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 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 the last lecture, we saw one way to get objects using functions and non-local variables.
- This is not the usual way it’s done, however.

Extending the Mutable Objects: Classes

- In languages such as Python, Java, and C++, an object is an instance of a class.
- The Python class statement defines new classes or types, creating new, vaguely dictionary-like varieties of object.

Simple Classes: Bank Account

```python
# type name
class Account:
    # constructor method
def __init__(self, initial_balance):
        self._balance = initial_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 and using 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):
        self._balance = initial_balance
    # how to initialize
    # create/modify state
    def deposit(self, amount):
        self._balance += amount
    # define new operation on Accounts
    # access state of an Account
    def balance(self):
        return self._balance

myAccount = Account(1000)
print(myAccount.balance())
```

Class Attributes

- 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 a class attribute.

Class Attributes in Python

class Account:
    _total_deposits = 0  # Define/initialize 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
    @staticmethod
    def total_deposits():
        return Account._total_deposits

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

Modeling Attributes in Python

- Unlike C++ or Java, Python takes a very dynamic approach.
- Classes and class instances behave rather like environment frames.

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).

Methods

- Consider

```python
>>> class Foo:
...     def set(self, x):
...         self.value = x
>>> aFoo = Foo()
>>> aFoo.set(13)  # The first parameter of set is aFoo.
>>> aFoo.value
13
>>> aFoo.set
<bound method Foo.set of ...>
```

- Selection of function-valued attributes from objects (other than classes) creates bound methods: first parameter is bound to the selected-from object, leaving one fewer parameters.
- Effect of selecting `aFoo.set` is like calling `partial_bind(aFoo, Foo.set)`, where
  ```python
def partial_bind(obj, func): return lambda x: func(obj, x)
```

Class Machinery: Summary

- Classes have attributes, created by assignment statements and `def` 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.