Object Oriented Programming

Guillaume Macneil

December 8, 2020

Abstract

This is the seventh lesson on the 'Introduction to Python' course. This course is loosely based around the 'MIT Introduction to Computer Science - Fall 2016' course. Our course is more focused towards the Python programming side of things.

1 Introduction to OOPs

In the field of programming, there are many different kinds of programming *paradigms*. Here are the some of the important ones:

- **Object Oriented Programming:**
  'A programming paradigm based on the concepts of "objects", which contain data and code; data in the form of fields, and code, in the form of procedures.'
- **Functional Programming:**
  'A programming paradigm where programs are constructed by applying and composing functions. It is a declarative paradigm in which function definitions are trees of expressions that each return a value.'
- **Logical Programming:**
  'A programming paradigm mainly based on formal logic. Any program written in a logic programming language is a set of sentences in logical form, expressing facts and rules.'

Python is an object oriented programming language (OOP) because every piece of data supported by Python is an 'object'. Each object is comprised of:

- A **type**, e.g. *int, str, bool, etc.*
- An internal **data representation**, e.g. *primitive or composite.*
- A set of procedures for **interaction** with the object.

An object is defined as an instance of a type (*42* is an instance of an *int*). When programming in Python (or any truly object oriented programming language) we are simply creating, manipulating and destroying[1] objects. Put loosely, objects are abstractions of data which can be easily manipulated and interacted with.

'But what makes OOPs better than their alternatives?', you may be thinking, this is an understandable question but also a slippery slope. As with many things in Computer Science, there is no 'best' programming paradigm, we simply have to make certain trade-offs based on our scenario. Here are some of the advantages as well as the disadvantages that come with OOPs:

- Advantages:
    1. Due to the **modular** nature of OOPs, it is often easier to write code with them. Code is reusable in the form of functions, classes and methods making the development process more straightforward.
    2. Software is generally more **maintainable** based on the reusable and easy-to-follow structure of the code. Sections of the code can often be altered without breaking the rest of the program.
    3. Data can be bundled into **packages**, alongside the procedures that work with them, creating reusable *'libraries'* of code.
- Disadvantage:
    1. Programs written with OOPs are often **larger** than their functional counterparts.
    2. They are generally also **slower** because more instructions often need to be executed to reach the same result.
    3. The inter-linked nature of OOP based programs can sometimes be **confusing** if the program is large and contains many different objects to be interacted with.

2 The Intricacies of OOPs

In this section I will go over how the object oriented nature of Python can be used to its fullest extent.

2.1 Classes

Classes are often confusing for beginners due to the wy that they are defined. I view classes like types which are made by the programmer and which can be used to define the interactions that happen with a certain kind of data. To create a class, we need to define the class' name and attributes. Then to use the

[1] The destruction of objects is actually quite an interesting problem to solve when it comes to OOPs. If you'd like to know more, look into 'garbage collectors' and 'object scopes'.

class we need to create new instances of it (objects!) and perform operations on them.

```
class particle(object):
        def __init__(self, mass, radius):
                self.mass = mass
                self.radius = radius
                self.volume = (4/3) * 3.1415 * (radius**3)
                self.density = mass / self.volume
        def does_it_sink(self, fluid_density):
                if self.density > fluid_density:
                        return True
                else:
                        return False

water_density = 997
gallium_density = 6095
mercury_density = 13593

iron_particle = particle(32.966, 0.1)
gold_particle = particle(80.844, 0.1)
ice_particle = particle(3.841, 0.1)

# Print the material densities
print(iron_particle.density)
print(gold_particle.density)
print(ice_particle.density)

# Does iron sink?
print(iron_particle.does_it_sink(water_density))
print(iron_particle.does_it_sink(gallium_density))
print(iron_particle.does_it_sink(mercury_density))
# Prints True, True, False

# Does gold sink?
print(gold_particle.does_it_sink(water_density))
print(gold_particle.does_it_sink(gallium_density))
print(gold_particle.does_it_sink(mercury_density))
# Prints True, True, True

# Does ice sink?
print(ice_particle.does_it_sink(water_density))
print(ice_particle.does_it_sink(gallium_density))
print(ice_particle.does_it_sink(mercury_density))
# Prints False, False, False
```

In the above example, we define a class called *particle*. In the *'__init__'* method, we define what happens when we create a new particle, you can see that we define a series of variables, but we prefix them with *'self.'*. This just means that these variables are specific to the individual instance of the particle (mass, radius, volume and density). After that we define another method called *'does_it_sink'* which contains an if statement that checks whether the density of the particle is greater than the density of a given fluid. You can then see how we initialise new particles and use methods in the print statements below the class.

There are also certain 'special' methods that can be defined in order to provide certain functionalities (this isn't an exhaustive list, there are many more):

- **__add__(self, other) : Defines functionality for [self + other]**
- **__sub__(self, other) : Defines functionality for [self - other]**
- **__eq__(self, other) : Defines functionality for [self == other]**
- **__len__(self) : Defines functionality for [len(self)]**
- **__str__(self) : Defines functionality for [print(self)]**