Objects and classes

CMPT 145

Objectives

After this topic, students are expected to be able to

1. Explain the differences between Procedural and Object Oriented Programming (OOP).
2. Explain the difference between a object and a record.
3. Explain the difference between a class and an object.
4. Explain what attributes and methods are in terms of object oriented programming.
5. Define simple classes, including data and methods, in Python.

Procedural programming

- In CMPT 141 and CMPT 145 (so far), our programs consisted of
  - data: variables, list, dictionaries.
  - computation: loops, conditionals, functions
- Procedural programming uses functions (procedures) to encapsulate (contain) algorithms.

Procedural programming and ADTs

- ADTs encapsulate data, and organize programs
- ADTs are implemented using dictionaries and functions; every operation required a reference to a record.
- The dictionary used to store the data and the operations are conceptually related...
- BUT! the data and the operations are not a single entity.
    - The data was stored in a dictionary
    - The operations were in the global scope

Object oriented programming can make data and operations a single entity.

Object Oriented programming

- Object oriented (OO) programming is a different style.
- Object oriented programming is the paradigm most often used today for large projects.
- OO is focused on creating *objects* who communicate to each other.
- An object has data, like a record, but also has operations attached. The data and operations are part of the same entity!
- Data hidden inside an object literally cannot be accessed outside that object.

Not everything...

- There are good reasons to use OOP
- There are good reasons not to use OOP
- OOP is not the answer to every problem
- E.g., When dealing with hardware, device drivers, operating systems software, OOP is rarely used.

Object Oriented concepts: Object

- An *object* consists of
  - **data** stored in the object (similar to a record defined by a record type)
  - **operations** on the data (in the form of functions)
- An object is like a record that also contains functions.
- The data in an object are stored using variables local to the object. These variables are called attributes or fields or instance variables.
- The operations in an object are called member functions or methods or messages.
- An object is self-contained. A well-designed object contains data and has methods to operate on that data.
- By default, the object's attributes cannot be touched except by calling the object's methods.

Object Oriented concepts: Class

- A class is like a blue-print for objects.
- An object is created from its class.
    - You can create many objects from the same class.
    - The class name is also the object's type.
- A class defines the attributes and the functions that the object will have.
    - The class doesn't usually do work; objects do work.
    - The class doesn't store attributes; the objects do.

Classes you already know about

- String (immutable)

```
alist = 'Jan Feb Mar Apr May'.split()
```

- List

```
astring = alist.append('Jun')
```

- Dictionary

```
addict = {'one': 1}
print(addict.keys())
```

A simple class

```
class Hero(object):
    def __init__(self, nn, pp):
        self.name = nn
        self.power = pp
```

Class definitions:

- A class definition starts with the keyword `class`
- Everything in the class is indented relative to `class`
  - (rather like internal functions)
- The class name is conventionally capitalized
- The class name is followed by `(object):`
  - Looks like a function-parameter list, but it's not
  - More about this later!

Class definitions: `__init__()`

- A class definition should always have an `__init__()` method
- When an object is created, Python calls `__init__()` implicitly
- The first parameter for `__init__()` is always `self`
- `__init__()` initializes the object `self` by creating attributes using assignment statements.
- `__init__()` has no return statement

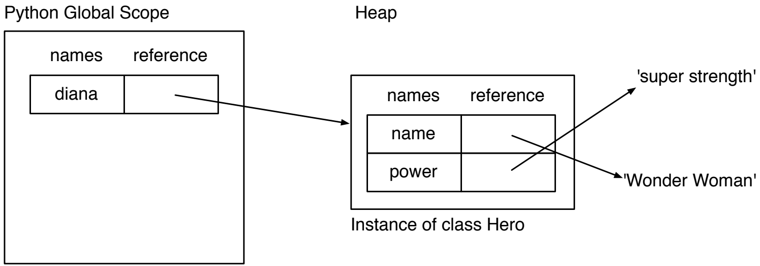
A hero is born

```python
class Hero(object):
    def __init__(self, nn, pp):
        self.name = nn
        self.power = pp

if __name__ == '__main__':
    diana = Hero('Wonder Woman', 'super strength')
```

There are two attributes, `self.name` and `self.power` are created by the assignment statements.

A hero is born



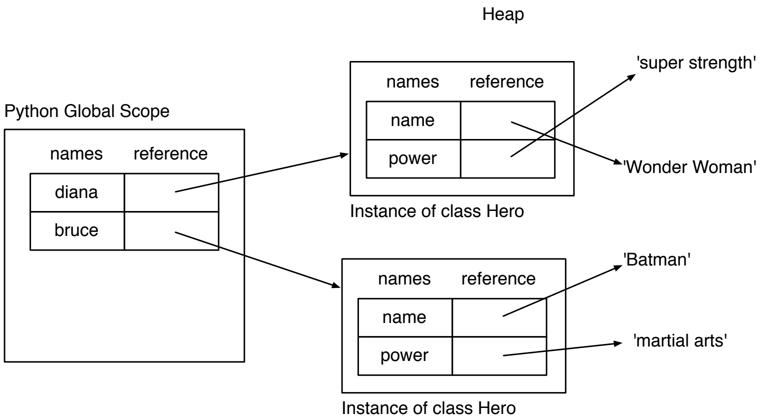
Towards a league

```python
class Hero(object):
    def __init__(self, nn, pp):
        self.name = nn
        self.power = pp

if __name__ == '__main__':
    diana = Hero('Wonder Woman', 'super strength')
    bruce = Hero('Batman', 'martial arts')
```

There are now two objects, each has two attributes. The attributes have the same names, but different values.

Towards a league



Object attributes

- The `__init__()` method should initialize attributes
- Attributes are variables local to the object `self`
- Attributes are accessed using the dot-notation, e.g., `self.name`
- Many objects can be created from the same class:
  - All the objects have the same attribute names
  - The attribute values can be different

Object methods

The class defines what objects do by defining methods:

```
class Hero(object):
    def __init__(self, nn, pp):
        self.name = nn
        self.power = pp

    def say_hello(self):
        print('Hello, evil-doers! My name is', self.name)
        print('My super power is', self.power)
```

- The function `say_hello()` is a method for the class Hero.
- Every method's first parameter is always `self`.
- More parameters are allowed, after `self`. All the parameters are normal function parameters.

Calling Object methods

```python
class Hero(object):
    def __init__(self, nn, pp):
        self.name = nn
        self.power = pp

    def say_hello(self):
        print('Hello, evil-doers! My name is', self.name)
        print('My super power is', self.power)

if __name__ == '__main__':
    diana = Hero('Wonder Woman', 'super strength')
    bruce = Hero('Batman', 'martial arts')
    bruce.say_hello()
    diana.say_hello()
```

Calling a method uses the dot-notation: `var.method(args)`
`var` is a variable or expression that refers to an object.

Calling Object Methods

- The class defines what objects do by defining methods
- In a definition, a method's first parameter is always `self`
- Calling a method uses the dot-notation.
- Calling a method never gives an argument for `self`
    - We write `bruce.say_hello()`
    - Python calls the `say_hello()` method, giving `bruce` as the value of the first parameter, `self`.

An old friend: The Statistics ADT

We used a Python dictionary to implement it:

```
# CMPT 145: Abstract Data Types
# Implements the Statistics ADT

def create():
    """
Purpose:
    Create a Statistics record.
Return:
    A Statistics record.
    """
    b = {}
    b['count'] = 0       # how many data values have been seen
    b['avg'] = 0         # the running average so far
    b['sumsqdiff'] = 0   # the sum of the square differences
    return b
```

An old friend: The Statistics ADT

We used normal functions as operations:

```
def add(stat, value):
    """
    Purpose:
        Use the given value in the calculation....
        ...
    """
    stat['count'] += 1
    k = stat['count']               # convenience
    diff = value - stat['avg']  # convenience
    stat['avg'] += diff/k
    stat['sumsqdiff'] += ((k-1)/k)*(diff**2)
```

Notice that our convention was to put the data structure, `stat`, as the first argument

The Make-over: The Statistics ADT

We can use a Python class to implement it:

```python
# CMPT 145: Objects
# Implements the Statistics ADT

class Statistics(object):
    def __init__(self):
        """
    Purpose:
        Initialize a Statistics object instance.
        """
        self.count = 0      # number data values seen
        self.avg = 0        # the running average
        self.sumsqdiff = 0  # sum of square differences
```

The Make-over: The Statistics ADT

We can define methods as its operations:

```
    def add(self, value):
        """
        Purpose:
            Use the given value in the calculations....
            ...
        """
        self.count += 1
        k = self.count              # convenience
        diff = value - self.avg     # convenience
        self.avg += diff / k
        self.sumsqdiff += ((k - 1) / k) * (diff ** 2)
```

Notice that `self` is the first parameter. This should feel familiar!

Rationale

- Python classes make our existing data structures a little nicer
- We can use records as data structures in any language
- Object oriented tools add value by decreasing the amount of work a programmer has to do
- Object oriented programming adds cost by increasing the amount of knowledge a programmer needs to learn.
- We'll learn just the basics. You can study OOP a lot deeper!

Classes provide encapsulation

- A class contains data and methods
- Everything an ADT needs to do is contained (encapsulated) in the class definition
- The ADT concept still applies: We can use a class as an ADT in the same way that we used a dictionary as an ADT.

Classes provide data hiding

- Our ADTs were designed to hide data behind operations.
    - E.g., the Statistics ADT.
- Classes provide extra safety for data by restricting access to attributes.
- Python does this by a convention:
    - `self.attribute1`: public. Anyone can access `attribute1`
    - `self._attribute2`: protected. Anyone can access `_attribute2` but doing so is considered ill-advised.
    - `self.__attribute3`: private. Leave `__attribute3` alone.

Access to attributes

```
class Hero(object):
    def __init__(self, nn, pp, sid):
        self.name = nn
        self.power = pp
        self.__secret = sid

if __name__ == '__main__':
    bruce = Hero('Batman', 'martial arts', 'Bruce Wayne')
    print(bruce.name)
    print(bruce.__secret)
```

There are two public attributes, `self.name` and `self.power`
There is one private attribute, `self.__secret`.

Public attributes

- All languages allow access to public attributes.
- Public attributes can be accessed in any script.
- Class designers decide to make attributes public because:
  - Access does not put data at risk.
  - Access simplifies coding for scripts using the class.

Public attributes example

```python
# CMPT 145: Objects and Classes
# Defines the tree node class

class TreeNode(object):

    def __init__(self, data, left=None, right=None):
        """
        Create a new treenode for the given data.
        """
        self.data = data
        self.left = left
        self.right = right

if __name__ == '__main__':
    anode = TreeNode(5)
    bnode = TreeNode(2)
    cnode = TreeNode(8)

    anode.left = bnode
    anode.right = cnode
```

Protected attributes

- Python leaves protected attributes public.
- Protected attributes are accessible by any script.
  - But the programmer doesn't really think you should be using them.
  - "Don't touch, but go ahead if you think you know what you're doing."
- In other languages (e.g., Java, C++), the term protected carries a bit more weight. Access to protected attributes is limited to modules in the same library.

Private attributes

- All languages prevent access to private attributes.
- In Python, trying to access a private attribute naively raises a run-time error.
- If you work hard enough, you can find a way to access private attributes in Python.
- Private attributes are used when the programmer knows you'll only mess things up.

Private attributes example: The Statistics ADT

```
# CMPT 145: Objects
# Implements the Statistics ADT

class Statistics(object):
    def __init__(self):
        """
    Purpose:
        Initialize a Statistics object instance.
        """
        self.__count = 0       # number data values seen
        self.__avg = 0         # the running average
        self.__sumsqdiff = 0   # sum of square differences
```

Private attributes example: The Statistics ADT

```
    def add(self, value):
        """
        Purpose:
            Use the given value in the calculations....
            ...
        """
        self.__count += 1
        k = self.__count              # convenience
        diff = value - self.__avg     # convenience
        self.__avg += diff / k
        self.__sumsqdiff += ((k - 1) / k) * (diff ** 2)
```

Notice that `self` is the first argument

An inconvenient implementation

```
# CMPT 145: Objects and Classes
# Defines the tree node class

class TreeNode(object):

    def __init__(self, data, left=None, right=None):
        """
        Create a new treenode for the given data.
        """
        self.__data = data          # private!
        self.__left = left          # private!
        self.__right = right        # private!

if __name__ == '__main__':
    anode = TreeNode(5)
    bnode = TreeNode(2)
    cnode = TreeNode(8)

    anode.__left = bnode        # causes error
```

Private attributes: getters and setters

- Making attributes protected or private allows programmers to control access
- Access can be granted by getters and setters.

```python
class TreeNode(object):
    def __init__(self, data, left=None, right=None):
        ... # as above

    def get_data(self):
        return self.__data
    def set_data(self, val):
        self.__data = val

if __name__ == '__main__':
    anode = TreeNode(5)

    print(anode.get_data())
    anode.set_data(500)
    print(anode.get_data())
```

Using getters and setters

```python
def member(tnode, value):
    '''
    Check if value is stored in the binary tree.
    '''
    if tnode is None:
        return False
    else:
        cval = tnode.get_data()
        if cval == value:
            # found the value
            return True
        else:
            return member(tnode.get_left(), value) \
                or member(tnode.get_right(), value)
```

But getters and setters are ugly...

```python
def member(tnode, value):
    '''
    Check if value is stored in the binary tree.
    '''
    if tnode is None:
        return False
    else:
        cval = tnode.data
        if cval == value:
            # found the value
            return True
        else:
            return member(tnode.left, value) \
                or member(tnode.right, value)
```

Access advice

- For ADTs, when data should be hidden, use private
- For simple data structures, allow public if there's no chance that the encapsulated data can be messed up.
- Use private for everything else.
- Don't be optimistic. Better to protect your data than to open your code up to errors.