OOP and Growth

Abhinav Ashar
CS 61A: Structure and Interpretation of Computer Programs

January 18, 2019

1 Object Oriented Programming (OOP)

OOP is a critical feature in many languages. Essentially, OOP involves utilizing classes and instances of those classes. In Python, classes are a way to abstract a group of data (Ex. If you have dogs and you want to store information about their breed and height, you would likely create a Dog class that stores the breed and height variables). These classes are templates or blueprints for instances of the class.

a) Structure

   (a) At the top of the class, there are class attributes

   (b) There is sometimes (but not always!) an init function that allows you to create instances and define instance attributes. If a class does not have an init function, check whether its parent class (or the parent class of the parent class ...) has an init function, which you will use to create your instance instead.

b) Functions

   (a) Classes often have functions associated with them

   (b) The self keyword in the parameter of a function means that the function is looking for an instance/sub-instance of the class

   (c) Let's say we have poodle, which is an instance of the Dog class. If the signature for the bark function is def bark(self), poodle.bark() is the same as Dog.bark(poodle) because there is a self attribute.

c) Inheritance

   (a) class Poodle(Dog) indicates that Poodle is a subclass of Dog

   (b) Subclasses share the attributes from the parent class (not a copy!)

   (c) Subclasses can override certain attributes from a parent class

d) Attributes

   (a) There are two types of attributes: class attributes and instance attributes

(b) Class attributes are defined at the top of a class and all instances have these attributes

(c) Instance attributes are defined in the init function and are more exclusive to that particular instance

(d) When searching for an attribute, look at the instance attributes before you look at the class attributes

(e) Class attributes can be referred to by (className).(attribute)

2 Orders of Growth

The goal of order of growth is to analyze how efficient our code is. For example, if we double the size of our input, does the code take twice as long, or four times as long, or more?

a) Drop constants in orders of growth: $\theta(2n) \rightarrow \theta(n)$

b) $\theta(1)$ means that the function does not take any longer if size of the input increases (does not depend on the input at all)

c) When you have for or while loops, understand where the starting point is, where the ending point is, and what the step size is

d) With recursive functions, look at how many branches are produced (Often, the answer is X^n where X is the number of branches