Object oriented programming

An integer is a set in mathematics. This set has whole numbers. 0 belongs to this set. If x belong to this set, so also x + 1 and x – 1.
On this set, we can do the following operations.
Add
Subtract
Multiply
All these result in integers.
We divide two integers the result may not be integer.
A set like integer specifies what it contains and what we can do with them. Such a set in programming is called a type.
A type specifies what it contains and what we can do with its elements.
What is the result of 25 + 36? It is definitely 61. How do we add? Table lookup? Add digits from right to left and propagate carry? Add row wise? Use tally marks? Use fingers and toes? Set of integer which is a type says what the result of an operation is, but does not force how to add.
A pure type specifies 'what' and not 'how'.
'what' specifies an interface.
'how' specifies the implementation.

A language a few types. It can not provide all possible types we may want to have – like desks, projector, chalkpiece. A language should provide a mechanism to make our own type. That is called a class. A class is a type and implementation. A variable of the class type is called an object.

This brief discussion slated above should be sufficient for our course.

We shall look at a few examples and add a few more points about objects as we go along.

Let us look at code from the file 0_intro.py.

```
class Ex0:
	pass
```

```
print(Ex0, type(Ex0)) #<class '__main__.Ex0'> <class 'type'>
```

So, Ex0 is a type in the package __main__.

A class can have its attributes and behaviour.

```
# a class can have functions(behaviour)class Ex1:
    def foo():
        print("foo of Ex")
Ex1.foo()
```

The function(behaviour) foo belongs to Ex1 and is invoked using the class name.

```
# a class can have attributes(fields or variables)
class Ex2:
    a = "test"
print(Ex2.a)
```

The attribute a with the value "test" belongs to Ex2 and is accessed using the class name.

```
# we can create variables of Ex3 type : that is called an instance or an object
class Ex3:
    pass
a = Ex3()
print(a, type(a))
```

```
# When an object is created, a special function of the class is called for initializing.
# That is called a constructor. The name of the constructor in Python is __init__
# gives an error
class Ex4:
	def __init__():
		print("constructor called")
a = Ex4()
```

```
# a = Ex4() conceptually becomes Ex4.__init__(a)
# The instance is passed as the first argument to the constructor(ctor).
# Even though the argument is implicit, in Python, we require an explicit parameter.
# This can be given any name - is normally called self.
class Ex4:
	def __init__(self):
		print("constructor called")
		print('self : ', self)
a = Ex4()
print('a : ', a)
```

Observe that the both the outputs have the same value indicating self and a refer to the same object.

# an object can have attributes. These are normally added and initialized in the constructor.

```
class Rect:
	def __init__(self, l, b):
		self.length = l
		self.breadth = b
		# print(length) # error
		# all names should be fully qualified
```

```
        print("__init__", self.length, self.breadth)

# initialize a Rect object r1 with length 20 and breadth 10
r1 = Rect(20, 10)
print(r1.length, r1.breadth)
```

Also, observe that these attributes can be accessed any where with a fully qualified name. In some languages, we have access control(private, public ...). We do not have them in Python.
We can create any number of objects - we have to invoke the constructor to initialize them.

```
# Integers support functions like __add__.
# Strings support functions like upper.
# There are called behaviours of that particular type
# Our class can also have behaviour through functions in the class.

# Rect.area
# An object of rectangle contains length and breadth.
# Given the rectangle, we can extract the length and the breadth to find the area.
# So, the function area is invoked with an object and does not require any other parameter.
```

The object encapsulates both attributes and behaviour. Encapsulation is putting together data and functions(attributes and behaviour).

```
class Rect:
    def __init__(self, l, b):
        self.length = l
        self.breadth = b
    def area(self):
        return self.length * self.breadth

# initialize a Rect object r1 with length 20 and breadth 10
r1 = Rect(20, 10)
r2 = Rect(40, 30)
print('area of r1 : ', r1.area()) # Rect.area(r1)
print('area of r2 : ', r2.area()) # Rect.area(r2)
```

```
# let us look at a few more behaviours
# - change length : requires the new length apart from the object
# - we require one explicit argument and two explicit parameters -
#   the first parameter always refers to the object through which the call is made.

# Rect.change_length, Rect.change_breadth
class Rect:
    def __init__(self, l, b):
        self.length = l
        self.breadth = b
    def area(self):
        return self.length * self.breadth
    def change_length(self, l):
        self.length = l
    def change_breadth(self, b):
        self.breadth = b

    def disp(self):
        print('length : ', self.length)
        print('breadth : ', self.breadth)

r1 = Rect(20, 10)
print("before change length ")
```

```
r1.disp()
r1.change_length(40)
print("after change length ")
r1.disp()

print("before change breadth ")
r1.disp()
r1.change_breadth(30)
print("after change breadth ")
r1.disp()
```

This completes a simple introduction to programming with objects.

A cat is a mammal. Tiger is a cat. House cat is a cat. Lion is a cat. All these have common characteristics. We can address all of them as cats. They walk on their toes. They climb trees. There could be differences in the way they walk or they climb. Because of common characteristics, we can consider a tiger or a lion as a cat. This helps in maintenance of programs.
We call this “is a” relationship between classes as inheritance.

```
class P2D:
        def __init__(self, x, y):
                self.x = x
                self.y = y
        def disp(self):
                print ("x : ", self.x)
                print ("y : ", self.y)

p2 = P2D(3, 4)
p2.disp()
```

```
# P3D inherits from P2D
# an object of P3D gets everything P2D has
class P3D(P2D) :
	def __init__(self, x, y, z):
		P2D.__init__(self, x, y)
		self.z = z

p3 = P3D(11, 12, 13)
p3.disp() # no function in the derived class; calls the base class function by default
print(isinstance(p3, P2D))  #True
```

A point in three dimensions(class P3D) is also a point in two dimensions(P2D).

```
class P3D(P2D) :
```

The above statement indicates that relationship. We say that P3D is a derived class and P2D is the base class.

An object of derived class with have a reference to an embedded object of base class – normally referred to as the base class sub object.

When we create an object of the derived class, we invoke the constructor of the derived class which in turn calls the base class constructor on the base class sub object.

The base class provides a default implementation of the behaviour for the derived class.

```
p3.disp() # no function in the derived class; calls the base class function by default
```

As there is no function called disp in the derived class, the base class function gets called.

```
class P3D(P2D) :
	def __init__(self, x, y, z):
		P2D.__init__(self, x, y)
		self.z = z
	#overriding the base class function
	def disp(self):
		P2D.disp(self) # delegate work to the base class
		print("z : ", self.z)
```

It is possible for the derived class to provide its own function disp. This concept is called overriding. The derived class modifies the function of the base class. In many such cases, this overriding function may in turn call the base class function.

We have discussed a relationship between classes – inheritance.
Inheritance is “is a” relationship between classes. A cat is a mammal. A tiger is a cat and so on.
We shall not discuss a relationship between objects – composition.
Composition is “has a” relationship between objects.
A cat has paws.
In the example below, MyEvent object has MyDate object as part of it. An Event occurs on a date. An event is not a date. A date is not an event. An event has a date as part of it. So observe that the constructor of MyEvent class initializes a field called date by creating and initializing an object of MyDate class.
All functions of the MyEvent class in turn invoke the functions of MyDate class through the attribute date – this is called delegation or forwarding.
We have been using the function str on different types like int. When we invoke this function, a special function __str__ of that class will be called. We can also provide such a function in our class to convert an object of our class to a string. In the print context, this function gets called automatically.

```
# filename : 2_composition.py

# composition
class MyDate:
	def __init__(self, dd, mm, yy):
		self.dd = dd
		self.mm = mm
		self.yy = yy
	def __str__(self):
		return str(self.dd) + "-" + str(self.mm) + "-" + str(self.yy)
```

```
    def key(self):
        return self.yy * 365 + self.mm * 30 + self.dd

d = MyDate(15, 8, 1947)
print(d)

class MyEvent:
    def __init__(self, dd, mm, yy, detail):
        self.date = MyDate(dd, mm, yy)
        self.detail = detail
    def __str__(self):
        return str(self.date) + " => " + self.detail

    def key(self):
        #return self.detail
        return self.date.key()

e = MyEvent(15, 8, 1947, "Independence Day")
print(e)
```

The rest of the code creates a list of user defined objects – creates a list of events.
Then the list is sorted and displayed using the for statement.
We would like sort these dates based on the date. We provide keyword parameter key which converts the date into a single number. The callback MyEvent.key in turn delegates the work to MyDate.key. This function converts a given date to a single integer by using some approximate formula to count the number of days from the beginning of the era. The sorted function converts each event into this number and sorts the events based on this number.

```
mylist = [
    MyEvent(26, 1, 1956, "Republic Day"),
    MyEvent(1, 11, 1973, "Karnataka born"),
    MyEvent(2, 10, 1868, "Gandhi jayanthi"),
    MyEvent(15, 8, 1947, "Independence Day"),
    MyEvent(16, 12, 1971, "Amar sonar bangla")
]

for e in sorted(mylist, key = MyEvent.key):
    print(e)
```

The number of students in a class is not an attribute of any one particular student. It is an attribute of the class itself. In such cases, we create attributes outside of the functions with in the class. In some languages, these are called static fields or variables of the class.
We can access this variable using the classname or an object name as long as the object does not have a field by the same name.
To count the number of objects, we can count up in the constructor. We know that the constructor is always called when an object is created.
Similarly when an object dies, a function call the destructor will be called. In Python, the name of this function is __del__.
In the destructor, we can count down.
The destructor is called when an object dies or when the object type is changed or when del is called on the object explicitly. But remember that objects are also reference counted like any other entity in Python. The destructor gets called only when the reference count becomes zero.

We may want to invoke a function to display or play with a static attribute. Such a function should be callable using the class name or an object without passing object as an implicit parameter. So, the function should have no parameters. We achieve by using the following magic.

```
    @staticmethod
    def disp_count():
        print("disp_count : count : ", MyDate.date_count)
```

@staticmethod will allow us to call disp_count using an object or a classname. These is no self.

```
# filename: 3_static.py

# count the number of objects
class MyDate:
        date_count = 0
        def __init__(self, dd, mm, yy):
                self.dd = dd
                self.mm = mm
                self.yy = yy
                MyDate.date_count += 1
        def __str__(self):
                return str(self.dd) + "-" + str(self.mm) + "-" + str(self.yy)

        def __del__(self):
                MyDate.date_count -= 1

        # this notation is used to indicate that the function should be changed
from an object
        # function to a static function - this does not have any parameter - no
self.
        @staticmethod
        def disp_count():
                print("disp_count : count : ", MyDate.date_count)
# count the number of dates
# is a property of the class and not any date object
# such an attribute is called a class attribute
print("count : ", MyDate.date_count)
d1 = MyDate(11, 11, 11)
print("count : ", MyDate.date_count)
d2 = MyDate(12, 12, 12)
print("count : ", MyDate.date_count)
# should decrease when the object is destroyed.
# a special function called destructor will be called - name __del__
# we can take care of resources in this function
del d2
```

```
print("count : ", MyDate.date_count)

# we could have a function to display static attributes
# called static function; there do not depend on any object of the class.
# this is a class behaviour and not object behaviour
MyDate.disp_count()
```

Masala Dosa does not grow on trees. You may think that it can be plucked from a tree. Nothing wrong. We should be able to support your way of thinking.
A rectangle has length and breadth. We can compute the area given an object of rectangle. The user may want to look upon area as an attribute. We can support it using the concept called property.
d1 is an object of MyDate class.
The functions get_dd gets us the date of MyDate object and set_dd changes the date of MyDate object.
We create a property called DD and associate in our example with a pair of functions. We can use expression DD of MyDate object either to the right of assignment or to the left of assignment. These are called rvalue and lvalue usages of that field.
When we use the expression d1.DD as an rvalue, the first function in the pair is called. So d1.DD becomes d1.get_dd().
d1.DD = 22
When we use the expression d1.DD as a lvalue, the second function in the pair is called. So this statement above becomes d1.set_dd(22).
A property allows the user to consider an entity as an attributes of a class.
This is the way we support the user’s view.

```
# get and set
class MyDate:
	date_count = 0
	def __init__(self, dd, mm, yy):
		self.dd = dd
		self.mm = mm
```

```
            self.yy = yy
        def get_dd(self):
            return self.dd
        def set_dd(self, dd) :
            self.dd = dd
        DD = property(get_dd, set_dd)

d1 = MyDate(11, 11, 11)
print(d1.DD)
d1.DD = 22
print(d1.DD)
```

This is a brief introduction to Object Oriented Programming in Python.