Object-oriented Programming and Workflows

A. Richards

02.08.2017

Objectives

Morning

- Compare and contrast functional and object oriented programming
- Given the code for a python class be able to: instantiate, call methods and access the attributes
- Write the python code for a simple class
- Design a program or algorithm in object oriented fashion
- Be able to implement several magic methods

Afternoon

- Intro to design patterns – thinking in an object oriented fashion
- Intro to functional programming – be able to compare and contrast functional and object oriented programming
- Intro to workflows

Programming Paradigms

Paradigm	Notable languages	Description
imperative	Fortran, Pascal, BASIC, C	*how to do this and how to do that*
declarative	SQL, Wolfram Language	*the how being left up to the language*
functional	Haskell, Erlang, Lisp	*evaluate an expression and use the result*
logic	Prolog, ALF, Leda	*answer a question via search for a solution*
object-oriented	Python, Java, Smalltalk	*pass messages between meaningful objects*

- imperative programming languages make use of procedural programming (functions)
- A heavy lean on procedural and we move towards structured and modular programming– fundamental concepts in OOP
- all (pure) functional and logic-based programming languages are also declarative.
- functional and logical constitute subcategories of the declarative category
- Here is a link to the more comprehensive wiki comparison

What is Python then?

Python

Is an imperative programming language that has both functional and object-oriented aspects. It is also interpreted, interactive, iterative, and more.

Like many popular languages today it is multiparadigm.

- Interpreted languages are programming languages in which programs may be executed from source code form, by an interpreter
- This is in contrast to compiled languages
- In theory any language can be compiled or interpreted—it is just that one is generally done more often than the other
- Iterative languages are built around or offering generators

Object-oriented programming

- Object-oriented programming (OOP) is a programming paradigm based on the concept of objects
- In Python, objects are data structures that contain
  1. data → attributes
  2. procedures → methods

Important

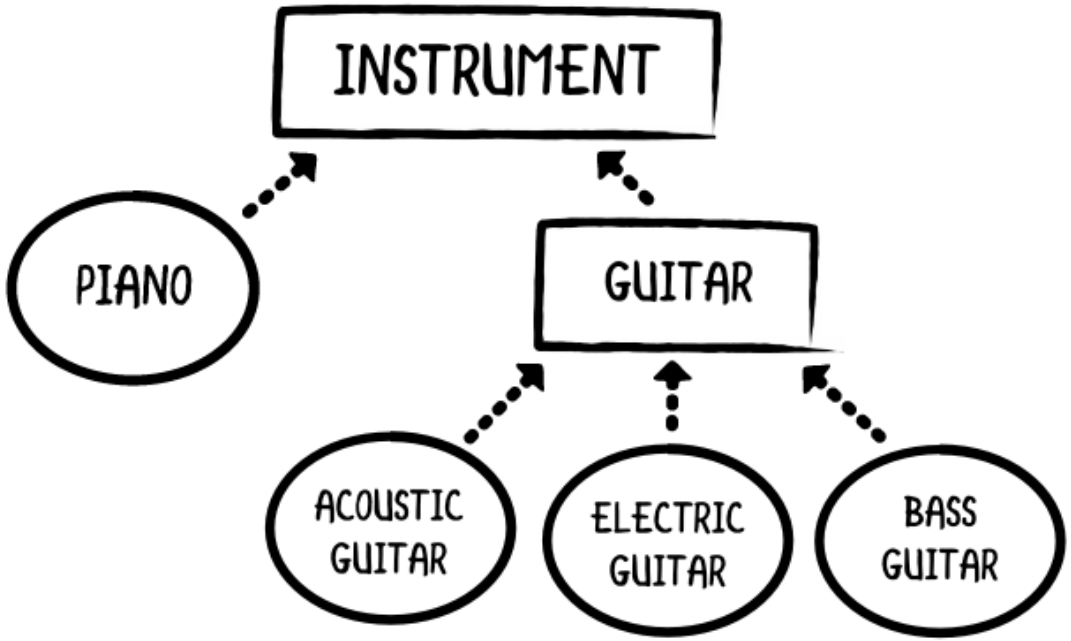
Languages are not object-oriented as much as the language environment supports OOP. Scala is another great example of a language that supports both functional and object-oriented scripting.

Message passing, Polymorphism, Encapsulation, and Inheritance

1. Message passing - objects communicate by sending and receiving messages (methods)
2. Polymorphism - also called subtype polymorphism (can create instances of classes)
3. Encapsulation - A language construct that facilitates the bundling of data with the methods operating on that data
4. Inheritance - reuse of base classes to form derived classes

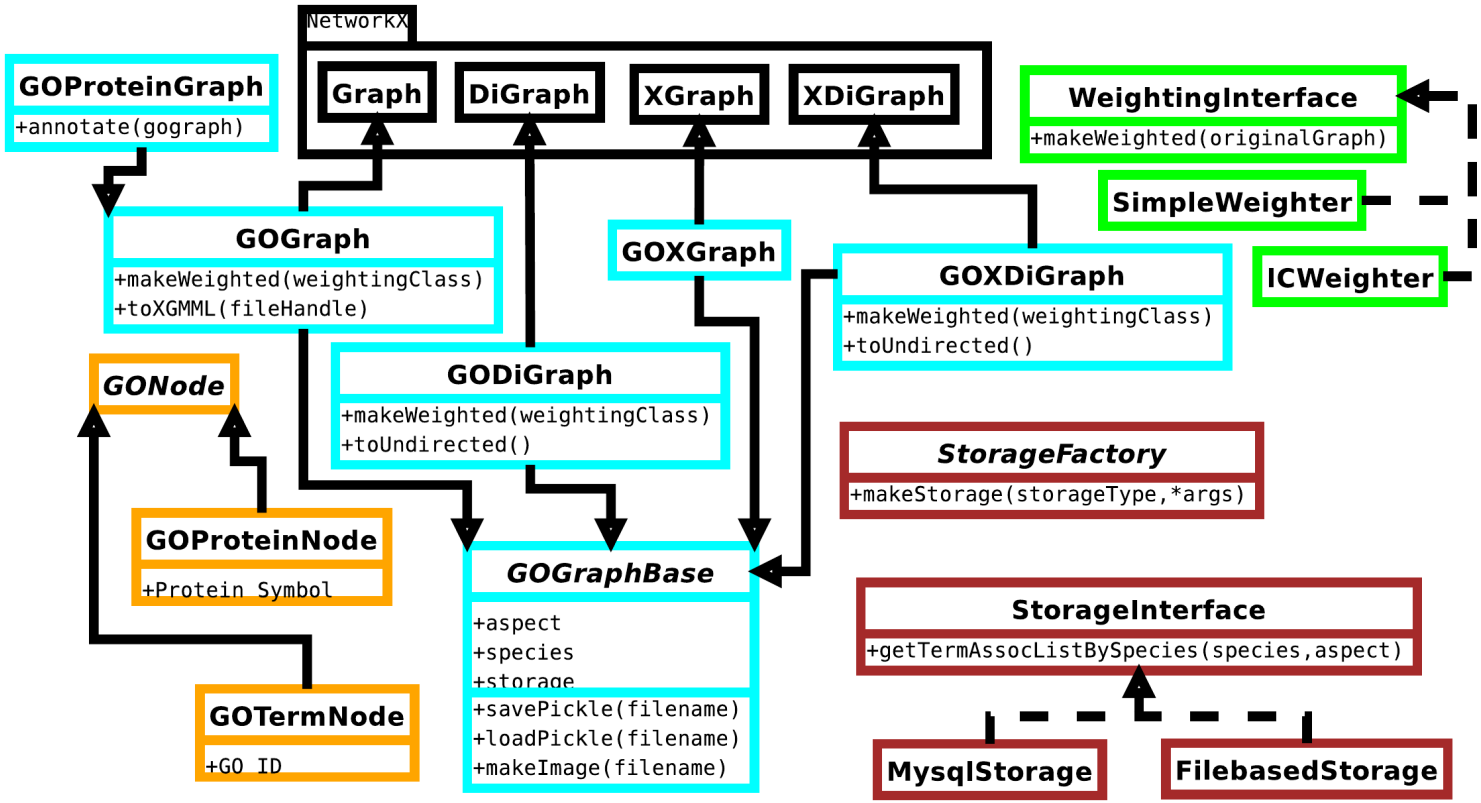
Python does not support full encapsulation

A paradigm for the real world



https://www.raywenderlich.com/160728/object-oriented-programming-swift

A library to model functional annotation in Biology



This figure is a UML class diagram in which classes are grouped according to their functionalities....

You have been working with objects all along

```
print(type([1,2,3]))
<class 'list'>

print(type(1))
<class 'int'>

print(type(lambda x: x**2))
<class 'function'>
```

In Ipython or Jupyter tab-completion shows the methods and attributes, but with `dir` you can filter

```
print([t for t in dir(np.array([1,2,3])) if 'set' in t.lower()])
['__setattr__', '__setitem__', '__setstate__', 'itemset', 'setfield', '
    setflags']
```

A class is a blueprint that describes the format of an object. How many classes? How many objects?








*args and **kwargs

Convention and shorthand to refer to a variable number of arguments:

```
For regular arguments, use *args:
    *args is a list
    def a_func(*args): takes multiple arguments
    Can also call function using a list

For keyword arguments, use **kwargs:
    **kwargs is a dict
    def a_func(**kwargs): takes multiple keyword arguments
    Can also call function using a dict
```

```
def print_all(*args):
    for count,item in enumerate(args):
        print('{0}. {1}'.format(count, item))

print(print_all('earth','air','water','fire'))
```

```
0. earth
1. air
2. water
3. fire
```

```
def list_all(**kwargs):
    for name, item in kwargs.items():
        print('{0} = {1}'.format(name, item))

list_all(blue_whale = 'Mammalia', lady_bug = 'Insecta')
```

```
blue_whale = Mammalia
lady_bug = Insecta
```

Magic Methods

Special methods, indicated by double underscore, that you can use to give ubiquitous functionality of some operators to objects defined by your class.

Magic method	Purpose
`__init__(self, ...)`	Constructor, initializes the class
`__repr__(self)`	Defines format for how object should be represented
`__len__(self)`	Returns number of elements in an object
`__gt__(self, other)`	Provides functionality for the $>$ operator
`__add__(self, other)`	Provides functionality for the $+$ operator
`__iadd__(self, other)`	Provides functionality for the $+=$ operator

https://www.python-course.eu/python3_magic_methods.php

Morning Breakout

Write a class to make an n-sided die

After a die is instantiated let the user be able to query:

1. How many sides it has
2. What number is face up (its value)

Also, let the user be able to:

1. Roll the die
2. Check compare the values of two die ($>, <, ==, \geq, \leq$)

You may use the `Simple.py` or the `Fruit.py` template. Also, before you write any code **Write out the pseudocode**.

Check-in questions

Core questions

1. What is the difference between an object and a class?
2. What is the difference between an attribute and a method?
3. What is the role of `self` in defining a class?
4. What can be used to give a custom class functionality similar to other classes?
5. How can we see the attributes and methods available on an object?

Objectives

Morning

- ✓ Compare and contrast functional and object oriented programming
- ✓ Given the code for a python class be able to: instantiate, call methods and access the attributes
- ✓ Write the python code for a simple class
- ✓ Design a program or algorithm in object oriented fashion
- ✓ Be able to implement several magic methods

Afternoon

- Intro to design patterns – thinking in an object oriented fashion
- Intro to functional programming – be able to compare and contrast functional and object oriented programming
- Intro to workflows

Functions, classes, modules and packages

- function - A block of organized, reusable code that is used to perform a single related action
- class - A template of reusable code that creates objects containing attributes and methods
- module - A file containing Python definitions and statements (e.g `mylib.py`)
- package - Packages are a way of structuring Python's module namespace
- library - A generic term for code designed to be usable by many applications
- script - a executable module

- All packages are modules, but not all modules are packages
- Any module that contains a `__path__` attribute is considered a package.
- Packages may be installable via `run.py` and registered in PyPI

The Python Package index is where many packages live,

OOP design

Build classes via:

- Composition/aggregation:
  - Class contains an object of another class with the desired functionality
  - Often, just basic types: `str`, `float`, `list`, `dict`, etc.
  - *HasA* $\Rightarrow$ use members, aggregation
- Inheritance
  - Class specializes behavior of a base class
  - *IsA* $\Rightarrow$ use inheritance
  - In some cases, derived class uses a *mix-in* base class only to provide functionality, not polymorphism

Interfaces

An interface is a contract between the client and the service provider:

- Isolates client from details of implementation
- Client must satisfy preconditions to call method/function
- Respect boundary of interface:

- Library/module provides a service
- Clients only access resource/service via library
- Then bugs arise from arise incorrect access or defect in library

Test Driven Development (TDD)

Make sure your interface is intuitive and friction-free:

- Use unit tests or specification test
  - To verify interface is good before implementation
  - To exercise individual functions or objects before application is complete
  - Framework can setup and tear-down necessary test fixture
- Stub out methods using pass
- Test Driven Development (TDD):
  - Red/Green/refactor
  1. Write unit tests
  2. Verify that they fail (red)
  3. Implement code (green)
  4. Refactor code (green)
- Use a unit test framework – unittest (best), doctest, or nose

Verification and Validation in Scientific Computing discusses rigorous framework to ensure correctness

Afternoon Breakout

Write a unit test to test for object equality

1. `run-tests.py` - controls all tests
2. `unittests` - is a folder were the test scripts live
3. `unittests/__init__.py` - controls what is run by `run-tests.py`
4. `unittests/DieTest.py` - contains the actual tests for the Dice class

1. Fill in the `testEquality` function
2. Using the red/green/refactor procedure to write a new test for the `__lt__` magic method in `BaseDie`

In part II, Just test something simple—exposure to this framework is what is important not the coverage

Design Patterns

Many design patterns exist to standardize best practice and they are worth learning if you regularly develop software

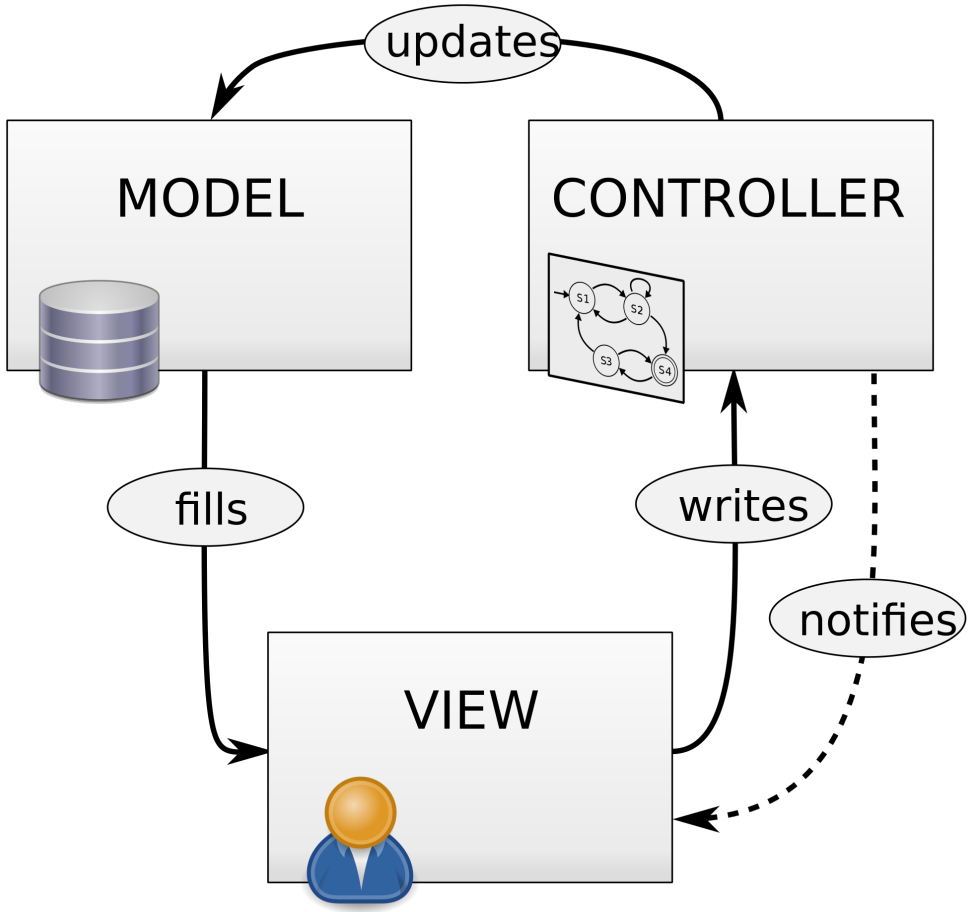
Examples

- MVC (Separate out Model, View, Controller)
- Abstract Factory - Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- Decorator - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality
- Observer - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Builder - Separate the construction of a complex object from its representation so that the same construction process can create different representations

Check out the book:

Design Patterns: Elements of reusable Object-Oriented Software

Design Patterns



https://commons.wikimedia.org/wiki/File:MVC_Diagram_(Model-View-Controller).svg

Check-in questions

Core questions

1. How do we know if we need functions, modules, or packages?
2. What are the three key components of OOP? How do they lead to better code?
3. How should I implement my code if the relationship is Is A? What if the relationship is Has A?
4. What is duck typing?
5. What should you do ensure an object is initialized correctly?
6. What are the benefits of TDD? What does Red/Green/Refactor mean?