Introduction to Python

Lecture 2
2/8/2009

Today

Object-oriented programing is an exceptionally bad idea which cold only have originated in California.

--- Dijkstra

Today

- Everything is an Object
- Principals of Object Orient Programming
- A Class is different then an Object
    - unless it is a *class object*
- Python Classes
- Inheritance
    - exceptions

Last Time

- Basic of Python
- Basic Types
- Control Flow
- builtins

Objects in Python

- **EVERYTHING IS AN OBJECT**
- Basic types are all objects
  - Some behave more like and less like *classic* objects
- You can write your own class descriptions for objects
  - Even the description is an object!
- This concept is different then
  Object Oriented Programming

So, What is OOP?

- Why do we use objects?
- What is an object?
- Principles of OOP?
- Wait, isn't everything an object?
- How do we make our own objects?

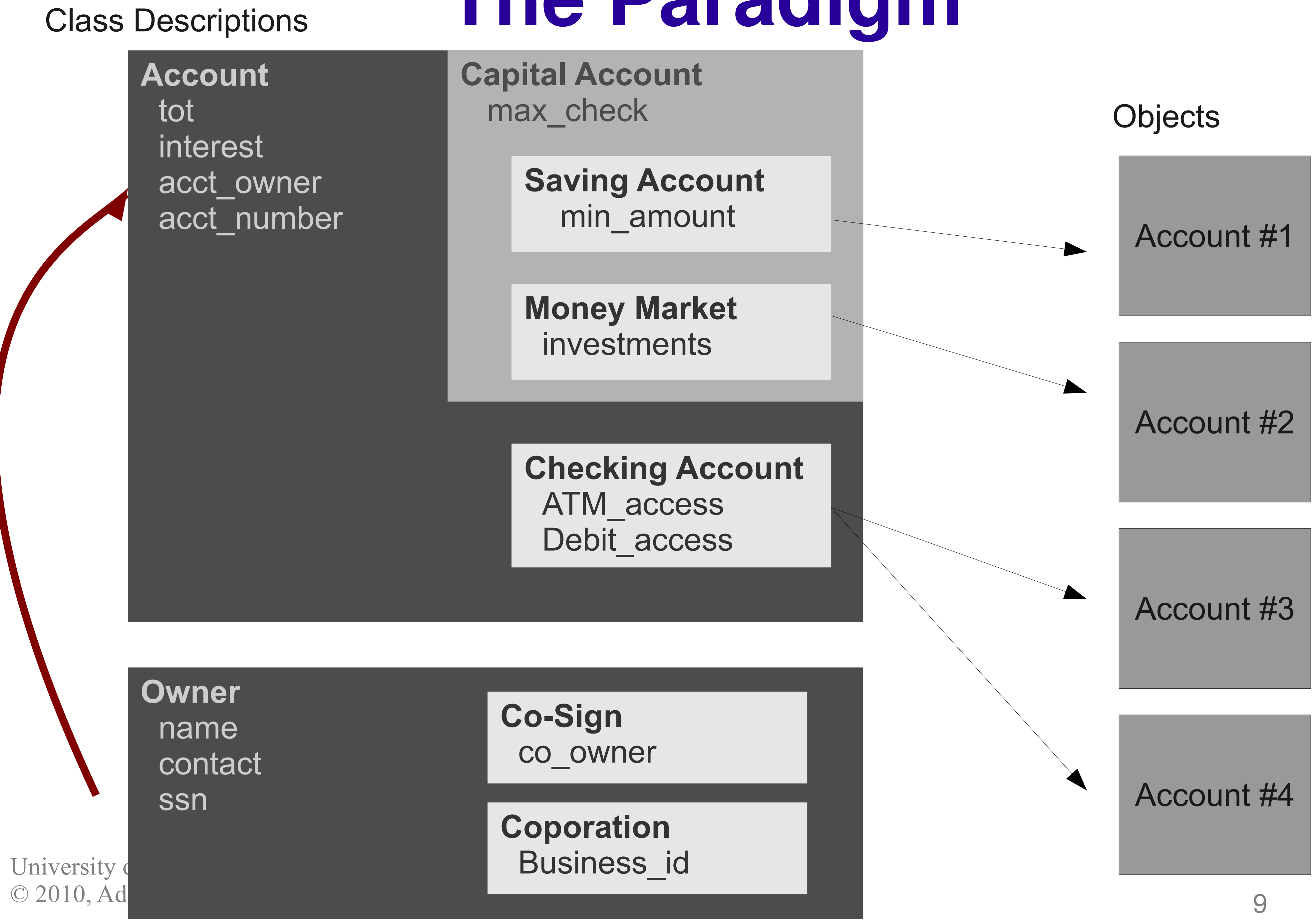
Suppose we are a Bank

- We have accounts
- Some accounts are savings accounts some are checking account, and other are money market account
- All accounts have a total, but different interest rates
- Some accounts limit the number of checks
- How do we organize this system?

We are still a bank?

- We designed our system to store accounts
  - Checking, saving, and money market
- What if we need to add another account type?
- What about the owners of the account?
- Some is a single person, or co-signers, or corporations
- What about combining accounts? Defining '+' between accounts?

The Paradigm



Principles of OOP

- Multiple Instances
  - Define it once, use it many times
- Inheritance/Customization
  - Customize Someone else's object to fit your need
- Operation Overloading
  - Overwrite for example the str() function or what it means to + to objects

Class vs. Instance

- Class is a definition
- Instance is an object created using the definition in a class
- Most languages, changing a class does not effect an instance
- Python is more dynamic (for good and bad)
  - Altering a class can effect an instance of it

First Example

```
#!/usr/bin/python
# first_class.py

class storage:
    'simple storage object'
    def set_data(self,data):
        'set the data'
        self.data = data
    def get_data(self,data):
        'get the data'
        return self.data
    def rm_data(self,data):
        'delete the data'
        del self.data
```

Using the Class

```
$>python
>>> from first_class import storage
>>> print storage
first_class.storage
>>> print type(storage)
<type 'classobj'>
>>> s = storage()
>>> print s
<first_class.storage instance at 0x98a128c>
>>> print type(s)
<type 'instance'>
>>> s.set_data('Spam')
>>> s.get_data()
'Spam'
>>> s.rm_data()
>>> s.get_data()
   ...
AttributeError: storage instance has no attribute 'data'
```

Class vs. Class Object

- `class` is an executable statement
  - Like `def` or assignment
  - Enters the class name into the local context
- Output of `class` statement is a ***class object***
  - The class name references the class object
- Calling a class object returns an ***instance object*** of the class
- Make Sense?

example

```
$>python
>>> class test: pass
...
>>> dir()#entered into local scope
['__builtins__', '__doc__', '__name__', 'test']
>>> type(test)
<type 'classobj'>
>>> t = test() #call to generate an instance
>>> type(t)
<type 'instance'>
>>> t.__class__ #get the class object from instance
<class __main__.test at 0xb70bd89c>
>>> isinstance(t,t.__class__) #instance of?
True
>>> issubclass(t.__class__,test) #subclass of?
True
```

`self`

- Not a reserved word, but treated as such
- Refers to the current instance of the class
- Provides access to data and methods during class specification
  - **`self.set_data()`**
  - **`self.data = data`**
- Also, used to pass this instance to supper classes for inheritance
- Like ***this*** in Java

More about my-`self`

- Self is the first argument of class methods
  - `obj.fun(10) --> fun(obj,10)`
- Otherwise you get argument errors

```
>>> class test:
...     def hello():
...             return "Hello"
>>> test().hello()
TypeError: hello() takes no arguments
(1 given)
```

```
>>> class test:
...     def hello(self):
...             return "Hello"
...
>>> test().hello()
'Hello'
```

Must define function within the class enclosure?

- No! This is perfectly fine too.
- What else does this imply? (example later)

```
>>> def hello(self, x):
...   print "Hello", x
>>>
>>> class h: pass
...
>>> h.f = hello
>>> h.f("Adam")
Hello Adam
```

`__init__()`

- The **`__init__`** function is the constructor
  - General initialization
- If things are constructed statically, technically unneeded
- You are overwriting the generic **`__init__`** function

```
>>> class test:
...     a = 10
...
>>> class test_2:
...     def __init__(self,a):
...             self.a=a
...
>>> t = test()
>>> t2 = test_2(10)
>>> t.a,t2.a
(10, 10)
```

Why the __ ?

- The double underscore, front and back, is special
- Python's operator overloading **(do example!)**
- `__init__()` is called when you call the class object, i.e., `class_obj()`
- There are more overloading options
  - `__del__ (del obj), __add__ (obj + obj) __cmp__ (obj == obj, obj < obj)`
- Investigate more in book on pg. 492
- Section 3 of Python Library Reference

More __ !

- You can also define your own functions with a leading double underscore
  - But don't use a trailing one
- For *<air quote>* **private** *<air quote>* functions
  - Not really private
  - Just obfuscated, occurs at compile time
  - Add to class dynamically with __ won't obfuscate
  - For Example
    - `__priv(self,a) --> _class__priv(self,a)`
    - `__a = 10 --> _class__a = 10`

Pseudo Privacy Protections

- You can directly change data within an instance
  - Overwrite a function, change a variable value
- You can add new data to an instance
- You can delete data from an instance and from the class
- Not really private data
  - Functions or variables
- Alter the class object, and effect all instances that are already created!

Lack of Privacy Example

```
>>> class test:
...       a = 10
...
>>> t = test()
>>> t.a
10
>>> t.a += 1
>>> t.a
11
>>> test.b = 1
>>> t.b
1
>>> del test.a
>>> t.a
11
```

```
>>> class test:
...    a = 10
...    __a = 9
...    def pr(self):
...       print self.__a,self.a
>>> test.b = 20
>>> t = test()
>>> t.b
20
>>> t.__a
AttributeError:'__a' not found
>>> t._test__a #pseudo private
10
>>> t.pr()
10 9
>>> test.pr = 20 #classobj
>>> t.pr()
TypeError:... int not callable
```

Inheritance

```
>>> class Super:
...     def __init__(self,data):
...             self.data = data
...
>>> class Sub(Super):
...     def __init__(self,data,more_data):
...             Super.__init__(self,data)
...             self.more_data = more_data
...
>>> s = Sub("Spam","Eggs")
>>> s.data,s.more_data
('Spam', 'Eggs')
```

Passing Larger Class

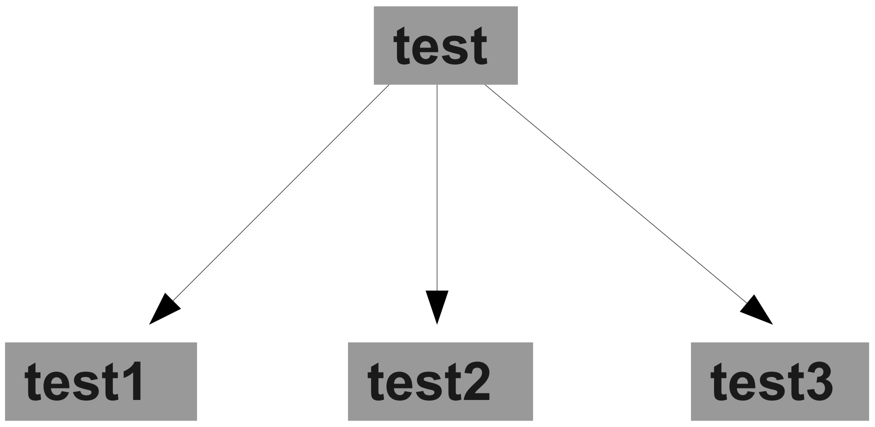
Overwriting **Super**'s **__init__**

calling **Super** class constructor

must pass **self** when calling larger class's functions

Attribute Look up

- Proceeds in a Depth First Search manner
- Ordering of the inheritance matter
  - `class test (test1,test2,test3):`
  - Look in test1 then test2 then test3



- Do Demo!!!

`isinstance()`

- Suppose you have an instance object, and you need to know it's type/class
  - **`isinstance(instance, class-obj)`**
  - **`isinstance([],type([]) == True`**
- Respects inheritance
  - **`isinstance([], object) --> True`**
  - **`object`** is the base object, everything inherits from it
- Not as precise as a type comparison with 'is'

`issubclass()`

- Need to know if one class is a subclass of another
  - **`issubclass(C,B)`** → **`bool`**
    "Is class C a sub-class of B?"
  - **`issubclass(C,(B1,B2,B3,...))`** → **`Bool`**
    "Is class C a sub-class of B1 or B2 or ...?"
- **`issubclass(C,object)`** → ?

Word of Caution

- Don't over classify
- Too much inheritance can be really hard to debug
- Brian W. Kerningham:
  - *"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."*
  - **Don't shoot your self in the foot**
- On that note, lets return to the bank example

Do Bank Example Demo

- We were a bank, we had different kind of accounts that were related to each other
- We had different types of ownership for the accounts
- Design questions in OOP?
- Show the demo!!
  - Don't program like this, to much inheritance

On Your Own!

- Investigate staticmethod()
  - How do you make static methods for an object?
- Investigate classmethod()
  - What is the difference between the two?
- Nested Classes
  - Within classes
  - Within functions
  - What is the scoping and how does it work?