of customer objects based on location.

class Customer:  
 def \_\_init\_\_(self, cust\_id, cust\_name, location):  
 self.cust\_id = cust\_id  
 self.cust\_name = cust\_name  
 self.location = location  
  
  
list\_of\_customers = [Customer(101, 'Mark', 'US'),  
 Customer(102, 'Jane', 'Japan'),  
 Customer(103, 'Kumar', 'India')]  
  
dict\_of\_customer = {}  
for c in list\_of\_customers:  
 dict\_of\_customer[c.location]=[c.cust\_id , c.cust\_name]  
  
print(dict\_of\_customer)

**Static Variable:**

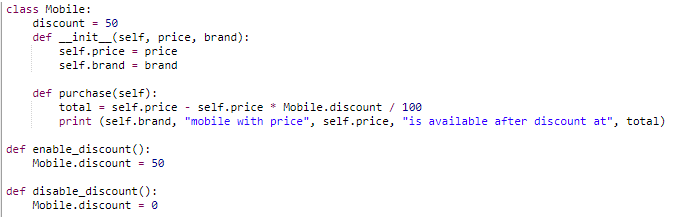
We can create shared attributes by placing them directly inside the class and not inside the constructor. And since this attribute is not owned by any one object, we don’t need the self to create this attribute. Such variables which are created at a class level are called **static variables**. Here discount is a static value.



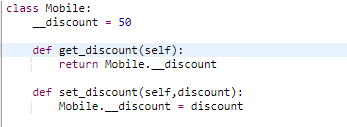
we can access them using the Class name itself. Static variable belong to the class and not an object. Hence we don’t need self to access static variables.

className.varName

We can also update static variable value using className.varName = <NewValue>; even outside of class



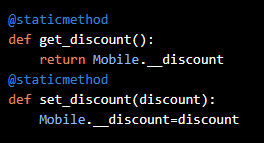
We can make our static variable as a private variable by adding a double underscore in front of it. We can also create getter and setter methods to access or modify it.



Since static variable is object independent, we need a way to access the getter setter methods without an object. This is possible by creating static methods. Static methods are those methods which can be accessed without an object. They are accessed using the class name.

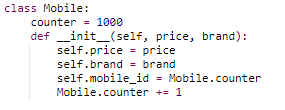
There are two rules in creating such static methods:

* The methods should not have self
* @staticmethod must be written on top of it



We can access static methods directly using the class name, even without creating objects.

Let us say we want to assign a unique number to each mobile object. The first object should be given a number 1000 and subsequent objects should have that value increased by 1. We can accomplish this by using a combination of static and instance variables as shown below:



self.variable\_name indicates it is an instance variable

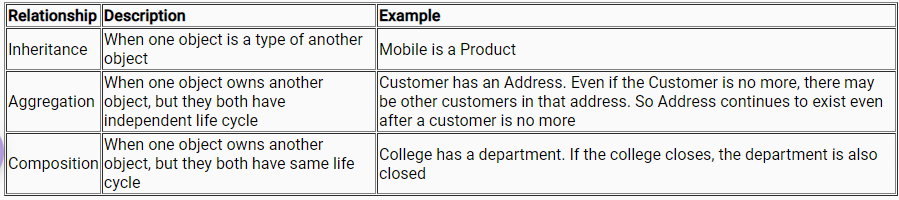
A variable which is created outside any method in a class is a static variable.

Any method created with @staticmethod is a static method.

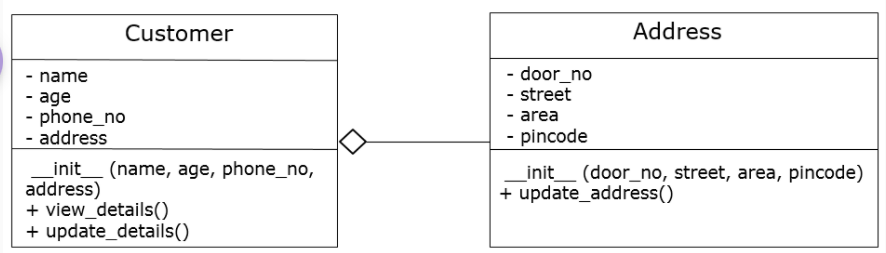
Any other method in a class is an instance method.

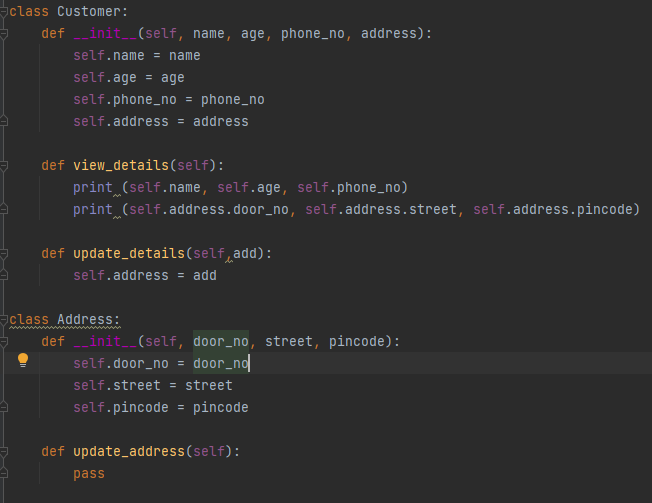
**Class Relationship:**

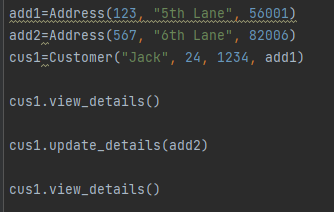
*Aggregation:*

**

**Note:** In class diagram, aggregation is represented by a line connecting the classes and a diamond symbol in the class which aggregates the other class. In the above example, the Customer aggregates the Address and hence the diamond symbol is near the Customer class

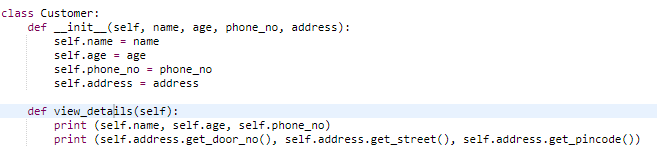


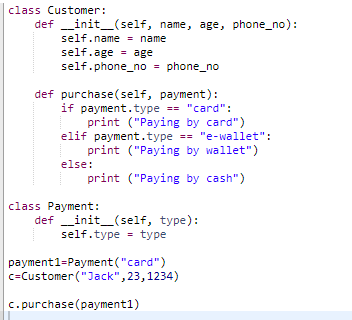


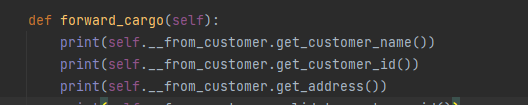


Private variables cannot be accessed outside the class. This is true even in aggregation. The owning class cannot access the private attributes of the aggregated class directly.

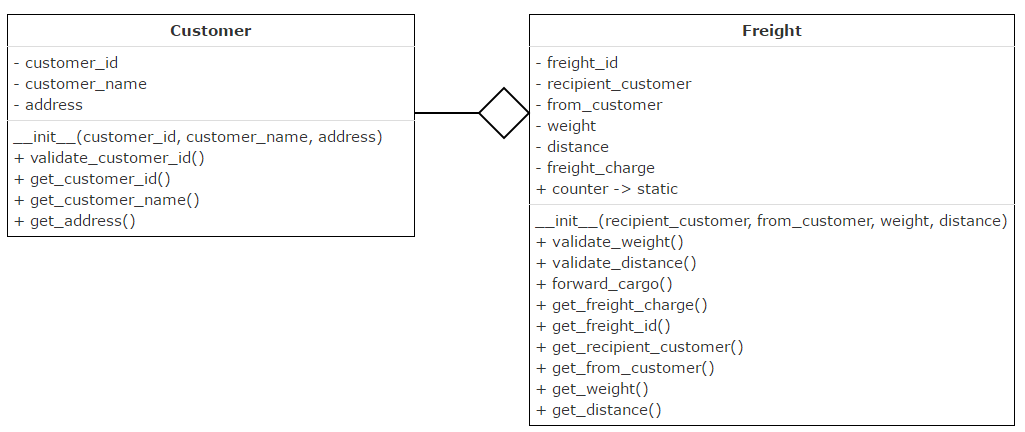
Once we have appropriate accessor and mutator methods we can start accessing the private variables of the aggregated class using those methods.





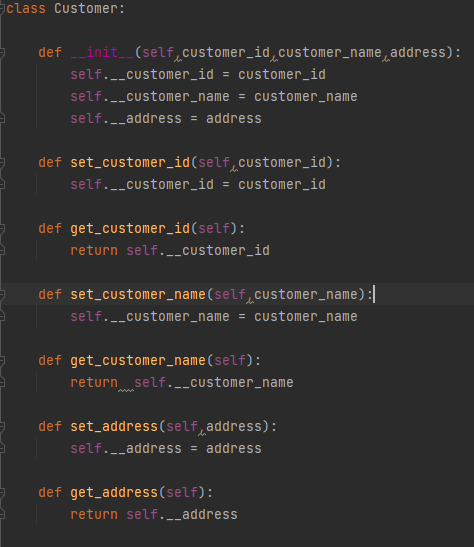


Freight Pvt. Ltd, a cargo company, forwards cargos/freights between its customers.  
Freight charges are applied based on weight and distance of the shipment.  
Write a python program to implement the class diagram given below.

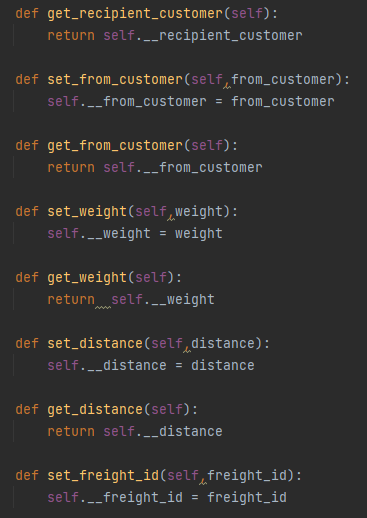


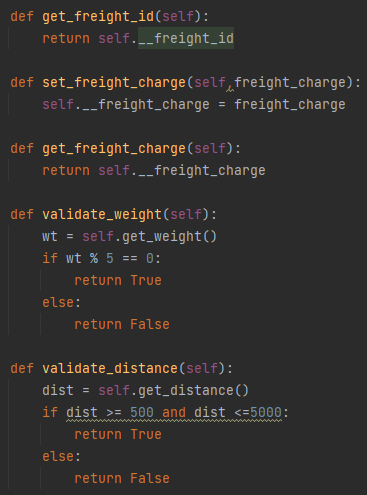
**Method description:**

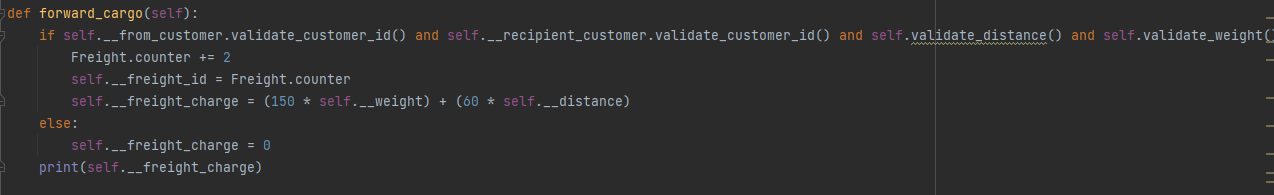
1. Initialize counter variable to 198 in Freight class
2. All validate methods should return true, if validation succeeds. Else it should return false
3. **validate\_customer\_id():** Customer id should be 6 digits and should begin with digit 1
4. **validate\_weight():** Weight should be a multiple of 5
5. **validate\_distance():**Distance should be between 500kms and 5000kms (both inclusive)
6. **forward\_cargo():**
   * Validate from\_customer.customer\_id, recipient\_customer.customer\_id, distance and weight of the freight
   * If valid,
     + auto-generate freight\_id starting from 200 and initialize it. freight\_id should be even
     + calculate freight\_charge based on weight (Rs.150/kg) and distance (Rs.60/km)
   * Else, set freight\_charge to 0









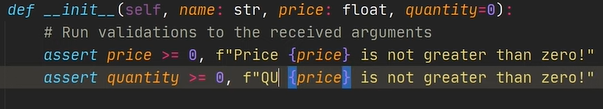


Assertion:

It is a debugging tool, and its primary task is to check the condition. If it finds that the condition is true, it moves to the next line of code, and If not, then stops all its operations and throws an error. It points out the error in the code.

Where Assertion in Python used

* Checking the outputs of the functions.
* Used for testing the code.
* In checking the values of arguments. Checking the valid input.



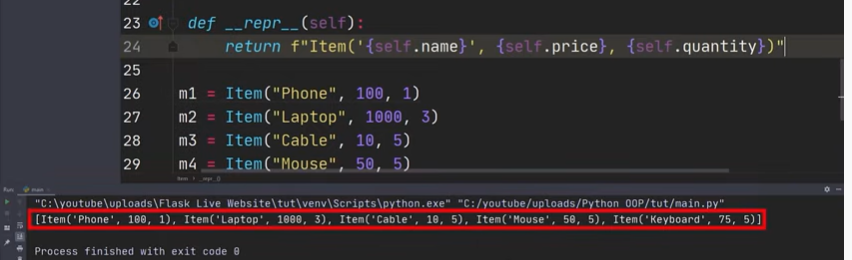
\_\_dict\_\_

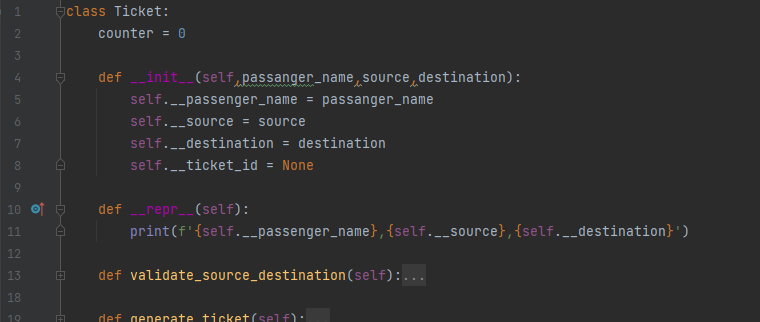
Can list down all the instance and class variables alon with their values, mainly used for debugging

\_\_repr\_\_

Python \_\_repr\_\_() function returns the object representation in string format. This method is called when repr() function is invoked on the object. If possible, the string returned should be a valid Python expression that can be used to reconstruct the object again.

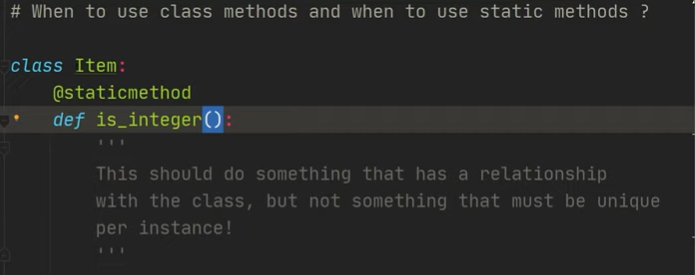
Print(object.repr)

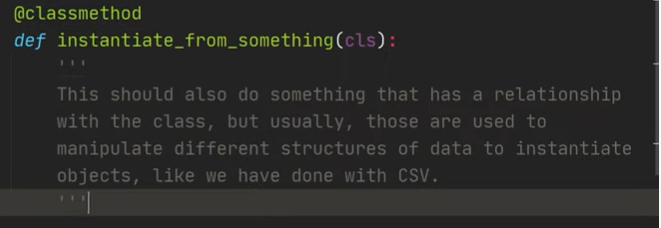




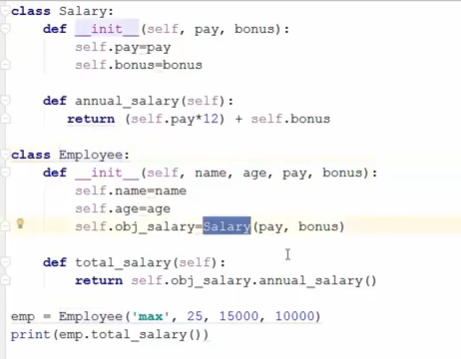




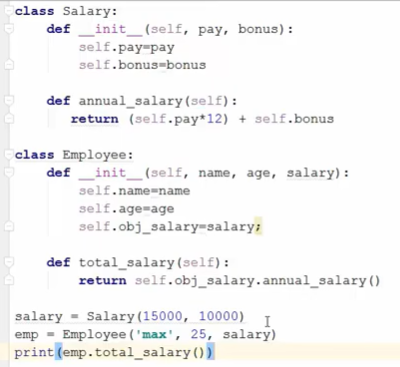




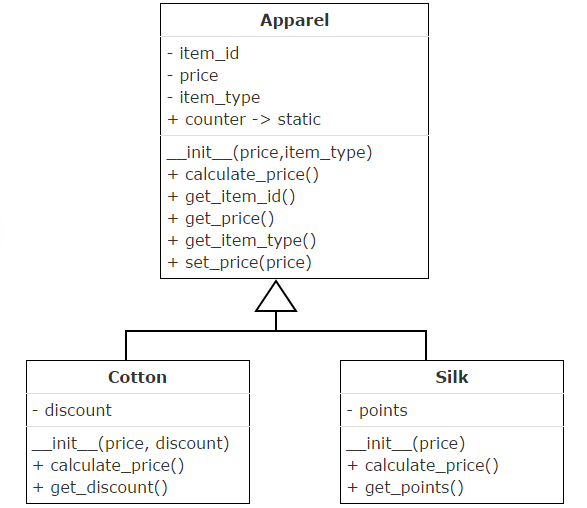
Composition: there is no “is-a” relation between classes, we delegate some responsiblity to other classes i.e. composition represents part-of realtionship



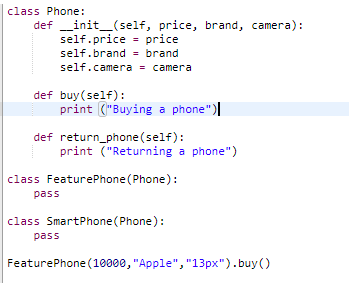
Aggregation: It represents “has-a” relationship. Associated classes has unidirectional association. Objects are indepenedent of each other



Inheritance: Is- A relationship; triangular shape on class diagram



To create an inheritance relationship between the classes, mention the name of the parent class in brackets as shown:

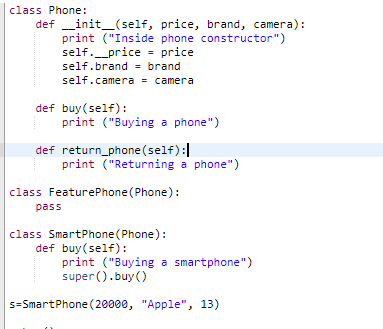


A child class cannot directly access the private attributes of the parent class.

Sometimes a child may not want to use what it has inherited from the parent. The same holds true for OOP as well. If the child class does not want to use a method inherited from the parent class then it may create its own method with the same name.

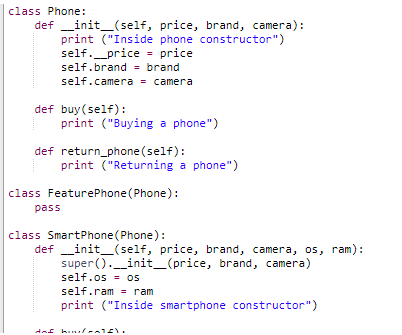
When the child has a method with the same name as that of the parent, it is said to override the parent’s method. This is called as **Method Overriding**. Method overriding is also called as **Polymorphism**.

To invoke anything belonging to the parent class, the child class needs to use the **super()** function



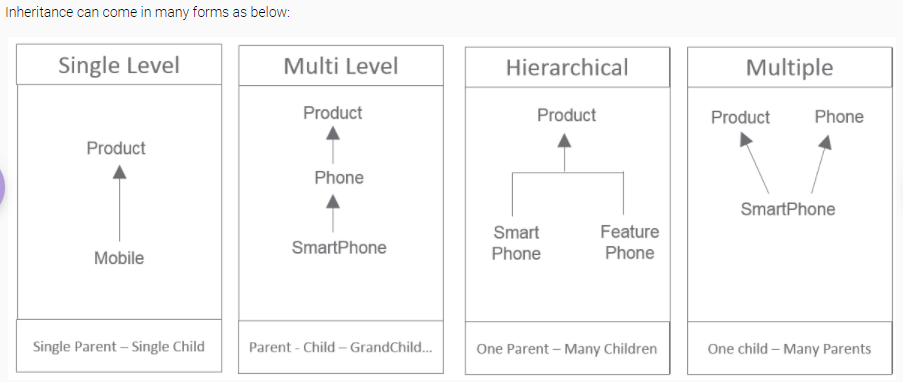
To access the parent class constructor we can use **super()**. Thus, the data is passed to the child class constructor, from there the data is sent to the parent class constructor and thus the attributes of the parent class get inherited.

super() function can be used to access the constructor or methods of the parent class, but not the attributes. Also super() function can be used only inside a class and not outside it

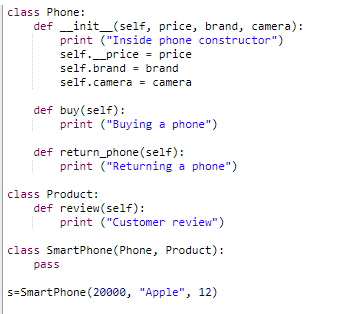


A class can inherit from another class.

* Inheritance improves code reuse
* Constructor, attributes, methods get inherited to the child class
* The parent has no access to the child class
* Private properties of parent are not accessible directly in child class
* Child class can override the attributes or methods. This is called method overriding
* super() is an inbuilt function which is used to invoke the parent class methods and constructor

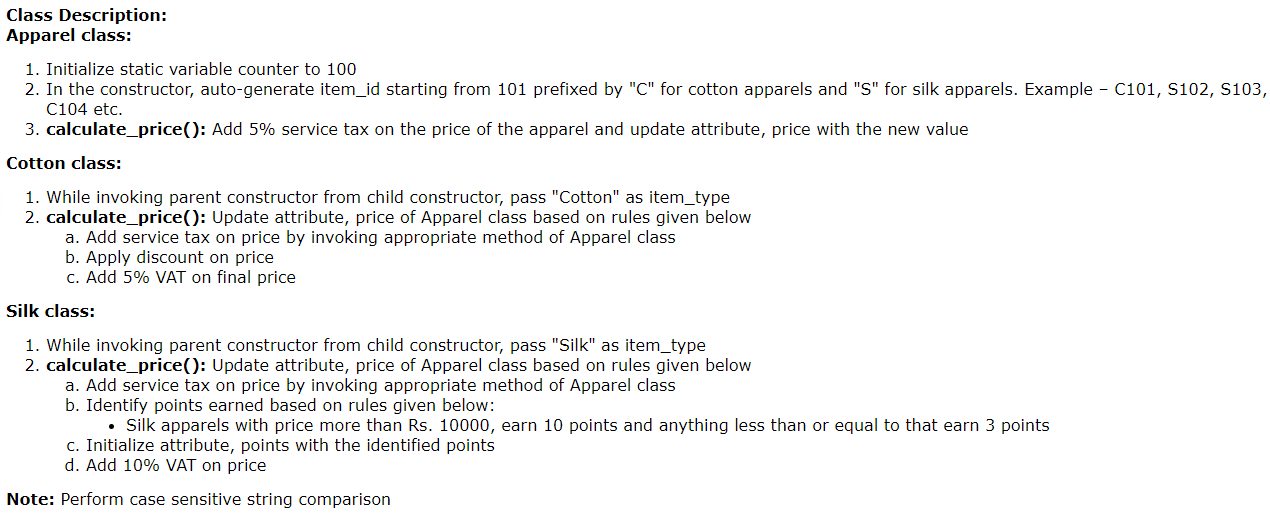


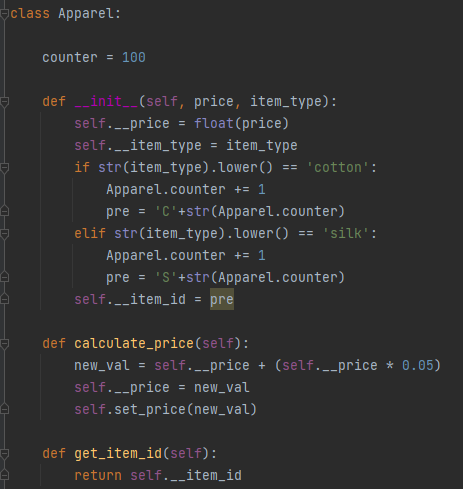
Multiple inheritance:

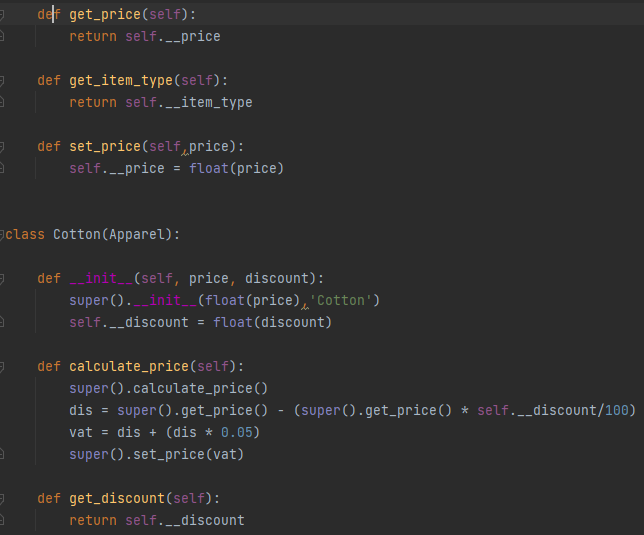


When a child is inheriting from multiple parents, and if there is a common behavior to be inherited, it inherits the method in Parent class which is first in the list. In our example, the buy() of Product is inherited as it appears first in the list.

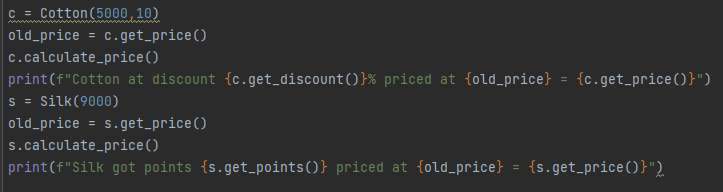
Sample:



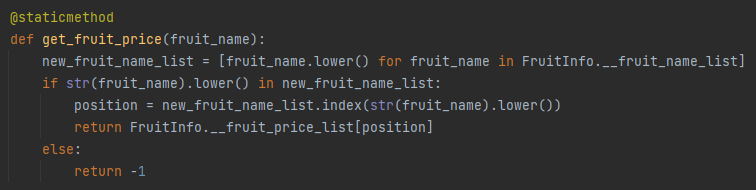




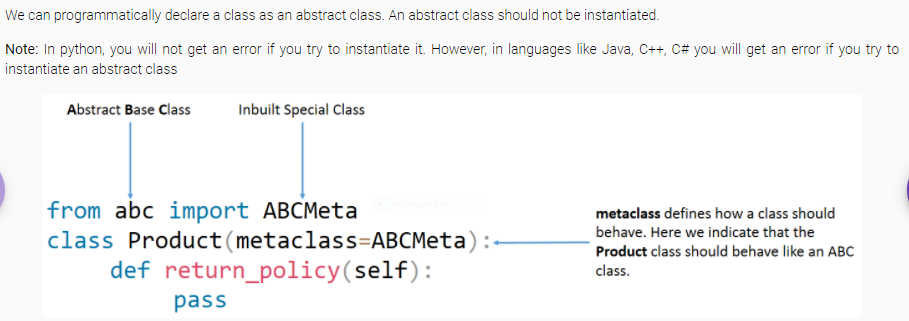




Case insensitive search in list:

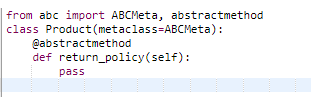


**Abstraction:**



If an abstract class should never be instantiated, then what is the use of such a class? The only way we can use an abstract class is to make other classes inherit from the abstract class. An abstract class is meant to be sub classed.

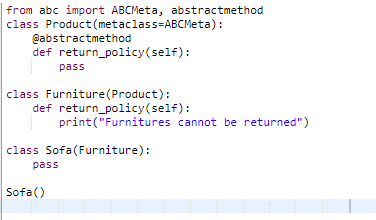
If we programmatically declare our return\_policy() of Product class as an abstract method, then every sub-class of Product class MUST override the abstract method. We can make our return\_policy() an abstract method as shown below:



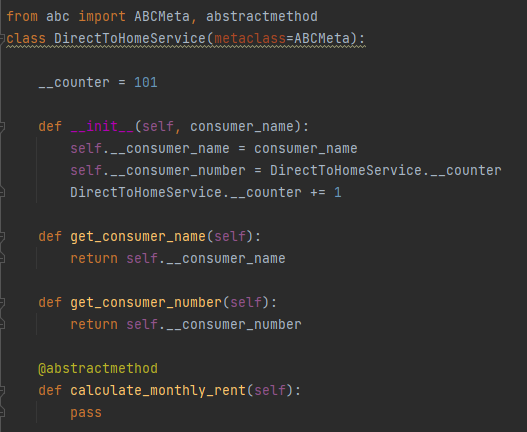
Even if one method is abstract, then we will get an error if we try to instantiate the class.

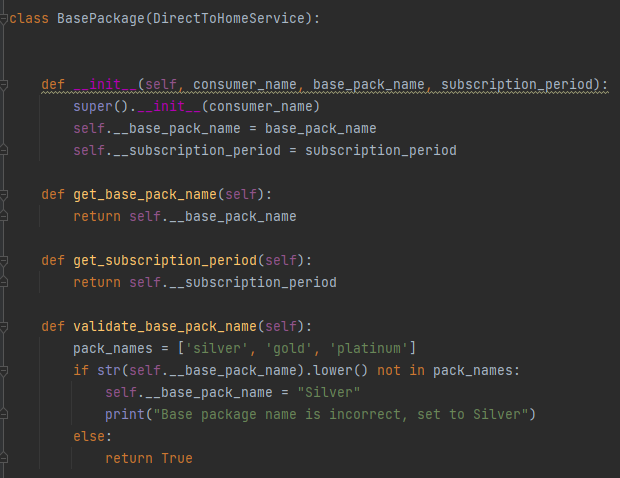
If a method is abstract, then the Subclass must override the abstract method. Else we cannot instantiate the subclass also.

If a child class overrides the abstract method, then its own child classes need not override the abstract method.



* Abstract classes should not be instantiated.
* An abstract class may contain 0 or many abstract methods.
* Usually the parent class is an abstract class.
* Abstract classes are meant to be inherited.
* If a class has an abstract method, then the class cannot be instantiated.
* The child class must implement/override all the abstract methods of the parent class. Else the child class cannot be instantiated.

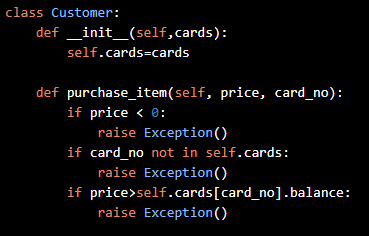


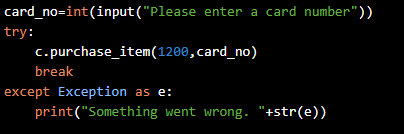




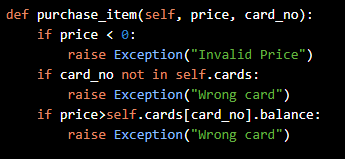
**Exception Handling:**

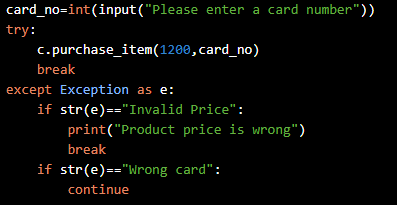
In Python, we can raise exception by using the raise keyword.





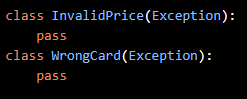
Custom message:

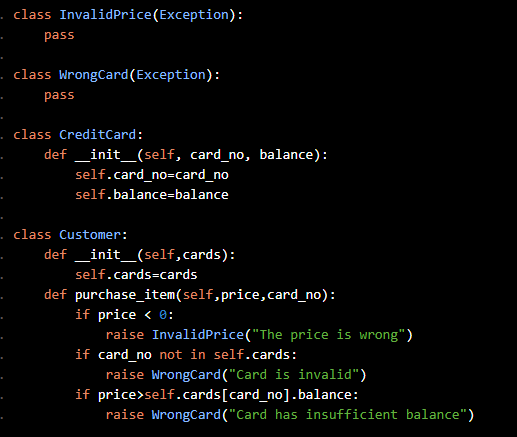


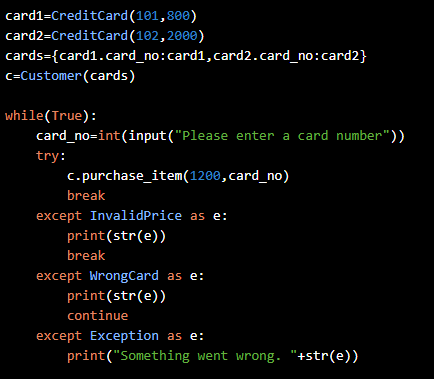


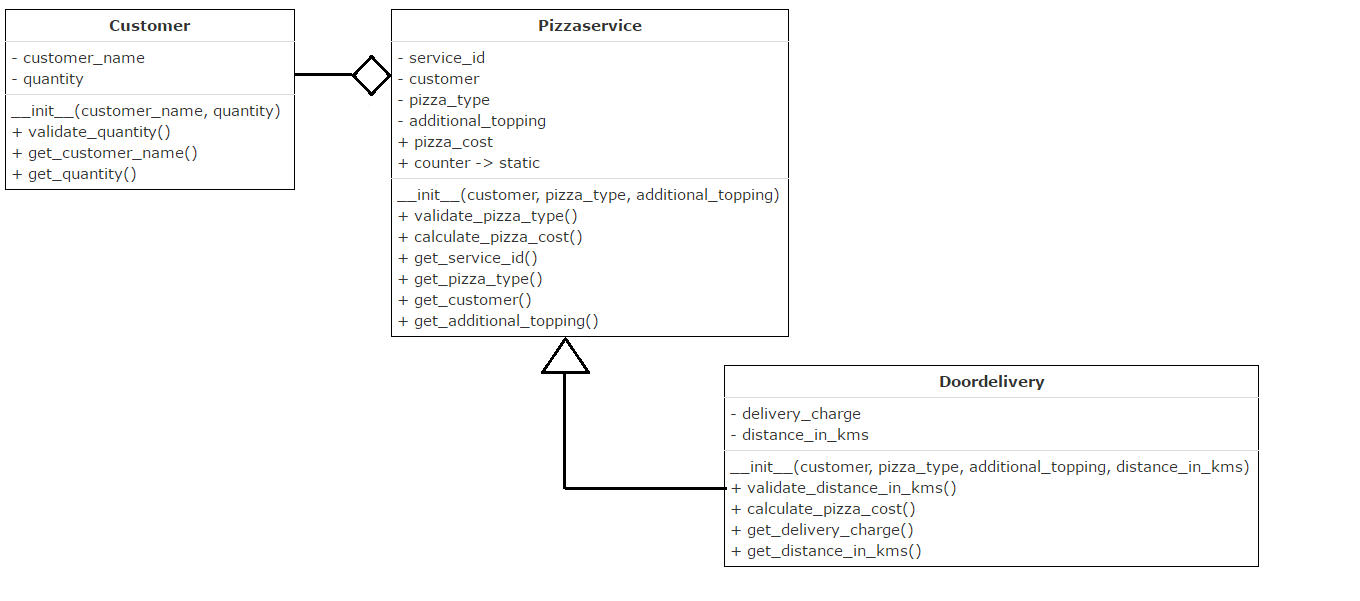
Custom Exception:

Exception is an inbuilt class in python and we can create our own exception by inheriting the Exception class as shown:

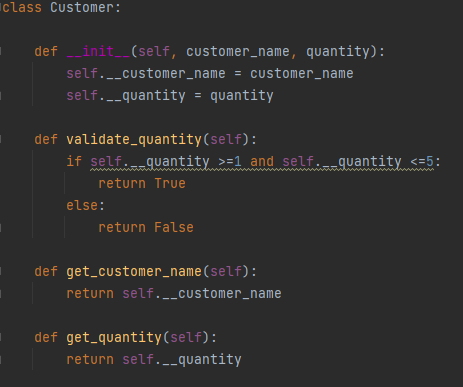


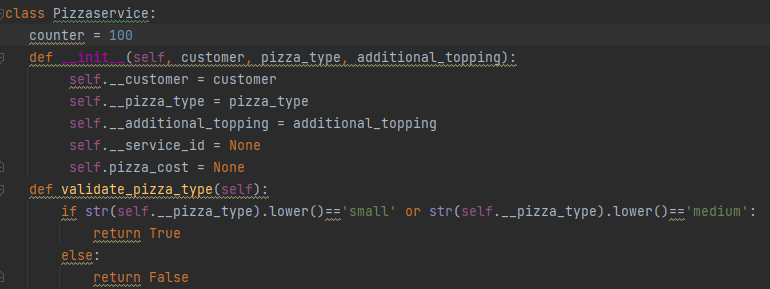


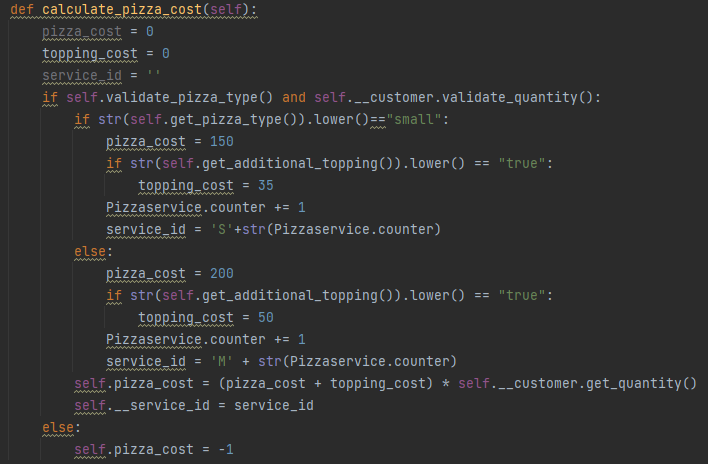


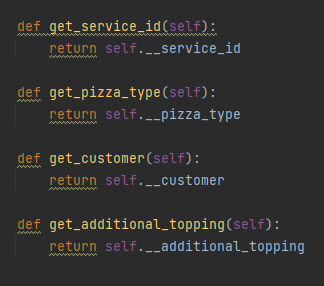


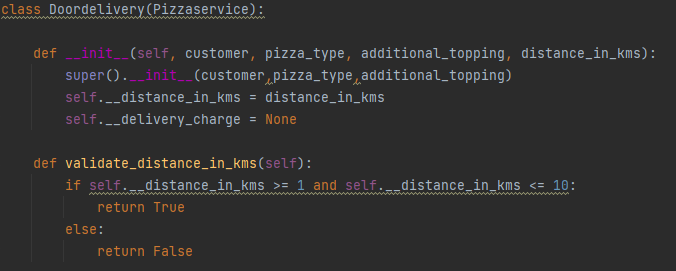
Note: Way to call inherited methods over child method

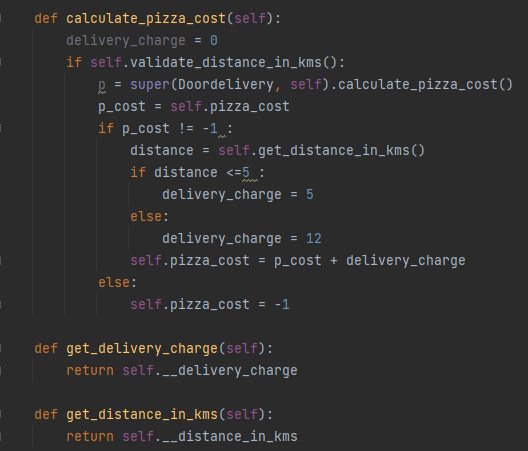












Essence:



Super(ParentClass, self).ParentMethod()