Data Abstraction vs. Function Abstraction

• Functions perform *computations*; their specifications abstract from possible implementations of a particular computation.

• In the old days, programs tended to be organized around functions or modules comprising related functions. The data were just the operands.

• Now we tend to organize instead around the data—around *objects* or types (*classes*) of objects.

• Objects have *state*, which is accessed and manipulated by means of *attributes*.

• The set of attributes and their behavior is analogous to the syntactic and semantic specification of a function.

• In previous lectures, we’ve seen standard Python objects and ways to get (in effect) new kinds of objects using functions and non-local variables. We’ve defined data types using them by defining a set of functions to be used to construct, query, and modify them.

• Python also provides a standard way to gather together state and attributes of new types of data: *classes*. 
Extending the Mutable Objects: Classes

- In languages such as Python, Java, and C++, an object is an instance of a class; the class is called the object's type.
- The Python `class` statement defines new classes or types, creating new, vaguely dictionary-like varieties of object.
Simple Classes: Bank Account

# type name
class Account:
    # constructor method
def __init__(self, initial_balance):
        self._balance = initial_balance

def balance(self):  # instance method
    # instance variable:
    return self._balance

def deposit(self, amount):
    if amount < 0:
        raise ValueError("negative deposit")
    self._balance += amount

def withdraw(self, amount):
    if 0 <= amount <= self._balance:
        self._balance -= amount
    else: raise ValueError("bad withdrawal")

>>> mine = Account(1000)
>>> mine.deposit(100)
>>> mine.balance()
1100
>>> mine.withdraw(200)
>>> mine.balance()
900
Class Concepts

• Just as `def` defines functions and allows us to extend Python with new operations, `class` defines types and allows us to extend Python with new kinds of data.

• What do we want out of a class?
  - A way of defining named `new types` of data.
  - A means of defining and accessing `state` for these objects.
  - A means of defining `operations` specific to these objects.
    * In particular, an operation for `initializing` the state of an object.
  - A means of `creating` new objects.
Class Machinery

• The Account type illustrated how we do each of these

```python
class Account:    # Define named new type
    def __init__(self, initial_balance):  # How to initialize
        self._balance = initial_balance  # Create/modify state

    def balance(self):  # Define new operation on Accounts
        return self._balance  # Access state of an Account

...

myAccount = Account(1000)  # Create a new Account object,
print(myAccount.balance())  # Operate on an Account object.
```
Attribute Access

• In general, the notation $X.Y$ means “The value named $Y$ in the object pointed to by $X$.”

• Unlike C++ or Java, Python takes a very dynamic approach.

• Classes and class instances behave rather like environment frames.

• Given a pointer to some object, $\text{obj}$,
  
  - $\text{obj}.x = \text{value}$ looks for a definition of $x$ in the object referenced by $\text{obj}$, creating one if it doesn’t exist, and assigning $\text{value}$ to it.
  
  - When not being assigned to, $\text{obj}.x$ returns the definition of $x$ in the object referenced by $\text{obj}$, if any,
  
  - ...and if there is no such definition, it returns the value defined for $x$ in the class itself, if any.
class Account:
    _total_deposits = 0

    def __init__(...):
        self._balance = ...
        Account._total_deposits = ...

acct1 = Account(1000)
acct2 = Account(10000)
acct1.deposit(300)

• Curved boxes are objects.
• Flat-bottomed boxes are class objects.
• 'x.y': look for 'y' starting at 'x'
Assigning to Attributes

- Assigning to an attribute of an object (including a class) is like assigning to a local variable: it creates a new binding for that attribute in the object selected from (i.e., referenced by the expression on the left of the dot).

```python
>>> class Value:
    ...    value = 0
    ...
>>> val1 = Value()
>>> val2 = Value()
>>> val2.value = 3
>>> val1.value
0
>>> Value.value
0
>>> val2.value
3
```
Attributes of Classes

• In Python classes themselves are objects.

• (You might well ask “What is the type of a class?” Answer: a builtin class called type, whose type is itself.)

• Therefore, classes themselves have attributes.

• Assignments and defs immediately inside a class define class attributes.

• Since obj.x looks for x in the class of obj if it doesn’t find it in obj itself, the attributes defined in a class provide default values for attributes of the object that are instances of the class.
**Methods**

- **Consider**
  ```python
  >>> class Foo:
  ...     def set(self, x):
  ...         self.value = x
  >>> aFoo = Foo(10)
  ```

- **The access** `aFoo.set` returns the `set` method defined in `Foo` (since we haven’t set it in `aFoo`).

- However, in this particular case (function retrieved from the class of an object), what gets returned is a little different.
  ```python
  >>> aFoo.set
  <bound method Foo.set of ...>
  ```

- A **bound method** is an ordinary function that has its first parameter “pre-bound” to a particular value—in this case to `aFoo`.
  ```python
  >>> aFoo.set(13)  # First parameter (self) of set is aFoo, x is 13.
  >>> aFoo.value
  13
  ```

- **The effect is (almost) the same as**
  ```python
  >>> Foo.set(aFoo, 13)
  ```
Class Attributes in Python

• Sometimes, a quantity applies to a type as a whole, not a specific instance.

• For example, with Accounts, you might want to keep track of the total amount deposited from all Accounts.

• This is an example of something confusing called a class attribute.
class Account:

    _total_deposits = 0                   # Define/initilize a class attribute

    def __init__(self, initial_balance):
        self._balance = initial_balance
        Account._total_deposits += initial_balance

    def deposit(self, amount):
        self._balance += amount
        Account._total_deposits += amount

    def total_deposits():              # Define a class method.
        return Account._total_deposits

>>> acct1 = Account(1000)
>>> acct2 = Account(10000)
>>> acct1.deposit(300)
>>> Account.total_deposits()
11300
Classes and Operators

- Many standard operators defined in Python are essentially “syntactic sugar” for method calls.

- Examples:
  - $x+y$ becomes $x.__add__(y)$ if $__add__$ is defined for $x$.
  - $x[k]$ becomes $x.__getitem__(k)$.
  - $x[k] = 3$ becomes $x.__setitem__(k, 3)$.
  - $\text{len}(x)$ calls $x.__len__()$.
  - $\text{repr}(x)$ calls $x.__repr__()$, which is what the interpreter uses to print the value of expressions you type.
Class Machinery: Summary

- Classes have **attributes**, created by assignment statements and `defs` in the class body.

- Function-values attributes of classes are called **methods**.

- Classes beget objects called **instances**, created by “calling” the class: `Account(1000)`.

- Each such `Account` object initially shares the attributes of its class.

- Attributes can be accessed using `object.attribute` notation.

- A method call `mine.deposit(100)` is essentially the same as `Account.deposit(mine, 100)`.

- By convention, we call the first argument of a method `self` to indicate that it is the object from which we got the method.

- When an object is created, the special `__init__` method is called on it first.

- Assigning to an attribute of an object (`a.b = v`) gives that object its own attribute (not shared with the class), if it doesn’t have it already.