Object Oriented Programming

David E. Culler

CS8 – Computational Structures in Data Science

http://inst.eecs.berkeley.edu/~cs88

Lecture 8

March 28, 2016

Computational Concepts Toolbox

- Data type: values, literals, operations,
- Expressions, Call expression
- Variables
- Assignment Statement
- Sequences: tuple, list
- Dictionaries
- Data structures
- Tuple assignment
- Function Definition Statement
- Conditional Statement
- Iteration: list comp, for, while
- Lambda function expr.
- Higher Order Functions
  - Functions as Values
  - Functions with functions as argument
  - Assignment of function values
- Higher order function patterns
  - Map, Filter, Reduce
- Function factories – create and return functions
- Recursion
  - Linear, Tail, Tree
- Abstract Data Types
- Mutation

Today: class

- Language support for object oriented programming
- Defining a class introduces a new type of object
- It has attributes
- It has methods
- These implement its behaviors

Review: Objects

- Objects represent information
- Consist of data and behavior, bundled together to create abstractions
  - Abstract Data Types
- They can have state
  - mutable vs immutable
- Object-oriented programming
  - A methodology for organizing large programs
  - So important it is supported in the language (classes)
- In Python, every value is an object
  - All objects have attributes
  - Manipulation happens through methods
- Functions do one thing (well)
  - Object do a collection of related things

Administrative Issues

- Maps project part II due 3/30
- HW05 is lighter, but due 3/28

- Midterm “breakthrough” opportunity
  - Thurs 9 - 1

Review: Bank account using dict

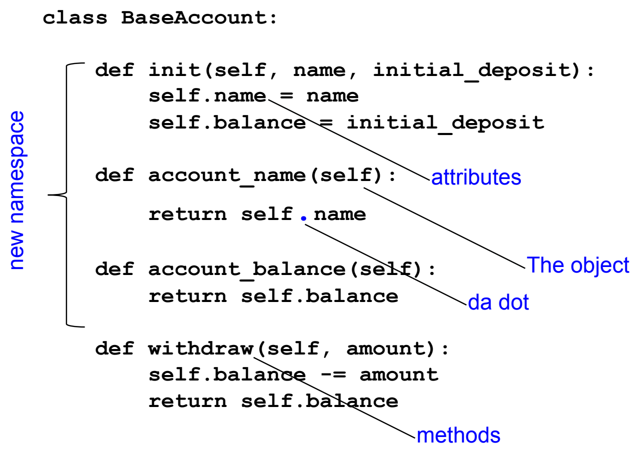
```
account_number_seed = 1000

def account(name, initial_deposit):
    global account_number_seed
    account_number_seed += 1
    return {'Name' : name, 'Number': account_number_seed,
            'Balance' : initial_deposit}

def account_name(acct):
    return acct['Name']

def account_balance(acct):
    return acct['Balance']

def account_number(acct):
    return acct['Number']

def deposit(acct, amount):
    acct['Balance'] += amount
    return acct['Balance']

def withdraw(acct, amount):
    acct['Balance'] -= amount
    return acct['Balance']
```

```
>>> my_acct = account('David Culler', 100)
>>> my_acct
{'Name': 'David Culler', 'Balance': 100,
'Number': 1001}
>>> account_number(my_acct)
1001
>>> your_acct = account("Fred Jones", 475)
>>> account_number(your_acct)
1002
>>>
```

Python class statement

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Example: Account



```
class BaseAccount:

    def init(self, name, initial_deposit):
        self.name = name
        self.balance = initial_deposit

    def account_name(self):

        return self.name

    def account_balance(self):
        return self.balance

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

Creating an object, invoking a method

The Class Constructor

```
my_acct = BaseAccount()
my_acct.init("David Culler", 93)
my_acct.withdraw(42)
```

da dot

Special Initialization Method

```
class BaseAccount:

    def __init__(self, name, initial_deposit):
        self.name = name
        self.balance = initial_deposit

    def account_name(self):

        return self.name

    def account_balance(self):
        return self.balance

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

return None

Attributes and "private"

- Attributes of an object accessible with 'dot' notation

  `obj.attr`

- Alternative to selector/mutator methods
- Most OO languages provide private instance fields
  - Python leaves it to convention

Example

```
class BaseAccount:

    def __init__(self, name, initial_deposit):
        self.name = name
        self.balance = initial_deposit

    def name(self):
        return self.name

    def balance(self):
        return self.balance

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

Example

```
class BaseAccount:

    def __init__(self, name, initial_deposit):
        self.name = name
        self.balance = initial_deposit

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

Example: “private” attributes

```
class BaseAccount:

    def __init__(self, name, initial_deposit):
        self._name = name
        self._balance = initial_deposit

    def name(self):
        return self._name

    def balance(self):
        return self._balance

    def withdraw(self, amount):
        self._balance -= amount
        return self._balance
```

Class attributes

- Pertain to the class as a whole
- Not to individual objects
- Name relative to class, not self

Example: class attribute

```
class BaseAccount:
    account_number_seed = 1000

    def __init__(self, name, initial_deposit):
        self._name = name
        self._balance = initial_deposit
        self._acct_no = BaseAccount.account_number_seed
        BaseAccount.account_number_seed += 1
    def name(self):
        return self._name

    def balance(self):
        return self._balance

    def withdraw(self, amount):
        self._balance -= amount
        return self._balance
```

More class attributes

```
class BaseAccount:
    account_number_seed = 1000
    accounts = []
    def __init__(self, name, initial_deposit):
        self._name = name
        self._balance = initial_deposit
        self._acct_no = BaseAccount.account_number_seed
        BaseAccount.account_number_seed += 1
        BaseAccount.accounts.append(self)

    def name(self):
        ...

    def show_accounts():
        for account in BaseAccount.accounts:
            print(account.name(),
                  account.account_no(),account.balance())
```

Inheritance

- Define a class as a specialization of an existing class
- Inherent its attributes, methods (behaviors)
- Add additional ones
- Redefine (specialize) existing ones
  - Ones in superclass still accessible in its namespace

```
class ClassName ( inherits ):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Example

```
class Account(BaseAccount):
    def deposit(self, amount):
        self._balance += amount
        return self._balance
```

More special methods

```
class Account(BaseAccount):
    def deposit(self, amount):
        self._balance += amount
        return self._balance

    def __repr__(self):
        return '< ' + str(self._acct_no) +
               '[' + str(self._name) + '] >'

    def __str__(self):
        return 'Account: ' + str(self._acct_no) +
               '[' + str(self._name) + ']'

    def show_accounts():
        for account in BaseAccount.accounts:
            print(account)
```

Classes using classes

```
class Bank:
    accounts = []

    def add_account(self, name, account_type,
                    initial_deposit):
        assert (account_type == 'savings') or
               (account_type == checking), "Bad Account type"
        assert initial_deposit > 0, "Bad deposit"
        new_account = Account(name, account_type,
                              initial_deposit)
        Bank.accounts.append(new_account)

    def show_accounts(self):
        for account in Bank.accounts:
            print(account)
```

Key concepts to take forward

- **Class definition**
- **Class namespace**
- **Methods**
- **Instance attributes (fields)**
- **Class attributes**
- **Inheritance**
- **Superclass reference**