Object Oriented Programming

David E. Culler

CS8 – Computational Structures in Data Science

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

Lecture 9

October 22, 2018

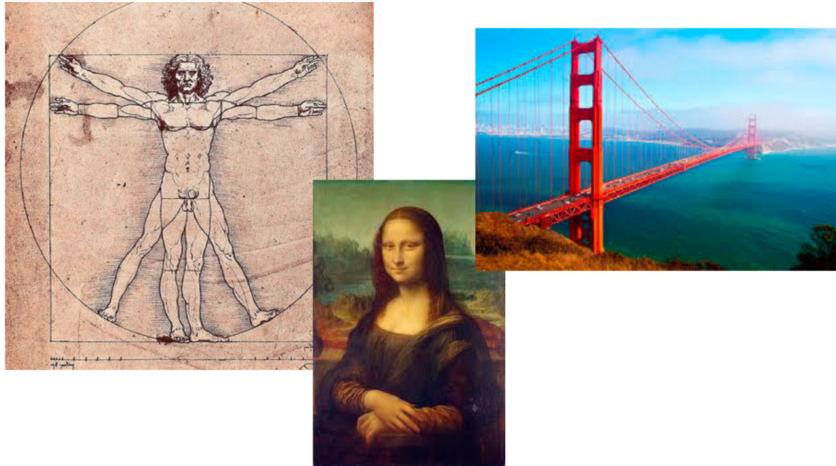
Today's notebooks: http://bit.ly/cs88-fa18-L09

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.

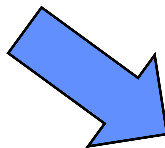
- Higher Order Functions
  - Functions as Values
  - Functions with functions as argument
  - Assignment of function values
- Higher order function patterns
  - Map, Filter, Reduce
- Function factories – create and return functions
- Recursion
  - Linear, Tail, Tree
- Abstract Data Types
- Mutation

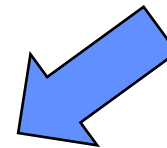
Today: Resolution



Elegance and beauty of Functional Pgm

Power of mutation of state





Structured Object Oriented Programming

Today: class

- Language support for object oriented programming
- Defining a class introduces a new type of object
  - class is the type
- 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 due 10/24**

Review: Bank account using dict

```python
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>:

    def <method-1>(self, ..)
      self.<instance_attr> = …
    .
    .
    .
    def <method-N>
```

https://docs.python.org/3/tutorial/classes.html

Class names should normally use the **CapWords** convention.

https://www.python.org/dev/peps/pep-0008/

Example: Account

```
class Account:
    # Constructor
    def init(self, name, initial_deposit):
        # Initialize the instance attributes
        self.name = name
        self.balance = initial_deposit

    # Selectors
    def account_name(self):
        return self.name

    def account_balance(self):
        return self.balance

    # Operations
    def deposit(self, amount):
        self.balance += amount
        return self.balance
```

The object

Instance attributes

Dot opens the object namespace

Methods

new namespace

Creating an object, invoking a method

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

The Class Constructor

da dot

Special Initialization Method

```
class Account:
    # Constructor
    def __init__(self, name, initial_deposit):
        # Initialize the instance attributes
        self.name = name
        self.balance = initial_deposit
        # Return None

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

    # Display representation
    def __repr__(self):
        return '<Acct: ' + str(self.account_name()) + '>'
```

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, use `_`**

Example

```
class Account:
    # Constructor
    def __init__(self, name, initial_deposit):
        # Initialize the instance attributes
        self._name = name
        self._balance = initial_deposit
        # Return None

    # Selectors
    def account_name(self):
        return self._name

    def account_balance(self):
        return self._balance

    # Operations
    def deposit(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 Account:
    # Class astributes outside and class defs
    _account_number_seed = 1000

    # Constructor
    def __init__(self, name, initial_deposit):
        # Initialize the instance attributes
        self._name = name
        self._acct_no = Account._account_number_seed
        Account._account_number_seed += 1
        self._balance = initial_deposit
        # Return None

    # Selectors
    def account_name(self):
        return self._name
    . . .
    def account_number(self):
        return self._acct_no
    . . .
```

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 CheckingAccount(Account):

    def __init__(self, name, initial_deposit):
        # Use superclass initializer
        Account.__init__(self, name, initial_deposit)
        # Additional initialization
        self._type = "Checking"

    def account_type(self):
        return self._type

    # Display representation
    def __repr__(self):
        return '<' + str(self.account_type()) + 'Account:…'
```

Attribute in subclass, not in superclass

Another Example

```
class SavingsAccount(Account):

    interest_rate = 0.02

    def __init__(self, name, initial_deposit):
        # Use superclass initializer
        Account.__init__(self, name, initial_deposit)
        # Additional initialization
        self._type = "Savings"

    def account_type(self):
        return self._type

    def acrue_interest(self):
        self._balance = self._balance *
                            (1 + SavingsAccount.interest_rate)
```

Methods in subclass, not in superclass

Classes using classes

```
class Bank:
    _accounts = []

    def add_account(self, name, account_type, initial_deposit):
        if account_type == 'Savings':
            new_account = SavingsAccount(name, initial_deposit)
        elif account_type ==  'Checking':
            new_account = CheckingAccount(name, initial_deposit)
        else:
            assert True, "Bad Account type: " + account_type
        assert initial_deposit > 0, "Bad deposit"

        Bank._accounts.append(new_account)
        return new_account

    def accounts(self):
        return self._accounts[:]

    def show_accounts(self):
        for acct in self.accounts():
            print(acct.account_number(), acct.account_type(),
                  acct.account_name(), acct.account_balance())
```

Key concepts to take forward

- Classes embody and allow enforcement of ADT methodology
- Class definition
- Class namespace
- Methods
- Instance attributes (fields)
- Class attributes
- Inheritance
- Superclass reference

Additional examples

- Redesign our KV as a class
- How should “new KV” vs mutation be handled
- Inheritance and “new object” in superclass

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
- Class
  - Object Oriented Programming