15-112
Fundamentals of Programming

Week 4 - Lecture 3:
Intro to Object Oriented Programming (OOP)
Important terminology

- data type (type)
- class

$s = \text{set()}$  
Create an object/instant of type/class set.

$s$ is then a reference to that object/instant.
What is object oriented programming (OOP)?

1. The ability to create your own data types.

```python
s = "hello"
print(s.capitalize())
```

```python
s = set()
s.add(5)
```

These are built-in data types.

2. Designing your programs around the data types you create.
What is object oriented programming (OOP)?

Is every programming language object-oriented?

No. e.g. C

(So OOP is not a necessary approach to programming)

What have we been doing so far?

Procedural programming.

Designing your programs around functions (actions)

Is OOP a useful approach to programming?

Make up your own mind about it.
1. Creating our own data type

2. OOP paradigm
Suppose you want to keep track of the books in your library.

For each book, you want to store:
- title,
- author,
- year published

How can we do it?
Motivating example

Option 1:

book1Title = “The Catcher in the Rye”
book1Author = “J. D. Sallinger”
book1Year = 1951

book2Title = “The Brothers Karamazov”
book2Author = “F. Dostoevsky”
book2Year = 1880;

Would be better to use one variable for each book.

One variable to hold logically connected data together.
(like lists)
Motivating example

Option 2:

book1 = [“The Catcher in the Rye”, “J.D. Sallinger”, 1951]

book2 = list()
book2.append(“The Brothers Karamazov”)
book2.append(“F. Dostoevsky”)
book2.append(1880)

Can forget which index corresponds to what.

Hurts readability.
Motivating example

**Option 3:**

```python
dict()
```

Doesn’t really tell us what type of object book1 and book2 are.

They are just dictionaries.
Option 3:

```python
book1 = {
    "title": "The Catcher in the Rye",
    "author": "J.D. Sallinger",
    "year": 1951
}

book2 = {
    "title": "The Brothers Karamazov",
    "author": "F. Dostoevsky",
    "year": 1880
}

article1 = {
    "title": "On the Electrodynamics of Moving Bodies",
    "author": "A. Einstein",
    "year": 1905
}
```

Better to define a new data type.
Defining a data type (class) called Book

```python
class Book(object):
    def __init__(self):
        self.title = None
        self.author = None
        self.year = None
```

This defines a new data type named Book.

__init__ is called a constructor.
Defining a data type (class) called Book

class Book(object):
    def __init__(self):
        self.title = None
        self.author = None
        self.year = None
Defining a data type (class) called Book

class Book(object):
    def __init__(self):
        self.title = None
        self.author = None
        self.year = None

call __init__ with
    self = b

Creates an object
    of type Book

b refers to that object.

b = Book()
b.title = “Hamlet”
b.author = “Shakespeare”
b.year = 1602

Compare to:
b = dict()
b[“title”] = “Hamlet”
b[“author”] = “Shakespeare”
b[“year”] = 1602
class Book(object):
    def __init__(self):
        self.title = None
        self.author = None
        self.year = None

b = Book()
b.title = “Hamlet”
b.author = “Shakespeare”
b.year = 1602

b2 = Book()
b2.title = “It”
b2.author = “S. King”
b2.year = 1987

b refers to an object of type Book.
b2 refers to another object of type Book.
Creating 2 books

```python
b = Book()
b.title = "Hamlet"
b.author = "Shakespeare"
b.year = 1602

b2 = Book()
b2.title = "It"
b2.author = "S. King"
b2.year = 1987
```
class Book(object):
    def __init__(self, t, a, y):
        self.title = t
        self.author = a
        self.year = y

b = Book(“Hamlet”, “Shakespeare”, 1602)

b.title = “Hamlet”
b.author = “Shakespeare”
b.year = 1602
class Book(object):
    def __init__(self, title, author, year):
        self.title = title
        self.author = author
        self.year = year

b = Book("Hamlet", "Shakespeare", 1602)
class Book(object):
    def __init__(self, title, author):
        self.title = title
        self.author = author
        self.year = None

b = Book(“Hamlet”, “Shakespeare”)

b.title = “Hamlet”
b.author = “Shakespeare”
class Book(object):
    def __init__(self, title, author):
        self.title = title
        self.author = author
        self.year = None

b = Book(“Hamlet”, “Shakespeare”)
Using Book data type for library

```python
library = list()
userInput = None

while (userInput != "3"):
    print ("1. Add a new book")
    print ("2. Show all books")
    print ("3. Exit")
    userInput = input("Enter choice: ")

if (userInput == "1"):
    title = input("Enter title: ")
    author = input("Enter author: ")
    year = input("Enter year: ")
    b = Book(title, author, year)
    library.append(b)

elif (userInput == "2"):
    for book in library:
        print ("Title: " + book.title)
        print ("Author: " + book.author)
        print ("Year: " + book.year)

elif (userInput == "3"):
    print ("Exiting system.")

else:
    print ("Not valid input. Try again.")
```
Imagine you have a website that allows users to sign-up.

You want to keep track of the users.

class User(object):
    def __init__(self, username, email, password):
        self.username = username
        self.email = email
        self.password = password
userList = list()
userInput = None
while (userInput != "3"):
    print ("1. Login")
    print ("2. Signup")
    print ("3. Exit")
    userInput = input("Enter choice: ")
    if (userInput == "1"):
        username = input("Enter username: ")
        password = input("Enter password: ")
        if (findUser(userList, username, password) != None):
            loggedInMenu()
    elif (userInput == "2"):
        username = input("Enter username: ")
        password = input("Enter password: ")
        email = input("Enter email: ")
        user = User(username, password, email)
        userList.append(user)
    elif (userInput == "3"):
        print ("Exiting system.")
    else:
        print ("Not valid input. Try again.")
class Account(object):
    def __init__(self):
        self.balance = None
        self.numWithdrawals = None
        self.isRich = False

Account is the type.

a1 = Account()
a1.balance = 1000000
a1.isRich = True

Creating different objects of the same type (Account).

a2 = Account()
a2.balance = 10
a2.numWithdrawals = 1
```python
class Cat(object):
    def __init__(self, name, age, isFriendly):
        self.name = None
        self.age = None
        self.isFriendly = None

Cat is the type.

c1 = Cat("Tobias", 6, False)

Creating different objects of the same type (Cat).

c2 = Cat("Frisky", 1, True)
```
```python
class Rectangle(object):
    def __init__(self, x, y, width, height):
        self.x = x
        self.y = y
        self.width = width
        self.height = height
```

Rectangle is the type.

Creating different objects of the same type (Rectangle).

```python
r1 = Rectangle(0, 0, 4, 5)
r2 = Rectangle(1, -1, 2, 1)
```
```python
class Aircraft(object):
    def __init__(self):
        self.numPassengers = None
        self.cruiseSpeed = None
        self.fuelCapacity = None
        self.fuelBurnRate = None

Aircraft is the type.

a1 = Aircraft()
a1.numPassengers = 305
...

Creating different objects of the same type (Aircraft).
a2 = Aircraft()
...
```
class Time(object):
    def __init__(self, hour, minute, second):
        self.hour = hour
        self.minute = minute
        self.second = second

    Time is the type.

    Creating different objects of the same type (Time).

t1 = Time(15, 50, 21)
...
t2 = Time(11, 15, 0)
...
An object has 2 parts

1. **instance variables**: a collection of related data

2. **methods**: functions that act on that data

```
    s = set()
s.add(5)
```

This is like having a function called `add`:
```
    add(s, 5)
```

How can you define methods?
1. Creating our own data type

   Step 1: Defining the instance variables
   Step 2: Adding methods to our data type

2. OOP paradigm
Example: Rectangle

class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height

def getArea(rec):
    return rec.width * rec.height

r = Rectangle(3, 5)
print(“The area is”, getArea(r))

Defining a function that acts on a rectangle object
Example: Rectangle

```python
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def getArea(self):
        return self.width*self.height

r = Rectangle(3, 5)
print(“The area is”, r.getArea())
```

Defining a method that acts on a rectangle object
```python
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def getArea(self):
        return self.width * self.height

    def getPerimeter(self):
        return 2 * (self.width + self.height)

    def doubleDimensions(self):
        self.width *= 2
        self.height *= 2

    def rotate90Degrees(self):
        (self.width, self.height) = (self.height, self.width)
```

Example: Rectangle

r1 = Rectangle(3, 5)
r2 = Rectangle(1, 4)
r3 = Rectangle(6, 7)

print ("The width of r1 is \%d." % r1.width)
r1.width = 10

print ("The area of r2 is \%d." % r2.getArea())

print ("The perimeter of r3 is \%d." % r.getPerimeter())
r3.doubleDimensions()

print ("The perimeter of r3 is \%d." % r.getPerimeter())
class Employee(object):
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def printEmployee(self):
        print("Name: ", self.name)
        print("Salary: ", self.salary)

    def getNetSalary(self):
        return 0.75*self.salary

    def isRich(self):
        return (self.salary > 100000)

    def salaryInFuture(self, years):
        return self.salary * 1.03**years

    def fire(self):
        self.salary = 0
Example 2: Employee

e1 = Employee(“Frank Underwood”, 200000)
e1.printEmployee()  
print (e1.isRich())  
print (e1.salaryInFuture(10))  
print (e1.fire())  
print (e1.salary)
Example 3: Cat

class Cat(object):
    def __init__(self, weight, age, isFriendly):
        self.weight = weight
        self.age = age
        self.isFriendly = isFriendly

    def printInfo(self):
        print("I weigh ", self.weight, "kg.")
        print("I am ", self.age, " years old.")
        if (self.isFriendly):
            print("I am the nicest cat in the world.")
        else:
            print("One more step and I will attack!!!")

...
def feed(self, food):
    self.weight += food
    print(“It was not Fancy Feast’s seafood”)
    self.wail()

def wail(self):
    print(“Miiiiaaaaawwwwww”)
    self.moodSwing()

def moodSwing(self):
    self.isFriendly = (random.randint(0,1) == 0)
frisky = Cat(4.2, 2, True)
tiger = Cat(102, 5, False)

frisky.printInfo()
tiger.printInfo()

frisky.feed(0.2)
tiger.feed(3)

frisky.printInfo()
tiger.printInfo()
1. Creating our own data type
   Step 1: Defining the instance variables
   Step 2: Adding methods to our data type

2. OOP paradigm
The general idea behind OOP

1. Group together **data** together with the **methods** into one unit.

2. Methods represent the interface:
   - control how the object should be used.
   - hide internal complexities.

3. Design programs around objects.
Idea 1: group together data and methods

*Encapsulate* the **data** together with the **methods** that act on them.

**data**  
(fields/properties)

**methods**  
that act on the data

All in one unit
Idea 1 advantages

Adds another layer of organizational structure.

Our data types better correspond to objects in reality.

Objects in real life have
- properties
- actions that they can perform

Your new data type is easily shareable.
- everything is in one unit.
- all you need to provide is a documentation.
Example: Representing fractions

Rational numbers: a number that can be expressed as a ratio of two integers.

Also called fractions.

\[ \frac{a}{b} \]

- \( a \) = numerator
- \( b \) = denominator (cannot be 0)
Example: Representing fractions

class Fraction(object):
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

    def toString(self):
        return str(self.numerator) + " / " + str(self.denominator)

    def toFloat(self):
        return self.numerator / self.denominator

    def simplify(self):
        # code for simplifying

    def add(self, other):
        # code for adding

    def multiply(self, other):
        # code for multiplying

    ...
Example: Representing fractions

Everything you might want to do with rational numbers is packaged up nicely into one unit:

the new data type `Fraction`.
The general idea behind OOP

1. Group together data together with the methods into one unit.

2. Methods represent the interface:
   - control how the object should be used.
   - hide internal complexities.

3. Design programs around objects.
Idea 2: Methods are the interface

Methods should be the only way to read and process the data/fields.

  - don’t access data members directly.

If done right, the hope is that the code is:

  - easy to handle/maintain
  - easy to fix bugs

Can modify classes independently as long as the interface stays the same.
class Cat(object):

    def __init__(self, n, w, a, f):
        self.name = n
        self.weight = w
        self.age = a
        self.isFriendly = f

...
Expanding the Cat class (2/3)

```python
... def setWeight(self, newWeight):
    if (newWeight > 0):
        self.weight = newWeight

def getWeight(self):
    return self.weight

def getAge(self):
    return self.age

def setAge(self, newAge):
    if(newAge >= 0):
        self.age = newAge

... Instead of:
c = Cat(“tiger”, 98, 2, False)
c.weight = -1

do:
c = Cat(“tiger”, 98, 2, False)
c.setWeight(-1)
```
... 
def getName(self):
    return self.name

def getIsFriendly(self):
    return self.isFriendly

def feed(self, food):
    self.weight += food
    self.isFriendly = (random.randint(0,1) == 0)

There are no methods to directly change the name or isFriendly fields.
A comment about Struct
Idea 2: Methods are the interface

The Cat data type

- getName
- setName
- setWeight
- setAge
- isFriendly
- name
- weight
- age
- ...
Common Types of Methods

Observers

```python
def getName(self):
    return self.name

def getAge(self):
    return self.age
```

Usually named `getBla()`, where `Bla` is the field name.

Modifiers

```python
def setWeight(self, newWeight):
    if (newWeight > 0):
        self.weight = newWeight
```

Usually named `setBla(input)`, where `Bla` is the field name.
Common Types of Methods

... def getWeight(self):
    return self.weight

def getAge(self):
    return self.age

... def setWeight(self, newWeight):
    if (newWeight > 0):
        self.weight = newWeight

def setAge(self, newAge):
    if (newAge >= 0):
        self.age = newAge

...
The general idea behind OOP

1. Group together **data** together with the **methods** into one unit.

2. Methods represent the interface:
   - control how the object should be used.
   - hide internal complexities.

3. Design programs around objects.
Idea 3: Objects are at the center

Privilege data over action

**Procedural Programming Paradigm**

Decompose problem into a series of actions/functions.

**Object Oriented Programming Paradigm**

Decompose problem first into bunch of data types.

In both, we have actions and data types. Difference is which one you end up thinking about first.
Simplified Twitter using OOP

User
- name
- username
- email
- list of tweets
- list of following
- changeName
- printTweets

Tweet
- content
- owner
- date
- list of tags
- printTweet
- getOwner
- getDate

Tag
- name
- list of tweets
- ...

...
Managing my classes using OOP

**Grade**
- type
- value
- weight
- get value
- change value
- get weighted value
- ...

**Student**
- first name
- last name
- id
- list of grades
- add grade
- change grade
- get average
- ...

**Class**
- list of Students
- num of Students
- find by id
- find by name
- add Student
- get class average
- fail all
- ...

...
Summary

Using a class, we can **define** a new data type.

The new data type encapsulates:

- **data members** (usually called *fields* or *properties*)
- **methods** (operations acting on the data members)

The **methods** control how you are allowed to **read** and **process** the **data members**.

Once the new data type is defined:

  Can create **objects** (*instances*) of the new data type.

  Each **object** gets its own copy of the **data members**.

  Data type’s methods = allowed operations on the object