Python classes

Contents

A very popular concept in Computer Science is *an object*. Objects are theoretical entities that may contain data (*fields*) as well as specific functionalities (*methods*). Classes in Python provide a means of implementing *types* objects.

Oof… that is a lot of abstract terminology. Let us have a look at an example:

```
class Dog:

    # initialise an INSTANCE of the object
    def __init__(self, name):
        # assign the supplied name to the Dogs name
        self.name = name

    # only dogs can bark
    def bark(self):
        print("woof-woof!")

# lets make some dogs
d1 = Dog("Princess")
d2 = Dog("Chloe")
d3 = Dog("Spooky")

# make them bark
d1.bark()
d3.bark()

print(d2.name)
```

```
woof-woof!
woof-woof!
k
```

`class Dog` defines a conceptual dog. In the simple model dogs only have a name and can bark, this is what characterises them.

The `__init__` method is used when a dog is initialised e.g. in the `d1 = Dog("Princess")`. Now `d1` is a member of the `Dog` class as well as `d2` and `d3`.

The `bark()` method is specific to dogs, so we can call it on any instance of a `Dog`.

IMPORTANT: Notice the `self` keyword in the definition. `self` will always be used as the first argument of the class methods. `self` is also used to reference the fields of the object (e.g. `self.name`).

Let us give the `Dog` class a bit more functionality:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.happy = True

    def makeHappy(self):
        self.happy = True

    def makeSad(self):
        self.happy = False

    def bark(self):
        print("woof-woof!")
```

Now we have methods that allow us to set the value of the `happy` attribute. Every dog is happy when instantiated.

Inheritance

Some concepts can be treated as subsets of others. In this inheritance relation we distinguish two types of classes:

1. **Parent class** (base class): the class being inherited
2. **Child class**: the class that inherits

Consider the following two classes:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.happy = True

    def makeHappy(self):
        self.happy = True

    def makeSad(self):
        self.happy = False

    def bark(self):
        print("woof-woof!")

class HappyDog(Dog):

    def __init__(self, name):
        super().__init__(name)

    def makeSad(self):
        print("Ha ha! I am always happy! Can't make me sad!")

hd1 = HappyDog("Scooby")

hd1.makeSad()
hd1.bark()
```

```
Ha ha! I am always happy! Can't make me sad!
woof-woof!
```

As you can probably see, `HappyDog` inherits the methods from the `Dog` class. It also has its version of the `makeSad` method. We say that `HappyDog` class overrides the `makeSad` method of the `Dog` class. We use the `super()` method to denote inheritance from the `Dog` class.

Whereas child classes can have multiple parent classes, it usually complicates the code and is not considered a good practice.

Classes without `__init__`

Classes without the `__init__` method cannot produce objects, they cannot be instantiated. They can still provide functionality, however. On a basic level, they can be treated as our own library. Notice that there is no `self` argument in the methods of such class:

```
#regular class
class ComplexNum():
    def __init__(self, real, im):
        self.real = real
        self.im = im

    def __str__(self):
        return str(self.real) +" + " + str(self.im)+"i"

#without __init__
class ComplexMath():

    def add(com1,com2):
        return ComplexNum(com1.real+com2.real, com1.im+com2.im)

    def mul(com1,com2):
        return ComplexNum(com1.real*com2.real-com1.im*com2.im,com1.im*com2.real+com


com1 = ComplexNum(1,1)
com2 = ComplexNum(1,-1)

print(ComplexMath.add(com1,com2))
print(ComplexMath.mul(com1,com2))
```

```
2 + 0i
2 + 2i
```

Here `ComplexNum` is a regular class used to represent complex numbers. `ComplexMath` provides some basic arithmetic operations on those numbers. There is no need to instantiate it, we just need the functionality.

REMARK: `__str__` method must be defined if you want to have a nice way of printing the instances of your class.

Classes are a very broad field of programming in Python, and this is just a brief introduction. We will explore classes in the Fundamentals of Computer Science more.

Exercises

- **Aliens** Define an `Alien` class which instantiates aliens with `age` equal to 1. It should also be able to increase the age of an alien by 1 by calling the `birthday()` method.

Answer

```
class Alien:

    def __init__(self):
        self.age = 1

    def birthday(self):
        self.age+=1
```

- **Living Aliens** Now upgrade your `Alien` class. Alien is instantiated with the field `isAlive` set to `True`. Now, an alien can `considerDying` by taking a random integer from 0 to 10 (inclusive) and seeing if it is smaller than its age. If it is, the alien dies. You should also add the following method to the class definition:

```
def reproduce(self):
    return self.isAlive and random.randint(0,6) > self.age
```

Answer

```
import random

class Alien:

    def __init__(self):
        self.age = 1
        self.isAlive = True

    def birthday(self):
        self.age+=1

    def considerDying(self):
        if random.randint(0,10) < self.age:
            self.isAlive = False

    def reproduce(self):
        return self.isAlive and random.randint(0,6) > self.age
```

- **Simulating population** Now let us simulate the population of aliens which can die and reproduce.

1. On each time step (`for` loop) we will make all aliens celebrate `birthday`, `reproduce` and `considerDying`.
2. An alien should be removed from the population when it dies.
3. Also, when reproduction is successful, a new alien is added to the population. Make sure to add the new aliens at the end of the time step, so they are not considered it the timestep they were born in.
4. Do the simulation for `100` timesteps.
5. You might want to print the size of the population vs time using matplotlib.
6. Start with only one alien in the population.

REMARK: This model of the population does not ensure that the size of the population will be stable. For now, the population ceases in the first couple of days or grows exponentially. If you want to explore the simulation more, consider adding population size in `reproduce` method, it should limit the exponential growth.

Answer

```
import random
import matplotlib.pyplot as plt

class Alien:

    def __init__(self):
        self.age = 1
        self.isAlive = True

    def birthday(self):
        self.age+=1

    def considerDying(self):
        if random.randint(0,10) < self.age:
            self.isAlive = False

    def reproduce(self):
        return self.isAlive and random.randint(0,6) > self.age


aliens = [Alien()]
timeSeries = []

for i in range(100):


    aliensToAdd = 0

    for alien in aliens:
        alien.birthday()
        if alien.reproduce():
            aliensToAdd+=1
        alien.considerDying()

    aliens = [x for x in aliens if x.isAlive]

    for new in range(aliensToAdd):
        aliens.append(Alien())

    timeSeries.append(len(aliens))

x = list(range(100))
y = timeSeries

plt.plot(x,y)
plt.show()
```