Oops Notes:

**Object** - Real world entities. Almost everything in Python is an object, with its properties and methods.

**Class** – A cluster/group of similar type of objects. A Class is like an object constructor, or a "blueprint" for creating objects.

#create/define a class

class demo\_class():

a=10 #class attribute

#create a object

demo = demo\_class() #The demo variable now holds a reference to this newly created object.

print(demo.a)

"""Here, we're accessing the class attribute a using the object demo.

In Python, you can access attributes (variables or methods) of an object using dot notation (object\_name.attribute\_name).

So, demo.a accesses the attribute a of the object demo.

The output of this line will be 10, because a is assigned the value 10 in the class definition."""

**The \_\_init\_\_() Function**

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in \_\_init\_\_() function.

All classes have a function called \_\_init\_\_(), which is always executed when the class is being initiated.

Use the \_\_init\_\_() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

class Person:  
  def \_\_init\_\_(self, name, age):  
    self.name = name  
    self.age = age  
  
p1 = Person("John", 36)  
  
print(p1.name)  
print(p1.age)

## The \_\_str\_\_() Function

The \_\_str\_\_() function controls what should be returned when the class object is represented as a string.

If the \_\_str\_\_() function is not set, the string representation of the object is returned:

# The string representation of an object WITHOUT the \_\_str\_\_() function:

class Person:  
  def \_\_init\_\_(self, name, age):  
    self.name = name  
    self.age = age  
  
p1 = Person("John", 36)  
  
print(p1)

#The string representation of an object WITH the \_\_str\_\_() function:

class Person:  
  def \_\_init\_\_(self, name, age):  
    self.name = name  
    self.age = age  
  
  def \_\_str\_\_(self):  
    return f"{self.name}({self.age})"  
  
p1 = Person("John", 36)  
  
print(p1)

**Object Methods/Functions:**

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

Let us create a method in the Person class:

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()

## The self Parameter

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named self, you can call it whatever you like, but it has to be the first parameter of any function in the class:

# Use the words mysillyobject and abc instead of self:

class Person:  
  def \_\_init\_\_(mysillyobject, name, age):  
    mysillyobject.name = name  
    mysillyobject.age = age  
  
  def myfunc(abc):  
    print("Hello my name is " + abc.name)  
  
p1 = Person("John", 36)  
p1.myfunc()

**Play with Objects:**

1. Modify object property

Example: p1.age = 40

1. Delete object property

Example: del p1.age

1. Delete object as whole

Example: del p1

## The pass Statement

class definitions cannot be empty, but if you for some reason have a class definition with no content, put in the pass statement to avoid getting an error.

class Person:  
  pass

**Inheritance**

Inheritance allows us to define a class that inherits all the methods and properties from another class.

**Parent class** is the class being inherited from, also called base class.

**Child class** is the class that inherits from another class, also called derived class.

## Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

# Create a class named Person, with firstname and lastname properties, and a printname method:

class Person:  
  def \_\_init\_\_(self, fname, lname):  
    self.firstname = fname  
    self.lastname = lname  
  
  def printname(self):  
    print(self.firstname, self.lastname)  
  
#Use the Person class to create an object, and then execute the printname method:  
  
x = Person("John", "Doe")  
x.printname()

## Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

# Create a class named Student, which will inherit the properties and methods from the Person class:

class Student(Person):  
  pass

Note: Use the pass keyword when you do not want to add any other properties or methods to the class.

Now the Student class has the same properties and methods as the Person class.

# Use the Student class to create an object, and then execute the printname method:

x = Student("Mike", "Olsen")  
x.printname()

## Add the \_\_init\_\_() Function

So far we have created a child class that inherits the properties and methods from its parent.

We want to add the \_\_init\_\_() function to the child class (instead of the pass keyword).

Note: The \_\_init\_\_() function is called automatically every time the class is being used to create a new object.

# Add the \_\_init\_\_() function to the Student class:

class Student(Person):  
  def \_\_init\_\_(self, fname, lname):  
    #add properties etc.

When you add the \_\_init\_\_() function, the child class will no longer inherit the parent's \_\_init\_\_() function.

Note: The child's \_\_init\_\_() function overrides the inheritance of the parent's \_\_init\_\_() function.

To keep the inheritance of the parent's \_\_init\_\_() function, add a call to the parent's \_\_init\_\_() function:

class Student(Person):  
  def \_\_init\_\_(self, fname, lname):  
    Person.\_\_init\_\_(self, fname, lname)

Now we have successfully added the \_\_init\_\_() function, and kept the inheritance of the parent class, and we are ready to add functionality in the \_\_init\_\_() function.

## Use the super() Function

Python also has a super() function that will make the child class inherit all the methods and properties from its parent:

class Student(Person):  
  def \_\_init\_\_(self, fname, lname):  
    super().\_\_init\_\_(fname, lname)

By using the super() function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

**Add Properties**

# Add a property called graduationyear to the Student class:

class Student(Person):  
  def \_\_init\_\_(self, fname, lname):  
    super().\_\_init\_\_(fname, lname)  
    self.graduationyear = 2019

In the example below, the year 2019 should be a variable, and passed into the Student class when creating student objects. To do so, add another parameter in the \_\_init\_\_() function:

# Add a year parameter, and pass the correct year when creating objects:

class Student(Person):  
  def \_\_init\_\_(self, fname, lname, year):  
    super().\_\_init\_\_(fname, lname)  
    self.graduationyear = year  
  
x = Student("Mike", "Olsen", 2019)

**Add Methods/Functions**

# Add a method called welcome to the Student class:

class Student(Person):  
  def \_\_init\_\_(self, fname, lname, year):  
    super().\_\_init\_\_(fname, lname)  
    self.graduationyear = year  
  
  def welcome(self):  
    print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

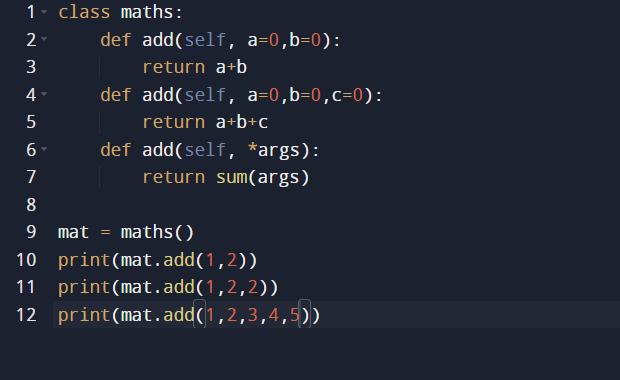
**Polymorphism**

Polymorphism in Python refers to the ability of different objects to respond to the same method or operation in a manner appropriate to their respective types. This concept allows for a unified interface to interact with objects of different classes, facilitating flexibility and code reusability. For instance, both a `Dog` class and a `Cat` class might have a `make\_sound` method, but the implementation of this method will differ between the two classes. When you call `make\_sound` on an instance of either class, Python will invoke the appropriate method based on the object's type, demonstrating polymorphism. This principle is central to object-oriented programming and enables functions and methods to operate on objects of various types seamlessly, provided they share the same interface or method names.

**Compile-time Polymorphism**

Compile-time polymorphism, also known as static polymorphism, occurs when the method to be invoked is determined at compile time. This is typically achieved through method overloading, where multiple methods share the same name but differ in their parameter lists within the same scope. Although Python does not support traditional method overloading as seen in statically typed languages like Java or C++, it can be mimicked using default arguments or variable-length arguments (\*args and \*\*kwargs). Since the binding of the method call to its definition happens at compile time, it allows for type checking and validation before runtime, reducing runtime errors and potentially optimizing performance.

Example:



**Run-time Polymorphism**

Run-time polymorphism, also known as dynamic polymorphism, occurs when the method to be invoked is determined during the program's execution. This is achieved through method overriding, where a subclass provides a specific implementation of a method that is already defined in its superclass. In Python, this is a core aspect of its object-oriented capabilities. When a method is called on an object, Python uses the object's class type to determine which method implementation to execute. This allows for more flexible and extensible code, as new subclasses can be created with specialized behavior without modifying the existing codebase, thereby adhering to the open/closed principle.

Example:

