Lecture Notes

Overview

Chapter 8 provides an introduction to objects and classes with Python. Students learn about object-oriented design and how to create object-oriented programs in Python. Important OO concepts like constructors, instance variables, methods, inheritance, polymorphism, and encapsulation are introduced. Special topics like exception handling (try-exception) and how to transfer objects to and from files (pickling/unpickling) are introduced as well.

Objectives

After completing this chapter, students will be able to:

- Determine the attributes and behavior of a class of objects required by a program
- List the methods, including their parameters and return types, that realize the behavior of a class of objects
- Choose the appropriate data structures to represent the attributes of a class of objects
- Define a constructor, instance variables, and methods for a class of objects
- Recognize the need for a class variable and define it
- Define a method that returns the string representation of an object
- Define methods for object equality and comparisons
- Exploit inheritance and polymorphism when developing classes
- Transfer objects to and from files

Getting Inside Objects and Classes

Review *object-oriented programming*, objects, and classes, noting that objects are abstractions that package their state and methods in a single entity which can be referenced with a name.

Review the components of a class definition. Refer to the bullet points on Pages 294-295.

Note for C/C++/Java/C# Programmers (taken from www.ibiblio.org/g2swap/byteofpython/read/oops.html): "Note that even integers are treated as objects (of the int class). This is unlike C++ and Java (before version 1.5) where integers are primitive native types. See help(int) for more details on the class. C# and Java 1.5 programmers will be familiar with this concept since it is similar to the boxing and unboxing concept."

For an introduction to object-oriented programming with Python, read: www.voidspace.org.uk/python/articles/OOP.shtml.

A First Example: The Student Class

Use the sample code provided in the book for the role of the Student class for a course-management application that needs to represent information about students in a course.

Use Table 8.1 for a description of the interface of the Student class.

Review the syntax of a simple class definition. The class name is typically capitalized.

Python classes are organized in a tree-like *class hierarchy*. Review the terms: object (class), *subclass*, and *parent class*.

Use the student.py code to show how to define a class in Python. For more information on how to define classes in Python, read: http://diveintopython.org/object_oriented_framework/defining_classes.html.

Docstrings

docstrings can appear at three levels: module, just after the class header, and after each method header.

Note that entering help (Student) at the shell prints the documentation for the class and all of its methods.

Method Definitions

Method definitions are indented below the class header.

Note the similarities and differences between function definitions and method definitions.

Each method definition **must** include a first parameter named self, which allows the interpreter to bind the parameter to the object used to call the method.

The __init__ Method and Instance Variables

The __init__ method is the class's *constructor* and is run automatically when a user instantiates the class. The syntax of this method is fixed and that it must begin and end with two consecutive underscores.

Review the role of *instance variables*. Their scope is the entire class definition. "Some people regard it as a Python 'wart' that we have to include self. Java includes it automatically and calls it this. The main argument in favour of self is the Pythonic principle explicit is better than implicit. This way we can see exactly where all our variable names come from." *Reference:* www.voidspace.org.uk/python/articles/OOP.shtml.

The __str__ Method

__str__ builds and returns a string representation of an object's state.

Note that when the str or print functions are called with an object, that object's __str__ method is automatically invoked.

Accessors and Mutators

Review *accessor* and *mutator* methods.

The Lifetime of Objects

The lifetime of an object's instance variables is the lifetime of that object.

Use the sample code provided in the book to illustrate that an object becomes a candidate for the graveyard when it can no longer be referenced. Review the term *garbage collector*.

Rules of Thumb for Defining a Simple Class

Review each of the rules of thumb for defining a simple class listed in Page 302 of the book.

Data-Modeling Examples

Review the data modeling examples that are discussed in this section.

Rational Numbers

Use the sample code provided in the book to see why it would be useful to implement a class for *rational numbers*. Review the term *overloading*.

Briefly go over the Rational class definition found on Pages 310-311 in the book.

Methods that are not intended to be in a class's interface are typically given names that begin with the _ symbol.

Rational Number Arithmetic and Operator Overloading

Use Tables 8.3, 8.4, and the __add__ example for how the built-in arithmetic operators can be overloaded.

Operator overloading is an abstraction mechanism. For more information on operator overloading in Python, read: www.learningpython.com/2008/06/21/operator-overload-learn-how-to-change-the-behavior-of-equality-operators/.

Comparison Methods

Use the example provided in the book to see how to overload and use the various comparison operators. Note the importance of overloading these operators using the appropriate comparison logic.

Once an implementer of a class has defined methods for == and <, the remaining methods can be defined in terms of these two methods.

If new class objects are comparable and the comparison methods are included in that class, other built-in methods—such as the sort method for lists—will be able to use the new class objects correctly.

Equality and the __eq__ Method

Use the example provided in the book to see how to overload and use the __eq__ method.

This method should also be applicable when comparing objects of two different types.

Note that one should include __eq__ in any class where a comparison for equality uses a criterion other than object identity.

Savings Accounts and Class Variables

Use Table 8.5 for a description of the interface for the SavingsAccount class.

Review the concept of a *class variable*. Review why such a variable would be useful in the SavingsAccount class. Review how a class variable is introduced and how it is referenced.

Putting the Accounts into a Bank

Use the sample code provided in the book and Table 8.6 to illustrate the functionality of the Bank class. The choice of a directory to represent the collection of accounts.

Using pickle for Permanent Storage of Objects

Review the term *pickling*, and use the save example provided in the book to see how to use pickle for permanent storage of objects.

Input of Objects and the try-except Statement

Review the syntax of the try-except statement and how it functions when an exception is generated.

Use the code on Page 321 of the book to see how to use a try-expect statement in a method. For more information on errors and exceptions in Python, read: http://docs.python.org/tut/node10.html.

Playing Cards

Use the sample code provided in the book to see the functionality of the Card class. Note why there is no need to include any methods other than __str__ for the string representation.

Use Table 8.7 and the sample code provided in the book to illustrate the functionality of the Deck class.

Structuring Classes with Inheritance and Polymorphism

Most object-oriented languages require the programmer to master the following techniques: *data encapsulation*, *inheritance*, and *polymorphism*.

While inheritance and polymorphism are built into Python, its syntax does not enforce data encapsulation. "Python does not really support encapsulation because it does not support data hiding through private and protected members. However some pseudo-encapsulation can be done. If an identifier begins with a double underline, i.e. __a, then it can be referred to within the class itself as self.__a, but outside of the class, it is named instance._classname__a. Therefore, while it can prevent accidents, this pseudo-encapsulation cannot really protect data from hostile code." *Reference:* www.astro.ufl.edu/~warner/prog/python.html.

Inheritance Hierarchies and Modeling

Use Figure 8.3 to review the role of *inheritance hierarchies*.

In Python, all classes automatically extend the built-in object class.

Inheritance provides an abstraction mechanism that allows programmers to avoid writing redundant code.

Example: A Restricted Savings Account

Use the RestrictedSavingsAccount class example to see how to extend an existing class while overloading some of the methods of the parent class.

Example: The Dealer and a Player in the Game of Blackjack

Use Figure 8.4 for a description of the role of the classes in the blackjack game application.

Use the sample code provided in the book to see how the Player class can be used to simulate a blackjack player.

Use the Dealer class example to see how to extend an existing class while overloading some of the methods of the parent class.

Use the sample code provided in the book to see how the Blackjack class can be used to set up and manage the interactions with the user

Polymorphic Methods

We subclass when two classes share a substantial amount of *abstract behavior*. Point out that a subclass usually adds something extra, such as a new method or data attribute, to the state provided by its superclass.

Review the role of *polymorphic methods* in enabling the two subclasses to have the same interface while performing different operations when the polymorphic method is called.

Abstract Classes

Use Figure 8.5 to see that an *abstract class* includes data and methods common to its subclasses but is never instantiated. Review how an abstract class differs from a *concrete class*, which can be instantiated.

"Abstract classes and interfaces are not supported in Python. In Python, there is no difference between an abstract class and a concrete class. Abstract classes create a template for other classes to extend and use. Instances can not be created of abstract classes but they are very useful when working with several objects that share many characteristics. For instance, when creating a database of people, one could define the abstract class Person, which would contain basic attributes and functions common to all people in the database. Then child classes such as SinglePerson, MarriedCouple, or Athlete could be created by extending Person and adding appropriate variables and functions. The database could then be told to expect every entry to be an object of type Person and thus any of the child classes would be a valid entry. In Python, you could create a class Person and extend it with the child classes listed above, but you could not prevent someone from instantiating the Person class." *Reference:* www.astro.ufl.edu/~warner/prog/python.html.

The Costs and Benefits of Object-Oriented Programming

Analyze the advantages and disadvantages of *imperative programming*, *procedural programming*, *functional programming*, and *object-oriented programming*.

With OO programming, although well-designed objects decrease the likelihood that a system will break when changes are made within a component, this technique can sometimes be overused.

Additional Resources

1. Python Basics:
   www.astro.ufl.edu/~warner/prog/python.html

2. Introduction to OOP with Python:
   www.voidspace.org.uk/python/articles/OOP.shtml

3. Defining Classes:
   http://diveintopython.org/object_oriented_framework/defining_classes.html

4. An Introduction to Python: Classes:
   www.penzilla.net/tutorials/python/classes/

5. Classes:
   http://docs.python.org/tut/node11.html

6. Errors and Exceptions:
   http://docs.python.org/tut/node10.html

Key Terms

- **abstract class:** A class that defines attributes and methods for subclasses, but is never instantiated.

- **accessor:** A method used to examine an attribute of an object without changing it.
- **behavior:** The set of actions that a class of objects supports.
- **class:** A description of the attributes and behavior of a set of computational objects.
- **class diagram:** A graphical notation that describes the relationships among the classes in a software system.
- **class variable:** A variable that is visible to all instances of a class and is accessed by specifying the class name.
- **concrete class:** A class that can be instantiated.
- **constructor:** A method that is run when an object is instantiated, usually to initialize that object's instance variables. This method is named _init_ in Python.
- **encapsulation:** The process of hiding and restricting access to the implementation details of a data structure.
- **garbage collection:** The automatic process of reclaiming memory when the data of a program no longer need it.
- **helper:** A method or function used within the implementation of a module or class but not used by clients of that module or class.
- **inheritance:** The process by which a subclass can reuse attributes and behavior defined in a superclass. See also subclass and superclass.
- **instance variable:** Storage for data in an instance of a class.
- **model/view/controller pattern (MVC):** A design plan in which the roles and responsibilities of the system are cleanly divided among data management (model), user interface display (view), and user event-handling (controller) tasks.
- **mutator:** A method used to change the value of an attribute of an object.
- **object-oriented programming:** The construction of software systems that define classes and rely on inheritance and polymorphism.
- **overloading:** The process of using the same operator symbol or identifier to refer to many different functions.
- **parent:** The immediate superclass of a class.
- **pickling:** The process of converting an object to a form that can be saved to permanent file storage.
- **polymorphism:** The property of one operator symbol or method identifier having many meanings. See also overloading.
- **procedural programming:** A style of programming that decomposes a program into a set of methods or procedures.
- **subclass:** A class that inherits attributes and behaviors from another class.
- **superclass:** The class from which a subclass inherits attributes and behavior.
- **Unified Modeling Language (UML):** A graphical notation for describing a software system in various phases of development.