User-Defined Types

George Mason University
Today’s topics

• User-defined types in python
Classes

• Classes are a user-defined type
• We create the blueprint for the class
  – this is python code
• We can then use the class
Class example: maps

• Let’s *pretend* the map type did not exist in python
  – let’s declare it ourselves!
  – it is not going to be identical to real python dictionaries, but it will start a discussion
class MyMap:
    def __init__(self):
        self.keys = []
        self.values = []

    def keys(self):
        return self.keys

    def values(self):
        return self.values

    def getValue(self, key):
        if key in self.keys():
            index = keys.index(key)
            return values[index]
        else:
            return "No such key!"
Class setup

class MyMap:
    def __init__(self):
        self.keys = []
        self.values = []

• **class** is a keyword that defines a type (MyMap in this example)

• **__init__** is a method called the **constructor** that defines the attributes
  – attributes are data that is stored

• **self** is a keyword and is a reference to the memory address of the **MyMap** variable
  – *must be first argument in every method*
Object creation

class MyMap:
    def __init__(self):
        self.keys = []
        self.values = []

• Later in the code, we can create an object of type **MyMap** by calling its constructor:

    `map = MyMap()`

    – this is a funny, ugly call to the **__init__** above
    – `self` is implicit as an argument
    – what does memory look like now?
Calling methods

```python
def keys(self):
    return self.keys

def values(self):
    return self.values

def getValue(self, key):
    if key in self.keys():
        index = keys.index(key)
        return values[index]
    else:
        return "No such key!"

map = MyMap()
print map.keys()
print map.values()
print map.getValue("dog")
```

- Later in the code, we can call the methods we wrote (top left) on the object we created (top right)
Writing methods

class MyMap:
    def __init__(self):
        self.keys = []
        self.values = []

def keys(self):
    return self.keys

def values(self):
    return self.values

def getValue(self,key):
    if key in self.keys():
        index = keys.index(key)
        return values[index]
    else:
        return "No such key!"

• MyMap is pretty useless right now; how do we add a key-value pair into the map?
  – Let’s write this method
Printing objects

• What happens when I do `print map`:
  – It’s ugly; prints out memory address

```python
def __str__(self):
    ctr = 0
    result = ""
    while ctr < len(self.keys()):
        result = result + self.keys()[ctr]
        result = result + " ---> "
        result = result + self.values()[ctr]
    ctr = ctr + 1
    return result
```

• `__str__` is a special method that is called with `print`, to pretty-print objects
Exercise

• Define a **Person** and **Address** classes
  – constructors (with arguments besides **self**)
  – **Person** stores name, age, and a list of addresses
  – **Addresses** have number, street, and zipcode
  – **__str__**
  – method to add an **Address** to a **Person** object
Exercise continued

• Create three addresses: home, florida, and mansion

• Create two people, bill and sally
  – They are brother and sister; add home to both of them
  – bill lives in florida, and sally is rich

• Show memory

• Print out bill and sally
Exercise Recap

• This was an example of **aggregation**, where one object is composed of other objects

• As Martha Stewart would say, “It’s a good thing.”

• We can create very complex objects now!
  – *now* you can go and implement Facebook...
Object Oriented Design

• Good interview buzzwords!
• **Abstraction**: details of classes are hidden inside the classes
  – we see all the methods
  – we don’t see or care how/what attributes are stored
Abstraction

```python
class MyMap:
    def __init__(self):
        self.keys = []
        self.values = []
    def add(self, key, value):
        self.keys.append(key)
        self.values.append(value)

class MyMap:
    def __init__(self):
        self.map = {}
    def add(self, key, value):
        self.map[key] = value
```

- Do you know which one is actually used? Do you care?
Violation of abstraction

```python
class MyMap:
    def __init__(self):
        self.keys = []
        self.values = []
    def add(self, key, value):
        self.keys.append(key)
        self.values.append(value)

class MyMap:
    def __init__(self):
        self.map = {}
    def add(self, key, value):
        self.map[key] = value
```

- Turns out, you can access the data of an object directly with the dot operator.
- This is bad, because we now need to use and stick to one implementation.
- Violates abstraction, because we end up relying on implementation-level details.

```python
animals = MyMap()
animals.add("cats", 1)
animals.add("dogs", 2)
print(animals.keys)
print(animals.values)
```
Encapsulation

```python
class MyMap:
    def __init__(self):
        self.__keys = []
        self.__values = []
    def add(self, key, value):
        self.__keys.append(key)
        self.__values.append(value)

animals = MyMap()
animals.add("cats", 1)
animals.add("dogs", 2)
print animals.keys
print animals.values
```

- **Private attributes** forbid direct access
- **Encapsulates** implementation-level details 😊
Inheritance

• With aggregation, a **Person** is composed of **Addresses**
  – You can see the composition in memory

• **Inheritance** is a central OO concept that says your blueprint (code) can be based off of existing classes
  – One class will **extend** another
    – A **Student** might extend a **Person**, by adding new attributes (major, gpa, etc) and method (signUpForClass, postToPiazza, etc)
Inheritance continued

• A **Person** is composed of a name, age, and **Address**
  – this is aggregation of objects

• A **Student** extends a **Person**:

```python
class Student(Person):
    def __init__(self, name, age, major):
        Person.__init__(self, name, age)
        self.__major = major
        self.__gpa = 0.0

def getGPA(self):
    return self.__gpa
```
Inheritance continued

class Student(Person):
    def __init__(self, name, age, major):
        Person.__init__(self, name, age)
        self.__major = major
        self.__gpa = 0.0

    def getGPA(self):
        return self.__gpa

home = Address(2304, "Main Street", 22202)
sally = Person("Sally", 23)
john = Student("John", 18, "underwater basketweaving")
sally.addAddress(home)
john.addAddress(home)
print "John's GPA: " + str(john.getGPA())
Questions?