Object Oriented

Aula 3

Object-oriented?

Object-oriented simply means, "functionally directed toward modeling objects"

It is one of many techniques used for modeling complex systems by describing a collection of interacting objects via their data and behavior

- Object-oriented Analysis (OOA)
- Object-oriented Design (OOD)
- Object-oriented Programming (OOP)

Objects and classes

As an example, we can assume that apples go in barrels and oranges go in baskets. Now, we have four kinds of objects:

- apples, oranges, baskets, and barrels

In object-oriented modeling, the term used for kinds of objects is class

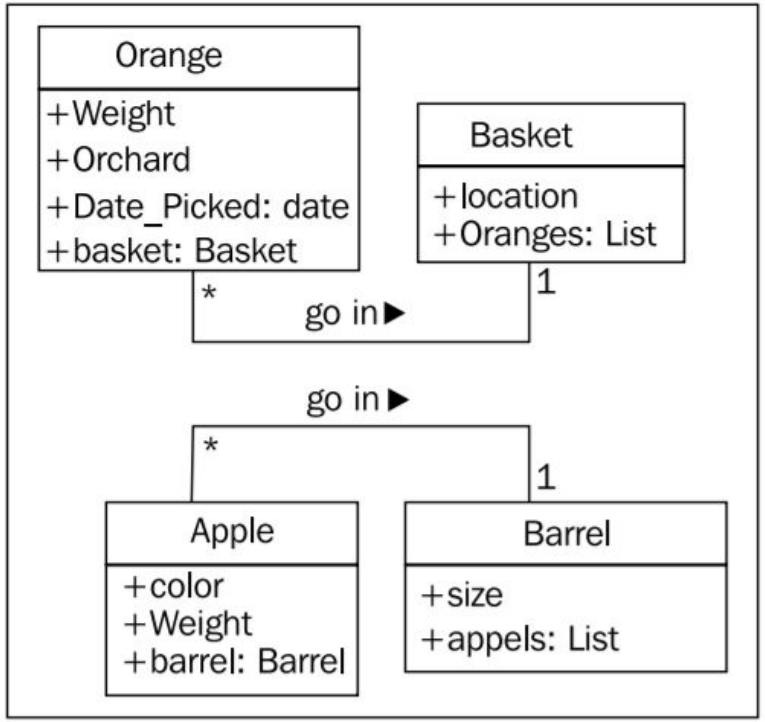
- In technical terms, we now have four classes of objects

Classes describe objects

Objects are instances of classes that can be associated with each other

Specifying attributes and behaviors

- An object instance is a specific object with its own set of data and behaviors
- We can also specify the type for each attribute
- In our fruit farming example, so far, our attributes are all basic **primitives**
- But there are implicit attributes that we can make explicit: the **associations**
- Behaviors are actions called **methods**



Hiding details and creating the public interface

The interface is the collection of attributes and methods that other objects can use to interact with that object:

- They do not need, and are often not allowed, to access the internal workings of the object

Ex: Interface to the television is the remote control. Each button on the remote control represents a method that can be called on the television object

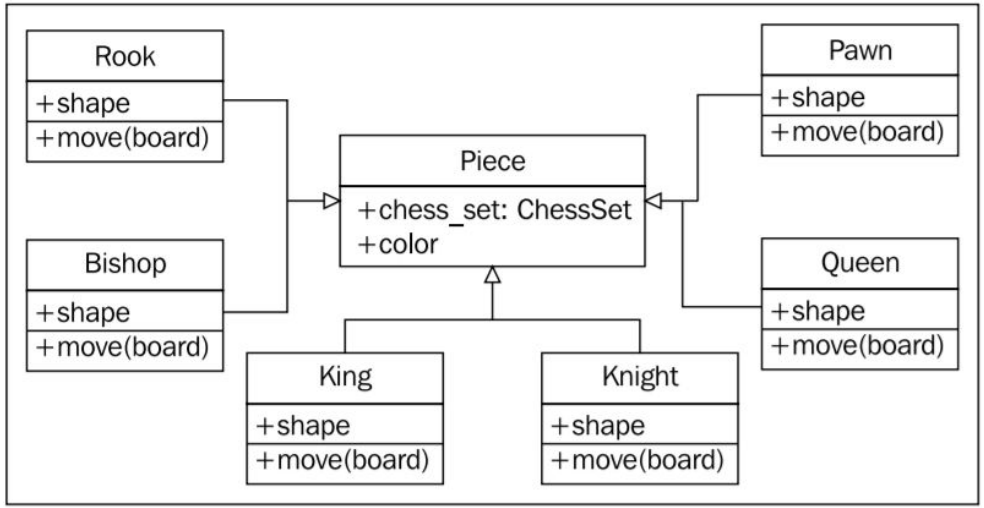
Information hiding is also sometimes referred to as encapsulation

The model is an abstraction of a real concept

Composition and inheritance

Composition is the act of collecting together several objects to compose a new one

Object-oriented programming, instead of inheriting features and behaviors from a person, a class can inherit attributes and methods from another class. Ex. Chess:



Inheritance provides abstraction

Polymorphism is the ability to treat a class differently depending on which subclass is implemented. In the Chess example:

- The board need not ever know what type of piece it is dealing with. All it has to do is call the move method and the proper subclass will take care of moving it as a Knight or a Pawn

Polymorphism is a word that is rarely used in Python programming

Multiple inheritance

Allows a subclass to inherit functionality from multiple parent classes

For example, an object designed to connect to a scanner and send a fax of the scanned document might be created by inheriting from two separate ***scanner*** and ***faxer*** objects

Must be used with caution. Is important to recognize that owning a hammer does not turn screws into nails

Explaining communication between objects

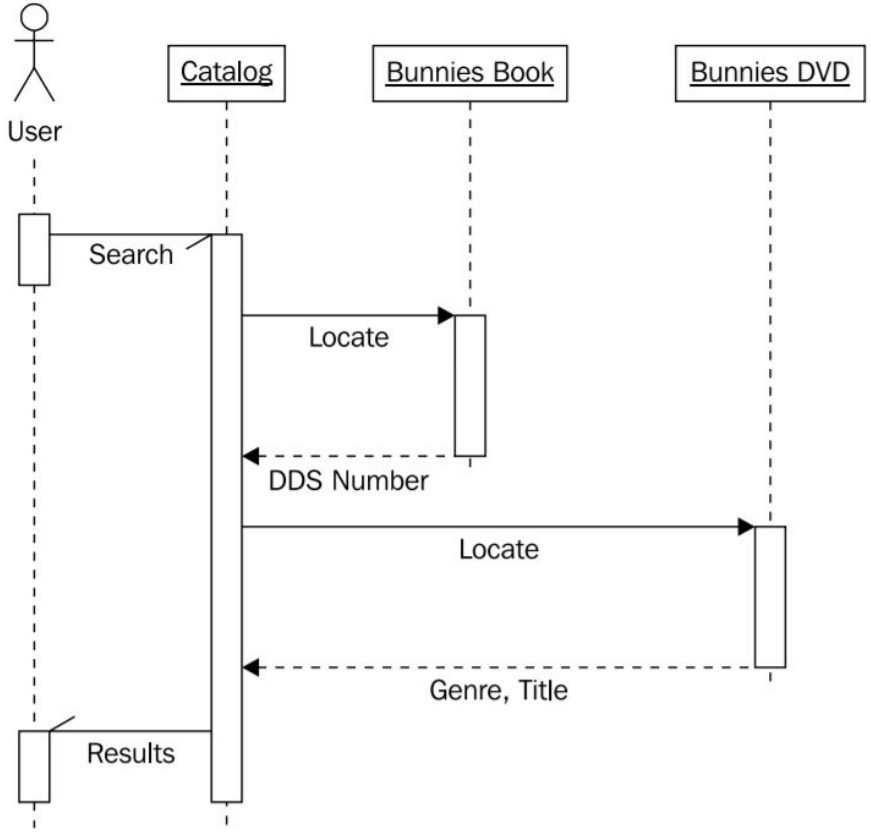
UML **sequence diagram**

Dashed line = lifetime of the object

Horizontal arrows between the lifelines indicate specific messages

Solid arrows represent methods being called

Dashed arrows with solid heads represent the method return values



Exercises

UML Diagrams: https://www.smartdraw.com/uml-diagram/examples/

Pick one and model a “student - professor” classes

Creating Python classes

```
class MyFirstClass:

    Pass

a = MyFirstClass()

b = MyFirstClass()

print(a) ## <__main__.MyFirstClass object at 0xb7b7faec>

print(b) ## <__main__.MyFirstClass object at 0xb7b7fbac>
```

Adding attributes

```
class Point:
    pass

p1 = Point()
p2 = Point()
p1.x = 5
p1.y = 4
```

```
p2.x = 3
p2.y = 6

print(p1.x, p1.y)
print(p2.x, p2.y)
```

Making it do something

```
class Point:
        def reset(self):
                self.x = 0
                self.y = 0

p = Point()
p.reset()
print(p.x, p.y)

# Same behaviour
p = Point()
Point.reset(p)
print(p.x, p.y)
```

- The **self** argument to a method is simply a reference to the object that the method is being invoked on
- Calling a method on the p object automatically passes that object to the method

Multiple arguments to a method

```
import math
class Point:
        def move(self, x, y):
                self.x = x
                self.y = y
        def reset(self):
                self.move(0, 0)
        def calculate_distance(self, other_point):
                 return math.sqrt((self.x - other_point.x)**2 + (self.y - other_point.y)**2)

point1 = Point()
point2 = Point()
```

Multiple arguments to a method

```
point1.reset()
point2.move(5,0)
print(point2.calculate_distance(point1))
assert (point2.calculate_distance(point1) == point1.calculate_distance(point2))
point1.move(3,4)
print(point1.calculate_distance(point2))
print(point1.calculate_distance(point1))

# Exception #
point = Point()
point.x = 5
print(point.x)
print(point.y)
```

Constructor

```
class Point:
        def __init__(self, x, y): # def __init__(self, x=0, y=0)
                self.move(x, y)
        def move(self, x, y):
                self.x = x
                self.y = y
        def reset(self):
                self.move(0, 0)

# Constructing a Point
point = Point(3, 5)
print(point.x, point.y)
```

Docstrings

```
import math
class Point:
        'Represents a point in two-dimensional geometric coordinates'
        def __init__(self, x=0, y=0):
                '''Initialize the position of a new point. The x and y coordinates can be
specified. If they are not, the point defaults to the origin.'''
                self.move(x, y)
        def move(self, x, y):
                "Move the point to a new location in two-dimensional space."
                self.x = x
                self.y = y
```

Modules and packages

Modules are simply Python files

If we have two files in the same folder, we can load a class from one module for use in the other module.Ex:

```
from database import Database as DB
db = DB()
# Do queries on db
```

Organizing the modules

Package is a collection of modules in a folder. Name of the package is the name of the folder

```
parent_directory/
        main.py
        ecommerce/
                __init__.py
                database.py
                products.py
                payments/
                        __init__.py
                        paypal.py
                        authorizenet.py
```

Absolute imports

specify the complete path to the module, function, or path we want to import

```
import ecommerce.products
product = ecommerce.products.Product()
```

or

```
from ecommerce.products import Product
product = Product()
```

or

```
from ecommerce import products
product = products.Product()
```

Relative imports

If we are working in the products module and we want to import the Database class from the database module "next" to it, we could use a relative import:

- from .database import Database

"Use the database package inside the parent package", instead:

- from ..database import Database

Relative imports

We could delay creating the database until it is actually needed by calling an **initialize_database** function to create the module-level variable:

```
class Database: # the database implementation
    pass

database = None
def initialize_database():
    global database

database = Database()
```

Typically defined at the module but can also be defined inside a function

Classes can be defined anywhere

```
def format_string(string, formatter=None):
    '''Format a string using the formatter object, which is expected to have a
    format() method that accepts a string.'''

    class DefaultFormatter:
        '''Format a string in title case.'''
        def format(self, string):
            return str(string).title()

    if not formatter:
        formatter = DefaultFormatter()
    return formatter.format(string)
```

Classes can be defined anywhere

The **format_string** function accepts a string and optional formatter object, and then applies the formatter to that string
If no formatter is supplied, it creates a formatter of its own as a local class and instantiates it:

- hello_string = "hello world, how are you today?"
  print(" input: " + hello_string)
  print("output: " + format_string(hello_string))

Who can access my data?

Outside objects don't access a property or method (double underscore: __):

```
class SecretString: # A not-at-all secure way to store a secret string.
    def __init__(self, plain_string, pass_phrase):
        self.__plain_string = plain_string
        self.__pass_phrase = pass_phrase

    def decrypt(self, pass_phrase): #Show string if the pass_phrase is correct.
        if pass_phrase == self.__pass_phrase:
            return self.__plain_string
        else:
            return ''
```

Who can access my data?

```
secret_string = SecretString("ACME: Top Secret", "antwerp")
print(secret_string.decrypt("antwerp"))
print(secret_string.__plain_text)
```

It works !! … maybe not …

```
print(secret_string._SecretString__plain_string)
```

Exercises

Implement “Student” and “Professor” classes using:

- Class syntax
- Attributes and methods
- Initializers and constructors
- Modules and packages
- Relative and absolute imports
- Access control

Basic inheritance

Allows us to create "is a" relationships between two or more classes, abstracting common details into superclasses and storing specific ones in the subclass

Technically, every class we create uses inheritance

All Python classes are subclasses of the special class named object

- class MySubClass(object):

    pass

Class variables

```
class Contact:
    all_contacts = []
    def __init__(self, name, email):
        self.name = name
        self.email = email
        Contact.all_contacts.append(self)
```

The all_contacts list, because it is part of the class definition, is actually **shared by all instances of this class**

But what if some of our contacts are also suppliers that we need to order supplies from?

Supplier class that *“acts”* like a Contact

```
class Supplier(Contact):
    def order(self, order):
        print("We would send {} order to {}".format(order, self.name))

c = Contact("Some Body", "somebody@example.net")
s = Supplier("Sup Plier", "supplier@example.net")
print(c.name, c.email, s.name, s.email)
c.all_contacts
c.order("I need pliers")
s.order("I need pliers")
```

Supplier can do what Contact can do, and all the things it needs as supplier

Overriding and super

```
class Friend(Contact):
    def __init__(self, name, email, phone):
    self.name = name
    self.email = email
    self.phone = phone

class Friend(Contact):
    def __init__(self, name, email, phone):
    super().__init__(name, email)
    self.phone = phone
```

Multiple inheritance

“As a rule of thumb, if you think you need multiple inheritance, you're probably wrong, but if you know you need it, you're probably right”

```
class MailSender:
        def send_mail(self, message):
        print("Sending mail to " + self.email)
        # Add e-mail logic here

# Defining a new class that is both a Contact and a MailSender

class EmailableContact(Contact, MailSender):
        pass
```

Multiple inheritance

```
e = EmailableContact("John Smith", "jsmith@example.net")
Contact.all_contacts
e.send_mail("Hello, test e-mail here")
```

Polymorphism

Different behaviors happen depending on which subclass is being used, without having to explicitly know what the subclass actually is:

```
class AudioFile:
    def __init__(self, filename):
        if not filename.endswith(self.ext):
            raise Exception("Invalid file format")
        self.filename = filename


class MP3File(AudioFile):
    ext = "mp3"
    def play(self):
        print("playing {} as mp3".format(self.filename))
```

Polymorphism

```
class WavFile(AudioFile):
        ext = "wav"
        def play(self):
                print("playing {} as wav".format(self.filename))

class OggFile(AudioFile):
        ext = "ogg"
        def play(self):
                print("playing {} as ogg".format(self.filename))
```

Polymorphism

```
ogg = OggFile("myfile.ogg")
ogg.play()

mp3 = MP3File("myfile.mp3")
mp3.play()

not_an_mp3 = MP3File("myfile.ogg")
```

Duck Typing

Allows us to use any object that provides the required behavior without forcing it to be a subclass. The following example does not extend AudioFile, but it can be interacted with in Python using the exact same interface:

```
class FlacFile:
    def __init__(self, filename):
        if not filename.endswith(".flac"):
            raise Exception("Invalid file format")
    self.filename = filename

    def play(self):
        print("playing {} as flac".format(self.filename))
```

Exercises

- Add functionality to existing classes and built-ins using inheritance
- Share similar code between classes by abstracting it into a parent class
- Combine multiple threads of functionality using multiple inheritance
- Use polimorphism