OBJECT ORIENTED PROGRAMMING

Brief Recap

- More on Lists
  - *Comparing List and String Objects*
  - *Aliasing*
- Introduction to Object Oriented Programming
  - *Data Abstraction*
  - *Python Classes*

Menu for Today!

- Classes, Objects and Methods
- Polymorphism
- Inheritance

Class of 2D points

```
import math

class point:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def distance(self, other):
        return ((self.x-other.x)**2 + (self.y-other.y)**2)**0.5
```

Class of 2D points

```
def getPolar(self):
    if (self.x > 0):
        theta = math.atan(self.y / self.x)
    elif (self.x < 0):
        theta = math.atan(self.y / self.x) + math.pi
    else:
        theta = math.pi
    r = (self.x**2 + self.y**2)**0.5
    return(r, theta)
```

Creating a Class

- A class is user defined type
  - *Has its own attributes*
  - *Attributes include methods to manipulate objects of the class*
  - *To access the methods, we need to first create an object (instance) of the class*
  - *Attributes and methods of the object can be accessed using the dot (.) operator*

```
class point:
# list attributes and methods
```

What are Attributes

- Data and procedures that “belong” to the class
- Data attributes
  - *think of data as other objects that make up the class*
  - *for example, a coordinate is made up of two numbers*
- Methods (procedural attributes)
  - *think of methods as functions that only work with this class*
  - *how to interact with the object*
  - *for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects*
- Attributes of the instance can be accessed as `self.attributeName` within the class
  - *Thus* `self.x` *refers the x coordinate of the point*

Creating an Object of A Class

- The following code create an object (or an instance) of class `point`

```
p1 = point(1,2)
```

- The attributes of the object can be accessed using the dot notation.

```
p1.x
p1.y
```

Initializing an Object

- Whenever an instance of a class is created, a special method called `__init__` is automatically called.
- The first parameter of `__init__` is always `self`; `self` refers to the instance being created
  - *We don't need to provide argument for* `self`*, Python does this automatically*
  - `self` *is not a keyword;* `self` *is a convention*
- This method initializes the attributes of the instance
- We can also give default values to attributes in `__init__`

```
def __init__(self, x = 0,y = 0):
        self.x = x
        self.y = y
origin = point()
```

Init (always) creates a New Object

```
myPoint = point(1,2)
myPoint1 = point(2,1)
myPoint == myPoint1
```

- Evaluates to `False` as expected

```
myPoint2 = point(1,2)
myPoint == myPoint2
```

- Also evaluates to `False`!!! Why?

```
id(myPoint)
id(myPoint2)
```

Methods

- Methods are attributes which are functions
  - *These functions work only with this class*
- Python always passes the object as the first argument
  - *We don't need to do this explicitly; Python will take care of it for us*
  - *Convention is to use self as the name of the first argument of all methods*
- As we saw earlier, the "." operator is used to access any attribute
  - *a data attribute of an object*
  - *a method of an object*

Invoking Methods

- Methods behave just like functions
  - *take params, do operations, return*
  - *other than self and dot notation*
- We can invoke the method either as `class.method` or as `object.method`
  - *If we use* `object.method` *format, self is an implicit argument*
  - *If we use* `class.method`*, the self parameter must be explicitly passed*

```
origin = point()
point1 = point(1,2)
point1.distance(origin)
point.distance(point1, origin)
```

Printing an Object

- Every class has a default print function
  - *Rather uninformative*

```
print(a)
<point object at 0x107ed75b0>
```

- To do something meaningful, define a method `__str__`

```
def __str__(self):
    return(f'<{self.x},{self.y}>')
a = point(1,2)
print(a)
<1,2>
```

Special Methods in Python

- Did you notice double underscores (`__`) before and after some method names : `__init__` and `__str__`
- These are special methods that can be invoked using special syntax
  - *arithmetic operations, comparisons, subscripting and slicing*

```
__add__(self, other)  → self + other
__sub__(self, other)  → self - other
__lt__(self, other) → self < other
__len__(self) → len(self)
__str__(self) → print self
__getitem__(self, i) → self[i]
```

Polymorphism

- Polymorphism : poly (many) + morphos (forms)
  - *capability of existing in different forms*
- Polymorphism in programming: the same function name (but different signatures) being used for different types.
  - `int + int → int`
  - `int + float → float`
  - `float + float → float`
  - `string + string → string`
- Operators like + , == and index ([.]) have different meanings depending on the type of object that they are applied to
  - *Such operator are implemented with special methods of the class*
  - *If the class has the underlying special method defined (*`__add__`, `__getitem__`*), then the corresponding syntax can be used on an object of that class (*`a + b`, `a[i]`*)*
  - `x + 3` *and* `x.__add__(3)` *have exactly the same effect*

One More Example: Person

```
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last
    def __str__(self):
        return self.first + ‘ ‘ + self.last
somePerson = person(‘Rishi’, ‘Dutt’)
```

Child Classes

- We are interested in students, not persons
  - *Every student is also a person*
- `Student` is said to be a child class of `Person`
  - *Child classes model "is a" relationships. Every* `Student` ***is-a*** `Person`
  - *Every student "inherits" all methods associated with a person*

```
class Student(Person):
    pass
student1 = student('Rishi', 'Dutt')
print(student1)
```

Additional Attributes in a Child Class

- A Student are a Person who has an additional attribute: Roll Number

```
class Student(Person):
    def __init__(self, first, last, rollNo):
        self.rollNo = rollNo
```

- Now try

```
student1 = Student('Rishi', 'Dutt', 'IMT2021001')
print(student1)
```

Enabling Inheritance in __init__

- Since we defined `Student.__init___`, the initiation of `Person` attributes are no longer inherited
  - *We need to explicitly call* `Person.__init__`

```
class Student(Person):
    def __init__(self, first, last, rollNo):
        Person.__init__(first, last)
        self.rollNo = rollNo
```

- Again try

```
student1 = Student('Rishi', 'Dutt', 'IMT2021001')
print(student1)
```

Manipulating Student Data

```
class Student(Person):
    def __init__(self, first, last, rollNo):
        Person.__init(first, last)
        self.rollNo = rollNo
   def getGradYear(self):
        if rollNo[0:3] == ‘IMT’:
            gradYear = int(self.rollNo[3:7]) + 5
        else:
            gradYear = int(self.rollNo[2:6]) + 2
        return gradYear
```

Associating Methods with Classes

- `getGradYear` is a custom method associated with class `Student`

```
student1 = Student("Rishi", "Dutt", "IMT2021001")
print(student1.getGradYear())
```

Class Variables

- Suppose we want to know how many students there are
  - *Create a "class variable"* `count` *inside the class* `Student`
  - This variable is shared by all instances of the class
  - Increment this variable every time we create a new instance of Student