Scopes & NameSpaces

Object: In general, anything that can be assigned to a variable in Python is referred to as an object. Strings, Integers, Floats, Lists, Functions, Module etc. are all objects.

Namespaces: A namespace is a collection of currently defined names along with information about the object that the name references. It ensures that names are unique and won't lead to any conflict.

```
def greet_1():
    a = "Hello"
    print(a)
    print(id(a))

def greet_2():
    a = "Hey"
    print(a)
    print(id(a))

print("Namespace - 1")
greet_1()
print("Namespace - 2")
greet_2()
```

```
# Output is:
Namespace - 1
Hello
140639382368176
Namespace - 2
Hey
140639382570608
```

Types of namespaces:

Built-in Namespace: Created when we start executing a Python program and exists as long as the program is running. This is the reason that built-in functions like `id()`, `print()` etc. are always available to us from any part of the program.

Global Namespace: This namespace includes all names defined directly in a module (outside of all functions). It is created when the module is loaded, and it lasts until the program ends.

Local Namespace: Modules can have various functions and classes. A new local namespace is created when a function is called, which lasts until the function returns.

Scope of a Name:

In Python, the scope of a name refers to where it can be used. The name is searched for in the local, global, and built-in namespaces in that order.

Global variables: In Python, a variable defined outside of all functions is known as a global variable. This variable name will be part of Global Namespace.

```
x = "Global Variable"
print(x) # Global Variable

def foo():
```

```python
    print(x) # Global Variable

foo()
```

Local Variables: In Python, a variable defined inside a function is a local variable. This variable name will be part of the Local Namespace.

```python
def foo():
    x = "Local Variable"
    print(x) # Local Variable

foo()
print(x) # NameError: name 'x' is not defined
```

Local Variables & Global Variables:

```python
x = "Global Variable"

def foo():
    x = "Local Variable"
    print(x)

print(x)
foo()
print(x)
```

```
# Output is:
Global Variable
Local Variable
Global Variable
```

Modifying Global Variables: `global` keyword is used to define a name to refer to the value in Global Namespace.

```python
x = "Global Variable"

def foo():
    global x
    x = "Global Change"
    print(x)

print(x)
foo()
print(x)
```

```
# Output is:
Global Variable
Global Change
Global Change
```

Python Standard Library

The collection of predefined utilities is referred as the Python Standard Library. All these functionalities are organized into different modules.

Module: In Python context, any file containing Python code is called a module.

Package: These modules are further organized into folders known as packages.

Importing module: To use a functionality defined in a module we need to import that module in our program.

```
import module_name
```

Importing from a Module: We can import just a specific definition from a module.

```
from math import factorial
print(factorial(5)) # 120
```

Aliasing Imports: We can also import a specific definition from a module and alias it.

```
from math import factorial as fact
print(fact(5)) # 120
```

Random module: Randomness is useful in whenever uncertainty is required.
Example: Rolling a dice, flipping a coin, etc,.

`random` module provides us utilities to create randomness.

Randint: `randint()` is a function in random module which returns a random integer in the given interval.

```
import random
random_integer = random.randint(1, 10)
print(random_integer) # 8
```

Choice: `choice()` is a function in random module which returns a random element from the sequence.

```
import random
random_ele = random.choice(["A","B","C"])
print(random_ele) # B
```

Classes

Classes: Classes can be used to bundle related attributes and methods. To create a class, use the keyword `class`

```
class className:
    attributes
    methods
```

Self: `self` passed to method contains the object, which is an instance of class.

Special Method: In Python, a special method `__init__` is used to assign values to attributes.

```
class Mobile:
    def __init__(self, model):
        self.model = model
```

Instance of Class: Syntax for creating an instance of class looks similar to function call. An instance of class is an Object.

```
class Mobile:
    def __init__(self, model):
        self.model = model
```

```python
mobile_obj = Mobile("iPhone 12 Pro")
```

Class Object: An object is simply a collection of attributes and methods that act on those data.

```python
class Mobile:
    def __init__(self, model):
        self.model = model
    def make_call(self,number):
        return "calling..{}".format(number)
```

Attributes of an Object: Attributes can be set or accessed using `.` (dot) character.

```python
class Mobile:
    def __init__(self, model):
        self.model = model

obj = Mobile("iPhone 12 Pro")
print(obj.model) # iPhone 12 Pro
```

Accessing in Other Methods: We can also access and update attributes in other methods.

```python
class Mobile:
    def __init__(self, model):
        self.model = model

    def get_model(self):
        print(self.model) # iPhone 12 Pro

obj_1 = Mobile("iPhone 12 Pro")
obj_1.get_model()
```

Updating Attributes: It is recommended to update attributes through methods.

```python
class Mobile:
    def __init__(self, model):
        self.model = model

    def update_model(self, model):
        self.model = model

obj_1 = Mobile("iPhone 12")
obj_1.update_model("iPhone 12 Pro")
print(obj_1.model) # iPhone 12 Pro
```

Instance Attributes: Attributes whose value can differ for each instance of class are modelled as instance attributes.

Accessing Instance Attributes: Instance attributes can only be accessed using instance of class.

```python
class Cart:
  def __init__(self):
      self.items = {'book': 3}
  def display_items(self):
      print(self.items)  # {'book': 3}
```

```
a = Cart()
```

```
a.display_items()
```

Class Attributes: Attributes whose values stay common for all the objects are modelled as Class Attributes.

Accessing Class Attributes:

```
class Cart:
    flat_discount = 0
    min_bill = 100
    def __init__(self):
        self.items = {}

print(Cart.min_bill) # 100
```

Updating Class Attribute:

```
class Cart:
    flat_discount = 0
    min_bill = 100
    def print_min_bill(self):
        print(Cart.min_bill) # 200
a = Cart()
b = Cart()
Cart.min_bill = 200
b.print_min_bill()
```

Methods: Broadly, methods can be categorized as

Instance Methods
Class Methods
Static Methods

Instance Methods: Instance methods can access all attributes of the instance and have `self` as a parameter.

```
class Cart:
    def __init__(self):
        self.items = {}
    def add_item(self, item_name,quantity):
        self.items[item_name] = quantity
    def display_items(self):
        print(self.items) # {'book': 3}

a = Cart()
a.add_item("book", 3)
a.display_items()
```

Class Methods: Methods which need access to class attributes but not instance attributes are marked as Class Methods. For class methods, we send `cls` as a parameter indicating we are passing the class.

```
class Cart:
    flat_discount = 0
    @classmethod
    def update_flat_discount(cls, new_flat_discount):
        cls.flat_discount = new_flat_discount
```

```
Cart.update_flat_discount(25)
```

```python
print(Cart.flat_discount) # 25
```

Static Method: Usually, static methods are used to create utility functions which make more sense to be part of the class. `@staticmethod` decorator marks the method below it as a static method.

```python
class Cart:
   @staticmethod
   def greet():
       print("Have a Great Shopping") # Have a Great Shopping

Cart.greet()
```

Instance Methods	Class Methods	Methods
self as parameter	cls as parameter	No cls or self as parameters
No decorator required	Need decorator `@classmethod`	Need decorator `@staticmethod`
Can be accessed through object(instance of class)	Can be accessed through class	Can be accessed through class

OOPS

OOPS: Object-Oriented Programming System (OOPS) is a way of approaching, designing, developing software that is easy to change.

Bundling Data: While modeling real-life objects with object oriented programming, we ensure to bundle related information together to clearly separate information of different objects.

Encapsulation: Bundling of related properties and actions together is called Encapsulation. Classes can be used to bundle related attributes and methods.

Inheritance: Inheritance is a mechanism by which a class inherits attributes and methods from another class. Prefer modeling with inheritance when the classes have an `IS-A` relationship.

```python
class Product:
    def __init__(self, name):
        self.name = name

    def display_product_details(self):
        print("Product: {}".format(self.name)) # Product: TV


class ElectronicItem(Product):
    pass

e = ElectronicItem("TV")
e.display_product_details()
```

Super Class & Sub Class:

Superclass cannot access the methods and attributes of the subclass.
The subclass automatically inherits all the attributes & methods from its superclass.

```python
class Product:
   def __init__(self, name):
```

```python
        self.name = name
    def display_product_details(self):
        print("Product: {}".format(self.name)) # Product: TV

class ElectronicItem(Product):
    def set_warranty(self, warranty_in_months):
        self.warranty_in_months = warranty_in_months

e = ElectronicItem("TV")
e.display_product_details()
```

Calling Super Class Method: We can call methods defined in the superclass from the methods in the subclass.

```python
class Product:
    def __init__(self, name):
        self.name = name
    def display_product_details(self):
        print("Product: {}".format(self.name)) # Product: TV

class ElectronicItem(Product):
    def set_warranty(self, warranty_in_months):
        self.warranty_in_months = warranty_in_months

    def display_electronic_product_details(self):
        self.display_product_details()

e = ElectronicItem("TV")
e.display_electronic_product_details()
```

Composition: Modeling instances of one class as attributes of another class is called Composition. Prefer modeling with inheritance when the classes have an HAS-A relationship.

```python
class Product:
    def __init__(self, name):
        self.name = name
        self.deal_price = deal_price

    def display_product_details(self):
        print("Product: {}".format(self.name)) # Product: Milk

    def get_deal_price(self):
        return self.deal_price

class GroceryItem(Product):
    pass

class Order:
    def __init__(self):
        self.items_in_cart = []

    def add_item(self, product, quantity):
        self.items_in_cart.append((product, quantity))

    def display_order_details(self):
        for product, quantity in self.items_in_cart:
```

```
            product.display_product_details()

milk = GroceryItem("Milk")
order.add_item(milk, 2)
order.display_order_details()
```

Overriding Methods: Sometimes, we require a method in the instances of a sub class to behave differently from the method in instance of a superclass.

```
class Product:
    def __init__(self, name):
        self.name = name

    def display_product_details(self):
        print("Superclass Method")

class ElectronicItem(Product):
    def display_product_details(self): # same method name as superclass
        print("Subclass Method")

e = ElectronicItem("Laptop")
e.display_product_details()
```

```
# Output is:
Subclass Method
```

Accessing Super Class's Method: super() allows us to call methods of the superclass (Product) from the subclass. Instead of writing and methods to access and modify warranty we can override __init__.

```
class Product:
  def __init__(self, name):
      self.name = name

  def display_product_details(self):
      print("Product: {}".format(self.name)) # Product: Laptop

class ElectronicItem(Product):

  def display_product_details(self):
      super().display_product_details()
      print("Warranty {} months".format(self.warranty_in_months)) # Warranty 10 months

  def set_warranty(self, warranty_in_months):
      self.warranty_in_months = warranty_in_months

e = ElectronicItem("Laptop")
e.set_warranty(10)
e.display_product_details()
```

MultiLevel Inheritance: We can also inherit from a subclass. This is called MultiLevel Inheritance.

```
class Product:
    pass

class ElectronicItem(Product):
```

```python
    pass

class Laptop(ElectronicItem):
    pass
```

Inheritance & Composition:

Inheritance	Composition
Car is a vehicle	Car has a Tyre
Truck is a vehicle	Order has a product

Errors & Exceptions

Errors & Exceptions: There are two major kinds of errors:

Syntax Errors
Exceptions

Syntax Errors: Syntax errors are parsing errors which occur when the code is not adhering to Python Syntax.

```python
if True print("Hello") # SyntaxError: invalid syntax
```

When there is a syntax error, the program will not execute even if that part of code is not used.

Exceptions: Errors detected during execution are called exceptions.

Division Example: Input given by the user is not within expected values.

```python
def divide(a, b):
    return  a / b

divide(5, 0)
```

```
# Output is:
ZeroDivisionError: division by zero
```

Working With Exceptions:

Raising Exceptions:

```python
raise ValueError("Unexpected Value!!") # ValueError:Unexpected Value
```

```python
def divide(x, y):
    if y == 0:
        raise ValueError("Cannot divide by zero")
    return x / y

print(divide(10, 0))  # ValueError: Cannot divide by zero
```

Handling Exceptions: Exceptions can be handled with `try-except` block. Whenever an exception occurs at some line in try block, the execution stops at that line and jumps to except block.

```python
try:
 # Write code that
 # might cause exceptions.
except:
```

```
# The code to be run when
# there is an exception.
```

```
def divide(x, y):
    try:
        result = x / y
    except TypeError:
        return "Invalid input"
    return result
print(divide(10, 5)) # 2.0
print(divide(10, "a"))  # Invalid input
```

Handling Specific Exceptions: We can specifically mention the name of exception to catch all exceptions of that specific type.

```
try:
    # Write code that
    # might cause exceptions.
except Exception:
    # The code to be run when
    # there is an exception.
```

```
try:
    result = 5/0
    print(result)
except ZeroDivisionError:
    print("Denominator can't be 0")
except:
    print("Unhandled Exception")
```

```
# Output is:
Denominator can't be 0
```

Handling Multiple Exceptions: We can write multiple exception blocks to handle different types of exceptions differently.

```
try:
    # Write code that
    # might cause exceptions.
except Exception1:
    # The code to be run when
    # there is an exception.
except Exception2:
    # The code to be run when
    # there is an exception.
```

```
try:
    result = 12/"a"
    print(result)
except ZeroDivisionError:
    print("Denominator can't be 0")
except ValueError:
    print("Input should be an integer")
```

```python
except:
    print("Something went wrong")
```

```
# Output is:
Denominator can't be 0
```

Working With Dates & Times

Datetime: Python has a built-in `datetime` module which provides convenient objects to work with dates and times.

```python
import datetime
```

Datetime classes: Commonly used classes in the datetime module are :

1. date class
2. time class
3. datetime class
4. timedelta class

Representing Date: A date object can be used to represent any valid date (year, month and day).

```python
import datetime

date_object = datetime.date(2022, 12, 17)
print(date_object) # 2022-12-17
```

Attributes of Date Object:

```python
from datetime import date

date_object = date(2019, 4, 13)
print(date_object.year) # 2019
print(date_object.month) # 4
print(date_object.day) # 13
```

Today's Date: Class method `today()` returns a date object with today's date.

```python
import datetime

date_object = datetime.date.today()
print(date_object) # 2022-12-17
```

Representing Time: A time object can be used to represent any valid time (hours, minutes and seconds).

```python
from datetime import time

time_object = time(11, 34, 56)
print(time_object) # 11:34:56
```

Attributes of Time Object:

```python
from datetime import time

time_object = time(11, 34, 56)
```

```
print(time_object.hour) # 11
print(time_object.minute) # 34
print(time_object.second) # 56
```

Datetime: The `datetime` class represents a valid date and time together.

```
from datetime import datetime

date_time_obj = datetime(2018, 11, 28, 10, 15, 26)
print(date_time_obj.year) # 2018
print(date_time_obj.month) # 11
print(date_time_obj.hour) # 10
print(date_time_obj.minute) # 15
```

Timedelta: Timedelta object represents duration.

```
from datetime import timedelta

delta = timedelta(days=365, hours=4)
print(delta) # 365 days, 4:00:00
```

Calculating Time Difference:

```
import datetime

dt1 = datetime.datetime(2021, 2, 5)
dt2 = datetime.datetime(2022, 1, 1)
duration = dt2 - dt1
print(duration) # 330 days, 0:00:00
print(type(duration)) # <class 'datetime.timedelta'>
```

Formatting Datetime: The datetime classes have `strftime(format)` method to format the datetime into any required format like

mm/dd/yyyy
dd-mm-yyyy

```
from datetime import datetime

now = datetime.now()
formatted_datetime_1 = now.strftime("%d %b %Y %I:%M:%S %p")
print(formatted_datetime_1) # 05 Feb 2021 09:26:50 AM

formatted_datetime_2 = now.strftime("%d/%m/%Y, %H:%M:%S")
print(formatted_datetime_2) # 05/02/2021, 09:26:50
```

Parsing Datetime: The class method `strptime()` creates a `datetime` object from a given string representing date and time.

```
from datetime import datetime

date_string = "28 November, 2018"
print(date_string) # 28 November, 2018
```

```
date_object = datetime.strptime(date_string, "%d %B, %Y")
print(date_object) # 2018-11-28 00:00:00
```