Financial Software Engineering
Lecture 2

Co-Pierre Georg
AIFMRM

July 9, 2018

Today

Lecture

1. Object oriented programming
2. Inheritance
3. Abstract base classes
4. Metaclasses

Programming language models and paradigms

Programming language models

- In the last class we briefly introduced the Python programming language
- Most of you may have experience with other languages: R, Java, C# ...
- At first sight, we may distinguish between these languages based on their syntax $\rightarrow$ how we implement similar procedures across different languages

Programming language models

- We can also distinguish between programming languages on a more fundamental level
- We could speak about compiled and interpreted languages
- Compiled languages $\rightarrow$ can be understood and run directly by a PC (C, Fortran, Haskell)
- Interpreted languages $\rightarrow$ must be interpreted into something that can then be compiled directly (Python, R)

Programming paradigm

- We could also distinguish between how programming languages and their implementations are *structured*
- $\rightarrow$ Which **programming paradigm** they adhere to
- The two major paradigms are the (i) functional paradigm and (ii) the object oriented paradigm
- Importantly, these categories need not be mutually exclusive
- $\rightarrow$ many programming languages allow you to program in either paradigm
- Rather, the choice of paradigm depends on the functionality you'll require and the nature of the application

Functional languages

- Key feature $\rightarrow$ separation between data and actions

1. Create a data frame of countries and their GDP
2. Write a change_gdp function which changes the GDP of a given country
3. Replace the GDP value in the original data frame with the new value generate by change_gdp

Object oriented languages

- Key feature $\rightarrow$ Integration of data and actions

1. Create a country class and assign a GDP attribute
2. Create a specific instance of the country class for each country
3. Write a method called SetGdp which changes the GDP of a specific object
4. Replace the GDP value for a specific country using the SetGdp method

Object oriented programming

- We'll cover the implementation of object oriented programming (OOP) using Python
- OOP is a powerful programming paradigm
- $\rightarrow$ useful in keeping code ordered, especially in large projects
- OOP languages have 4 main building blocks: (i) classes (ii) objects (iii) attributes and (iv) methods
- Classes **create** objects, which **have** attributes and **execute** methods
- These building blocks allow us to integrate data and action

OOP building blocks

- **Class** $\rightarrow$ a type which creates objects that share features of the type
- **Object** $\rightarrow$ instance of a certain class type with attributes and methods
- **Attributes** $\rightarrow$ features of the class and all instances of the class
- **Methods** $\rightarrow$ functions unique to a class and all instances of the class which use information about an object to return results or change the object

OOP features

- Two features make OOP special: **inheritance** and **encapsulation**
- Inheritance $\rightarrow$ allows new objects to take on the attributes and methods of other objects
- Encapsulation $\rightarrow$ data, attributes and methods are all contained within an object
- Inheritance enables the re-use of code and encapsulation helps structure data and functions
- These concepts will make more sense after some examples

Python objects

- In the previous lecture we spoke about various data types in Python
- Each of these data types represent what we call an object
- Each object is also a specific instance of a type
- We call this type, the class of an object
- When we do the following

```
name = "John Doe"
```

- The variable `name` represents an object of the type, or class, string

Python objects

- Since, everything in Python is an object, learning Python involves learning how to create, manipulate and work with objects
- We'll use classes to create objects
- Classes $\rightarrow$ provide the blueprint for creating objects: what objects need to have and need to do, to be a valid object
- We'll also use classes to assign attributes and methods to objects
- Attributes $\rightarrow$ features of the object
- Methods $\rightarrow$ functions which can manipulate the object

Python class syntax

- The blueprint for all Python classes looks something like this

```
class ClassName():

    class_feat = "example"

    def __init__(self, arg1, arg2):
        self.arg1 = arg1
        self.arg2 = arg2

    def some_method(self):
        print(self.arg1)
```

- We'll go through each of these aspects in detail

Class

- Represents a logical grouping of data and functions (methods)
- A class creates a new type where objects are instances of the class
- $\rightarrow$ a blueprint for creating objects
- When we create a variable and assign it a value of 10, we say that the variable belongs to the integer class
- Objects of the integer class have certain attributes that distinguish them from other classes like strings
- Furthermore, we can execute certain actions with integers (multiplication, division etc)

Class

- When we work with classes, we'll do one of two things (i) create a class or (ii) use an instance of a class
- *Creating* a class involves defining the class name, attributes and methods
- *Using* a class involves creates new instances of objects which we can then edit or manipulate
- We call this separation of the actions of a class, **abstraction**

Class

- To create a new class, we call the `class` statement and provide a class name

```
class ClassName():
```

- We'll now replace `ClassName()` with something more meaningful
- Creating the blueprint for a digital cat, we now create a cat class

```
class Cat():
```

- **Importantly** $\rightarrow$ creating a cat class does not mean we've created a cat object; rather, we've created the blueprint or instruction manual for how to create one

Class object attributes

- In the next step, we step we'll assign an attribute, `species`
- An attribute is a feature, or variable, which all instances of a class require
- We assign this variable within our class

```
class Cat():

    species = "mammal"
```

- We call this a class object attribute since it is true for every instantiation (object) of the `Cat()` class

Class object attributes

- To create our cat class we call the class and assign it to a name
- We now have two objects, called `garfield` and `kitty` which are specific instances of our cat class

```
garfield = Cat()
kitty = Cat()
```

- We can now examine the attribute `species`
- We have two different objects of the class cat which share a common species

```
garfield.species
>>> 'mammal'

kitty.species
>>> 'mammal'
```

Initializing a class

- While `species` represents a variable which is identical for all instances of cats, we may have other variables which are instance specific
- For example, all cats have a name and a color but this is unique to the specific instance of a cat
- In this case, we'll need instance specific attributes as opposed to class specific attributes
- To create these attributes, we'll use a special method in Python, namely `__init__`

Initializing a class

- The `init` method in Python has a special significance - it is the first method called when an instance of a class is created
- `init` takes arguments or parameters and assigns these to attributes of a specific instantiation of an object from the class
- We use the `init` method to create instance specific variables that are allowed to differ across different instances of a class

```
def __init__(self, arg1, arg2):
    self.arg1 = arg1
    self.arg2 = arg2
```

Initializing a class

- While functions can also take parameters, methods are special class of functions that are (i) class specific and (ii) typically inherit a very specific parameter, `self`
- `self` refers to the specific instance of an object being created
- When defining an instance specific attribute, `self` should always be passed as an attribute

```
def __init__(self, arg1, arg2):
    self.arg1 = arg1
    self.arg2 = arg2
```

Initializing a class

- `init` takes parameters or inputs and assigns these inputs to instance specific attributes

```
def __init__(self, arg1, arg2):
    self.arg1 = arg1
    self.arg2 = arg2
```

- `arg1` and `arg2` are passed as inputs and merely represents local variables
- `self.arg1` and `self.arg2` represents the attributes `arg1` and `arg2` that all objects of the class share, but who's value is unique to the object `self` currently being created
- This will make more sense after an example

Initializing a class

- Back to our example, we know that while all cats are mammals, they have different names and colors
- We now stipulate that every time a new object of the type cat is created, the parameters `name` and `color` must be included
- What makes this different to a function is that by including, `self` as an additional parameter, we can change the name and color of every instance of the Cat type

```
class Cat():

    species = "mammal"

    def __init__(self, name, color):
        self.name = name
        self.color = color
```

Initializing a class

```
class Cat():

    species = "mammal"

    def __init__(self, name, color):
        self.name = name
        self.color = color
```

- While every cat must have a name and color, these features will be different in every cat
- We implement this in `self.name = name` and in `self.color = color`
- In this way, the `self` variables allow us to assign an attribute to the specific cat we are creating, as opposed to all cats

Instantiation

- To create a specific instance of the cat class, we call the cat class with the required arguments

```
garfield = Cat('Garfield', 'Orange')
kitty = Cat('Kitty', 'Black')
```

- Note now that since the `init` methods requires two arguments, we cannot create a new instance of the cat class without these arguments

```
salem = Cat()
>>> TypeError: __init__() missing 2 required positional arguments:
               'name' and 'color'

salem = Cat('Salem')
>>> TypeError: __init__() 1 required positional argument: 'color'
```

Instantiation

- Examining our objects, we now see two cat objects, who share a species, but have different colors

```
garfield.species
>>> 'mammal'
kitty.species
>>> 'mammal'

garfield.color
>>> 'Orange'
kitty.color
>>> 'Black'
```

- What `__init__` together with `self` has allowed us to do is add attributes to our two objects, both of type cat, which are unique to each instantiation

Methods

- We call the function we've just used `__init__()`, a method
- Methods are special types of functions that only exist within classes
- They are special given that the method is unique to objects of a specific class
- For example, a method that takes the square root of an real number is unique to the integer or real number class

```
import math

math.sqrt(9)
>>> 3.0

math.sqrt('hello')
>>> TypeError: must be real number, not str
```

Methods

```
class ClassName():

    class_feat = "example"

    def __init__(self, arg1, arg2):
        self.arg1 = arg1
        self.arg2 = arg2

    def some_method(self):
        print(self.arg1)
```

- Once we have assigned arguments to attributes of the object `self`, any method in the class that accepts the `self` object is able to call these arguments
- This allows different methods to use object attributes as inputs, return results or change the objects themselves

Methods

- Implementing this in our cat class

```
class Cat():

    species = "mammal"

    def __init__(self, name, color):
        self.name = name
        self.color = color

    def change_name(self):
            self.name = 'KitKat'
```

- We create a method `change_name`, which takes `self` and uses it to change the name of our cat
- Importantly, the name is only changed if we call the method `change_name`

Methods

- Calling the method on a cat object, we see the desired output

```
anon_cat = Cat("Anon", "Orange")
anon_cat.name
>>> 'Anon'

anon_cat.change_name()
anon_cat.name
>>> 'KitKat'
```

Special methods

- The `__init__()` method we've just used represents one of a range of special methods in Python
- These methods are identified by their use of underscores
- `__init__()`, `__str__()`, `__len__()` and `__del__()`
- Allow one to use Python specific functions on objects created via a class
- The `print()` method for example, exists natively in Python

```
print(garfield)
>>> <__main__.Cat object at 0x110594e48>
```

- When we print our cat object, we see it is indeed of type Cat
- What if we wanted the print function to return us some information about our object?

Special methods

- `__str__()` is a special method that tells Python what to do when a native Python method which takes an object of type string, should do on objects of another class

```
class Cat():

    species = "mammal"

    def __init__(self, name, color):
        self.name = name
        self.color = color

    def change_name(self):
            self.name = 'KitKat'

    def __str__(self):
        return "Name: " + self.name + ", color: " + self.color
```

```
print(garfield)
>>> Name: Garfield, Color: Orange
```

Polymorphism

- Another important Python feature is Polymorphism
- $\rightarrow$ different classes can share the same method and attributes names

```
class Dog:
    def __init__(self, name):
        self.name = name

    def speak(self, name):
        return "I am " + self.name

class Cat:
    def __init__(self, name):
        self.name = name

    def speak(self, name):
        return "I am " + self.name
```

Inheritance

Inheritance

- Our previous examples have highlighted one of OOP's key features, encapsulation
- $\rightarrow$ data, attributes and methods are all contained within an object
- Another powerful feature of OOP is inheritance $\rightarrow$ allows new objects to take on the attributes and methods of other objects
- This allows us to efficiently reuse code by implementing types and sub-types (parent/child)
- Instead of retyping these features for every class we type them once in the base class, and have every sub-class inherit this base class itself

Inheritance

- In our example, we created a specific animal, namely a cat, which had a name and a color
- What if we wanted to create a dog object? The dog object also represents an animal with a name and a color
- We are allowed to do this as a result of polymorphism

```
class Cat():

    def __init__(self, name, color):
        self.name = name
        self.color = color

class Dog():

    def __init__(self, name, color):
        self.name = name
        self.color = color
```

Inheritance

- This is an inefficient exercise - what if we wanted to add 50 more animals?
- We can solve this problem by creating a base animal class which has a name and color attribute and then link this to various sub-classes of specific animal types

Inheritance

```
class Animal():

    def __init__(self, name, color):
        self.name = name
        self.color = color

class Cat(Animal):
    pass
```

- By passing the animal class as a parameter to the cat class, the cat class inherits the animal class and it's attributes and methods

Inheritance

- Now, when we create a object of type cat, we inherit the attributes of the type animal

```
garfield = Cat('Garfield', 'Orange')

garfield.name
>>> 'Garfield'

garfield.color
>>> 'Orange'
```

- We call `Animal` the base or parent class and `Cat`, the subclass or child class

Inheritance

- Now, adding a dog class is trivial

```
class Animal():

    def __init__(self, name, color):
        self.name = name
        self.color = color

class Cat(Animal):
        pass

class Dog(Animal):
    pass
```

Let's give Dog and Cat some methods

```
class Animal():

    def __init__(self, name, color):
        self.name = name
        self.color = color

class Cat(Animal):

    def meow(self):
        return "Meow! My name is " + self.name

class Dog(Animal):

    def bark(self):
        return "Woof! My name is " + self.name
```

Inheritance

- Now, when we call the method meow (bark) on a object of type cat (dog) we receive the desired output
- Note however that when we call the method meow (bark) on a object of type dog (cat) we get a very informative error message

```
garfield = Cat('Garfield', 'Orange')
garfield.meow()
>>>'Meow! My name is Garfield'

garfield.bark()
>>> AttributeError: 'Cat' object has no attribute 'bark'

lassie = Dog('Lassie', 'Brown')
lassie.bark()
>>> 'Woof! My name is Lassie'

lassie.meow()
>>> AttributeError: 'Dog' object has no attribute 'Meow'
```

Abstract Base Classes

Abstract base classes

- Up to now we've used base classes to (i) instantiate objects and (ii) to create subclasses
- In this case, we can think of base classes as playing a prescriptive role $\rightarrow$ stipulates what attributes all objects and subclasses require and what methods these objects and subclasses can inherit
- This is especially true in the instantiation process
- The base class `Animal` requires two attributes, `name` and `color` - without providing these parameters in the initialization, Python will not create the object

```
Salem = Cat()
TypeError: __init__() missing 2 required positional arguments: 'name'
           and 'color'
```

Abstract base classes

- Note that while a base class can prescribe what attributes subclasses require, it does not do the same for methods
- So far, while our animals have all been required to have a name and a color, there has been no requirement that they need to perform certain actions, like walking or eating
- In this case, while all objects of type `Animal` share certain attributes, they may also share certain methods
- We could obviously implement these methods in the base class
- This approach however only makes sense if the action is executed in the same way for all subclasses

Abstract base classes

- In our example however, while all animals are capable of moving, animals do this in different ways: walking, slithering, swimming ...
- In cases like this, we would be unable to implement a method called `move` in the base class, given each animal may move differently
- However, we would still require all subclasses of animals to be able to `move`

Abstract base classes

- We can address this making use of abstract base classes or ABCs
- $\rightarrow$ a type of base class which contains one or more abstract methods
- $\rightarrow$ declared methods that contain no implementation of the method
- $\rightarrow$ meant to be inherited from and never instantiated
- $\rightarrow$ allow us to specify which methods a subclass requires while still ensuring each subclass has flexibility in its implementation of the method

Abstract base classes

- In our example, all subclasses of animal need to be able to move, in whichever way they choose to do so
- In addition to our base class, let's now create an abstract base class

```
from abc import ABC

class BaseAnimal(ABC):

    @abc.abstractmethod
    def move(self):
        pass

class Animal():

    def __init__(self, name, color):
        self.name = name
        self.color = color
```

Abstract base classes

- We now add a new Snake class which inherits from BaseAnimal and Animal

```
class Snake(BaseAnimal,Animal):

    def hiss(self):
        return "Hiss! My name is " + self.name
```

- If we try and create a new Snake object without the subclass having a method move, we see an error

```
nagini = Snake("Nagini", "Green")
>>> TypeError: Cant instantiate abstract class Snake with abstract
                methods move
```

Abstract base classes

```
nagini = Snake("Nagini", "Green")
>>> TypeError: Cant instantiate abstract class Snake with abstract
                methods move
```

- Without a method called move we cannot create an object from a class that inherits from BaseAnimal
- Adding the method now, we are able to successfully create an object of the class Snake, which is a subclass of both Animal and BaseAnimal

```
class Snake(BaseAnimal, Animal):

    def __init__(self, name, color):
        Animal.__init__(self, name, color)

    def hiss(self):
        return "Hiss! " + Animal.speak(self)

    def move(self):
        return "Slithers"
```

Abstract base classes

- Abstract base classes are a powerful tool to create subclasses with clearly defined requirements in terms of their functionality in a way that still retains flexibility
- As a result, abstract base classes are typically seen as "templates"
- ABCs can be very useful especially in large projects where many individuals contribute and use the code
- These templates effectively serve as a blueprint or formula for creating generic classes reducing many of the (preventable) errors that typically occur

Metaclasses

Metaclasses

- Earlier we spoke about how everything in Python is an object
- `my_int = 2` is of type **int**, `my_tuple = c(1,2)` is of type **tuple** etc
- Then we looked at how to use classes to create objects
- Objects represented instances of the class
- The cats we created were objects of the type Cat
- Turns out, classes are objects too!
- Specifically, classes are an object of the type, `type`

Metaclasses

```
class Cat():
    pass

c = Cat()
type(c)

# c is of type, Cat
>>> __main__.Cat

type(Cat)

# Cat is of type, type
>>> type

# The same is true for all the native type classes
type(int), type(list), type(tuple), type(float)
>>> (type, type, type, type)

# Also, the type of a type, is a type!
type(type)
>>> type
```

Metaclasses

- We call the class, `type`, a **metaclass**
- $\rightarrow$ a class whose instances are classes
- Any class in Python 3 is an instance of a metaclass, the default being `type`
- Writing and using metaclasses is a convenient way to dynamically create classes $\rightarrow$ a class factory

Metaclasses

- In reality, the average Python user will never come across a usecase for metaclasses
- Still, it's a useful concept and framework to know about, should you ever require the functionality
- Tim Peters - *Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why)*[1]

[1] `https://realpython.com/python-metaclasses/`

Advantages of OOP

- We've touched on OOP briefly, but as you spend more time with it you'll realize many of the benefits
- We can effectively store data and methods together
- Abstraction allows us to separately define how to implement an object vs how to use an object
- Inheritance allows us to effectively re-use code and better structure our classes and objects
- Abstract base classes provide a framework for better creating templates for our classes

OOP going forward

- For now it's important to understand the concepts involved in OOP rather than the implementation
- We'll continue with more OOP in the next class
- You'll learn how to set up with Python and cover extensive examples of OOP in the next two tutorials with lots of exercises and take-home practice questions
- Understanding OOP now will make the move to JavaScript easier later in the course