Object-Oriented Programming in Python

- Arianne Dee

Today's schedule

- **What is Object-Oriented Programming?** (35 mins)
  - Break, Q&A
- **Use and Create Classes in Python** (45 mins)
  - Break, Q&A
- **Intermediate Topics** (45 mins)
  - Break
- **Refactoring to OOP** (60 mins)
- **Course wrap-up** (5 mins)

Break format

- 3 Breaks (10 mins)
  - Step away or work through code
- Q&A (5 mins)
  - Use Q&A feature
- Use group chat throughout for questions that anyone can answer

Introduction

About me

- Started learning Python 10 years ago through tutorials
- CS degree from UBC
- Have been programming in Python primarily since 2015
  - Primarily web apps in Django

Poll (single choice)

How long have you been programming?

- Less than a month
- 1 - 6 months
- 6 - 12 months
- 1 - 3 years
- 3 - 10 years
- 10+ years

Poll (single choice)

How much Python do you know?

- Basically nothing
- A bit, but I'm an experienced developer
- A bit, and I'm a new developer
- Enough to get by, but I'm not comfortable with classes
- Quite a bit, but I want to learn more advanced concepts

Poll (multi choice)

- What are you hoping to learn from this class?
  - Understand what OOP is
  - Figure out when to use it
  - Learn how to read Python code better
  - Learn how to create new classes in Python
  - Learn some intermediate/advanced stuff
  - Design patterns in Python
  - Learn something specific: answer in group chat

Set up

- Python 3.6+ installed
- An IDE for Python (I'm using PyCharm)
- Course material downloaded and unzipped
- Go to https://github.com/ariannedee/oop-python for step-by-step instructions

What is Object-Oriented Programming?

What is Object-Oriented programming?

- A style of programming
- Instead of writing what to do line by line, your code is organized based on “objects”
- Objects are basically groups of data, variables and functions that are all related

Why should you learn it?

- Use many libraries and frameworks
- Fuller understanding of the Python language
- More tools for writing code
- Think in a new way about programming
  - Or program Python similar to Java

Objects

What data do you need to describe it?

What do you want it to do?

Data (properties/attributes)

What data do you need to describe it?

Retail store

- Description
- Cost
- Sale price
- Condition
- Sold?

Mechanic

- Owner
- Description
- Frame size
- Wheel size
- Last service date

Function (methods)

What do you want it to do?

Retail store

- Add to inventory
- Update sale price
- Sell

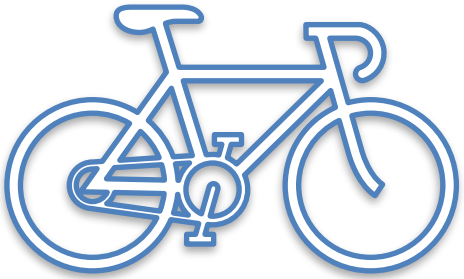
Mechanic

- Drop off
- Service
- Pick up
- Charge client

We can do this with functions and dicts

```
bike1 = create_bike('Univega Alpina, orange', cost=100, sale_price=500, condition=0.5)
# bike1 = {
#         'cost': 100,
#         'condition': 0.5,
#         'description': 'Univega Alpina, orange',
#         'sale_price': 500,
#         'sold': False,
#     }

update_sale_price(bike1, 350)
# bike1['sale_price'] = 350.00

sell(bike1)
# bike1['sold'] = True
```

Classes vs instances



Class

Instance

Using classes

```
bike = Bike('Univega Alpina, orange', Condition.OKAY, sale_price=500, cost=100)

bike.service(spent=30, sale_price=600)  # cost = $130, sale_price=600

print(bike.sale_price)                  # sale price = 600

bike.sell()                             # profit = 470
```

Why use Object-Oriented programming?

- Makes it easier to:
  - Organize code
  - Change functionality
  - Add functionality (extend)
  - Reuse code for other programs

We'll go over

- Using classes in Python
- Creating classes in Python
  - Attributes and methods
  - Properties, class methods, class attributes
  - Constructors and destructors
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Questions and break

Use and Create Classes in Python

Using classes

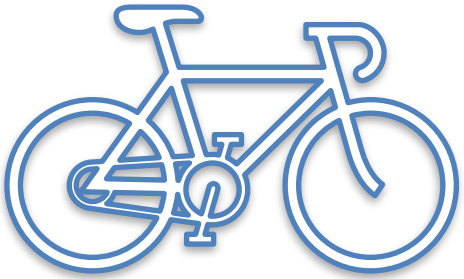
```
bike = Bike('Univega Alpina, orange', Condition.OKAY, sale_price=500, cost=100)

bike.service(spent=30, sale_price=600)  # cost=$130, sale_price=$600

print(bike.sale_price)                  # 600

bike.sell()                             # profit = 470
```

Using classes

```
bike = Bike('Univega Alpina, orange', Condition.OKAY, sale_price=500, cost=100)

bike.service(spent=

print(bike.sale_price)                    # 600

bike.sell()                               # sold=True
```

Create a new instance of `Bike` and set attributes

Classes vs instances



Class

Instance

Using classes

```
bike = Bike('Univega Alpina, orange', Condition.OKAY, sale_price=500, cost=100)

bike.service(spent=30, sale_price=600)  # cost=$130, sale_price=$600

print(bike.sale_price)

bike.sell()                             # sold=True
```

Call a method. It can update attributes.

Using classes

```
bike = Bike('Univega Alpina, orange', Condition.OKAY, sale_price=500, cost=100)

bike.service(spent=30, sale_price=600)  # cost=$130, sale_price=$600

print(bike.sale_price)                  # 600

bike.sell()                             d=True
```

Access an attribute directly

Using classes

```
bike = Bike('Univega Alpina, orange', Condition.OKAY, sale_price=500, cost=100)

bike.service(spent=30, sale_price=600)  # cost=$130, sale_price=$600

print(bike.sale_price)                  # 600

bike.sell()                             # sold=True
```

Call a method without passing parameters

```
bike = Bike('Univega Alpina, orange', Condition.OKAY, sale_price=500, cost=100)

bike.service(spent=30, sale_price=600)  # cost=$130, sale_price=$600

print(bike.sale_price)                  # 600

bike.sell()                             # sold=True
```

Create class

```
class Bike(object):
    pass
```

OR

```
class Bike:
    pass
```

Create method stubs

```
class Bike(object):
    def update_sale_price(self):
        pass

    def sell(self):
        pass

    def service(self):
        pass
```

Create initializer method

```
class Bike(object):
    def __init__(self):
        pass

    def update_sale_price(self):
        pass

    def sell(self):
        pass

    def service(self, cost, new_condition):
        pass
```

Gets called on `Bike()`

Set properties/attributes

```
class Bike(object):
    def __init__(self, description, condition, sale_price, cost=0):
        # Different initial values for every new instance
        self.description = description
        self.condition = condition
        self.sale_price = sale_price
        self.cost = cost

        # Same initial value for every new instance
```

Set properties/attributes

```
class Bike(object):
    def __init__(self, description, condition, sale_price, cost=0):
        # Different initial values for every new instance
        self.description = description
        self.condition = condition
        self.sale_price = sale_price
        self.cost = cost

        # Same initial value for every new instance
```

Set properties/attributes

```
class Bike(object):
    def __init__(self, description, condition, sale_price, cost=0):
        # Different initial values for every new instance
        self.description = description
        self.condition = condition
        self.sale_price = sale_price
        self.cost = cost

        # Same initial value for every new instance
```

Value from user

Saved to object

Fill out methods

```
class Bike(object):
    def __init__(self, description, condition, sale_price, cost=0):...

    def update_sale_price(self, sale_price):...

    def sell(self):
        """
        Mark as sold and determine the profit received from selling the bike
        """
        self.sold = True
        profit = self.sale_price - self.cost
        return profit
```

Fill out methods

```
class Bike(object):
    def __init__(self, description, condition, sale_price, cost=0):...

    def update_sale_price(self, sale_price):...

    def sell(self):
        """
        Mark as sold and determine the profit received from selling the bike
        """
        self.sold = True
        profit = self.sale_price - self.cost
        return profit
```

Set attribute

Fill out methods

```
class Bike(object):
    def __init__(self, description, condition, sale_price, cost=0):...

    def update_sale_price(self, sale_price):...

    def sell(self):
        """
        Mark as sold and determine the profit received from selling the bike
        """
        self.sold = True
        profit = self.sale_price - self.cost
        return profit
```

Use attributes

Enums

```
from enum import Enum


class Condition(Enum):
    NEW = 1
    GOOD = 0.8
    OKAY = 0.5
    BAD = 0.2
```

Questions and break

That’s all for the basics

Let’s look at some intermediate concepts

`__init__`

What's with the underscores?

Dunder/magic methods

Python's hidden magic

Dunder/magic methods

What happens when you call

`Bike()`, `1+2`, `my_list[0]`, or `len(my_list)`?

Common dunder/magic methods

- +
  - __add__
- len()
  - __len__
- str()
  - __str__
- ==
  - __eq__

Dunder/magic methods

- Functions, methods, and modules that start and end with '__'
- Meant to be called indirectly
  - e.g. `n1.__add__(n2)` gets called on `n1 + n2`
- Allows for overloading of common functions
  - You can define how the + operator works on an object by defining its `__add__()` method

Dunder/magic methods for classes

- __init__ - Initializer
  - `Bike()`
- __del__ - Deconstructor (delete)
  - `del bike`
- __str__ - String for display purposes (string)
  - `print(bike)`
- __repr__ - String for development purposes (representation)
  - `repr(bike)`

Class attributes

```
class Bike(object):
    num_wheels = 2
```

```
# All print 2
print(bike2.num_wheels)
print(bike1.num_wheels)
print(Bike.num_wheels)
```

@property

Defined like a method, called like an attribute

@property

```
@property
def profit(self):
    if not self.sold:
        return None
    return self.sale_price - self.cost


print(bike.profit)    # Call property
```

Decorators

Decorates a function and alters its functionality

Decorator

- A function that takes a function as input and returns a function out
- Ways of calling decorators:

```
decorated_function = decorator(function)
```

or

```
@decorator
def function():
    pass
```

Other useful decorators

`@staticmethod` and `@classmethod`

@staticmethod

```
@staticmethod
def age(year):
    current_year = datetime.now().year
    age = current_year - year
    if age < 1:
        return "New"
    elif age < 5:
        return "Recent"
    elif age < 40:
        return "Old"
    else:
        return "Vintage"
```

Static methods

- Can be called without creating an instance first
- A method that does not depend on object state
- `self` doesn't get passed as the first parameter
- Use it when you don't need `self`

@classmethod

```
@classmethod
def get_default_bike(cls):
    return cls(
        cost=0,
        make='A make',
        model='A model',
        year=2010,
        condition=Condition.GOOD
    )
```

```
bike = Bike.get_default_bike()  # Class method
```

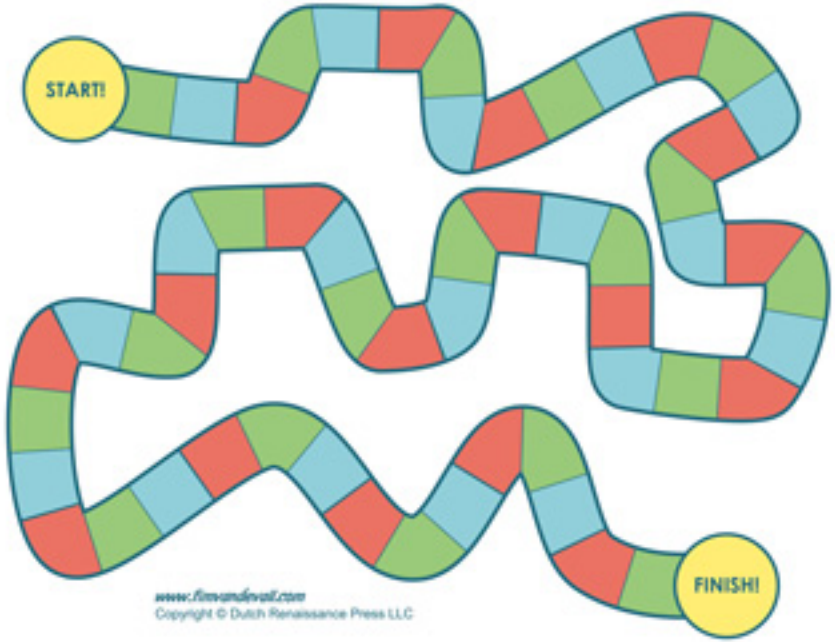
Class methods

- Called on a class, not an instance
- `cls` gets passed as the first parameter, not `self`
- Useful for factories
- Use it when you don't need `self`, but do need `type(self)`

Refactoring to OOP

Merge theory with practise

Example



Example game

- 2-player game
- Take turns rolling a die, then move that many spaces
- First player to 100 wins

Procedural programming

Let’s code it without using objects and classes

Break while we work on game

Game rules:

- 2-player game
- Each player takes turns rolling a 6-sided die, then move that many spaces
- First player to 100 wins

Abstraction

Show only “relevant” data and “hide” unnecessary details of an object from the user.

Encapsulation

Combining relevant data and functionality into a single unit

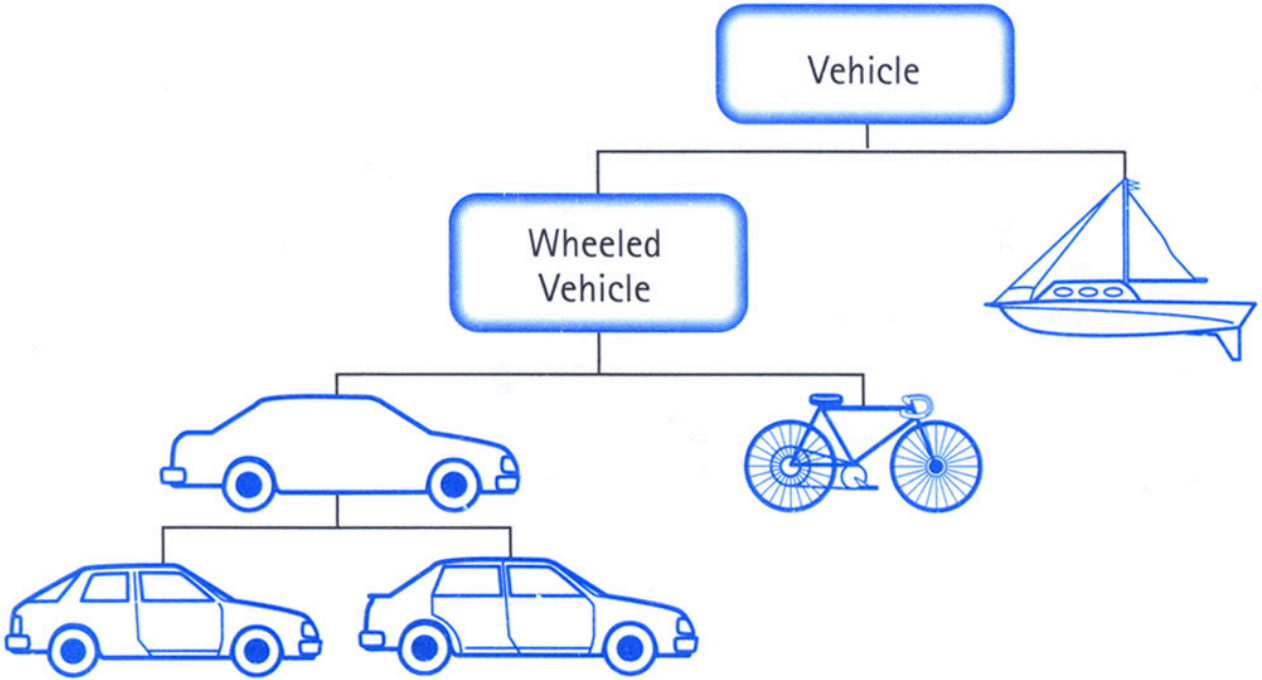
Private vs public?

Nothing in Python is strictly “private”.
Use a ‘_’ to denote something as private.

Inheritance

New classes can **inherit** the properties and methods of existing classes

Inheritance



What is object?

```
class Bike(object):
    pass
```

The `object` object

- The generic base class of all Python objects
- In Python 2, you must **inherit** from `object`
- In Python 3, you don't need to do it explicitly
  - All classes inherit from `object` by default
  - `class Bike:` is okay
    ```
    class Bike:
        pass
    ```
- Read more

What about Mixins?

Add a small amount of functionality to many different types of objects

Mixins are not superclasses and cannot be instantiated by themselves, but they are used in the same way as inheritance.

They are like class extensions or interfaces.

For example, you could write a **AsDict** mixin that adds an **as_dict** function to every object that uses it. So now all of your custom classes can create a dictionary from an object instance.

Polymorphism

Subclasses can define the same method but do different things

*args and **kwargs?

*args and **kwargs?

args = positional arguments

kwargs = keyword arguments

*args and **kwargs?

args = positional arguments

If args is a list, *args is item1, item2, …

kwargs = keyword arguments

If kwargs is a dict, **kwargs is key1=value1, key2=value2, …

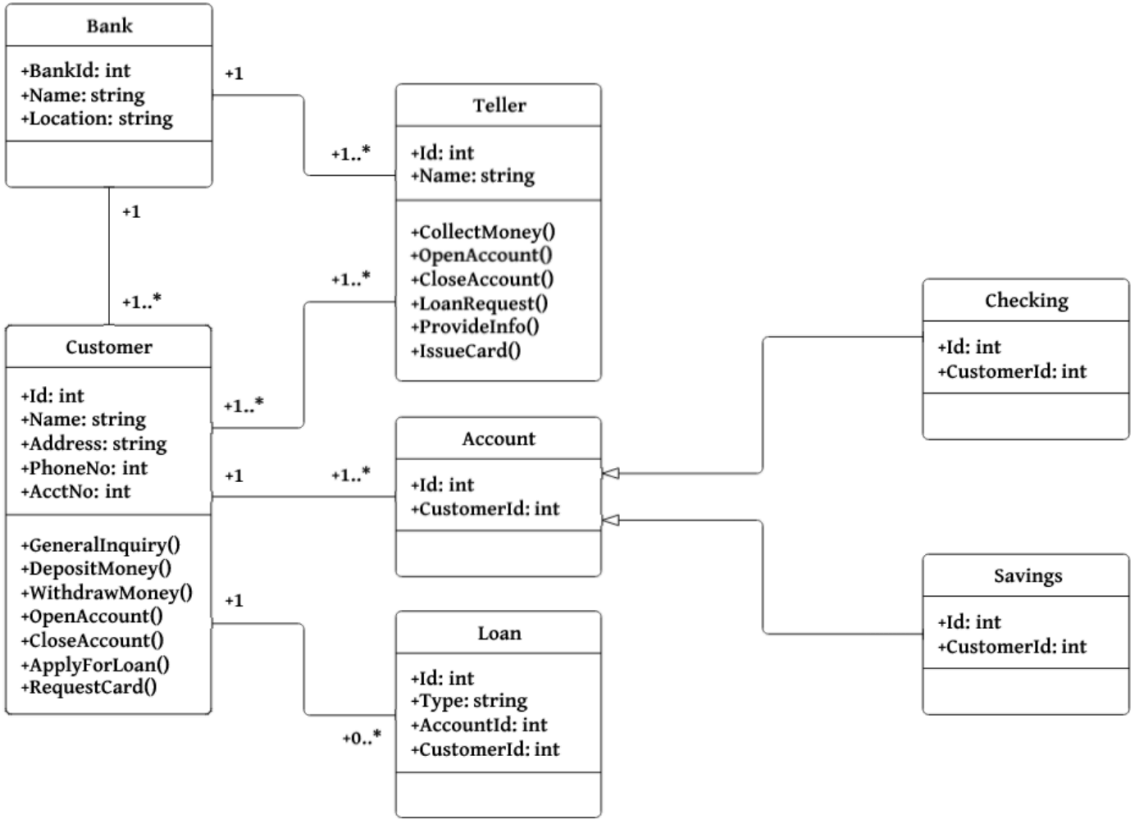
Course wrap-up

Class Diagrams

- Think about your program at a high level
  - What objects is your program comprised of?
  - How are those object related, how do they interact?
  - What attributes and methods do they need?

- Draw a diagram to show relationships and attributes

- Translate diagram to code



UML class diagrams

- Tutorial
  - https://medium.com/@smagid_allThings/uml-class-diagrams-tutorial-step-by-step-520fd83b300b

Homework: Make a game using PyGame

- Documentation
  - https://www.pygame.org/docs/
- Tutorial
  - https://realpython.com/pygame-a-primer/

- Ideas
  - Our board game with a UI - visualize track, dice, score
  - Paint program
  - Side scroller

Practise: Make a card game

- Classes:
  - Card
  - Player
  - Game

- Create a simple game, e.g. war, big 2, hearts
- Build an AI to play against

Homework: Make a game using PyGame

- Design game
- Create a class diagram
- Code
- Test, play, add features, share!
  - Email arianne.dee.studios@gmail.com if you want to share them with me or if you are quite stuck

More courses by me, Arianne

Live Trainings

- **Introduction to Python Programming**
  - Very beginner
- **Python Environments *new***
  - Beginner - link to Oct 20 class
- **Programming with Python: Beyond the Basics**
  - Beginner - link to Nov 2 class
- **Rethinking REST: A hands-on guide to GraphQL**
  - Advanced

Videos

- **Introduction to Python LiveLessons** - link
- **Rethinking REST: A hands-on guide to GraphQL** - link

More reading

- Dunder/magic methods
  - https://www.tutorialsteacher.com/python/magic-methods-in-python
- Private variable docs
  - https://docs.python.org/3.7/tutorial/classes.html#private-variables
- Decorators
  - https://realpython.com/primer-on-python-decorators/

More reading

- *args and **kwargs
  - https://realpython.com/python-kwargs-and-args/
- Abstract Base Class
  - https://docs.python.org/3/library/abc.html
- Design patterns in Python
  - https://www.tutorialspoint.com/python_design_patterns/index.htm

More reading

- Mixins tutorial
  - https://pythonpedia.com/en/tutorial/4359/mixins

- Inheritance, compositions, and mixins
  - https://realpython.com/inheritance-composition-python/

E-books

- Mastering PyCharm
  - https://learning.oreilly.com/library/view/mastering-pycharm/9781783551316/index.html

- Fluent Python (intermediate -> advanced)
  - https://learning.oreilly.com/library/view/fluent-python/9781491946237/

Thanks!

Questions?

Email me at arianne.dee.studios@gmail.com