Getting Inside Objects and Classes

- Programmers who use objects and classes know:
  - Interface that can be used with a class
  - State of an object
  - How to instantiate a class to obtain an object
- Objects are abstractions
  - Package their state and methods in a single entity that can be referenced with a name
- Class definition is like a blueprint for each of the objects of that class

A First Example: The `Student` Class

- A course-management application needs to represent information about students in a course

```
>>> from student import Student
>>> s = Student("Maria", 5)
>>> print(s)
Name: Maria
Scores: 0 0 0 0 0
>>> s.setScore(1, 100)
>>> print(s)
Name: Maria
Scores: 100 0 0 0 0
>>> s.getHighScore()
100
>>> s.getAverage()
20
>>> s.getScore(1)
100
>>> s.getName()
'Maria'
>>>
```


The `Student` Class (continued)

Student METHOD	WHAT IT DOES
s = Student(name, number)	Returns a **Student** object with the given **name** and **number** of scores. Each score is initially 0.
s.getName()	Returns the student's name.
s.getScore(i)	Returns the student's **i**th score. **i** must range from 1 through the number of scores.
s.setScore(i, score)	Resets the student's **i**th score to **score**. **i** must range from 1 through the number of scores.
s.getAverage()	Returns the student's average score.
s.getHighScore()	Returns the student's highest score.
s.__str__()	Same as **str(s)**. Returns a string representation of the student's information.

[TABLE 8.1] The interface of the **Student** class

The `Student` Class (continued)

- Syntax of a simple class definition:

```
class <class name>(<parent class name>):
    <method definition-1>
    …
    <method definition-n>
```

← class header

  - Class name is a Python identifier
    - Typically capitalized
- Python classes are organized in a tree-like **class hierarchy**
  - At the top, or root, of this tree is the **`object`** class
  - Some terminology: **subclass**, **parent class**

The `Student` Class (continued)

```
def getAverage(self):
    """Returns the average score."""
    return sum(self._scores) / len(self._scores)

def getHighScore(self):
    """Returns the highest score."""
    return max(self._scores)

def __str__(self):
    """Returns the string representation of the student."""
    return "Name: " + self._name  + "\nScores: " + \
           " ".join(map(str, self._scores))
```

Docstrings

- Docstrings can appear at three levels:
  - Module
  - Just after class header
    - To describe its purpose
  - After each method header
    - Serve same role as they do for function definitions
- **`help(Student)`** prints the documentation for the class and all of its methods

Method Definitions

- Method definitions are indented below class header
- Syntax of method definitions similar to functions
  - Can have required and/or default arguments, return values, create/use temporary variables
  - Returns **`None`** when no **`return`** statement is used
- Each method definition must include a first parameter named **`self`**
- Example: `s.getScore(4)`
  - Binds the parameter **`self`** in the method **`getScore`** to the **`Student`** object referenced by the variable **`s`**

The `__init__` Method and Instance Variables

- Most classes include the **`__init__`** method

```
def __init__(self, name, number):
    """All scores are initially 0."""
    self._name = name
    self._scores = []
    for count in range(number):
        self._scores.append(0)
```

  - Class's **constructor**
  - Runs automatically when user instantiates the class
- Example: `s = Student("Juan", 5)`
- **Instance variables** represent object attributes
  - Serve as storage for object state
  - Scope is the entire class definition

The `__str__` Method

- Classes usually include an **`__str__`** method
  - Builds and returns a string representation of an object's state

```
def __str__(self):
    """Returns the string representation of the student."""
    return "Name: " + self._name  + "\nScores: " + \
           " ".join(map(str, self._scores))
```

- When **`str`** function is called with an object, that object's **`__str__`** method is automatically invoked
- Perhaps the most important use of **`__str__`** is in debugging

Accessors and Mutators

- Methods that allow a user to observe but not change the state of an object are called **accessors**
- Methods that allow a user to modify an object's state are called **mutators**

```
def setScore(self, i, score):
    """Resets the ith score, counting from 1."""
    self._scores[i - 1] = score
```

- Tip: if there's no need to modify an attribute (e.g., a student's name), do not include a method to do that

The Lifetime of Objects

- The lifetime of an object's instance variables is the lifetime of that object
- An object becomes a candidate for the graveyard when it can no longer be referenced

```
>>> s = Student("Sam", 10)
>>> csci111 = [s]
>>> csci111
[<__main__.Student instance at 0x11ba2b0>]
>>> s
<__main__.Student instance at 0x11ba2b0>
>>> s = None
>>> csci111.pop()
<__main__.Student instance at 0x11ba2b0>
>>> print(s)
None
>>> csci111
[]
```

`Student` object still exists, but interpreter will recycle its storage during **garbage collection**

Rules of Thumb for Defining a Simple Class

- Before writing a line of code, think about the behavior and attributes of the objects of new class
- Choose an appropriate class name and develop a short list of the methods available to users
- Write a short script that appears to use the new class in an appropriate way
- Choose appropriate data structures for attributes
- Fill in class template with **`__init__`** and **`__str__`**
- Complete and test remaining methods incrementally
- Document your code

Case Study: Implementation (Coding)

```
"""
File: die.py

This module defines the Die class.
"""

from random import randint

class Die(object):
    """This class represents a six-sided die."""

    def __init__(self):
        """The initial face of the die."""
        self._value = 1

    def roll(self):
        """Resets the die's value to a random number
        between 1 and 6."""
        self._value = randint(1, 6)

    def getValue(self):
        return self._value

    def __str__(self):
        return str(self._value)
```

Data-Modeling Examples

- As you have seen, objects and classes are useful for modeling objects in the real world
- In this section, we explore several other examples

Savings Accounts and Class Variables

`SavingsAccount` METHOD	WHAT IT DOES
`a = SavingsAccount(name, pin, balance = 0.0)`	Returns a new account with the given name, PIN, and balance.
`a.deposit(amount)`	Deposits the given amount from the account's balance.
`a.withdraw(amount)`	Withdraws the given amount from the account's balance.
`a.getBalance()`	Returns the account's balance.
`a.getName()`	Returns the account's name.
`a.getPin()`	Returns the account's PIN.
`a.computeInterest()`	Computes the account's interest and deposits it.
`__str__(a)`	Same as `str(a)`. Returns the string representation of the account.

[TABLE 8.5] The interface for `SavingsAccount`

Savings Accounts and Class Variables (continued)

```
class SavingsAccount(object):
    """This class represents a Savings account
    with the owner's name, PIN, and balance."""

    RATE = 0.02

    def __init__(self, name, pin, balance = 0.0):
        self._name = name
        self._pin = pin
        self._balance = balance

    def __str__(self):
        result =  'Name:    ' + self._name + '\n'
        result += 'PIN:     ' + self._pin + '\n'
        result += 'Balance: ' + str(self._balance)
        return result

    def getBalance(self):
        return self._balance

    def getName(self):
        return self._name

    def getPin(self):
        return self._pin
```


Savings Accounts and Class Variables (continued)

```
def deposit(self, amount):
    """Deposits the given amount and returns the
    new balance."""
    self._balance += amount
    return self._balance

def withdraw(self, amount):
    """Withdraws the given amount.
    Returns None if successful, or an
    error message if unsuccessful."""
    if amount < 0:
        return 'Amount must be >= 0'
    elif self._balance < amount:
        return 'Insufficient funds'
    else:
        self._balance -= amount
        return None

def computeInterest(self):
    """Computes, deposits, and returns the interest."""
    interest = self._balance * SavingsAccount.RATE
    self.deposit(interest)
    return interest
```

Putting the Accounts into a Bank

```
>>> from bank import Bank, SavingsAccount
>>> bank = Bank()
>>> bank.add(SavingsAccount("Wilma", "1001", 4000.00))
>>> bank.add(SavingsAccount("Fred", "1002", 1000.00))
>>> print(bank)
Name:    Fred
PIN:     1002
Balance: 1000.00
Name:    Wilma
PIN:     1001
Balance: 4000.00
>>> account = bank.get("1000")
>>> print(account)
None
>>> account = bank.get("1001")
>>> print(account)
Name:    Wilma
PIN:     1001
Balance: 4000.00
>>> account.deposit(25.00)
4025
>>> print(account)
Name:    Wilma
PIN:     1001
Balance: 4025.00
>>> print(bank)
```


Putting the Accounts into a Bank (continued)

Bank METHOD	WHAT IT DOES
b = Bank()	Returns a bank.
b.add(account)	Adds the given account to the bank.
b.remove(pin)	Removes the account with the given PIN from the bank and returns the account. If the pin is not in the bank, returns **None**.
b.get(pin)	Returns the account associated with the PIN if the PIN is in the bank. Otherwise, returns **None**.
b.computeInterest()	Computes the interest on each account, deposits it in that account, and returns the total interest.
__str__(b)	Same as **str(b)**. Returns a string representation of the bank (all the accounts).

[TABLE 8.6] The interface for the **Bank** class

Putting the Accounts into a Bank (continued)

```
class Bank(object):

    def __init__(self):
        self._accounts = {}

    def __str__(self):
        """Return the string rep of the entire bank."""
        return '\n'.join(map(str, self._accounts.values()))

    def add(self, account):
        """Inserts an account using its PIN as a key."""
        self._accounts[account.getPin()] = account

    def remove(self, pin):
        return self._accounts.pop(pin, None)

    def get(self, pin):
        return self._accounts.get(pin, None)

    def computeInterest(self):
        """Computes interest for each account and
        returns the total."""
        total = 0.0
        for account in self._accounts.values():
            total += account.computeInterest()
        return total
```


Using `pickle` for Permanent Storage of Objects

- `pickle` allows programmer to save and load objects using a process called **pickling**
  - Python takes care of all of the conversion details

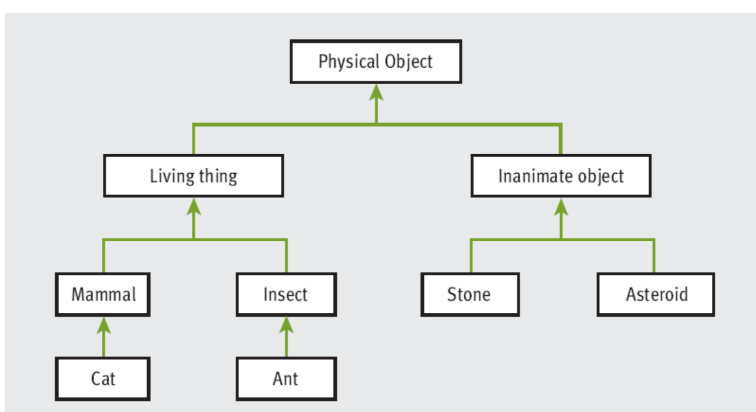
```
import pickle

def save(self, fileName = None):
    """Saves pickled accounts to a file.  The parameter
    allows the user to change filenames."""
    if fileName != None:
        self._fileName = fileName
    elif self._fileName == None:
        return
    fileObj = open(self._fileName, 'wb')
    for account in self._accounts.values():
        pickle.dump(account, fileObj)
    fileObj.close()
```

Structuring Classes with Inheritance and Polymorphism

- Most object-oriented languages require the programmer to master the following techniques:
  – **Data encapsulation:** Restricting manipulation of an object's state by external users to a set of method calls
  – **Inheritance:** Allowing a class to automatically reuse/ and extend code of similar but more general classes
  – **Polymorphism:** Allowing several different classes to use the same general method names
- Python's syntax doesn't enforce data encapsulation
- Inheritance and polymorphism are built into Python

Inheritance Hierarchies and Modeling



[FIGURE 8.3] A simplified hierarchy of objects in the natural world

Inheritance Hierarchies and Modeling (continued)

- In Python, all classes automatically extend the built-in **`object`** class
- It is possible to extend any existing class:

```
class <new class name>(<existing class name>):
```

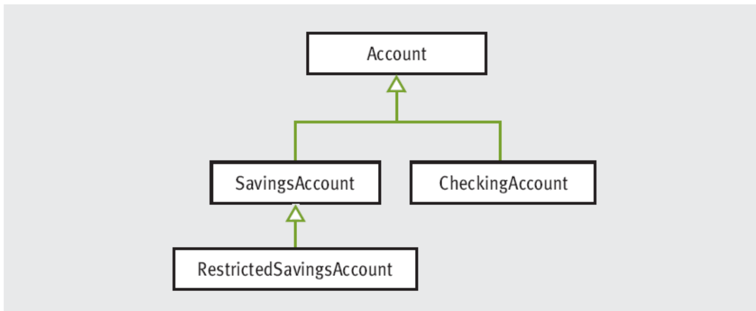
- Example:
  - **`PhysicalObject`** would extend **`object`**
  - **`LivingThing`** would extend **`PhysicalObject`**
- Inheritance hierarchies provide an abstraction mechanism that allows the programmer to avoid reinventing the wheel or writing redundant code

Polymorphic Methods

- We subclass when two classes share a substantial amount of **abstract behavior**
  - The classes have similar sets of methods/operations
  - A subclass usually adds something extra
- The two classes may have the same interface
  - One or more methods in subclass override the definitions of the same methods in the superclass to provide specialized versions of the abstract behavior
    - **Polymorphic methods** (e.g., the **`__str__`** method)

Abstract Classes

- An **abstract class** includes data and methods common to its subclasses, but is never instantiated



[FIGURE 8.5] An abstract class and three concrete classes

The Costs and Benefits of Object-Oriented Programming

- **Imperative programming**
  - Code consists of I/O, assignment, and control (selection/iteration) statements
  - Does not scale well
- Improvement: Embedding sequences of imperative code in function definitions or subprograms
  - **Procedural programming**
- **Functional programming** views a program as a set of cooperating functions
  - No assignment statements

The Costs and Benefits of Object-Oriented Programming (continued)

- Functional programming does not conveniently model situations where data must change state
- Object-oriented programming attempts to control the complexity of a program while still modeling data that change their state
  - Divides up data into units called objects
  - Well-designed objects decrease likelihood that system will break when changes are made within a component
  - Can be overused and abused

Summary

- A simple class definition consists of a header and a set of method definitions
- In addition to methods, a class can also include instance variables
- Constructor or `__init__` method is called when a class is instantiated
- A method contains a header and a body
- An instance variable is introduced and referenced like any other variable, but is always prefixed with `self`

Summary (continued)

- Some standard operators can be overloaded for use with new classes of objects
- When a program can no longer reference an object, it is considered dead and its storage is recycled by the garbage collector
- A class variable is a name for a value that all instances of a class share in common
- Pickling is the process of converting an object to a form that can be saved to permanent file storage
- **`try-except`** statement is used to catch and handle exceptions

Summary (continued)

- Most important features of OO programming: encapsulation, inheritance, and polymorphism
  - Encapsulation restricts access to an object's data to users of the methods of its class
  - Inheritance allows one class to pick up the attributes and behavior of another class for free
  - Polymorphism allows methods in several different classes to have the same headers
- A data model is a set of classes that are responsible for managing the data of a program


EAST CENTRAL UNIVERSITY
1909