Chapter 8

Design with Classes

At a Glance

Instructor’s Manual Table of Contents

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

Teaching Tips

Getting Inside Objects and Classes

1. Provide a brief introduction to *object-oriented programming*, objects, and classes, stressing that objects are abstractions that package their state and methods in a single entity which can be referenced with a name.

2. Provide an overview of the components of a class definition. Refer to the bullet points on Pages 294-295.

Teaching Tip	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."

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

A First Example: The `Student` Class

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

2. Use Table 8.1 to describe the interface of the `Student` class.

3. Describe the syntax of a simple class definition. Stress that the class name is typically capitalized.

4. Explain that Python classes are organized in a tree-like *class hierarchy*. Introduce the terms: `object` (class), *subclass*, and *parent class*.

5. Use the `student.py` code to show how to define a class in Python.

Teaching Tip	For more information on how to define classes in Python, read: http://diveintopython.org/object_oriented_framework/defining_classes.html.

Docstrings

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

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

Method Definitions

1. Explain that method definitions are indented below the class header.

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

3. Explain that 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

1. Use an example to explain the role of the `__init__` method. Explain that this method is the class's *constructor* and is run automatically when a user instantiates the class. Point out that the syntax of this method is fixed and that it must begin and end with two consecutive underscores.

2. Describe the role of *instance variables*. Stress that their scope is the entire class definition.

Teaching Tip	"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

1. Explain that `__str__` builds and returns a string representation of an object's state.

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

Accessors and Mutators

1. Remind students what *accessor* and *mutator* methods are. Use one or more examples to help students understand these concepts.

The Lifetime of Objects

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

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

Rules of Thumb for Defining a Simple Class

1. Introduce and explain each of the rules of thumb for defining a simple class listed in Page 302 of the book. You may add your own tips or ask students with previous OO experience to add their own.

Case Study: Playing the Game of Craps

1. Guide students as they step through this section.

Request

1. Ask students to write a program that allows the user to play and study the game of craps.

Analysis

1. Explain to students how they can use the nouns in the problem description to determine what types of classes are required to solve a problem. Introduce the role of the `Player` and `Die` classes.

2. Show and explain the sample user interface provided in the book.

Design

1. Use Table 8.2 to describe the interfaces of the `Player` and `Die` classes.

2. Review the pseudocode of the play method, and make sure students understand how this code captures the logic of playing a game of craps and tracking its results.

Implementation (Coding)

1. Review the code in Pages 306-309 of the book, specifying that the `Die` class is defined in a file named `die.py`. The `Player` class and the top-level functions are defined in a file named `craps.py`.

Quick Quiz 1

1. Why is an object considered an abstraction?
   Answer: Like functions, objects are abstractions. A function packages an algorithm in a single operation that can be called by name. An object packages a set of data values—its state—and a set of operations—its methods—in a single entity that can be referenced with a name. This makes an object a more complex abstraction than a function. To get inside a function, you must view the code contained in its definition. To get inside an object, you must view the code contained in its class. A class definition is like a blueprint for each of the objects of that class.

2. True or False: In a class, all of the method definitions are indented below the class header.
   Answer: True

3. True or False: Within the class definition, the names of instance variables must begin with `this`.
   Answer: False

4. Methods that allow a user to modify an object's state are called ____________________.
   Answer: mutator methods

Data-Modeling Examples

1. Provide students of an overview of the data modeling examples that are discussed in this section.

Rational Numbers

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

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

3. Explain that 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

1. Use Tables 8.3, 8.4, and the `__add__` example to explain how the built-in arithmetic operators can be overloaded.

2. Stress that *operator overloading* is an abstraction mechanism.

Teaching Tip	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

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

2. Point out that once an implementer of a class has defined methods for == and <, the remaining methods can be defined in terms of these two methods.

3. Stress that 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

1. Use the example provided in the book to show how to overload and use the `__eq__` method.

2. Point out that this method should also be applicable when comparing objects of two different types.

3. 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

1. Use Table 8.5 to describe the interface for the `SavingsAccount` class.

2. Explain the concept of a *class variable*. Tell students why such a variable would be useful in the `SavingsAccount` class. Demonstrate how a class variable is introduced and how it is referenced.

3. Briefly step through the code of this class, pointing out the class variable and the use balance as an optional argument.

4. Briefly outline the guidelines for using class variables.

Putting the Accounts into a Bank

1. Use the sample code provided in the book and Table 8.6 to describe the functionality of the `Bank` class. Briefly step through the code of this class. Explain the choice of a directory to represent the collection of accounts.

Using `pickle` for Permanent Storage of Objects

1. Introduce the term *pickling,* and use the `save` example provided in the book to show how to use `pickle` for permanent storage of objects.

Input of Objects and the `try-except` Statement

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

2. Use the code on Page 321 of the book to show how to use a `try-expect` statement in a method.

Teaching Tip	For more information on errors and exceptions in Python, read: http://docs.python.org/tut/node10.html.

Playing Cards

1. Use the sample code provided in the book to show the functionality of the `Card` class. Explain why there is no need to include any methods other than `__str__` for the string representation. Briefly step through the code of this class.

2. Use Table 8.7 and the sample code provided in the book to describe the functionality of the `Deck` class. Briefly step through the code of this class.

Case Study: An ATM

1. Guide students as they step through this section.

Request

1. Ask students to write a program that simulates a simple ATM.

Analysis

1. Use Figure 8.1 to show sample terminal-based interface for the program.

2. The *class diagram* in Figure 8.2 shows the relationships among the classes. Explain how to interpret such diagrams.

3. Introduce the *model/view* pattern and explain how it will be used in this program.

Design

1. Use Table 8.8 to describe the interface of the `ATM` class. Be sure to explain why the `run` method is the only method in this class that does not begin with the _ symbol.

Implementation (Coding)

1. Review the code in `atm.py` with students. Explain that the code defines the `ATM` class, instantiates a `Bank` and an `ATM`, and executes the ATM's `run` method. Explain how the `run` method is used to call other methods as requested by the user.

Quick Quiz 2

1. True or False: The code `x + y` is actually shorthand for the code `y.__add__(x)`.
   Answer: False

2. To overload the % arithmetic operator, you need to define a new method using ____________________ as the method name.
   Answer: `__mod__`

3. What is operator overloading?
   Answer: Operator overloading is another example of an abstraction mechanism. In this case, programmers can use operators with single, standard meanings even though the underlying operations vary from data type to data type.

4. What is a class variable?
   Answer: A class variable is visible to all instances of a class and does not vary from instance to instance. While it normally behaves like a constant, in some situations a class variable can be modified. But when it is, the change takes effect for the entire class.

Structuring Classes with Inheritance and Polymorphism

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

2. Stress that while inheritance and polymorphism are built into Python, its syntax does not enforce data encapsulation.

Teaching Tip	"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

1. Use Figure 8.3 to introduce the role of *inheritance hierarchies*.

2. Explain that, in Python, all classes automatically extend the built-in `object` class.

3. Explain how inheritance provides an abstraction mechanism that allows programmers to avoid writing redundant code.

Example: A Restricted Savings Account

1. Use the `RestrictedSavingsAccount` class example to show 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

1. Use Figure 8.4 to describe the role of the classes in the blackjack game application.

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

3. Use the `Dealer` class example to show how to extend an existing class while overloading some of the methods of the parent class. Step through the code of this class, explaining it to students.

4. Use the sample code provided in the book to show how the `Blackjack` class can be used to set up and manage the interactions with the user. Step through the code of this class, explaining it to students.

Polymorphic Methods

1. Explain that 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.

2. Describe 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

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

Teaching Tip	"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

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

2. Explain that 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.

Quick Quiz 3

1. _____________________ restricts the manipulation of an object's state by external users to a set of method calls.
   Answer: Data encapsulation

2. _____________________ allows a class to automatically reuse and extend the code of similar but more general classes.
   Answer: Inheritance

3. What is polymorphism?
   Answer: Polymorphism is an object-oriented technique that allows several different classes to use the same general method names.

4. True or False: In Python, all classes automatically extend the built-in `object` class.
   Answer: True

Class Discussion Topics

1. Are your students familiar with exceptions in C++ or Java? If so, ask them to compare the exception mechanisms provided by those languages to the one provided by Python (`try-except` statement).

2. Ask your students if they are familiar with other object-oriented languages. Do those languages provide support for data encapsulation? How?

Additional Projects

1. Ask students to do some research on multiple inheritance and Python's support for this object-oriented feature.

2. Some card games may need to use the Jokers in a deck of cards. Modify the `Card` class so that it supports Jokers. Modify the `Deck` class by adding a `hasJokers` method that can be used to test if the current deck has Jokers or not.

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.