Object Oriented Programming

OOP
ANOTHER PROGRAMMING PARADIGM

Encapsulation as Information hiding

- Data belonging to one **object** is hidden from other objects.
- Know what an object can do, not how it does it.
- Information hiding increases the level of *independence*.
- Independence of modules is important for large systems and maintenance.

Information Hiding

Code

```
class Member:

    def __init__(self, firstname, surname, postcode, uid):
        self.firstname = firstname
        self.surname = surname
        self.postcode = postcode
        self.uid = uid
        self.borrowed = [] ## a list of items uid

    def __repr__(self):
        return ('<Member: uid = ' + str(self.uid) + ', ' +
                self.surname + ', ' +
                str(self.borrowed) + '>')
```

- All Attributes are public

Private vs Protected vs Public Attributes

- Using the private modifier is the main way that an object encapsulates itself and hide data from the outside world.
- There is no completely private attributes in Python
- Private attributes can be simulated using two underscores for prefix
  - __attributename
- Protected attributes have a single underscore for prefix
  - _attributename
- No underscore prefix for public attributes

Private vs Protected vs Public Attributes

- If using private/protected attribute, you should provide adequate
  - Accessors method (read/get value)
  - Mutators method (change/set value)
- Which attribute must be public/protected/private is a design decision
- Which mutator/accessor to provide is also a design decision

Information Hiding

Code

```
class Member:

    def __init__(self, firstname, surname, postcode, uid):
        self._firstname = firstname
        self._surname = surname
        self._postcode = postcode
        self._uid = uid
        self._borrowed = [] ## a list of items uid

    def __repr__(self):
        return ('<Member: uid = ' + str(self._uid) + ', ' +
                self._surname + ', ' +
                str(self._borrowed) + '>')
```

How to access Information?

- All Attributes are protected

Defining Class Behaviour

- **Methods**, enable the behaviour of class **instances**.

```
def  methodName (self, <parameters>):
     <body>
```

- **Accessors** are methods to read/access the values of some attributes

Code Accessor

```
def getSurname(self):
   return self._surname

def getUID(self):
   return self._uid
```

Defining Class Behaviour

- **Mutators** are methods to set/modify the values of some attributes.

- We may want some attributes not to be modified by an external source (e.g. another class), so no mutator should be provided.
  - For example UID should never be changed (design decision)

Code Mutator

```
def setSurname(self, name):
    self._surname = name
```

Defining Class Behaviour

- How about modifying _borrowed attribute?

Code Mutator

```
def setBorrowed(self, borrowed):
    self._ borrowed = borrowed
```

  - What are the drawbacks of this design?

- Use another design

Code

```
def addBorrowed(self, borrowed_item):
   if borrowed_item in self._borrowed:
      raise Exception('Already in borrowed')
   else:
      self._ borrowed.append(borrowed_item)

Def removeBorrowed(self, borrowed_item):
   ...
```

Method calls

- Internal method calls (inside class definition)
  - `self.method_name(parameters)`
- External method calls
  - `object.method_name(parameters)`

Code External Method Call

```
# Constructor Call of class QueueOOP
lilian = Member('blot','lilian','yox xgh', '01')

print 'Surname is:', lilian.getSurname()
```

External method call (class Member)

Class Design

12

THE LIBRARY CLASS

Code quality

Two important concepts for quality of code:

1. Coupling
2. Cohesion

Coupling

- Coupling refers to links between separate units of a program.
- If two classes depend closely on many details of each other, we say they are tightly coupled.
- We **aim** for **loose coupling**.

Loose coupling

Loose coupling makes it possible to:

- understand one class without reading others;
- change one class without affecting others.
- Thus: improves maintainability.

Cohesion

- Cohesion refers to the number and diversity of tasks that a single unit is responsible for.
- If each unit is responsible for **one single logical task**, we say it has high cohesion.
- Cohesion applies to classes and methods.
- We **aim** for **high cohesion**.

High cohesion

High cohesion makes it easier to:

- understand what a class or method does;
- use descriptive names;
- reuse classes or methods.

Cohesion of methods

A method should be responsible for **one and only one** well defined task.

Cohesion of classes

Classes should represent **one single**, well defined entity.

Last Week Practical Design

- Two dictionaries, one handling members and one handling items
- Both dictionaries strongly coupled, however they are not part of the same data structure

Code

```
def add_member(firstname, surname, postcode, uid, listMembers):
    if uid in listMembers: ## uid already existing so must not add item
        return None
    else:
        member = Member(firstname, surname, postcode, uid)
        listMembers[uid] = member
        return member

members = {}    # keys are members UID, values are Member objects

print 'Add member :', add_member('lilian','blot','xxx', '007', members)
```

Design a Library Class

- Encapsulate the Items and Members into a third class Library
  - What are the attributes?
  - Should the attributes be public, protected, or private?
  - What are the Accessors?
  - What are the Mutators?

- To help in designing the class, think about a real world library:
  - what actions that can be done in a library?

Summary

By now, you should be able to:

- create small scripts/program using selection and repetition
- Decomposed complex problems into smaller sub-problems
- Use Modularisation
  - Separation of concerns
  - Semantically coherent
  - Function
  - Classes/OOP
- Write Documentation and use correct code style (conventions)