Part 2:

Object Oriented Analysis

OOP CHARACTERISTICS

OOP Terminology

- Each **object** created in a program is an **instance** of a **class**.
- Each class presents to the outside world a concise and consistent view of the objects that are instances of this class
  - without going into too much unnecessary detail or giving others access to the inner workings of the objects.
- The class definition typically specifies **instance variables**
  - Also known as **data member**s, that the object contains
  - As well as the **methods**, also known as **member functions**, that the object can execute.

Members and Methods

```
class BankAccount:
    def __init__(self, initial_balance=0):
        self.balance = initial_balance      # Member
        self.limit = -10000                 # Member

    def deposit(self, amount):              # Method
        self.balance += amount

    def withdraw(self, amount):             # Method
        if self.balance - amount < self.limit:
            print "Cannot withdraw!"
            return
        self.balance -= amount

    def overdrawn(self):                    # Method
        return self.balance < 0
```

- **Members**
  - `balance`
  - `limit`
- **Methods:**
  - `deposit`
  - `withdraw`
  - `overdrawn`

Members and Methods

```
class BankAccount:
    def __init__(self, initial_balance=0):
        self.balance = initial_balance      # Member
        self.limit = -10000                 # Member

    def deposit(self, amount):              # Method
        self.balance += amount

    def withdraw(self, amount):             # Method
        if self.balance - amount < self.limit:
            print "Cannot withdraw!"
            return
        self.balance -= amount

    def overdrawn(self):                    # Method
        return self.balance < 0

# Usage Example
if __name__ == "__main__":
    a = BankAccount(15)
    a.deposit(17)
    a.withdraw(12)
    print "a blanace =", a.balance          # Member
    print "a overdrawn?", a.overdrawn()     # Method

    b = BankAccount(100)
    b.deposit(200)
    b.withdraw(740)
    print "b blanace =", b.balance          # Member
    print "b overdrawn?", b.overdrawn()     # Method

    # a and b are married
    # What is their total balance?
    print "Total =", a.balance + b.balance
```

OOP Goals

- **Robustness (איתנות)**
  We want software to be capable of handling unexpected inputs that are not explicitly defined for its application
- **Adaptability (סגילות)**
  Software needs to be able to evolve over time in response to changing conditions in its environment
- **Reusability (שימוש חוזר)**
  The same code should be usable as a component of different systems in various applications

Quoted from: © 2013 Goodrich, Tamassia, Goldwasser

OOP Principles

- **Modularity (מודולריות Lego)**
  Property of a system which has been decomposed into a set of cohesive and loosely coupled modules. source code of the application is divided into small independent modules
- **Abstraction (הפשטה)**
  Essential characteristics of an object that we want to model. In the real world, the object usually has too many characteristics which do not interest us. Simplification.
- **Encapsulation (כימוס)**
  The ability of an object to hide its data and methods from the rest of the world. objects encapsulate data and implementation, the user of an object can view the object as a black box that provides services

Object-Oriented Software Design

- **Responsibilities**
  Divide the work into different actors, each with a different responsibility
- **Independence**
  Define the work for each class to be as independent from other classes as possible.
- **Behaviors**
  Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it.

Unified Modeling Language (UML)

A **class diagram** has three portions.

1. The name of the class
2. The recommended instance variables
3. The recommended methods of the class.

Class:	CreditCard	
Fields:	_customer _bank _account	_balance _limit
Behaviors:	get_customer( ) get_bank( ) get_account( ) make_payment(amount)	get_balance( ) get_limit( ) charge(price)

Class Definitions

- A class serves as the primary means for abstraction in object-oriented programming.
- In Python, every piece of data is represented as an instance of some class.
- A class provides a set of behaviors in the form of member functions (also known as **methods**), with implementations that belong to all its instances.
- A class also serves as a **blueprint** (שרטוט) for its instances, effectively determining the way that state information for each instance is represented in the form of **attributes** (also known as **fields**, **instance variables**, or **data members**).

The self Identifier

- In Python, the **self** identifier plays a key role.
- In any class, there can possibly be many different instances, and each must maintain its own instance variables.
- Therefore, each instance stores its own instance variables to reflect its current state. Syntactically, **self** identifies the instance upon which a method is invoked

Example: CreditCard Class

```
class CreditCard:
  """A consumer credit card."""

  def __init__(self, customer, bank, acnt, limit):
    """Create a new credit card instance.
    The initial balance is zero.
    customer the name of the customer (e.g., 'John Bowman')
    bank     the name of the bank (e.g., 'California Savings')
    acnt     the account identifier (e.g., '5391 0375 9387 5309')
    limit    credit limit (measured in dollars)
    """
    self._customer = customer
    self._bank = bank
    self._account = acnt
    self._limit = limit
    self._balance = 0
```

Example: CreditCard Class (2)

```
def get_customer(self):                          # Accessor
    """Return name of the customer."""
    return self._customer

def get_bank(self):                              # Accessor
    """Return the bank's name."""
    return self._bank

def get_account(self):                           # Accessor
    """Return the card identifying number"""
    return self._account

def get_limit(self):                             # Accessor
    """Return current credit limit."""
    return self._limit
```

Example: CreditCard Class (3)

```
def get_balance(self):                    # Accessor
    "Return current balance."
    return self._balance

def charge(self, price):                  # Mutator
    """Charge given price to the card, assuming sufficient
    credit limit. Return True if charge was processed;
    False if charge was denied.
    """
    if price + self._balance > self._limit:
        return False
    else:
        self._balance += price
        return True

def make_payment(self, amount):           # Mutator
    "Process customer payment that reduces balance."
    self._balance -= amount
```

Class Developer and Class Client

A class must be viewed from two angles:

- **Developer View**
  The developer is the programmer that develops the class, but not necessarily the one who is going to use it !!!
- **Client View**
  The Client is the user which is going to use the class, but usually not the one who developed it !!!

Developer View

- The developer is the programmer that develops the class, but not necessarily the one who is going to use it !
- To be able to do a good job, a developer must know
  - Who are his potential clients (or users)?
  - What do the clients really need?
    [Responsibilities and Behaviors]
  - To what information they should be exposed?
    [ADT, Abstraction, Encapsulation]
  - What information should be hidden from them?
    [Encapsulation]
  - Does the class fits well in the designated systems?
    [Modularity]

Client View

- The client is the programmer that is going to use the class (usually there are many clients).
- To help the class designer, the client must define
  - Who he is and what he needs?
    [Responsibilities and Behaviors]
  - What is the precise information that he needs to know?
    [Abstraction, ADT, Encapsulation]
  - In which contexts is he going to use the class?
    [Modularity]

CreditCard Class: Client View

```
class: CreditCard

A consumer credit card class
c = CreditCard(customer, bank, acnt, limit):
    Create a new CreditCard instance c
    The initial balance is zero
    customer    the name of the customer (e.g., 'John Bowman')
    bank        the name of the bank (e.g., 'California Savings')
    acnt        the account identifier (e.g., '5391 0375 9387 5309')
    limit       credit limit (measured in dollars)

c.get_customer()
    Get name of the customer

c.get_bank()
    Get the banks name

c.get_account()
    Get the card identifying number

c.get_limit()
    Get current credit limit

c.get_balance()
    Get current balance

c.charge(price):
    Charge given price to the card, assuming sufficient credit limit
    Return True if charge was processed; False if charge was denied.

c.make_payment(amount):
    Process customer payment that reduces balance
```

Client Code

```
c1 = CreditCard('John Bowman', 'California Savings', '5391 0375 9387 5309', 2500)
c2 = CreditCard('John Bowman', 'California Federal', '3485 0399 3395 1954', 3500)
c3 = CreditCard('John Bowman', 'California Finance', '5391 0375 9387 5309', 5000)
cards = [c1, c2, c3]

for val in range(1, 17):
    cards[0].charge(val)
    cards[1].charge(2*val)
    cards[2].charge(3*val)

for c in cards:
    print 'Customer =', c.get_customer()
    print 'Bank =', c.get_bank()
    print 'Account =', c.get_account()
    print 'Limit =', c.get_limit()
    print 'Balance =', c.get_balance()
    while c.get_balance() > 100:
        c.make_payment(100)
        print 'New balance =', c.get_balance()
    print '--------------------'
```

Constructors

- A user can create an instance of the CreditCard class using a syntax as:

```
c = CreditCard('John Bowman', 'California Savings', '5391 0375 9387 5309', 2500)
```

- Internally, this results in a call to the specially named **__init__** method that serves as the constructor of the class.
- Its primary responsibility is to establish the state of a newly created credit card object with appropriate instance variables.

Operator Overloading

- Python's built-in classes provide natural semantics for many operators (+, *, -, /, …)
- For example, the syntax **'a + b'** invokes addition for numeric types, yet concatenation for sequence types

```
3 + 4                    3 * 4
3.14 + 5.807             3.14 * 5.807
"abc" + "xyz"            5 * "abc"
[1,2,3] + [3,4,5]        7 * [1,2,3]
```

- When defining a new class, we must consider whether a syntax like a + b should be defined when a or b is an instance of that class.

Operator Overloading with dunder methods

- A **dunder** method is a special Python method of the form **`__method__`** which is reserved for operators such as ‘+’, ‘*’, ‘==‘, and print

```
class Pluver:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def __add__(self, other):
        a = self.a + other.a
        b = self.b * other.b
        c = max(self.c, other.c)
        return Pluver(a, b, c)

    def __str__(self):
        return "<<%d,%d,%d>>" % (self.a, self.b, self.c)
```

Operator Overloading with dunder methods

- Examples:

```
p = Pluver(2,4,6)
q = Pluver(3,5,7)
r = p + q
print p
print q
print "%s + %s = %s" % (p,q,r)
```

```
p + q  == __add__(p, q)
```

```
<<2,4,6>>
<<3,5,7>>
<<5,20,7>>
<<2,4,6>> + <<3,5,7>> = <<5,20,7>>
```

```
# Operator overloading main advantage:
p + q + r + s
# is much better than:
add(add(add(p, q), r), s)
```

Iterators

- Iteration is an important concept in the design of data structures.
- An **iterator** for a collection provides one key behavior:
  - It supports a special method named **__next__** that returns the next element of the collection, if any, or raises a **StopIteration** exception to indicate that there are no further elements.

```
>>> L = range(1,29,3)
>>> it = iter(L)
>>> it.next()
1
>>> it.next()
4
>>> it.next()
7
>>> it.next()
10
```

```
>>> s = "Hello, World"
>>> it = iter(s)
>>> it.next()
'H'
>>> it.next()
'e'
>>> it.next()
'l'
>>> it.next()
'l'
```

Automatic Iterators

- Any class that defines a next() and __iter__ methods is called an iterator
- **Example**

```
class Counter:
    def __init__(self, start, stop):
        self.current = start
        self.stop = stop

    def __iter__(self):
        return self

    def next(self):
        if self.current > self.stop:
            raise StopIteration
        else:
            current = self.current
            self.current += 1
            return current
```

Class Iterator

- The following three tests do the same thing!

```
def test1():
    c = Counter(3,8)
    print c.next()
    print c.next()
    print c.next()
    print c.next()
    print c.next()
    print c.next()

def test2():
    it = Counter(3, 8)
    for i in it:
        print i

def test3():
    for i in Counter(3, 8):
        print i
```

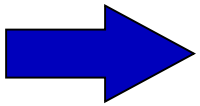
```
# Output:

3
4
5
6
7
8
```

How the “for” statement work?

- Here is what really happens behind the scenes when you run the **for** statement:

```
c = Counter(3, 8)
for i in c:
    print i
```



```
c = Counter(3, 8)
While True:
    try:
        i = c.next()
        print i
    except StopIteration:
        break
```

(syntactic sugar)

Fibonacci Iterator

- The Fibonacci class creates Fibonacci objects that can generate all the Fibonacci numbers up to some limit

```
class Fibonachi:
    def __init__(self, n):
        self.stop = n
        self.a = 1
        self.b = 1

    def __iter__(self):
        return self

    def next(self):
        c = self.a + self.b
        if c > self.stop:
            raise StopIteration
        self.a = self.b
        self.b = c
        return c
```

Fibonacci Tests

- The following three tests are equivalent and have the same output:

```
def test1():
    fit = Fibonachi(50)
    print fit.next()
    print fit.next()
    print fit.next()
    print fit.next()
    print fit.next()
    print fit.next()
    print fit.next()

def test2():
    fit = Fibonachi(50)
    for i in fit:
        print i

def test3():
    for i in Fibonachi(50):
        print i
```

```
test1 output:
2
3
5
8
13
21
34

test2 output:
2
3
5
8
13
21
34

test3 output:
2
3
5
8
13
21
34
```

When ‘list’ meets an ‘iterator’

- The list constructor can accept iterators !
- In such case it constructs a list by running the next() method until it gets all its elements
- It returns the list of all elements the next() generates

```
def test4():
    print list(Fibonachi(1000))

def test5():
    fit = Fibonachi(1000000)
    print list(fit)[-1]
```

```
test4 output:
  [2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]

test5 output:
  832040
```

Automatic Iterators

- An automatic iterator implementation for any class that defines the two methods:

__len__ **__getitem__**

```
class Series:
    def __init__(self, start, end, step=1):
        self.start = start
        self.step = step
        self.length = (end - start) / step

    def __getitem__(self, i):
        if i >= self.length:
            raise IndexError
        return self.start + i * self.step

    def __len__(self):
        return self.length
```

```
s = Series(3,24,5)
it = iter(s)
```

Automatic Iterators

- If a class C has the two methods **__getitem__** and **__len__** then any object c of this lass can be transformed into an iterator by simply: it = iter(c)

```
s = Series(3,24, 5)
print "Length =", len(s)
print s[0]
print s[1]
print s[2]
print s[3]
```

Output

```
Length = 4
3
8
13
18
```

```
s = Series(3,24, 5)
it = iter(s)
print it.next()
print it.next()
print it.next()
print it.next()
```

Output

```
3
8
13
18
```

Automatic Iterators

- You get to enjoy all the benefits that a usual iterator object has!
- Note that '**for**' and '**list**' automatically advance **s** to **iter(s)**

```
s = Series(3,24, 5)
for i in s:
    print i
```

Output

```
3
8
13
18
```

```
for i in Series(3, 24, 5):
    print i
```

```
s = Series(3,24, 5)
print list(iter(s))
```

Output

```
[3, 8, 13, 18]
```

```
# surprise! don't need iter(s)
# list already takes care of it!
s = Series(3,24, 5)
print list(s)
```

Surprise: why Series is better than range ?

- Take the time to check out why Series(0,1000000) is much more better than range(0,1000000) ?

```
>>> memory_usage()
('21.93M', '26.00M')
>>> A = range(0, 1000000)
>>> memory_usage()
('33.99M', '37.72M')
>>> s = Series(0, 1000000)
>>> memory_usage()
('33.99M', '37.72M')
```

```
def memory_usage(pid=0):
    import psutil
    if pid == 0:
        pid = os.getpid()
    p = psutil.Process(pid)
    m = p.get_memory_info()
    vms = "%.2fM" % (m.vms / (1024.0**2))
    rss = "%.2fM" % (m.rss / (1024.0**2))
    return vms, rss
```

Directory Walker Abstract Data Type

- `dw = DirectoryWalker(directory_path)` — **Constructor**
  - Create a new directory walker
- `dw.next()` — **Accessor**
  - Get next file in directory
  - All files and subdirectories must be traversed!
  - Raise **StopIteration** exception when done

Directory Walker in Action 1

- Example 1:

```
dw = DirectoryWalker("c:/windows")
dw.next()
⇒c:/windows
dw.next()
⇒c:/windows/addins
dw.next()
⇒c:/windows/ADFS
dw.next()
⇒c:/windows/AppCompat
dw.next()
⇒c:/windows/apppatch
...
```

Directory Walker in Action 2

- Example 2:

```
dw = DirectoryWalker("c:/anaconda")
for filepath in dw:
    if filepath[-4:] == '.exe':
        print filepath
```

```
c:/anaconda/python.exe
c:/anaconda/pythonw.exe
c:/anaconda/Removepandas.exe
c:/anaconda/Removepy2exe.exe
c:/anaconda/Uninstall-Anaconda.exe
c:/anaconda/Scripts/ipengine.exe
c:/anaconda/Scripts/ipython.exe
c:/anaconda/Scripts/irunner.exe
c:/anaconda/Scripts/pep8.exe
c:/anaconda/Scripts/pip.exe
c:/anaconda/Scripts/py.test.exe
c:/anaconda/Scripts/skivi.exe
```

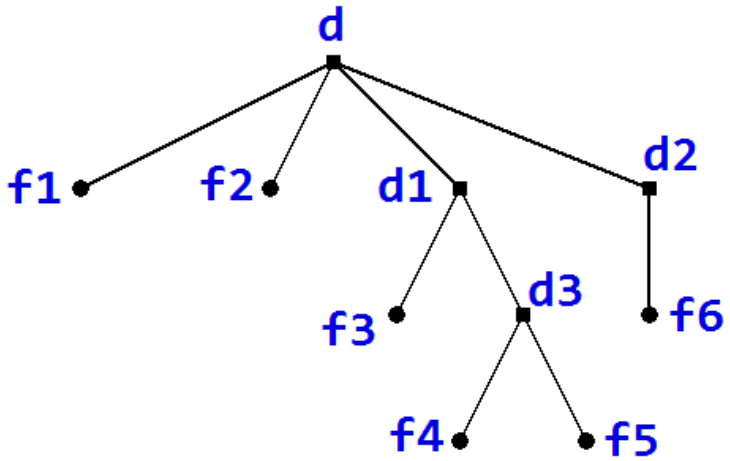
DirectoryWalker Implementation

```
class DirectoryWalker:
    def __init__(self, directory_path):
        self.que = [directory_path]

    def next(self):
        if not self.que:
            raise StopIteration
        else:
            path = self.que.pop(0)
            if os.path.isdir(path):
                for filename in os.listdir(path):
                    file_path = path + '/' + filename
                    self.que.append(file_path)
            return path

    def __iter__(self):
        return self
```

DirectoryWalker: State Diagram




```
w = DirectoryWalker(d)          w.queue = [d]
w.next() => d                   w.queue = [f1, f2, d1, d2]
w.next() => f1                  w.queue = [f2, d1, d2]
w.next() => f2                  w.queue = [d1, d2]
w.next() => d1                  w.queue = [d2, f3, d3]
w.next() => d2                  w.queue = [f3, d3, f6]
w.next() => f3                  w.queue = [d3, f6]
w.next() => d3                  w.queue = [f6, f4, f5]
w.next() => f6                  w.queue = [f4, f5]
w.next() => f4                  w.queue = [f5]
w.next() => f5                  w.queue = []
w.next() => StopIteration Exception!
```

Walking Advantages

- Today's file trees are very large!
- Impossible to hold all the paths in a single memory list!
- Walking holds a minimal subset at every given moment, but able to generate all the paths!
- There's no need to hold all the paths in memory, we only need to hold one path at time to do almost any computation on a files tree

Use Python's os.walk

- It is faster and better tuned to all operating systems and for most practical applications
- It is fully supported by the Python core libraries

```
def dir_size(directory_path):
    size = 0
    for path,dirs,files in os.walk(directory_path):
        for f in files:
            fpath = path + '/' + f
            size += os.path.getsize(fpath)
    return size
```

Generators

- Fast easy way to create iterators
- Looks like a function which saves it state …

```
def countdown(n):
    while n > 0:
        yield n
        n = n - 1
```

```
c = countdown(5)
print c.next()
print c.next()
print c.next()
print c.next()
print c.next()
```

Thanks Dave Beazley for the great Examples:
http://www.dabeaz.com/generators/Generators.pdf

Generators

- Only useful purpose is iteration
- Cannot be reused (after iteration done)

```
def pairs(n):
    "Generator of all pairs (i,j), i<j<n"
    for i in range(n):
        for j in range(i,n):
            yield (i,j)
```

```
def test2():
    g = pairs(10000)
    for i,j in g:
        if i+j == 10000 and j-i == 100:
            print i, j
```

http://www.dabeaz.com/generators/Generators.pdf

QUIZ

- Create an iterator class Pairs that does the same thing
  - Which is easier?
- How much memory is needed to create the list of all pairs (i,j), where i<j<n=10000?
- Exercise: generator for generating all subsets of size k of elements from [0,1,…,n-1]

Directory Walker – 1

```
def dirwalk(dirpath):
    "Generate all files in directory tree"
    que = [dirpath]
    while que:
        path = que.pop(0)
        yield path
        if os.path.isdir(path):
            for file in os.listdir(path):
                file_path = path + '/' + file
                que.append(file_path)
```

Directory Walker – 2

```
def dirwalk(dirpath):
    "Generate all files in directory tree"
    for f in os.listdir(dirpath):
        path = dirpath + '/' + f
        yield path
        if os.path.isdir(path):
            for x in dirwalk(path):
                yield x
```