Introduction to OOP in Python

Today's objectives

- What's Object-Oriented Programming (OOP)?
- Why OOP?
- Key OOP concepts
- OOP using Python

Today's plan

Morning:

- Introduction to OOP
- Core OOP using Python
  - Design and implement a basic class
  - Instantiate an object
  - Key OOP concepts
- Individual assignment: Modify an existing OOP-style program: The War card game

Afternoon:

- Advanced OOP using Python
- Pair programming: Simulate the Blackjack game in an OOP-style program

Introduction to OOP

Why OOP?

As a data scientist, you'll be reading and writing a lot of code. Good software engineering is key to be a good data scientist. Good OOP is key to good software engineering

Good code should be:

- Easy to understand
- Easy to use/re-use
- Easy to maintain

By splitting code into logical components, OOP helps you to write good code

What's in a program?

Program = Data + Instructions

- But for our purpose today, we'll say:

Program = State + Behavior

- E.g., consider a mug:
  - It has a **state**: color, volume, empty, broken, ...
  - It has a **behavior**: buy, fill, drink, clean, dispose of, ...

What's OOP? Classes and Objects

- OOP is a programming paradigm (a set of thought patterns) that models state and behavior into **classes** and **objects**
- A **class**:
  - Embody a concept (e.g., the concept of "mugs") and defines its blueprint by grouping the concept's state and behavior into **member variables** (for state) and **methods** (for behavior)
- An **object**:
  - Is an instance of a class (i.e., a specific mug, e.g., red, with a capacity of 20oz, half full)
  - Can create multiple objects of the same class

What's OOP? Member Variables and Methods

- Every object has a set of **member variables** that define its **state**. These are variables that are bound to that object and that object only; it "owns" them
- Every object has a set of **methods** that define its **behavior**. Each method is a procedure that the object knows how to perform. Most of the time the methods will change the object's state (i.e., modify its member variables)

Core OOP using Python

Very basic OOP design

Decompose your problem into nouns and verbs:

- To implement a **noun** $\Rightarrow$ use a class
- To implement a **verb** $\Rightarrow$ use a method

Defining a class in Python

Classes:

- Use `class` keyword
- Class names are nouns
- Class names use *UpperCamelCase*

Methods:

- Method names are (usually) verbs
- Method names use *snake_case*

Example: Defining a class in Python

Defining a class in Python (cont.)

`self`

- Use `self` (for 'self-reference') to refer to an instance's own member variables and methods

`def __init__(self, ...):`

- Define the special method `__init__` to initialize an instance of a class
- Called whenever an instance of the class is created to handle instance-specific initialization

Example: Instantiating objects in Python

OOP revolves around four key concepts:

- Encapsulation
- Inheritance
- (Polymorphism) (not really applicable with Python)
- Composition

Encapsulation

- What is encapsulation?
  - Encapsulation is about selectively hiding some member variables and methods from a class definition so as not to expose the internal representation of an object
  - Exposed member variables and methods are said to be **public**; hidden member variables and methods are said to be **private**
- Why encapsulation?
  - Hiding an object's internal representation protects its integrity by preventing users from setting it into an invalid or inconsistent state
- Good encapsulation $\Rightarrow$ Good interface
  - A class' public member variables and methods define an **interface** for that class, a **contract** between the client and the service provider

Encapsulation with Python

- That being said, **Python doesn't enforce encapsulation**:
  - *"We are all consenting adults"* (but you are warned: violating encapsulation will make code impossible to maintain)
- In Python, start names of classes, member variables, and methods with _ to indicate that they are private

Example: Encapsulation with Python

Inheritance

Derive a **child** class from a **parent** class:

- Parent class defines general behavior; also called base class
- Child class specializes behavior; also called derived class
  - Child gets all the functionality of the parent class
  - Child add new methods or can override parent methods of the same name

Example: Inheritance with Python

Inheritance with Python

`class DerivedClass(BaseClass)`

- To inherit from a base class, specify the base class instead of `object` when you define the derive class:
- Can check if an object is a specific class via `isinstance()`

`def __init__(self, ...):`

- When a class inherits from another, the derived class must call the base class's constructor:
  - Use `super().__init__()` to call base class's `__init__()`
- Always initialize base class before derived class

Polymorphism

OOP enables polymorphism:

- For objects to be treated identically, they typically need to be related via inheritance
- This isn't required in Python which uses *duck-typing*:
  - *"If it looks like a duck and quacks like a duck, it is a duck"*
- In Python, polymorphism works as long as objects instantiates the necessary interface:
  - At run-time, Python will check if an object has the desired method; if it is missing, Python will raise an `AttributeError`

IsA relationship

- "Derived/Child is a Base/Parent type of thing"
  - E.g., An "apple" is a "Fruit"
  - E.g., A "car" is a "Vehicle"

Composition

- Composition/aggregation is another way to build classes:
  - Class contains an object of a class with the desired functionality
  - Often, just basic types: `str`, `float`, `list`, `dict`, etc.
  - *HasA* $\Rightarrow$ use composition/aggregation
- E.g., an "Airplane" has an "engine"
- E.g., a "House" has a "bathroom"

Other programming paradigms

- **Imperative** and **procedural**
  - Explicit sequence of commands
- **Declarative**
  - Declare what result you want, not how to obtain it (e.g., SQL)
- **Functional**
  - Immutable state, deeply nested function calls (e.g., Spark)

Most programming languages (such as Python) offer a mix of paradigms

Individual assignment: Modify an existing OOP-style program: The War card game

Advanced OOP using Python

Advanced OOP using Python

- `*args` and `**kwargs`
- Decorators
- Properties
- Class-specific data and methods
- Static data and methods
- Magic methods
- Testing

`*args`

Shorthand to refer to a variable number of arguments:

- For regular arguments, use `*args`:
  - `def my_args(*args):` to define a function which takes multiple arguments
  - `*args` is a `list`
  - Can also call function using a `list`, if you dereference

`**kwargs`

- For keyword arguments, use `**kwargs`:
  - `def my_kwargs(**kwargs):` to define a function which takes multiple keyword arguments
  - `**kwargs` is a `dict`
  - Can also call function using a `dict`, if you dereference

Example: `*args` and `**kwargs`

Decorators

A *decorator* is a function which wraps another function:

- Looks like the original function, i.e., `help(myfunc)` works correctly
- But, decorator code runs before and after decorated function

Some common decorators are:

- `@property` often with `@<NameOfYourProperty>.setter`
- `@classmethod` - can access class specific data
- `@staticmethod` - group functions under class namespace

Properties with @property

Properties look like member data:

- Actually returned by a function which has been decorated with `@property`
- Cannot modify the field unless you also create a setter, by decorating with `@<field_name>.setter`
- Gives you flexibility to change implementation later

Example: @property

Class-specific data and methods with @classmethod

- Example: number of instances of class which have been created
- Decorate member function with `@classmethod`
- Use `cls` instead of `self` to refer class data
- . . . except in a method which already refers to instance data

Example: @classmethod

Static data and methods with @staticmethod

Static methods are normal functions which live in a class's namespace:

- Do not access class or instance data
- No `self` or `cls` argument
- Just access by prepending name with the class's name

Example: @staticmethod

Magic methods

Magic methods are special methods you can define to add "magic" to your classes, e.g.,

- To support math and relational operators
- To support iteration
- To create a new container, e.g., support `len()`

See: magic methods

Magic methods (cont.)

Popular magic methods:

Method	Purpose
`__init__`	Constructor
`__del__`	Destructor
`__str__`	Define behavior for `str(obj)`
`__repr__`	Define behavior for `repr(obj)`
`__len__`	Return number of elements in object
`__iter__`, `__next__`	Returns an iterable
`__eq__`, `__ne__`	Compare two objects (==, !=)
`__lt__`, `__le__`	Compare two objects (<, <=) (cont.)
`__gt__`, `__ge__`	Compare two objects (>, >=) (cont.)

Example: Magic methods

Testing

Testing

Testing your code, and finding and fixing bugs are critical skills:

- Just because your code runs, doesn't mean it is correct
- Write unit tests to exercise your code:
  - Ensures interfaces satisfy their contracts
  - Exercise key paths through code
  - Identify any bugs introduced by future changes which break existing code
  - Test code before implementing entire program
- When unit tests fail, use a debugger to examine how code executes
- Both are critical skills and will save you hours of time

Unit tests and TDD (test-driven development)

Unit tests exercise your code so you can test individual functions:

- Use a unit test framework, e.g., `nose`
- Unit tests should exercise key cases and verify interfaces
- A unit test can setup fixtures (i.e., resources) needed for testing
- *Test Driven Development* is a good approach to development:
  - *Red*: implement test and check it fails
  - *Green*: implement code and make sure it passes
  - *Green*: refactor and optimize implementation
- 'Only refactor in the presence of working tests'
- Save time by verifying interfaces and catching errors early
- Catch errors if a future change breaks things

Summary

- What is the difference between a class and an object?
- What are the three key components of OOP? How do they lead to better code?
- How should I implement my code if the relationship is *IsA*? What if the relationship is *HasA*?
- What is duck typing?
- What should you do ensure an object is initialized correctly?
- What are magic methods?
- What are the benefits of TDD? What does Red/Green/Green mean?

Pair programming: Simulate the Blackjack game in an OOP-style program