LECTURE 6 MODELLING THE WORLD WITH OBJECTS

Fundamentals of Programming - COMP1005

Department of Computing
Curtin University

Updated 12/9/19

Copyright Warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulation 1969

WARNING

This material has been copied and communicated to you by or on behalf of Curtin University of Technology pursuant to Part VB of the Copyright Act 1968 (the Act)

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice

Learning Outcomes

- Understand the main concepts in object-oriented programming and their value
- Read and explain object-oriented code
- Apply and create simple object-oriented Python code

CODE STRUCTURE
A REVIEW

Fundamentals of Programming

Lecture 6

Code Structure

- Across the semester we've learnt many elements of coding:
  - **Control structures**: if/else/elif, for loops, while loops
  - Creating and using **functions**
  - Creating and using **modules**
  - **Python style**: PEP-8, "readability counts"
  - **Data types**: int, float, string, list, array, set, dictionary
  - **Files**: text, csv
  - **Key packages**: numpy, scipy, matplotlib, pandas, random
  - **Environments**: python scripts, command lines, bash scripts, jupyter notebooks

Control structures: if, else, elif

- Truth values / boolean expressions can be used to control the flow of a program:

```
if (concession == "Y"):
    fare = 1.20
elif (multi == "Y"):
    fare = 5.00
else:
    fare = 3.00
```

- Note: the code within the if/elif/else block needs to be indented with <u>4 spaces</u>
- Multiple elif's can be used in sequence

Control structures: while loop

- While loops repeat a block of statements until the condition is false

```
finished = False

while not finished:
    if input("Finished? Y/N : ") == "Y":
        finished = True


n = 100
s = 0
counter = 1

while counter <= n:
    s = s + counter
    counter += 1

print("Sum of 1 until ",n ," : ", s)
```

Control structures: for loop

- For loops repeat for a set number of iterations

```
for num in range(10):
    print(num)                      # prints 0..9

for num in range(4,10):
    print(num)                      # prints 4..9

for num in range(20, -6, -2):
    print(num)                      # prints 20, 18, 16.. -4

n = 100
sum = 0
for counter in range(1,n+1):        # adds 1..100 = 5050
    sum = sum + counter
print("Sum of 1 until %d: %d" % (n,sum))
```

Why Functions?

- Makes your program easier to read and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it

https://en.wikibooks.org/wiki/Think_Python/Functions

Functions

Function name

Argument

Function header

```
def print_lyrics2(count):
    for ii in range(count):
        print("I'm a lumberjack, ",\
              "and I'm okay.")
        print("I sleep all night ",\
              "and I work all day.")
    return("I'm okay")
```

Function body

Return value

Return statement

General Code Structure

```
import matplotlib.pyplot as plt
import numpy as np

def calcheat(row,col):
    subgrid = b[row-1:row+2,col-1:col+2]
    result = 0.1 * (subgrid.sum()+ b[row,col])
    return result

size = 10
b = np.zeros((size,size))
b2 = np.zeros((size,size))

for i in range(size):
    b[i,0] = 10

for timestep in range(5):
    for r in range(1, size-1):
        for c in range (1, size-1 ):
            b2[r,c] =calcheat(r,c)
    for i in range(size):
        b2[i,0] = 10
    b = b2.copy()

plt.title('Heat Diffusion Simulation')
plt.imshow(b2, cmap=plt.cm.hot)
plt.show()
```

import statements

function definitions

set up variables

input data

process data

output data

Procedural Programming

- So far we've been applying a **procedural** programming approach
- We've been focusing on the steps of the problem, breaking it down into a sequence of instructions
  - We have control structures to help with the flow through the sequence
- When we've found repetition, we've used functions (procedures) to factor out that code
- Although our code has been procedural, we have been using Objects…

  ... and, **in Python, everything's an object**

OBJECT ORIENTATION

Fundamentals of Programming

Lecture 6

Object-Orientation

- In object-oriented programming, we bundle the behaviour (methods) and data (attributes) together
- Benefits:
  - OO protects data from being used incorrectly
  - Increases code reuse (fewer errors)
  - Makes code easier to read and maintain
  - Objects "know" how to respond to requests
  - Relates to how objects function in the real world

Reference: Module 4 of Object-oriented Program Design, Curtin University, 2017

Classes – Specifying Objects

- Before we can use an object, we need to describe it as a **class** (of objects).
  - Similar to how we define a function once and use it multiple times
- The class specifies the state and behaviour an object can have:
  - State: what the object is
    - attributes or member fields
  - Behaviour: what the object does
    - methods or functions

Encapsulation

- A (an object of a) class makes use of the "information hiding" principle
  - Communication with the rest of the software system is clearly defined
    - methods are the means for communication
  - Its obligations to the software system are clearly defined
    - what services the class offers (via data and methods)
  - Implementation details should be hidden from the user
    - don't need to know how it does things to use it

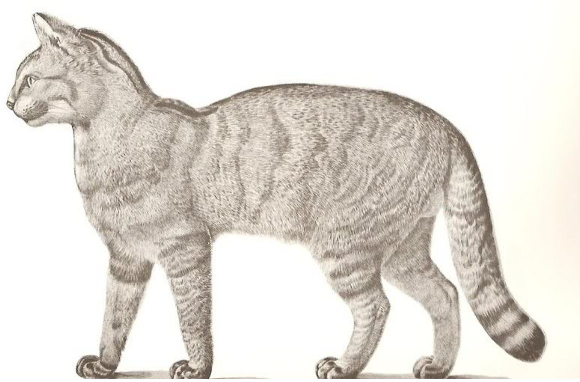
Class Specification

- Must include:
  - Details of the communication with the rest of the software system (method names)
  - The exact data representation required
  - Exactly how the required functionality is to be achieved (method implementation)

Classes and Objects

- An object is an **instance of** a class
- The class definition provides a template for an object
- An object gives details for a particular instance

Generic cat = class "cat"



Specific cat = instance
"Oogie" of class "cat"

http://s460.photobucket.com/user/stefer24/media/scan0024.jpg.html

Class roles

- Every class is designed with a specific **role** in mind.
- The total set of functional requirements for a software system is broken down into a set of tasks
- Collections of tasks are grouped together and mapped to roles
- Roles are mapped to specific classes

Class Responsibility

- Take the requirements for a software application:
  - Identify the classes required
  - Assign specific Responsibilities to each class
  - Determine relationships between classes (see later)
  - Repeat the above steps until the design is correct
  - Each responsibility should be handled by that class and no other
    - Example: If a responsibility for keeping track of a person's name is assigned to a class called PersonClass then:
    - No other class should have this information
    - Other classes which need this information should refer to this class when the information is required

Comparison to non-OO design

- In a top-down procedural approach, we design an algorithm by starting with a main module and using step-wise refinement to determine the processing steps
- Some of these steps get refined into sub modules and the process repeats until the design is refined enough to code

- Under Object Orientation this all changes…

OO design

- Before the algorithm is designed:
  - The classes are identified
  - Each class is assigned role(s) or responsibilities
  - The required sub modules are designed (i.e. Constructors, accessors, etc)
  - Each Class is thoroughly tested via a test harness

- Finally, the main algorithm and any required sub modules is designed (making use of the developed classes in the process)

Nouns and Verbs

- Like algorithm design, the determination of classes is still a bit of an art form
- One simple technique is the nouns and verb approach:
  - Nouns are mapped to classes
  - Verbs are mapped to sub modules within classes
  - The definition of noun and verb gets stretched to cover collections of words
  - Result is that:
    - Sub module names should always describe an action (i.e. getName)
    - Class names should always describe a thing (e.g. PersonClass)
- It is important to note that the set of classes proposed will change over the design phase

Object Communication

- Sometimes referred to as **message passing**:
  - When an object of one class calls an object of another class it is passing a message (i.e. A request to the object to perform some task)
- The [public] methods must provide the functionality required for the class to fulfill its role.
- There are five categories of methods in a class:
  - The Constructors
  - The Accessor Methods (aka Interrogative Methods)
  - The Mutator Methods (aka Informative Methods)
  - Doing Methods (aka Imperative Methods)
  - [Private] methods

Classes in Python

- Order your code consistently for FOP
- Declare the components of each class in the following order:
  - Declarations for class constants
  - Declarations for **class variables**/fields
    - variables which are global to the class
  - Declarations of **instance variables** (local to each instance)
  - Declarations for the Constructors (__init__)
  - *Accessor methods* *Python instance and class variables are*
  - *Mutator methods* *public, so basic set/gets are not req'd*
  - Doing methods ("public")
  - Internal methods ("private")
- Note that everything in Python is "public" (unlike Java, C++) so we can only **treat** methods and data as private

Example: song

```
class Song():

    def __init__(self, lyrics):
        self.lyrics = lyrics

    def sing_me_a_song(self):
        for line in self.lyrics:
            print(line)

lumberjack = Song(["I'm a lumberjack and I'm OK",
                   "I sleep all night",
                   "And I work all day"])

spam = Song(["SPAM, SPAM, SPAM, SPAM",
             "spam, spam, spam, spam"])

lumberjack.sing_me_a_song()
spam.sing_me_a_song()
```

Song: lumberjack
lyrics: ["I'm a lumberjack and I'm OK", "I sleep all night", "And I work all day"]

https://learnpythonthehardway.org/book/ex40.html

Example: dog tricks

```
class Dog():

    def __init__(self, name):
        self.name = name
        self.tricks = []

    def add_trick(self, trick):
        self.tricks.append(trick)

d1 = Dog('Brutus')
d2 = Dog('Dude')

d1.add_trick('roll over')
d1.add_trick('sit')
d2.add_trick('stay')
d2.add_trick('roll over')
print("Dog's name: ", d1.name, "\nDog's tricks: ", d1.tricks)
print("Dog's name: ", d2.name, "\nDog's tricks: ", d2.tricks)
```

Dog: d1
name: Brutus tricks: ["roll over", "sit"]

Example: bank account

BankAccount: bank

interest_rate: 0.3
name: 'Everyday'
number: '007'
balance: 2000

```
class BankAccount ():
    interest_rate = 0.03
    def __init__(self, name, number, balance):
        self.name = name
        self.number = number
        self.balance = balance


bank = BankAccount('Everyday', '007', 2000)

print("Name: ", bank.name, "\tNumber: ",
      bank.number, "\tBalance: ", bank.balance)
```

Output:

```
Name:  Everyday     Number:  007  Balance:  2000
```

Example: bank account

```
class BankAccount ():
    interest_rate = 0.03
    def __init__(self, name, number, balance):
        self.name = name
        self.number = number
        self.balance = balance
```

BankAccount: bank

interest_rate: 0.03
name: 'Everyday'
number: '007'
balance: 2000

Class variable

Instance variables

Example: bank account (v2)

```
class BankAccount ():
    interest_rate = 0.03
    def __init__(self, name, number, balance):
        self.name = name
        self.number = number
        self.balance = balance


accounts = []
bank = BankAccount('Everyday', '007', 2000)
accounts.append(bank)
bank = BankAccount('Cheque A/C', '008', 3000)
accounts.append(bank)
bank = BankAccount('Term Deposit', '009', 20000)
accounts.append(bank)

total = 0
for i in range(len(accounts)):
    print("Name: ", accounts[i].name, "\tNumber: ", accounts[i].number,
          "\tBalance: ", accounts[i].balance)
    total = total + accounts[i].balance
print("\t\t\t\t\tTotal: ", total)
```

Example: bank account (v2)

```
class BankAccount ():
    inte
    def
```

```
OUTPUT:
Name:  Everyday         Number:  007  Balance:  2000
Name:  Cheque A/C       Number:  008  Balance:  3000
Name:  Term Deposit     Number:  009  Balance:  20000
                                      Total:  25000
```

```
accounts = []
bank = BankAccount('Everyday', '007', 2000)
accounts.append(bank)
bank = BankAccount('Cheque A/C', '008', 3000)
accounts.append(bank)
bank = BankAccount('Term Deposit', '009', 20000)
accounts.append(bank)

total = 0
for i in range(len(accounts)):
    print("Name: ", accounts[i].name, "\tNumber: ", accounts[i].number,
          "\tBalance: ", accounts[i].balance)
    total = total + accounts[i].balance
print("\t\t\t\t\tTotal: ", total)
```

Example: bank account (v3)

```
class BankAccount ():
    interest_rate = 0.03
    def __init__(self, name, number, balance):
        self.name = name
        self.number = number
        self.balance = balance

    def withdraw(self, amount):
        self.balance = self.balance - amount

    def  deposit(self, amount):
        self.balance = self.balance + amount

    def add_interest(self):
        self.balance += self.balance * self.interest_rate

def balances():
    total = 0
    for i in range(len(accounts)):
        print("Name:", accounts[i].name, "\tNumber: ", accounts[i].number,
              "\tBalance: ", accounts[i].balance)
        total = total + accounts[i].balance
    print("\t\t\t\t\tTotal: ", total)
```

Example: bank account (v3)

```
accounts = []
bank = BankAccount('Everyday', '007', 2000)
accounts.append(bank)
bank = BankAccount('Cheque A/C', '008', 3000)
accounts.append(bank)
bank = BankAccount('Term Deposit', '009', 20000)
accounts.append(bank)
balances()

print("\nDoing some transactions...\n")
accounts[0].deposit(100)
accounts[1].withdraw(500)
accounts[2].add_interest()
balances()
```

Output:

```
Name:  Everyday          Number:  007     Balance:  2000
Name:  Cheque A/C        Number:  008     Balance:  3000
Name:  Term Deposit      Number:  009     Balance:  20000
                                          Total:  25000

Doing some transactions...

Name:  Everyday          Number:  007     Balance:  2100
Name:  Cheque A/C        Number:  008     Balance:  2500
Name:  Term Deposit      Number:  009     Balance:  20600.0
                                          Total:  25200.0
```

Self

- Why do I need self when I make __init__ or other functions for classes?
- If you don't have self, then code like cheese = 'Gorgonzola' is ambiguous.
- That code isn't clear about whether you mean the *instance's* cheese attribute/variable, *or* a local variable named cheese.
- With self.cheese = 'Gorgonzola' it's very clear you mean the instance attribute self.cheese.
- You can use any variable name, but self is the convention.

https://learnpythonthehardway.org/book/ex40.html

OO Design…Where to begin?

- Find your objects
- If we wanted to keep track of our household animals: cats, dogs and birds
- We could make classes for cats, dogs and birds
- For each animal, we might track:
  - name
  - date of birth
  - colour
  - breed

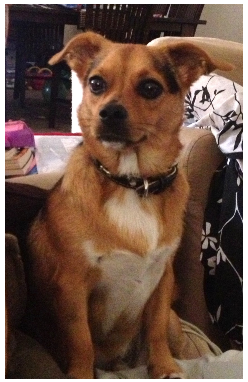
Test our objects out…

CAT

Name: Oogie
DOB: 1/1/2006
Colour: Grey
Breed: Fluffy

DOG

Name: Dude
DOB: 1/1/2011
Colour: Brown
Breed: Jack Russell



BIRD

Name: Big Bird
DOB: 10/11/1969
Colour: Yellow
Breed: Canary

animals.py - Dog Class (v2)

```
class Dog():

    myclass = "Dog"

    def __init__(self, name, dob, colour, breed):
        self.name = name
        self.dob = dob
        self.colour = colour
        self.breed = breed

    def printit(self):
        print('Name: ', self.name)
        print('DOB: ', self.dob)
        print('Colour: ', self.colour)
        print('Breed: ', self.breed)
        print('Class: ', self.myclass)
```

animals.py - Cat Class

```
class Cat():

    myclass = "Cat"

    def __init__(self, name, dob, colour, breed):
        self.name = name
        self.dob = dob
        self.colour = colour
        self.breed = breed

    def printit(self):
        print('Name: ', self.name)
        print('DOB: ', self.dob)
        print('Colour: ', self.colour)
        print('Breed: ', self.breed)
        print('Class: ', self.myclass)
```

animals.py - Bird Class

```
class Bird():

    myclass = "Bird"

    def __init__(self, name, dob, colour, breed):
        self.name = name
        self.dob = dob
        self.colour = colour
        self.breed = breed

    def printit(self):
        print('Name: ', self.name)
        print('DOB: ', self.dob)
        print('Colour: ', self.colour)
        print('Breed: ', self.breed)
        print('Class: ', self.myclass)
```

Pets.py

```
from animals import Dog
from animals import Cat
from animals import Bird

dude = Dog('Dude', '1/1/2011', 'Brown', 'Jack Russell')
oogs = Cat('Oogie', '1/1/2006', 'Grey', 'Fluffy')
bbird = Bird('Big Bird', '10/11/1969', 'Yellow', 'Canary')

dude.printit()
oogs.printit()
bbird.printit()
```

Or…

from animals import *

Pets.py

```
from animals import Dog
from animals import Cat
from animals import Bird

dude = Dog('Dude', '1/1/2011', 'Brown', 'Jack Russell')
oogs = Cat('Oogie', '1/1/2006', 'Grey', 'Fluffy')
bbird = Bird('Big Bird', '10/11/1969', 'Yellow', 'Canary')

dude.printit()
oogs.printit()
bbird.printit()
```

```
Name:  Dude
DOB:  1/1/2011
Colour:  Brown
Breed:  Jack Russell
Class:  Dog

Name:  Oogie
DOB:  1/1/2006
Colour:  Grey
Breed:  Fluffy
Class:  Cat

Name:  Big Bird
DOB:  10/11/1969
Colour:  Yellow
Breed:  Canary
Class:  Bird
```

If you try to print dude directly:

animals.Dog object at 0x10108d978

Summary

- Understand the main concepts in object-oriented programming and their value
- Read and explain object-oriented code
- Apply and create simple object-oriented Python code

Practical Sessions

- We'll be coding objects and using them

Assessments

- We've sat Prac Test 2 in the week preceding this lecture

- We have an in-class written test next week during the lecture
- The assignment will be released after the test

Mid-semester Test

- 60 minutes + 5 minutes reading time
- Covers weeks 1-6 of lectures, weeks 1-5 of practicals
  - i.e. just simple questions from OO
  - Questions based on lectures, practicals, practical tests, revision questions and review questions
- Examples are available on Blackboard
- Hopefully will have a ComSSA revision TBA

Next week…

- Test during lecture – 22/9/19
  - If you have a Curtin Access Plan, and haven't informed me, do so ASAP.

- Lecture 7:
  - Relationships in Object-orientation
  - Exception handling