Object Oriented Programming

Cary Goltermann

Galvanize

2016

Overview

OOP

- Motivation
- Definition
- Terminology

OOP - Tenets

- Encapsulation
- Composition
- Polymorphism
- Inheritance

Overview

OOP

- Motivation
- Definition
- Terminology

OOP - Tenets

- Encapsulation
- Composition
- Polymorphism
- Inheritance

Coding Concerns

Much of what we are going to talk about today is concerned with the overarching topic of good software engineering practices.

When we write code we are striving for it to be:

- Easy to maintain and modify
  - Easy to understand
  - Easy to use / re-use
  - Split into logical components
- Easy to test

What is OOP?

Object oriented programming is a paradigm that allows us to model real-world ideas as “objects” in code. These objects have data and associated actions that can be done with that data.

What is OOP?

Object oriented programming is a paradigm that allows us to model real-world ideas as “objects” in code. These objects have data and associated actions that can be done with that data.

In this way a programs can be built up by defining objects and relationships between those objects.

What is OOP?

Object oriented programming is a paradigm that allows us to model real-world ideas as "objects" in code. These objects have data and associated actions that can be done with that data.

In this way a programs can be built up by defining objects and relationships between those objects.

This model is quite popular for many reasons, but one of the main ones that proponents cite is that programming in this way models that the way that we think about the world, as things with relationships.

What is a Program?

A program is a set of instructions, **behavior**, that when performed on some data, **state**, results in some desired outcome. E.g. a calculation being performed or webpage rendered.

$$\text{Program} = \text{State} + \text{Behavior}$$

Overview

OOP

- Motivation
- Definition
- Terminology

OOP - Tenets

- Encapsulation
- Composition
- Polymorphism
- Inheritance

OOP - Formalization

OOP originated from a desire to have a way to program that mirrors how humans label, interpret and interact with the physical world.

In this way the state and behavior that make up a program are collected into **objects** and **classes**.

Objects

An **object** is a single entity containing:

State	Variable(s), representing some data.
Behavior	Action(s), defining ways to interact with the state.

Objects

An **object** is a single entity containing:

State	Variable(s), representing some data.
Behavior	Action(s), defining ways to interact with the state.

Example: Blender

State	Plugged in, On, Speed
Behavior	Plug in, Blend, Change Speed

Classes

A **class** is a meta-concept that describes how an object operates. You can think of it like a blueprint for an object. In this way an object is a single realization, or **instance**, of a class.

Classes

A **class** is a meta-concept that describes how an object operates. You can think of it like a blueprint for an object. In this way an object is a single realization, or **instance**, of a class.

In this way the nature of the state and behavior that an object can have is specified in this blueprint, when the class for it is defined.

Overview

OOP

- Motivation
- Definition
- Terminology

OOP - Tenets

- Encapsulation
- Composition
- Polymorphism
- Inheritance

Attributes

Every object has a group of attributes. These attributes are where the data, the state, of the object resides. You can think of the data in an object's attributes as being “owned” by that object.

Attributes

Every object has a group of attributes. These attributes are where the data, the state, of the object resides. You can think of the data in an object's attributes as being "owned" by that object.

Example: Blender Attributes

- `plugged_in` → Bool
- `on` → Bool
- `speed` → Int

Attributes

Every object has a group of attributes. These attributes are where the data, the state, of the object resides. You can think of the data in an object's attributes as being "owned" by that object.

Example: Blender Attributes

- `plugged_in` → Bool
- `on` → Bool
- `speed` → Int

The value of these attributes defines the object's state. To access that state that lives on the object we use *dot notation*. E.g. `my_blender.speed`.

Methods

In addition to the attributes which store an object's state, every object has a set of **methods** that define the object's behavior. A method is simply a procedure that has access to an object's state.

Methods

In addition to the attributes which store an object's state, every object has a set of **methods** that define the object's behavior. A method is simply a procedure that has access to an object's state.

Frequently, methods will change its object state, in fact, methods are the de facto correct way to interact with an object's state.

Methods

In addition to the attributes which store an object's state, every object has a set of **methods** that define the object's behavior. A method is simply a procedure that has access to an object's state.

Frequently, methods will change its object state, in fact, methods are the de facto correct way to interact with an object's state.

Example: Blender Methods

- `plug()` → Changes `plugged_in` attribute.
- `toggle_power()` → Changes `on` attribute.
- `set_speed(speed)` → Changes `speed` attribute to specified int.

Tenets

In object oriented programming there are three (and a half) overarching ideas that guide the way that we about making classes.

- Encapsulation
- Inheritance - but we'll mostly talk about Composition
- Polymorphism

Overview

OOP

- Motivation
- Definition
- Terminology

OOP - Tenets

- Encapsulation
- Composition
- Polymorphism
- Inheritance

Encapsulation

Expands the concept of abstraction that we take advantage of when writing functions to include both the: **state** and **behavior** that we deal with when working with objects.

Encapsulation

Expands the concept of abstraction that we take advantage of when writing functions to include both the: **state** and **behavior** that we deal with when working with objects.

This means that not only are the details of how the behavior are implemented, but also, the details about how are state is being stored.

Encapsulation Benefits

The implications of encapsulation make working with classes easy in a number of ways:

1. When using the class we don't have to worry about the details about how any of the state is being stored, or how the behavior is being implemented.

Encapsulation Benefits

The implications of encapsulation make working with classes easy in a number of ways:

1. When using the class we don't have to worry about the details about how any of the state is being stored, or how the behavior is being implemented.
2. This lack of concern makes it so that the implementations can changes without any user ever knowing, so long as the interface remains unchanged.

Encapsulation Example

```
In [1]: from collections import Counter

In [2]: c = Counter('Hello')

In [3]: c
Out[3]: Counter({'e': 1,'h': 1,'l': 2,'o': 1})

In [4]: c.most_common()
Out[4]: [('l', 2), ('o', 1), ('e', 1), ('h', 1)]
```

Encapsulation Example

```
In [1]: from collections import Counter

In [2]: c = Counter('Hello')

In [3]: c
Out[3]: Counter({'e': 1,'h': 1,'l': 2,'o': 1})

In [4]: c.most_common()
Out[4]: [('l', 2), ('o', 1), ('e', 1), ('h', 1)]
```

How does `most_common()` work? For that matter, how is the data in a Counter stored? Does it matter that we don't actually know?

Overview

OOP

- Motivation
- Definition
- Terminology

OOP - Tenets

- Encapsulation
- Composition
- Polymorphism
- Inheritance

Composition

When building up our own classes we want to strive to have each class concerned with a single specific concept. However, frequently we'll find that we want to make classes that encompass complicated ideas.

In these circumstances we can have the attributes of our complicated class be instances of another class. We call this class **composition**, in that we are composing the complicated class from other, specifically focused, hopefully simpler classes.

Example Class - Deck of Cards

What are the objects that we might want to make if we were codifying the concept of a deck of cards?

Example Class - Deck of Cards

What are the objects that we might want to make if we were codifying the concept of a deck of cards? **Card** & **Deck**?

Example Class - Deck of Cards

What are the objects that we might want to make if we were codifying the concept of a deck of cards? **Card** & **Deck**?

The language we use to describe this relationship is that a Deck **has-a** Card, many of them in fact.

Example Class Cont.

```
import random

class Card(object):
    def __init__(self, number, suit):
        self.suit = suit
        self.number = number

class Deck(object):
    def __init__(self):
        self.cards = []
        for num in ['2', '3', '4', '5', '6', '7', '8',
                    '9', '10', 'J', 'Q', 'K', 'A']:
            for suit in 'cdhs':
                self.cards.append(Card(num, suit))
```

Example Class Cont.

```
    ...
    def shuffle(self):
        random.shuffle(self.cards)

    def draw_card(self):
        if not self.isempty():
            return self.cards.pop()

    def add_cards(self, cards):
        self.cards.extend(cards)

    def isempty(self):
        return not self.cards
```

Built-in Operations

When we write custom classes we frequently want to enable instances of them to interact with built-in language operations so that we can use them like built in classes.

Consider that we can compare strings with '*brain*' $<$ '*pinky*', and we can find the length of containers with *len*().

Built-in Operations

When we write custom classes we frequently want to enable instances of them to interact with built-in language operations so that we can use them like built in classes.

Consider that we can compare strings with '*brain*' $<$ '*pinky*', and we can find the length of containers with *len*().

The functionality that we desire is known as operator overloading, and in Python it is available to us via **magic methods**.

Magic Methods

Magic methods are any methods that begin and end with a double underscore (the double underscore convention is exclusively reserved for magic methods). The functionality they afford us enable interaction with built-in language operations.

Example: Length and Stringing

```
class Deck(object):
    ...
    def __len__(self):
        return len(self.cards)
```

Magic Methods

Magic methods are any methods that begin and end with a double underscore (the double underscore convention is exclusively reserved for magic methods). The functionality they afford us enable interaction with built-in language operations.

Example: Length and Stringing

```
class Deck(object):
    ...
    def __len__(self):
        return len(self.cards)
```

```
class Card(object):
    ...
    def __str__(self):
        return '{}{}'.format(self.number, self.suit)
```

Two Object Magic Method Example

When we work with operations that involve two objects we need to have a way to access information about both. Python handles this by passing a reference to each of the objects to the magic method that implements the operation's functionality. Convention is to use the names **self** (per the usual) and **other**.

Two Object Magic Method Example

When we work with operations that involve two objects we need to have a way to access information about both. Python handles this by passing a reference to each of the objects to the magic method that implements the operation's functionality. Convention is to use the names **self** (per the usual) and **other**.

```
class Card(object):
    ...
        def __cmp__(self, other):
            return cmp(self.number, other.number)
```

Note: self.number and other.number may be characters.

The cmp() function is built into Python. Returns −1 if the first is less than the second, 1 if the reverse is true, and 0 if they are equal.

Overview

OOP

- Motivation
- Definition
- Terminology

OOP - Tenets

- Encapsulation
- Composition
- Polymorphism
- Inheritance

Polymorphism

Having the abstraction of a class' behavior span multiple classes of a similar type is frequently a desirable trait. This next level of abstraction is referred to as **polymorphism**, and is formally defined as:

Polymorphism

The provision of a single interface for objects of different classes.

Overview

OOP

- Motivation
- Definition
- Terminology

OOP - Tenets

- Encapsulation
- Composition
- Polymorphism
- Inheritance

Inheritance

Sometimes when we are designing classes we find that there exists a common structure and functionality that underlies multiple classes. In these situations we would like to abstract out these commonalities, and have a process by which we share that functionality.

Inheritance

Sometimes when we are designing classes we find that there exists a common structure and functionality that underlies multiple classes. In these situations we would like to abstract out these commonalities, and have a process by which we share that functionality.

Inheritance defines a framework in which a class, the child class, can be based off of another class, the parent class. When the child inherits from the parents the functionality defined on the parent class is accessible to the child for free. This process of inheriting allows for effective code reuse amongst like classes.

Inheritance Example: Is-a Relationship

Say we were trying to model some fruit that we frequently eat at home: bananas, oranges, and apples. What do we think the underlying structure and functionality of these classes might be? What would we call it?

Inheritance Example: Is-a Relationship

Say we were trying to model some fruit that we frequently eat at home: bananas, oranges, and apples. What do we think the underlying structure and functionality of these classes might be? What would we call it? **Fruit** - you can eat fruit, it has a weight, and it must be grown.

Inheritance Example: Is-a Relationship

Say we were trying to model some fruit that we frequently eat at home: bananas, oranges, and apples. What do we think the underlying structure and functionality of these classes might be? What would we call it? **Fruit** - you can eat fruit, it has a weight, and it must be grown.