Weeks 10 and 11

Chapter 1

Object-Oriented Programming (OOP)

This is a very broad topic: we will only cover a few critical concepts to help us understand OOP in Python.

Throughout, we will use two running examples to illustrate the concepts: a point in the two-dimensional Euclidean plane and a triangle made up of three such points.

- Classes, Objects and Constructors
- Instance Attributes and Methods
- Special Methods
- Iterators and Generators
- Simple Inheritance

Chapter 2

Classes, Objects and Constructors

A class is essentially a **datatype**: it consists of a domain (the possible **object instances** of the class) coupled with the different operations (or **methods**) that can be used with the objects.

In Python, a class definition provides the framework for specifying a way to **construct** new object instances and to delineate all the **attributes** of such instances, including the **methods** that the objects can use to interact with the world.

The entry-point into the definition is the `__init__` method (Python classes have several predefined hooks for such *special methods*: their names begin and end with **two underscores** and hence, are often referred to informally as *double-under* or *dunder* methods).

Here is an example of a class for creating points in 2D Euclidean space: we will build up the definition piece-by-piece to illustrate several key features/principles of the OOP paradigm.

```python
import math
class Point:
    def __init__(self, x, y):
        """Defines a 2D Euclidean point
        """
        self.x = float(x)
        self.y = float(y)
```

Chapter 3

Instance Attributes and Methods

So far, the definition just tells us how to construct a new `Point` instance via the **constructor** method `__init__`. This is an *instance* method: every instance including a nascent one that is just coming to life passes an *implicit* first argument (traditionally named `self`) to such a function. In addition, the function specifies two arguments that are used to **set** the **instance attributes** `x` and `y`, the Cartesian coordinates of the point.

The attributes `x` and `y` belong to the instance: they can be used within the class definition, for example, if we wanted to define *polar coordinates* (r, θ) for the point with $r = \sqrt{x^2 + y^2}$ and $\sim\theta = \arctan y/x$. We can do this by using a method that provides the polar coordinates:

```
class Point:
    def __init__(self, x, y):
    ...

    def get_polar(self):
        """Returns polar coordinates (r, theta)"""
        return (math.sqrt(self.x ** 2 + self.y ** 2), math.atan(self.y / self.x))
```

Now, a `Point` object has two **public** attributes `x` and `y` and one **public** method `get_polar`. Any code can access these via a `Point` object.

Note

Python does not have the same notion of *privacy* or *information-hiding* as in other OOP langauges: to indicate that some attribute (or method) is **non-public** (i.e., should not be used by other code), Python uses the *convention* of naming it with a name that starts with an underscore (`_`).

For example, we could modify the current definition so that direct access to the coordinates is discouraged, but is provided through so-called *getter* and *setter* methods:

```python
from math import sqrt, atan
class Point:
    def __init__(self, x, y):
        self.set_x(x)
        self.set_y(y)

    def get_polar(self):
        """Returns polar coordinates (r, theta)"""
        r = sqrt(self.__x ** 2 + self.__y ** 2)  # distance from origin
        theta = atan(self.__y / self.__x)  # angle  the x-axis
        return r, theta

    def set_x(self, x):
        self.__x = float(x)

    def get_x(self):
        return self.__x
    ...
```

Any code that tries to access the `__x` or `__y` attributes will raise an `AttributeError` exception, which is Python's way of indicating that these are non-public. Note that the attributes can still be used by **other parts of the class code** without a problem.

! Important

Everything in Python is an object! This includes classes themselves: for instance, `list`, `int`, `function` etc. are classes but they are also *class objects*. Be sure to appreciate the distinction between *class instances* (members of the domain of a class, e.g. the integers 0, 1, 2 etc. that are instances of `int`) and *class objects* (e.g., the datatype `int`).

Chapter 4

Special Methods

There are some methods that can be re-defined in a class definition to ensure that the corresponding protocol is executed correctly. For example, it would be natural to **print** a meaningful *string representation of a `Point` object and this can be done by re-defining the `__str__` method: the protocol is that when an object has to be printed, its `__str__` method is consulted to produce the desired string that needs to be output to the standard output!

In the same way, it would be meaningful to *add* or *subtract* two `Point` objects using the `+` or `-` arithmetic operators but ensuring that the operations *mean* that the `x` and `y` coordinates are added (or subtracted) to yield a new `Point`!

```
class Point:
    def __init__(self, x, y):
        ...

    def get_polar(self):
        ...

    def set_x(self, x):
        ...

    def get_x(self):
        ...

    def __str__(self):
        """Returns a printable representation"""
        return f'Point (x={self.__x}, y={self.__y})'

    def __add__(self, other):
        """Returns a new Point that is the sum of self and other"""
        return Point(self.__x + other.get_x(), self.__y + other.get_y())
```

We can now use these methods as follows:

```
point_1 = Point(2.0, 5.0)
point_2 = Point(3.0, -1.0)
print(point_1)  # should print Point(x=2.0, y =5.0)
print(point_1 + point_2)  # should print Point(x=5.0, y=4.0)
```

Chapter 5

Iterators and Generators

We now come to one of the coolest features of Python: a standard **iteration** protocol for stepping through an iterable collection. The protocol works as follows:

- Applying the `iter` function to the collection, we get an **iterator** object.
- The iterator is like a *cursor*: on demand, it steps to the next element from the iterable. This is done by calling the built-in `next` function *on the iterator object*.
- when no further elements are available, the `next` function raises the `StopIteration` exception.

Study the following example to see what really happens under the hood when we run a `for` loop!

```
>>> z = [2.0, 'x']
>>> z_iter = iter(z)   # initialize an iterator over the list
>>> next(z_iter)   # get the first element from the list
2.0
>>> next(z_iter)   # then, the next one
'x'
>>> next(z_iter)   # Nothing left in the list, so signal the end of iteration!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> for element in z:   # the entire protocol from start to finish.
    element
2.0
'x'
```

In summary, a `for` loop is designed to run the iteration protocol on the target iterable: the target variable successively takes on the `next` function's return values. When the `StopIteration` exception is raised, the loop quits!

Any class whose instances are meant to be *collections* can be **programmed to implement the iteration**

protocol as follows:

- In the class, define an `__iter__` method to return an iterator object.
- ensure that **the iterator object's class** implements a `__next__` method that correctly advances through the elements of the collection in the appropriate manner.

We can illustrate this using a `Triangle` class whose instances each *contain* three `Point`s making up the three corners of the triangle. Now, suppose that we want the iterator for the class to step through the corners of the `Triangle` in *non-decreasing order of distance from the origin.* So, for example, if the corners `x`, `y`, and `z` of a `Triangle` object are located at coordinates (5, -1), (-1, 1) and (2, 2) respectively, we then wish to iterate through them in the order `y`, then `z` and finally `x`. We will show three different versions of the `Triangle` class to illustrate this.

5.1 Triangle: Version 1

The iterator object is the triangle instance itself, and hence we need to have the `Triangle` class define both an `__iter__` and `__next__` method! The `__iter__` method ensures that we first set up a non-public list of corners sorted by distance (the `get_polar` method for `Point` objects returns a tuple with its first component being the distance from the origin).

```
class Point:
    ...

class Triangle:  # Version 1
    def __init__(self, pt1, pt2, pt3):
        """Initializes three Point objects to be the corners"""
        self.corners = [pt1, pt2, pt3]

    def __iter__(self):
        self.__cursor = -1
        self.__sorted = sorted(self.corners,
                               key=lambda pt: pt.get_polar()[0])
        return self

    def __next__(self):
        self.__cursor += 1
        if self.__cursor == 3:
            raise StopIteration
        return self.__sorted[self.__cursor]

trng = Triangle(Point(5, -1), Point(-1, 1), Point(2, 2))
for point in trng:
    print(f'{point} is at distance {point.get_polar()[0]} from origin')
```

5.2 Triangle: Version 2

If we instead make use of the builtin `enumerate` function to have `__iter__` return an *enumeration* of the list of corners (after sorting the list in place by distance to the origin), then the `Triangle` class does not need to implement its own `__next__` method. The enumeration already has the loop protocol built into it!

Notice, however, the subtle change in the target variable in the loop: since the iterator is an enumerate object, the variable is a tuple made up of the index and the triangle corner.

```
class Triangle:  # Version 2
    def __init__(self, pt1, pt2, pt3):
        """Initializes three Point objects to be the corners"""
        self.corners = [pt1, pt2, pt3]

    def __iter__(self):
        sorted(self.corners, key=lambda pt: pt.get_polar()[0])
        return enumerate(self.corners)

trng = Triangle(Point(5, -1), Point(-1, 1), Point(2, 2))
for _, point in trng:
    print(f'{point} is at distance {point.get_polar()[0]} from origin')
```

5.3 Generator functions and expressions

A **generator** function creates a **lazy iterator**: it is especially useful in cases where we need to iterate through a large (even infinite) collection without explicitly needing the entire collection to be available.

The `generated` object implements the iteration protocol as follows. The loop in the main body of the generator function is essentially like the `__next__` function in the iteration protocol, except that when a `yield` statement is encountered, it sends the calling code the object that is yielded and then *suspends itself* until the next (lazy) iteration. Here is an example.

5.3.1 Triangle: Version 3

```
def corner_genrtr(triangle):
    self.__sorted = sorted(triangle.corners,
                           key=lambda pt: pt.get_polar()[0])
    for corner in self__sorted:
        yield corner

class Triangle:  # Version 3
    def __init__(self, pt1, pt2, pt3):
        """Initializes three Point objects to be the corners"""
        self.corners = [pt1, pt2, pt3]
```

```python
    def __iter__(self):
        return corner_genrtr(self)

trng = Triangle(Point(5, -1), Point(-1, 1), Point(2, 2))
for point in iter(trng):
    print(f'{point} is at distance {point.get_polar()[0]} from origin')
```

Note

Observe that the `__iter__` method returns an object that is defined **outside the class**! This means that we can have multiple iterators being used on the same `Triangle` object simultaneously without the iterations interfering with one another.

5.4 Generating infinite streams

Since generator functions create lazy iterators, we can produce elements from an infinite stream on demand, without explicitly having to store such a stream, which would be impossible anyway! Here is an example:

```python
def gen_evens():
    val = 0
    while True:
        yield val  # suspends itself after this!
        val += 2

evens = gen_evens()
```

Now, `evens` will produce the even numbers 0, 2, 4 etc. in sequence. Note the infinite loop in the definition: we will never have a problem with it because the iteration is lazy. The next even number will be produced on demand only when the `next` function is applied to `evens`.

Here's another example: a generator that returns an iterator for the Fibonacci sequence:

```python
def fibo_iterator():
    current, next_val = 0, 1   # the frst two Fibonacci sequence numbers
    while True:
        yield current
        current, next_val = next_val, current + next_val

fibonacci = fibo_iterator()
first_15 = [str(next(fibonacci)) for _ in range(15)]
print(' '. join(first_15))
```

5.5 Generator Expressions

In many situations, we want to create lazy sequences instead of list comprehensions either (a) to minimize memory usage or (b) to transform a potentially infinite stream of data in a pipeline. In such cases, generator expressions are extremely useful.

For example, suppose we wish to produce, on demand, a stream of Fibonacci numbers in increasing order that are also divisible by 7. The following generator expression does the trick (it looks quite similar in syntax to a list comprehension but with parentheses replacing square braces):

```
fibonacci = fibo_iterator()
fibs_div_by_7 = (number for number in fibonacci if number % 7 == 0)
for _ in range(5):
    print(next(fibs_div_by_7))
```

Note that `fibs_div_by_7`, just like `fibonacci`, is a lazy iterator for an infinite stream.

Caution

Be aware of a few limitations of iterators:

- You can only iterate **once** over a collection with an iterator: it cannot be reset. A new iterator has to be created for this purpose.
- Iterators can only advance forward, unlike `next`, there is no `previous` builtin function (nor a corresponding `__prev__` kind of special method). If the collection is a finite sequence, then the `reversed` builtin function can be used to create an iterator that iterates backwards over a collection.
- while iterators and generators resemble sequences, one cannot use builtin functions like `len` to compute their length, or index into them using `[]` notation.

Chapter 6

Simple Inheritance

Large Python projects often have many classes that share common attributes and beavior, or may have small modifications in he way that they behave from other classes. To link such classes, there are two primary mechanisms: **composition** and **inheritance**. We saw an example of the former above: three `Point` objects are composed to form a `Triangle`. The latter mechanism, viz. inheritance, helps us organize code in a manner that avoids repetition and helps us *classify* datatypes in a natural way based on their properties.

For example, a triangle and a rectangle are dissimilar: one has three sides, the other four. They also have similarities in that both are examples of *polygons* in the plane and have areas that are related to the coordinates of the points forming the corners (see the Shoelace Formula)! So, a triangle **is a** polygon, as is a rectangle. On the other hand, a circle is not a polygon (well, in the limit it *is* one, but we'll let that go…) but both circles and polygons are examples of planar regions that have an *area*.

One of the primary goals of OOP is to facilitate systematic development of code that works consistently across such a **hierarchy** of classes. For instance, the hierarchy of planar regions identified in the previous paragraph might look something like this (with an auxiliary `Point` class defined as above):

```
Point

PlanarRegion
    Oval
        Circle
    Polygon
        Triangle
        Rectangle
```

In OOP terminology, `Polygon` is the **superclass** of `Triangle` and `Rectangle`: they can both share the methods of the `Polygon` class but can add specialized attributes and methods in their own definitions that provide distinctive state and behavior. Thus, the `Triangle` class is a **subclass** of `Polygon`; in turn, `Polygon` is a subclass of `PlanarRegion`.

The **Substitution Principle** in OOP asserts that any subclass object can be syntactically substituted in code where a superclass object is expected. For example, every `Polygon` object has a length (number of sides) so it is syntactically correct to expect a `Triangle` object to have a length. The reverse is not true: while a `Circle` is an `Oval`, an oval object has no radius (it has a major and minor axis that are not necessarily equal) and *cannot be substituted* in a place where we expect to use a circle.

6.1 Attributes, Methods, Information Hiding

Building out the hierarchy depicted above, we observe some features of Python that help with creating rich inheritance hierarchies.

1. `PlanarRegion` is an **abstract** class - we cannot *instantiate* an object directly as a `PlanarRegion` object. The only purpose of the class is to *root* the hierarchy, and to specify that all planar shapes that are derived from this class must define their own *concrete* **`area`** method.
2. Sibling classes like `Oval` and `Polygon` share the **`area`** method (because they inherit it from `PlanarRegion`) but also have bespoke attributes: ovals have major and minor axes, while polygons have corners.
3. It is often very easy to construct an instance of a subclass like `Circle` by invoking the constructor of the superclass (`Oval`).
4. Attributes whose names begin with two underscores are non-public, and are effectively **hidden** from other code, including from the subclasses! For instance, a `Circle` object cannot access the `__sem_major` attribute in its parent class `Oval`. This mechanism does has a backdoor (since Python does not have the same philosophy of public-protected-private access that other OOP languages like Java enforce for information hiding).

```python
from abc import ABC, abstractmethod
import math


class PlanarRegion(ABC):
    """An abstract base class (ABC) for planar regios"""
    @abstractmethod
    def area(self):
        raise NotImplementedError


class Oval(PlanarRegion):
    """An x-y axis aligned oval"""
    def __init__(self, semi_major_axis, semi_minor_axis):
        self.__semi_major = semi_major_axis
        self.__semi_minor = semi_minor_axis

    def area(self):
        return math.pi * self.__semi_major * self.__semi_minor
```

```
class Circle(PlanarRegion):
    def __init__(self, radius):
        self.set_radius(radius)

    def set_radius(self, radius)
        super().__init__(radius, radius)
        self.__radius = radius

    def get_radius(self):
        return self.__radius

    def get_diameter(self):
        return 2 * self.get_radius()


class Polygon(PlanarRegion):
    """A polygonal shape specified by corners in sequence with the
    assumption that the sides of the polygons do not self-intersect.
    """
    def __init__(self, *corners):
        """Set the corners of the polygon in order"""
        self.__corners = list(corners)  # each corner is a Point

    def area(self):
        """Implementation of the shoelace formula"""
        return 0

    def perimeter(self):
        """Implement this for practice!"""
        return 0


class Triangle(Polygon):
    def __init__(self, pt1, pt2, pt3):
        super().__init__(pt1, pt2, pt3)


class Rectangle(Polygon):
    def __init__(self, pt1, pt2, pt3, pt4):
        # Implement this: ensure that the sides form a rectangle!
        super().__init__(pt1, pt2, pt3, pt4)

    def diagonal(self):
        """Returns the length of the diagonal of the rectangle"""
        # Implement this
        return 0
```