1 Python Primer

2 Object-Oriented Programming

2.1 Goals, Principles, and Patterns

As the name implies, the main "actors" in the object-oriented paradigm are called **objects**. Each object is an **instance** of a **class**. Each class represents to the outside world a concise and consistent view of the objects that are instances of the class, without going into too much unnecessary detail or giving others access to the inner workings of the objects. The class definition typically specifies **instance variables**, also known as data numbers, that the object contains, as well as the methods, also known as number functions, that the object can execute. This view of computing is intended to fulfill several goals and incorporate several design principles, which we discuss in this chapter.

2.1.1 Object-Oriented Design Goals

Software implementations should achieve robustness, adaptability, and reusability. (see Figure 2-1.)

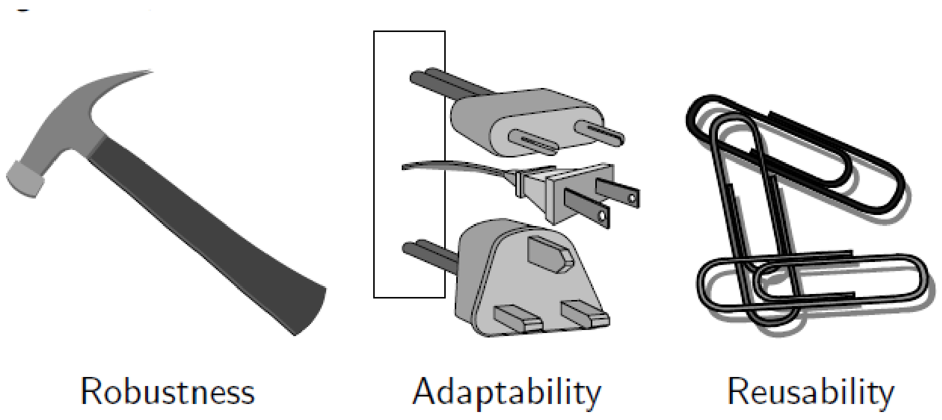


Figure 2-1 Goals of object-oriented design.

Robustness

Every good programmer wants to develop software that is correct, that means that a program produces the right output for all the anticipated inputs in the program's application. In addition, we want software to be **robust**, that is, capable of handling unexpected inputs that are not explicitly defined for its application. For example, if a program is expecting a positive integer, then the program should be able to recover graceful from this error. More importantly, in **life-critical applications**, where a software error can lead to injury or loss of life, software that is not robust could be deadly. This point was driven home in the late 1980s in accidents involving Therac-25, a radiation-therapy machine, which severely overdosed six patients between 1985 and 1987, some of whom died from complications resulting from their radiation overdose. All six accidents were traced to software errors.

Adaptability

Modern software applications, such as Web browsers and Internet search engines, typically involve large programs that are used for many years. Software, therefore, needs to be able to evolve overtime in response to changing conditions in its environment. Thus, another important goal of quality software is that it achieves **adaptability** (also called **evolvability**). Related to this concept is **portability**, which is the ability of software to run with minimal change on different hardware and operating system platforms. An advantage of writing software in Python is the portability provided by the language itself.

Reusability

Going hand in hand with adaptability is the desire that software be reusable, that is, the same code should be usable as a component of different systems in various applications. Developing quality software can be an expensive enterprise, and its cost can be offset somewhat if the software is designed in a way that makes it easily reusable in future applications. Such reuse should be done with care, however, for one of the major sources of software errors in the Therac-25 came from inappropriate reuse of Therac-20 software (which was not object-oriented and not designed for the hardware platform used with the Therac-25).

2.1.2 Object-Oriented Design Principles

Chief among the principles of the object-oriented approach, which are intended to facilitate the goals outlined above, are the following (see Figure 2-2)

i. Modularity
ii. Abstraction
iii. Encapsulation

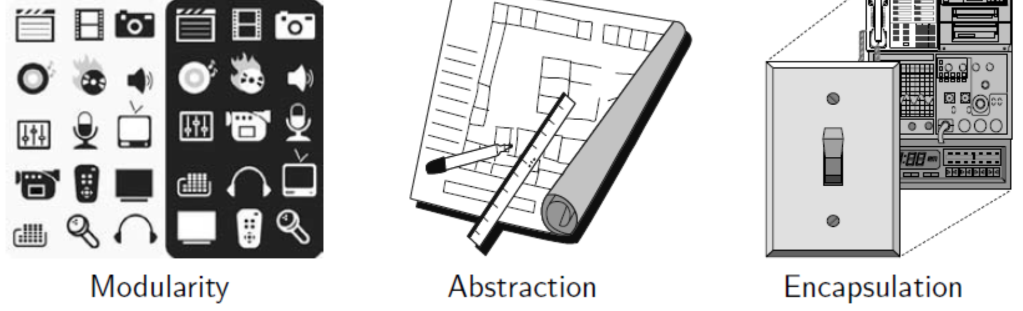


Figure 2-2 Principles of object-oriented design

Modularity

Modern software systems typically consist of several different components that must interact correctly in order for the entire system to work properly. Keeping these interactions straight requires that these different components be well organized. Modularity refers to an organizing principle in which different components of a software system are divided into separate functional units.

As a real-world analogy, a house or apartment can be viewed as consisting of several interacting units: electrical, heating and cooling, plumbing, and structural. Rather than viewing these systems as one giant jumble of wires, vents, pipes, and boards, the organized architect designing a house or apartment will view them as separate modules that interact in well-defined ways. In so doing, he or she is using modularity to bring a clarity of thought that provides a natural way of organizing functions into distinct manageable units.

In like manner, using modularity in a software system can also provide a powerful organizing framework that brings clarity to an implementation. In Python, we have already seen that a **module** is a collection of closely related functions and classes that are defined together in a single file of source code. Python's standard libraries include, for example, the math module, which provides definitions for key mathematical constants and functions, and the os module, which provides support for interacting with the operating system.

The use of modularity helps support the goal listed in Section 2.1.1. Robustness is greatly increased because it is easier to test and debug separate components before they are integrated into a larger software system. Furthermore, bugs that persist in a complete system might be traced to a particular component, which can be fixed in relative isolation. The structure imposed by modularity also helps enable software reusability. If software modules are written in a general way, the modules can be reused when related need arises in other contexts. This is particularly relevant in a study of data structure, which can typically be designed with sufficient abstraction and generality to be reused in many applications.

Abstraction

The notion of **abstraction** is to distill a complicated system down to its most fundamental parts. Typically, describing the parts of a system involves naming them and explaining their functionality. Applying the abstraction paradigm to the design of data structures gives rise to **abstract data types** (ADTs). An ADT is a mathematical model of a data structure that specifies the type of data sorted, the operations supported on them, and the types of parameters of the operations. An ADT specifies **what** each operation does, but not **how** it does it. We will typically refer to the collective set of behaviors supported by an ADT as its **public interface**.

As a programming language, Python provides a great deal of latitude in regard to the specification of an interface. Python has a tradition of treating abstractions implicitly using a mechanism known as **duck typing**. As an interpreted and dynamically typed language, there is no "compile time" checking of data types in Python, and no formal requirement for declarations of abstract base classes. Instead programmers assume that an object supports a set of known behaviors, with the interpreter raising a run-time error if those assumptions fail. The description of this as "dunk typing" comes from an adage attributed to p poet James

Whitcomb Riley, stating that "when I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

More formally, Python supports abstract data types using a mechanism known as an abstract base class (ABC). An abstract base class cannot be instantiated (i.e., you cannot directly create an instance of that class), but it defines one or more common methods that all implementations of the abstraction must have. An ABC is realized by one or more concrete classes that inherit from the abstract base class while providing implementations for those method declared by the ABC. Python's abc module provides formal support for ABCs, although we omit such declarations for simplicity. We will make use of several existing abstract base classed coming from Python's collections module, which includes definitions for several common data structure ADTs, and concrete implementations of some of those abstractions.

Encapsulation

Another important principle of object-oriented design is encapsulation. Different components of a software system should not reveal the internal details of their respective implementations. One of the main advantages of encapsulation is that it gives one programmer freedom to implement the details of a component, without concern that other programmers will be writing code that intricately depends on those internal decisions. The only constraint on the programmer of a component is to maintain the public interface for the component, as other programmers will be writing code that depends on that interface. Encapsulation yields robustness and adaptability, for it allows the implementation details of parts of a program to change without adversely affecting other parts, thereby making it easier to fix bugs or add new functionality with relatively local changes to a component.

Throughout this book, we will adhere to the principle of encapsulation, making clear which aspects of a data structure are assumed to be public and which are assumed to be internal details. With that said, Python provides only loose support for encapsulation. By convention, names of members of a class (both data members and member functions) that start with a single underscore character (e.g., _secret) are assumed to be nonpublic and should not be relied upon. Those conventions are reinforced by the intentional omission of those members from automatically generated documentation.

2.1.3 Design Patterns

Object-oriented design facilitates reusable, robust, and adaptable software. Designing good code takes more than simply understanding object-oriented methodologies, however. It requires the effective use of object-oriented design techniques.

Computing researchers and practitioners have developed a variety of organizational concepts and methodologies for designing quality object-oriented software that is concise, correct, and reusable. Of special relevance to this book is the concept of a **design pattern**, which describes a solution to a "typical" software design problem. A pattern provides a general template for a solution that can be applied in many different situations. It describes the main elements of a solution in an abstract way that can be specialized for a specific problem at hand. It consists of a name, which identifies the pattern; a context, which describes the scenarios for which this pattern can be applied; a template, which describes how the pattern is applied; and a result, which describes and analyzes what the pattern produces.

We present several design patterns in this book, and we show how they can be consistently applied to implementations of data structures and algorithms. These design patterns fall into two groups—patterns for solving algorithm design problems and patterns for solving software engineering problems. The algorithm design patterns we discuss include the following:

i. Recursion (Chapter 4)
ii. Amortization (Section 5.3 and 11.4)
iii. Divide-and-conquer (Section 12.2.1)
iv. Prune-and-search, also known as decrease-and-conquer (Section 12.7.1)
v. Brute force (Section 13.2.1)
vi. Dynamic programming (Section 13.3)
vii. The greedy method (Section 13.4.2, 14.6.2, and 14.7)

Likewise, the software engineering design patterns we discuss include:

i. Iterator (Section 1.8 and 2.3.4)
ii. Adapter (Section 6.1.2)
iii. Position (Section 7.4 and 8.1.2)
iv. Composition (Section 7.6.1, 9.2.1, and 10.1.4)
v. Template method (Section 2.4.3, 8.4.6, 10.1.3, 10.5.2, and 11.2.1)
vi. Locator (Section 9.5.1)
vii. Factory method (Section 11.2.1)

Rather than explain each of these concepts here, however, we introduce them throughout the text as noted above. For each pattern, be it for algorithm engineering or software engineering, we explain its general use and we illustrate it with at least one concept example.

2.2 Software Development

Traditional software development involves several phases. Three major steps are:

1. Design
2. Implementation
3. Testing and Debugging

In this section, we briefly discuss the role of these phases, and we introduce several good practices for programming in Python, including coding style, naming conventions, formal documentation, and unit testing.

2.2.1 Design

For object-oriented programming, the design step is perhaps the most important phase in the process of developing software. For it is in the design step that we decide how to divide the workings of our program into classes, we decide how these classes will interact, what data each will store, and what actions each will perform. Indeed, one of the main challenges that beginning programmers face is deciding what classes to define to do the work of their program. While general prescriptions are hard to come by, there are some rules of thumb that we can apply when determining how to design our classes:

i. **Responsibility**: Divide the work into different **actors**, each with a different responsibility. Try to describe responsibilities using action verbs. These actors will form the classes for the program.
ii. **Independence**: Define the work for each class to be as independent from other classes as possible. Subdivide responsibilities between classes so that each class has autonomy over some aspect of the program. Given data (as instance variables) to the class that has jurisdiction over the actions that require access to this data.
iii. **Behaviors**: Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it. These behaviors will define the methods that this class performs, and the set of behaviors for a class are the **interface** to the class, as these form the means for other pieces of code to interact with objects from the class.

Defining the classes, together with their instance variables and methods, are key to the design of an object-oriented program. A good programmer will naturally develop greater skill in performing these tasks over time, as experience teaches him or her notice patterns in the requirements of a program that match patterns that he or she has seen before.

A common tool for developing an initial high-level design for a project is the use of CRC cards. Class-Responsibility-Collaborator (CRC) cards are simple index cards that subdivide the work required of a program. The main idea behind this tool is to have each card represent a component, which will ultimately become a class in the program. We write the name of each component on the top of an index card. On the left-hand side of the card, we begin writing the responsibilities for this component, that is, the other components that this component will have to interact with to perform its duties.

The design process iterates through an action/actor cycle, where we first identify an action (that is, a responsibility), and we then determine an actor (that is, a component) that is best suited to perform that action. The design is complete when we have assigned all actions to actors. In using index cards for this process (rather than larger pieces of paper), we are relying on the fact that each component should have a small set of responsibilities and collaborators. Enforcing this rule helps keep the individual classes manageable.

As the design takes form, a standard approach to explain and document the design is the use of UML (Unified Modeling Language) diagrams to express the organization of a program. UML diagrams are a standard visual notation to express object-oriented software designs. Several computer-aided tools are available to build UML diagrams. One type of UML figure if known as a **class diagram**. An example of such a diagram is given in Figure 2-3, for a class that represents a consumer credit card. The diagram has three portions, with the first designating the name of the class, the second designating the recommended instance variables, and the third designating the recommended methods of the class. In section 2.2.3, we discuss our naming conventions, and in in Section 2.3.1, we provide a complete implementation of a Python CreditCard class based on this design.

Class:		**CreditCard**
Fields:	_customer _bank _account	_balance _limit

Behaviors:	Get_customer()
	Get_bank()
	Get_account()
	Make_payment(amount)
		Get_balance()
		Get_limit()
		Charge(price)

Figure 2-3 Class diagram for a proposed CreditCard class

2.2.2 Pseudo-Code

As an intermediate step before the implementation of a design, programmers are often asked to describe algorithms in a way that is intended for human eyes only. Such descriptions are called pseudo-code. Pseudo-code is not a computer program, but is more structured that usual prose. It is a mixture of natural language and high-level programming constructs that describe the main ideas behind a generic implementation of a data structure of algorithm. Because pseudo-code id designed for a human reader, not a computer, we can communicate high-level ideas, without being burdened with low-level implementation details. At the same time, we should not gloss over important skill that is refined through practice.

In this book, we rely on a pseudo-code style that we hope will be evident to Python programmers, yet with a mix of mathematical notations and English prose. For example, we might use the phrase "indicate an error" rather than a formal raise statement. Following conventions of Python, we rely on indentation to indicate the extent of control structures and on an indexing notation in which entries of a sequence A with length n are indexed from A[0] to A[n-1]. However, we choose to enclose comments within curly brace { like these } in our pseudo-code, rather than using Python's # character.

2.2.3 Coding Style and Documentation

Programs should be made easy to read and understand. Good programmers should therefore be mindful of their coding style, and develop a style that communicates the important aspects of a program's design for both humans and computers. Conventions for coding style tend to vary between different programming communities. The official *Style Guide for Python Code* is available online at

http://www.python.org/dev/peps/pep-0008/

The main principles that we adopt are as follows:

1. Python code blocks are typically indented by 4 spaces. However, to avoid having our code fragments overrun the book's margins, we use 2 spaces for each level of indentation. It is strongly recommended that tabs be avoided, as tabs are displayed with different widths across systems, and tabs and spaces are not viewed as identical by the Python interpreter. Many Python-aware editors will automatically replace tabs with an appropriate number of spaces.
2. Use meaningful names for identifiers. Try to choose names that can be read aloud, and choose names that reflect the action, responsibility, or data each identifier is naming.

    1) Classes (other than Python's built-in classes) should have a name that serves as a singular noun, and should be capitalized (e.g., Data rather that data or Dates). When multiple words are concatenated to form a class name, they should follow the so-called "CamelCase" convention in which the first letter of each word is capitalized (e.g., CreditCard).

    2) Functions, including member functions of a class, should be lowercase. If multiple words are combined, they should be separated by underscores (e.g., make_payment).

    3) Names that identify an individual object (e.g., a parameter, instance variable, or local variable) should be a lowercase noun (e.g., price). Occasionally, we stray from this rule when using a single uppercase letter to designate the name of a data structure (such as tree T).

    4) Identifiers that represent a value considered to be a constant are traditionally identified using all capital letters and with underscores to separate words (e.g., MAX_SIZE).

    Recall from our discussion of **encapsulation** that identifiers in any context that begin with a single leading underscore (e.g., _secret) are intended to suggest that they are only for "internal" use to a class or module, and not part of a public interface.

3. Use comments that add meaning to a program and explain ambiguous or confusing constructs. In-line comments are good for quick explanations; they are indicated in Python following the # character, as in

```
        if n % 2 == 1:      # n is odd
```

Multiple block comments are good for explaining more complex code sections. In Python, these technically multiline string literals, typically delimited with triple quotes ("""), which have no effect when executed. In the next section, we discuss the use of block comments for documentation.

Documentation

Python provides integrated support for embedding formal documentation directly in source code using a mechanism known as a **docstring**. Formally, any string literal that appears as the first statement within the body of a module, class, or function (including a member function of a class) will be considered to be a docstring. By convention, those string literals should be delimited within triple quote ("""). As an example, our version of the scale function from Section 1.5.1-Mutable Parameters could be documented as follows:

```
def scale(data, factor):
    """Multiply all entries of numeric data list by the given factor."""
    for j in range(len(data)):
        data[j] *= factor
```

It is common to use the triple-quoted string delimiter for a docstring, even when the string fits on a single line, as in the above example. More detailed docstrings should begin with a single line that summaries the purpose, followed by a blank line, and then further details. For example, we might more clearly document the scale function as follows:

```
def scale(data, factor):
    """Multiply all entries of numeric data list by the given factor.
    data      an instance of any mutable sequence type (such as a list)
        containing numeric elements.

    factor    a number that serves as the multiplicative factor for scaling
    """
    for j in range(len(data)):
        data[j] *= factor
```

A docstring is sorted as a field of the module, function, or class in which it is declared. It serves as documentation and can be retrieved in a variety of ways. For example, the command help(x), within the Python interpreter, produces the documentation associated with the identified object x. An external tool named pydoc is distributed with Python and can used to generate formal documentation as text or as a Web page. Guidelines for authoring useful docstrings are available at:

http://www.python.org/dev/peps/pep-0257/

In this book, we will try to present docstrings when space allows. Omitted docstrings can be found in the online version of our source code.

2.2.4 Testing and Debugging

Testing is the process of experimentally checking the correctness of a program, while debugging is the process of tracking the execution of a program and discovering the error in it. Testing and debugging are often the most time consuming activity in the development of a program.

Testing

A careful testing plan is an essential part of writing a program. While verifying the correctness of a program over all possible inputs is usually infeasible, we should aim at executing the program on a representative suspect of inputs. At the very minimum, we should make sure that every method of a class is tested at least once (method coverage). Even better, each code statement in the program should be executed at least once (statement coverage).

Programs often tend to fail on special cases of the input. Such cases need to be carefully identified and tested. For example, when testing a method that sorts (that is, puts in order) a sequence of integers, we should consider the following inputs:

i. The sequence has zero length (no element).
ii. The sequence has one element.
iii. All the elements of the sequence are the same.
iv. The sequence is already sorted.
v. The sequence is reverse sorted.

In addition to special inputs to the program, we should also consider special condition for the structures used by the program. For example, if we can use a Python list to store data, we should make sure that the boundary cases, such as inserting or removing at the beginning or end of the list, and properly handled.

While it is essential to use handcrafted test suites, it is also advantageous to run the program on a large collection of randomly generally inputs. The random module in Python provides several means for generating random numbers, or for randomizing the order of collections.

The dependencies among the classes and functions of a program induce a hierarchy. Namely, a component A is above a component B in the hierarchy if A depends upon B, such as when function A calls function B, or function A relies on a parameter that is an instance of class B. There are two main testing strategies, **top-down** and **bottom-up**, which differ in the order in which components are tested.

Top-down testing proceeds from the top to the bottom of the program hierarchy. It is typically used in conjunction with **stubbing**, a boot-strapping technique that replaces a lower-level component with a **stub**, a replacement for the component that simulates the functionality of the original. For example, if function A calls function B to get the first line of a file, when testing A we can replace B with a stub that returns a fixed string.

Bottom-up testing proceeds from lower-level components to higher-level component. For example, bottom-level functions, which do not invoke other functions, are tested first, followed by functions that call only bottom-level functions, and so on. Similarly a class that does not depend upon any other classes can be tested before another class that depends on the former. This form of testing is usually described as **unit testing**, as the functionality of a specific component is tested in isolation of the larger software project. If used properly, this strategy better isolates the cause of errors to the component being tested, as lower-level components upon which it relies should have already been thoroughly tested.

Python provides several forms of support for automated testing. When functions or classes are defined in a module, testing for that module can be embedded in the same file. The mechanism for doing so was described in Section 1.11. Code that is shielded in a conditional construct of the form:

```
if __name__ == '__main__':
    # perform tests···
```

Will be executed when Python is invoked directly on that module, but not when the module is imported for use in a larger software project. It is common to put tests in such a construct to test the functionality of the functions and classes specifically defined in that module.

More robust support for automation of unit testing is provided by Python's unittest module. This framework allows the grouping of individual test cases into larger test suites, and provides support for executing those suites, and reporting or analyzing the results of those tests. As software is maintained, the act of **regression testing** is used, whereby all previous tests are re-executed to ensure that changes to the software do not introduce new bugs in previously tested components.

Debugging

The simplest debugging technique consists of using **print statements** to track the values of variables during the execution of the program. A problem with this approach is that eventually the print statements need to be removed or commented out, so they are not executed when the software is finally released.

A better approach is to run the program with in a debugger, which is a specialized environment for controlling and monitoring the execution of a program. The basic functionality provided by a debugger is the insertion of **breakpoints** within the code. When the program is executed within the debugger, it stops at each breakpoint. While the program is stopped, the current value of variables can be inspected.

The standard Python distribution includes a module named phd, which provides debugging support directly within the interpreter. Most IDEs for Python, such as IDLE, provide debugging environments with graphical user interfaces.

2.3 Class Definitions

A class serves as the primary means for abstraction in object-oriented programming. In Python, every piece of data is represented as an instance of some class. A class provides a set of behaviors in the form of **member functions** (also known as **methods**), with implementations that are common to all instances of that class. A class also serves as a blueprint for its instances, effectively determining the way that state information for each instance is represented in the form of **attributes** (also known as **fields**, **instance variables**, or **data members**).

2.3.1 Example: CreditCard Class

As a first example, we provide an implementation of a CreditCard class based on the design we introduced in Figure 2-3 of Section 2.2.1. The instances defined by the CreditCard class provide a simple model for traditional credit cards. They have identifying information about the customer, bank, account number, credit limit, and current balance. The class restricts charges that would cause a card's balance to go over its spending limit, but it does not charge interest or late payments (we revisit such themes in Section 2.4.1).

Our code begins in Code Fragment 2.1 and continues in Code Fragment 2.2. The construct begins with the keyword, class, followed by the name of the class, a colon, and then an indented block of code that serves as the body of the class. The body includes definitions for all methods of the class. These methods are defined as functions, using techniques introduced in Section 1.5, yet with a special parameter, named self, that serves to identify the particular instance upon which a member is invoked.

The self identifier

In Python, the self identifier plays a key role. In the context of the CreditCard class, there can presumably be many different CreditCard instances, and each must maintain its own balance, its own credit limit, and so on. Therefore, each instance stores its own instance variables to reflect its current state.

Syntactically, self identifiers the instance upon which a method is invoked. For example, assume that a user of our class has a variable, my_card, that identifies an instance of the CreditCard class. When the user calls my_card.get_balance(), identifier self, within the definition of the get_balance method, refers to the card known as my_card by the callser. The expression, self._balance refers to an instance variable, named _balance, stored as part of that particular credit card's state.

```
class CreditCard:
    """A consumer credit card."""

    def __init__(self, customer, bank, acnt, limit):
        """Create a new credit card instance.

        The initial balance is zero.

        customer    the name of the customer (e.g.,'John Bowman')
        bank        the name of the bank (e.g., 'California Saving')
        acnt        the acount identifier (e.g., '5391 0375 9387 5309')
        limit       credit limit (measured in dollars)
        """
        self._coustomer = customer
        self._bank = bank
        self._account = acnt
        self._limit = limit
        self._balance = 0

    def get_customer(self):
        """Return name of the customer."""
        return self._customer

    def get_bank(self):
        """Return the bank's name"""
        return self._bank

    def get_account(self):
        """Return the card identifying number (typically sorted as a string)"""
        return self._account

    def get_limit(self):
        """Return current credit limit."""
        return self._limit

    def get_balance(self):
        """Return current balance."""
        return self._balance
```

Code Fragment 2.1 The beginning of the CreditCard class definition (continued in Code Fragment 2.2)

```
    def charge(self, price):
        """Charge given price to the card, assuming sufficient credit limit.

        Return True if charge was processed; False if charge was denied.
        """

        if price + self._balance > self._limit:
            return False
        else:
            self._balance += price
            return True

    def make_payment(self, amount):
        """Process customer payment that reduces balance."""
        self._balance -= amount
```

Code Fragment 2.2 The conclusion of the CreditCard class definition (continued from Code Fragment 2.1). These methods are indented within the class definition.

We draw attention to the difference between the method signature as declared within the class versus that used by a caller. For example, from a user's perspective we have seen that the get_balance method takes zero parameters, yet within the class definition, self is an explicit parameter (self and price), even though this method is called with one parameter, for example, as my_card.charge(200). The interpreter automatically binds the instance upon which the method is invoked to the self parameter.

The Constructor

We user can create an instance of the CreditCard class using a syntax as:

cc = CreditCard('John Doe','1st bank','5491 0375 9387 5309','1000')

Internally, this results in a call to the specially named __init__ method that serves as the **constructor** of the class. Its primary responsibility is to establish the state of a newly created credit card object with appropriate instance variables. In the case of the CreditCard class, each object maintains five instance variables, which we name: _customer, _bank, _account, _limit, and _balance. The initial values for the first four of those five are provided as explicit parameters that are sent by the user when instantiating the credit card, and assigned within the body of the constructor. For example, the command, self._customer = customer, assigns the instance variable self._customer to the parameter customer; note that because customer is **unqualified** on the right-hand side, it refers to the parameter in the local namespace.

Encapsulation

By the conventions described in Section 2.2.3, a single leading underscore in the name of a data member, such as _balance, implies that it is intended as nonpublic. Users of a class should not directly access such members.

As a general rule, we will treat all data members as nonpublic. This allows us to better enforce a consistent state for all instances. We can provide accessors, such as get_balance, to provide a user of our class read-only access to a trait. If we wish to allow the user to change the state, we can provide appropriate update methods. In the context of data structures, encapsulating the internal representation allows us greater flexibility to redesign the way a class works, perhaps to improve the efficiency of the structure.

Additional Methods

The most interesting behaviors in our class are charge and make_payment. The charge function typically adds the given price to the credit card balance, to reflect a purchase of said price by the customer. However, before accepting the charge, our implementation verifies that new purchase would not cause the balance to exceed the credit limit. The make_payment charge reflects the customer sending payment to the bank for the given amount, thereby reducing the balance on the card. We note that in the command, self._balance -= amount, the expression self._balance is qualified with the self identifier because it represents an instance variable of the card, while the unqualified amount represents the local parameter.

Error Checking

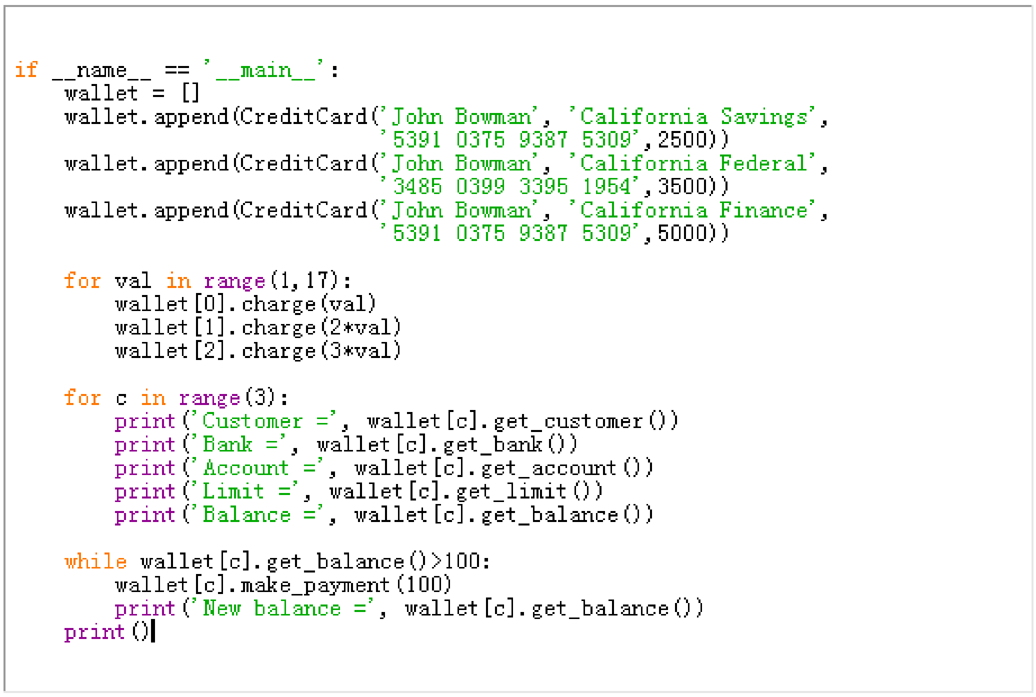
Our implementation of the CreditCard class is not particularly robust. First, we note that we did not explicitly check the types of the parameters to charge and make_payment, nor any of the parameters to the constructor. If a user were to make a call such as visa.charge('candy'), our code would presumably crash when attempting to add that parameter to the current balance. If this class were to be widely used in a library, we might use more rigorous techniques to raise a TypeError when facing such misuse (see Section 1.7).

Beyond the obvious type errors, our implementation may be susceptible to logical errors. For example, if a user were allowed to charge a negative price, such as visa.charge(-300), that would serve to lower the customer's balance. This provides a loophole for lowering a balance without making a payment. Of course, this might be considered valid usage if modeling the credit received when a customer returns merchandise to a store. We will explore some such issues with the CreditCard class in the end-of-chapter exercises.

Testing the Class

In Code Fragment 2.3, we demonstrate some basic usage of the CreditCard class, inserting three cards into a list named wallet. We use loops to make some charges and payments, and use various accessors to print results to the console.

These tests are enclosed within a condition, if __name__ == '__main__:', so that they can be embedded in the source code with the class definition. Using the terminology of Section 2.2.4, these tests provide **method coverage**, as each of the methods is called at least once, but it does not provide **statement coverage**, as there is never a case in which a charge is rejected due to the credit limit. This is not a particular advanced from of testing as the output of the given tests must be manually audited in order to determine whether the class behaved as expected. Python has tools for more formal testing (see discussion of the unittest module in Section 2.2.4), so that resulting values can be automatically compared to the predicted outcomes, with output generated only when an error is detected.



```
if __name__ == '__main__':
    wallet = []
    wallet.append(CreditCard('John Bowman', 'California Savings',
                             '5391 0375 9387 5309',2500))
    wallet.append(CreditCard('John Bowman', 'California Federal',
                             '3485 0399 3395 1954',3500))
    wallet.append(CreditCard('John Bowman', 'California Finance',
                             '5391 0375 9387 5309',5000))

    for val in range(1,17):
        wallet[0].charge(val)
        wallet[1].charge(2*val)
        wallet[2].charge(3*val)

    for c in range(3):
        print('Customer =', wallet[c].get_customer())
        print('Bank =', wallet[c].get_bank())
        print('Account =', wallet[c].get_account())
        print('Limit =', wallet[c].get_limit())
        print('Balance =', wallet[c].get_balance())

    while wallet[c].get_balance()>100:
        wallet[c].make_payment(100)
        print('New balance =', wallet[c].get_balance())
    print()
```

Code Fragment 2.3 Testing the CreditCard class

2.3.2 Operator Overloading and Python's Special Methods

Python's built-in classes provide natural semantics for many operators. For example, the syntax a + b invokes addition for numeric types, yet concatenation for sequence types. When defining a new class, we must consider whether a syntax like a + b should be defined when a or b is an instance of that class.

By default, the + operator is undefined for a new class. However, the author of a class may provide a definition using a technique known as **operator overloading**. This is done by implementing a specially named method. In particular, the + operator is overloaded by implementing a method named __add__, which takes the right-hand operand as a parameter and which returns the result of the expression. That is, the syntax, a + b, is converted to a method call on object a of the form, a.__add__(b). Similar specially named methods exist for other operators. Table 2-1 provides a comprehensive list of such methods.

When a binary operator is applied to two instances of different types, as in 3 * 'love me', Python gives deference to the class provides a sufficient definition for how to multiply an instance by a string, via the __mul__ method. However, if that class does not implement such a behavior, Python checks the class definition for the right-hand operand, in the form of a special method named __rmul__ (i.e., 'right multiply'). This provides a way for a new user-defined class to support mixed operations that involve an instance of an existing class (given that the existing class would presumably not have defined a behavior involving this new class). The distinction between __mul__ and __rmul__ also allows a class to define different semantics in cases, such as matrix multiplication, in which an operation is noncommutative (that is, A*x may differ from x * A).

Non-Operator Overloads

In addition to traditional operator overloading, Python relies on specially named methods to control the behavior of various other functionality, when applied to user-defined classes. For example, the syntax, str(foo), is formally a call to the constructor for the string class. Of course, if the parameter is an instance of a user-defined class, the original authors of the string class could not have known how that instance should be portrayed. So the sting constructor calls a specially named method, foo.__str__(), that must return an appropriate string representation.

Similar special methods are used to determine how to construct an int, float, or bool based on a parameter from a user-defined class. The conversion to a Boolean value is particularly important, because the syntax, if foo:, can be used even when foo is not formally a Boolean value (see Section 1.4.1). For a user-defined class, that condition is evaluated by the special method foo.__bool__().

Several other top-level functions rely on calling specially named methods. For example, the standard way to determine the size of a container type is by calling the top-level len function. Note well that the calling syntax, len(foo), is not the traditional method-calling syntax with the dot operator. However, in the case of a user defined class, the top-level len function relies on a specially named __len__ method of that class. That is, the call len(foo) is evaluated through a method call, foo.__len__(). When developing data structures, we will routinely define the __len__ method to return a measure of the size of the structure.

Table 2-1 Overloaded operations, implemented with Python's special methods.

Common Syntax	**Special Method Form**
a + b	a.__add__(b); alternatively b.__radd__(a)
a – b	a.__sub__(b); alternatevely b.__rsub__(a)
a * b	a.__mul__(b); alternatively b.__rmul__(a)
a / b	a.__truediv__(b); alternatively b.__rtruediv__(a)
a // b	a.__floordiv__(b); alternatively b.__rfloordiv(a)
a % b	a.__mod__(b); alternatively b.__rmoddiv__(a)
a ** b	a.__pow__(b); alternatively b.__rpow__(a)
a << b	a.__lshift__(b); alternatively b.__rlshift__(a)
a >> b	a.__rshift__(b); alternatively b.__rrshift__(a)
a & b	a.__add__(b); alternatively b.__and__(a)
a ^ b	a.__xor__(b); alternatively b.__rxor__(a)
a | b	a.__or__(b); alternatively b.__ror__(a)

a += b
a -= b
a *= b
…
	a.__iadd__(b)
	a.__isub__(b)
	a.__imul__(b)
	…

+a	a.__pos__()
-a	a.__neg__()
~a	a.__invert__()
abs(a)	a.__abs__()
a < b	a.__lt__(b)
a <= b	a.__le__(b)
a > b	a.__gt__(b)
a >= b	a.__ge__(b)
a == b	a.__eq__(b)
a != b	a.__ne__(b)
v in a	a.__contains__(v)
a[k]	a.__getitem__(k)
a[k] = v	a.setitem(k,v)
del a[k]	a.__delitem__(k)
a(arg1, arg2)	a.__call__(arg1, arg2, …)
len(a)	a.__len__()
hash(a)	a.__hash__()
iter(a)	a.__iter__()
next(a)	a.__next__()
bool(a)	a.__bool__()
float(a)	a.__float__()
int(a)	a.__int__()
repr(a)	a.__repr__()
reversed(a)	a.__reversed__()
str(a)	a.__str__()

Implied Methods

As a general rule, if a particular special method is not implemented in a user-defined class, the standard syntax that relies upon that method will raise an exception. For example, evaluating the expression, a + b, for instances of a user-defined class without __add__ or __radd__ will raise an error.

However, there are some operators that default definitions provided by Python, in the absence of special methods, and there are some operators whose definitions are derived from others. For example, the __bool__ method, which supports the syntax if foo:, has default semantics so that every object other than None is evaluated as True. However, for container types, the __len__ method is typically defined to return the size of the container. If such a method exists, then the evaluation of bool(foo) is interpreted by default to be True for instances with nonzero length, and False for instances with zero length, allowing a syntax such as if waitlist: to be used to test whether there are one or more entries in the waitlist.

In Section 2.3.4, we will discuss Python's mechanism for providing iterators for collections via the special method, __iter__. With that said, if a container class provides implementations for both __len__ and __getitem__, a default iteration is provided automatically (using means we describe in Section 2.3.4). Furthermore, once an iterator is defined, default functionality of __cointains__ is provided.

In Section 1.3 we drew attention to the distinction between expression a is b and expression a==b, with the former evaluating whether identifiers a and b are aliases for the same object, and the later testing a notion of whether the two identifiers reference equivalent values. The notion of "equivalence" depends upon the context of the class, and semantics is defined with the __eq__ method. However, if no implementation is given for __eq__, the syntax a==b is legal with semantics of a is b, that is, an instance is equivalent to itself and no others.

We should caution that some natural implications are not automatically provided by Python. For example, the __eq__ method supports the syntax a==b, but providing that method does not affect the evaluation of syntax a!=b. (The __ne__ method should be provided, typically returning not (a==b) as a result.) Similarly, providing a __lt__ method supports syntax a<b, and indirectly b>a, but providing both __lt__ and __eq__ does not imply semantics for a<=b.

2.3.3 Example: Multidimensional Vector Class

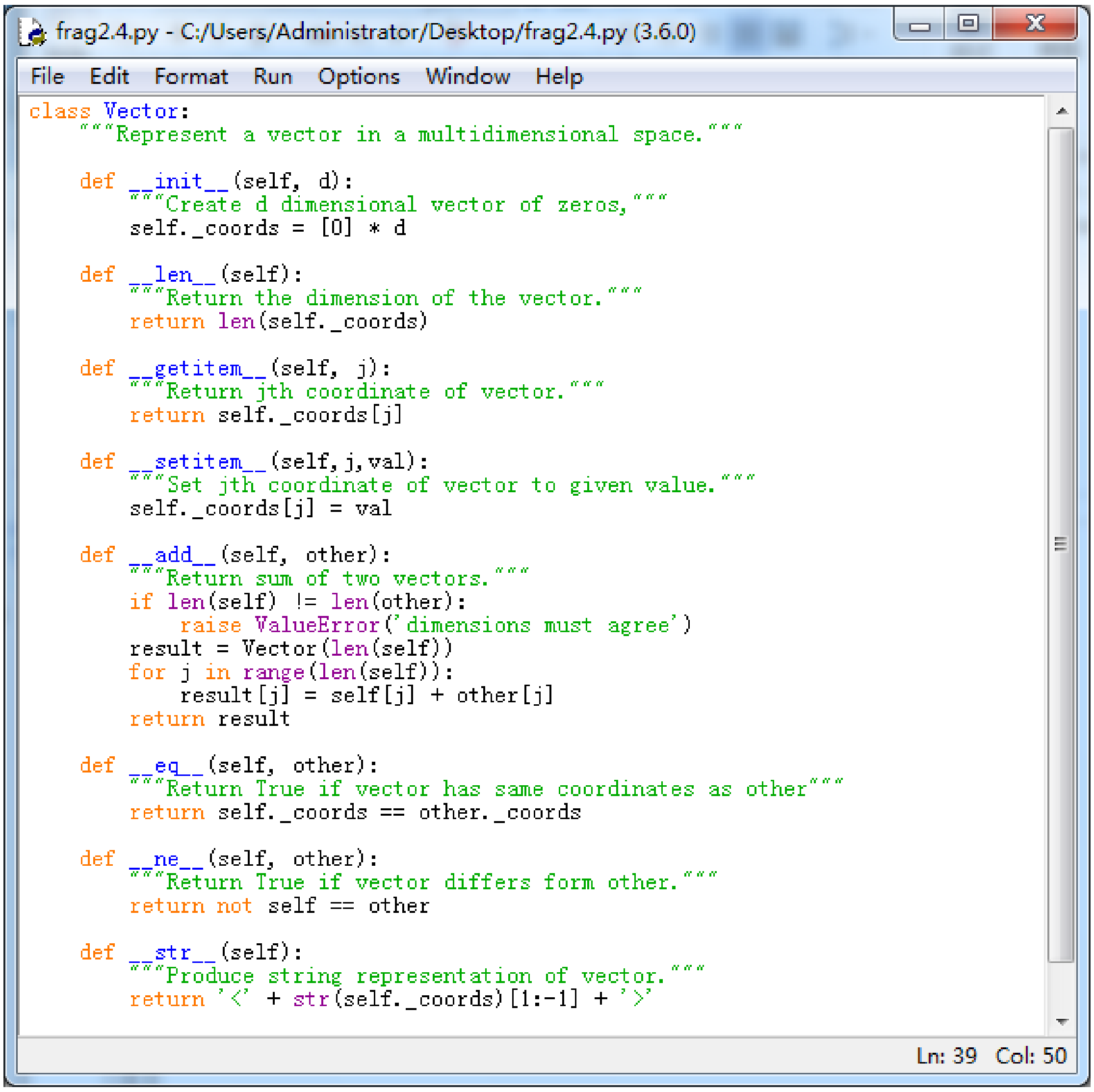
To demonstrate the use of operator overloading via special methods, we provide an implementation of a Vector class, representing the coordinates of a vector in a multidimensional space. For example, in a three-dimensional space, we might wish to represent a vector with coordinates <5, -2, 3>. Although it might be tempting to directly use a Python list to represent those coordinates, a list does not provide an appropriate abstraction for a geometric vector. In particular, if using lists, the expression [5, -2, 3] + [1, 4, 2] results in the list [5, -2, 3, 1, 4, 2]. When working with the vectors, if u = <5, -2, 3> and v = <1, 4, 2>, one would expect the expression, u + v, to return a three-dimensional vector with coordinates <6, 2, 5>.

We therefore define a Vector class, in Code Fragment 2.4, that provides a better abstraction for the notion of a geometric vector. Internally, our vector relies upon an instance of a list, named _coords, as its storage mechanism. By keeping the internal list encapsulated, we can enforce the desired public interface for instances of our class. A demonstration of supported behaviors includes the following:

```
>>> v = Vector(5)
>>> v[1] = 23
>>> v[-1] = 45
>>> print(v[4])
45
>>> u = v + v
>>> print(u)
<0, 46, 0, 0, 90>
>>> total = 0
for entry in v:
    total += entry

>>> total
68
>>>
>>> u = v + [5, 3, 10, -2, 1]
>>> u
<__main__.Vector object at 0x022D5690>
>>> print(u)
<5, 26, 10, -2, 46>
>>>
```

We implement many of the behaviors by trivially invoking a similar behavior on the underlying list of coordinates. However, our implementation of __add__ is customized. Assuming the two operands are vectors with the same length, this method creates a new vector and sets the coordinates of the new vector to be equal to the respective sum of the operands' elements.

It is interesting to note that the class definition, as given in Code Fragment 2.4, automatically supports the syntax u = v + [5, 3, 10, -2, 1], representing in a new vector that is the element-by-element "sum" of the first vector and the list instance. This is a result of Python's **polymorphism**. Literally, "polymorphism" means "many forms." Although it is tempting to think of the other parameter of our __add__ method as another Vector instance, we never declared it as such. Within the body, the only behaviors we rely on for parameter other is that it supports len(other) and access to other[j]. Therefore, our code executes when the right-hand operand is a list of numbers (with matching length).


frag2.4.py - C:/Users/Administrator/Desktop/frag2.4.py (3.6.0)
File Edit Format Run Options Window Help
Ln: 39 Col: 50

```
class Vector:
    """Represent a vector in a multidimensional space."""

    def __init__(self, d):
        """Create d dimensional vector of zeros,"""
        self._coords = [0] * d

    def __len__(self):
        """Return the dimension of the vector."""
        return len(self._coords)

    def __getitem__(self, j):
        """Return jth coordinate of vector."""
        return self._coords[j]

    def __setitem__(self,j,val):
        """Set jth coordinate of vector to given value."""
        self._coords[j] = val

    def __add__(self, other):
        """Return sum of two vectors."""
        if len(self) != len(other):
            raise ValueError('dimensions must agree')
        result = Vector(len(self))
        for j in range(len(self)):
            result[j] = self[j] + other[j]
        return result

    def __eq__(self, other):
        """Return True if vector has same coordinates as other"""
        return self._coords == other._coords

    def __ne__(self, other):
        """Return True if vector differs form other."""
        return not self == other

    def __str__(self):
        """Produce string representation of vector."""
        return '<' + str(self._coords)[1:-1] + '>'
```

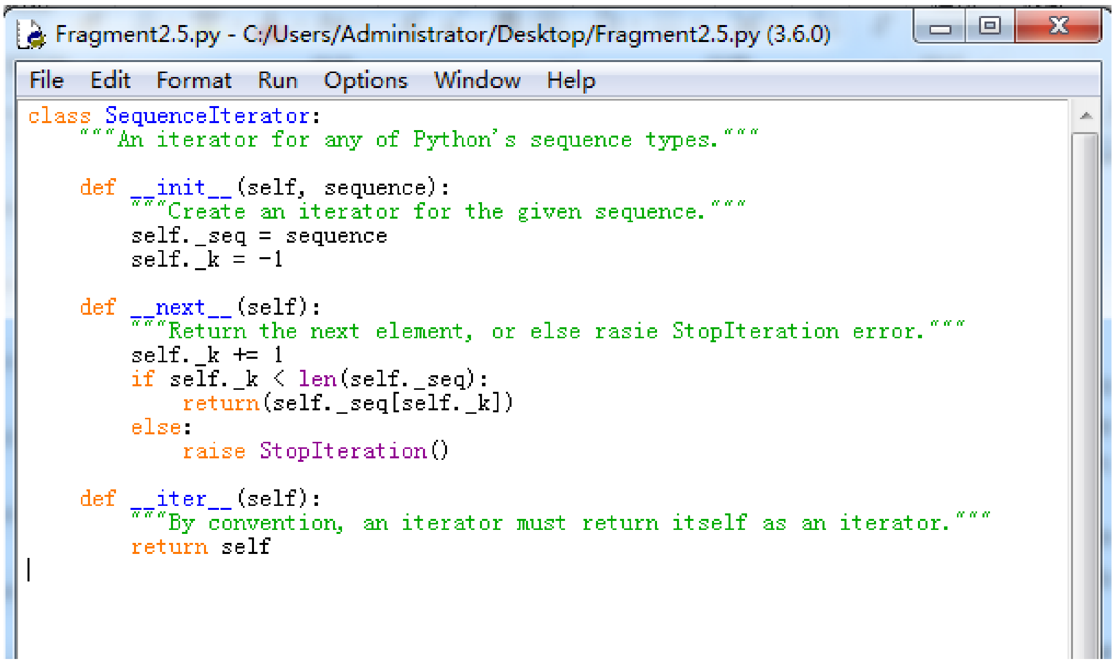
Code Fragment 2.4 Definition of a simple Vector class

2.3.4 Iterators

Iteration is an important concept in the design of data structures. We introduced Python's mechanism for iteration in Section1.8. In short, an **iterator** for a collection provides one key behavior: It supports a special method __next__ that returns the next element of the collection, if any, or raises a StopItration exception to indicate that there are no further elements.

Fortunately, it is rare to have directly implement an iterator class. Our preferred approach is the use of the **generator** syntax (also described in Section 1.8), which automatically produces an iterator of yielded values.

Python also helps by providing an automatic iterator implementation for any class that defines both __len__ and __getitem__. To provide an instructive example of a low-level iterator, Code Fragment 2.5 demonstrates just such an iterator class that works on any collection that supports both __len__ and __getitem__. This class can be instantiated as SequenceIterator(data). It operates by keeping an internal reference to the data sequence, as well as a current index into the sequence. Each time __next__ is called, the index is incremented, until reaching the end of the sequence.


Fragment2.5.py - C:/Users/Administrator/Desktop/Fragment2.5.py (3.6.0)
File Edit Format Run Options Window Help

```
class SequenceIterator:
    """An iterator for any of Python's sequence types."""

    def __init__(self, sequence):
        """Create an iterator for the given sequence."""
        self._seq = sequence
        self._k = -1

    def __next__(self):
        """Return the next element, or else rasie StopIteration error."""
        self._k += 1
        if self._k < len(self._seq):
            return(self._seq[self._k])
        else:
            raise StopIteration()

    def __iter__(self):
        """By convention, an iterator must return itself as an iterator."""
        return self
```

Code Fragment 2.5 An iterator class for any sequence type.

2.3.5 Example: Range Class

As the final example for this section, we develop our own implementation of a class that mimics Python's built-in range class. Before introducing our class, we discuss the history of the built-in version. Prior to Python being released, range(2,10,2) returned the list [2,4,6,8]. However, a typical use of the function was to support a for-loop syntax, such as for k in range(10000000). Unfortunately, this caused the instantiation and initialization of a list with the range of numbers. That was an unnecessarily expensive step, in terms of both time and memory usage.

This mechanism used to support ranges in Python 3 is entirely different (to be fair, the "new" behavior existed in Python 2 under the name xrange). It uses a strategy known as **lazy evaluation**. Rather than creating a new list instance, range is a class that can effectively represent the desired range of elements without ever storing them explicitly in memory. To better explore the built-in range class, we recommend that you create an instance as r = range(8, 140, 5). The result is a relatively lightweight object, an instance of the range class, that has only a few behaviors. The syntax len(r) will report the number of elements that are in the given range(27, in our example). A range also supports the __getitem__ method, so that syntax r[15] reports the sixteenth element in the range (as r[0] is the first element). Because the class supports both __len__ and __getitem__, it inherits automatic support for iteration (see Section 2.3.4), which is why it is possible to execute a for loop over a range.

At this point, we are ready to demonstrate our own version of such a class. Code Fragment 2.6 provides a class we name Range (so as to clearly differentiate it from built-in range). The biggest challenge in the implementation is properly computing the number of elements that belong in the range, given the parameters sent by the caller when constructing a range. By computing that value in the constructor, and storing it as self._length, it becomes trivial to return it from the __len__ method. To properly implement a call to __getitem__(k), we simply take the starting value of the range plus k times the step size (i.e., for k=0, we return the start value). There are a few subtleties worth examining in the code:

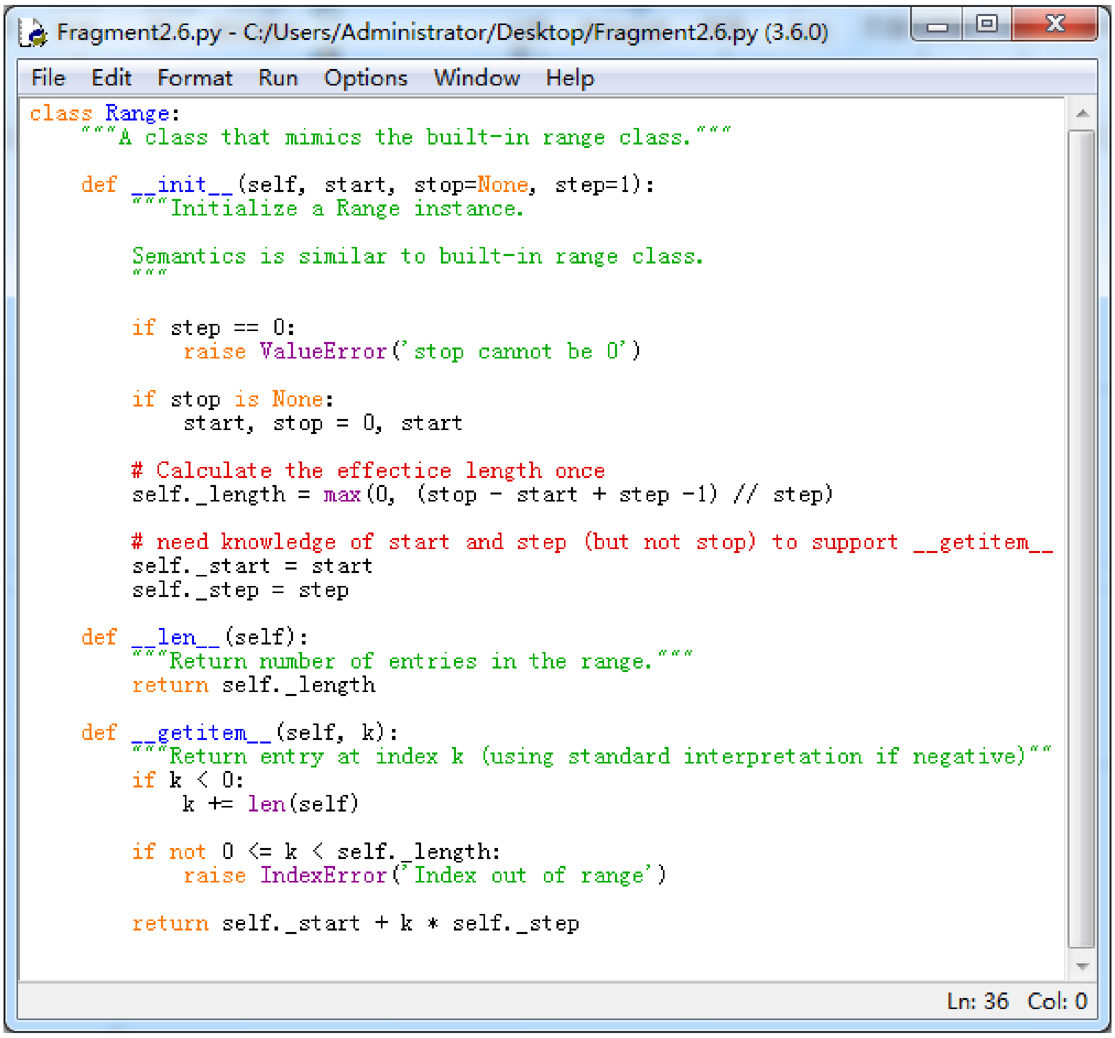
i. To properly support optional parameters, we rely on the technique described in Section 1.5.1-Keyword Parameters, when discussing a functional version of range.

ii. We compute the number of elements in the range as

   max(0, (stop – start + step -1) // step)

   It is worth testing this formula for both positive and negative step sizes.

iii. The __getitem__ method properly supports negative indices by converting an index -k to len(self)-k before computing the result.


Fragment2.6.py - C:/Users/Administrator/Desktop/Fragment2.6.py (3.6.0)
File Edit Format Run Options Window Help
Ln: 36 Col: 0

```python
class Range:
    """A class that mimics the built-in range class."""

    def __init__(self, start, stop=None, step=1):
        """Initialize a Range instance.

        Semantics is similar to built-in range class.
        """
        if step == 0:
            raise ValueError('stop cannot be 0')

        if stop is None:
            start, stop = 0, start

        # Calculate the effectice length once
        self._length = max(0, (stop - start + step -1) // step)

        # need knowledge of start and step (but not stop) to support __getitem__
        self._start = start
        self._step = step

    def __len__(self):
        """Return number of entries in the range."""
        return self._length

    def __getitem__(self, k):
        """Return entry at index k (using standard interpretation if negative)"""
        if k < 0:
            k += len(self)

        if not 0 <= k < self._length:
            raise IndexError('Index out of range')

        return self._start + k * self._step
```

Code Fragment 2.6 Our own implementation of a Range class.

2.4 Inheritance

A natural way to organize various structural components of a software package is in a **hierarchical** fashion, with similar abstract definitions grouped together in a level-by level manner that goes from specific to more general as one traverses up the hierarchy. An example of such a hierarchy is shown in Figure 2-4. Using mathematical notations, the set of houses is a subset of the set of buildings, but a superset of the set of ranches. The correspondence between levels if often referred to as an "is a" relationship, as a house is a building, and a ranch is a house.

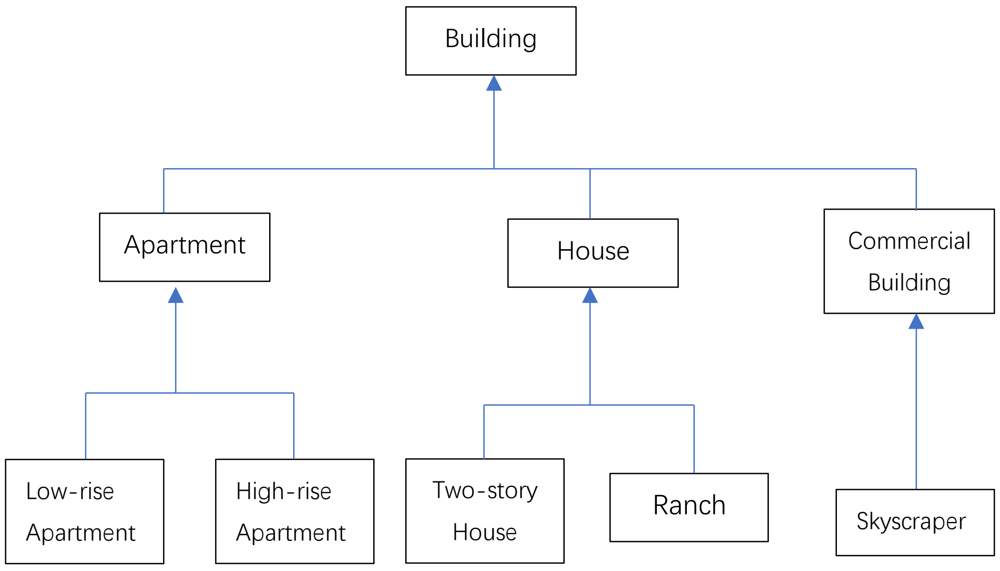


Figure 2-4 An example of an "is a " hierarchy involving architectural buildings.

A hierarchical design is useful in software development, as common functionality can be grouped at the most general level, thereby promoting reuse of code, while differentiated behaviors can be viewed as extensions of the general case. In object-oriented programming, the mechanism for a modular and hierarchical organization is a technique known as **inheritance**. This allows a new class to be defined based upon an existing class is typically described as the **base class**, **parent class**, or **superclass**, while the newly defined class is known as the **subclass** or **child class**.

There are two ways in which a subclass can differentiate itself from its superclass. A subclass may **specialize** an existing behavior by providing a new implementation that **overrides** an existing method. A subclass may also **extend** its superclass by providing brand new methods.

Python's Exception Hierarchy

Another example of a rich inheritance hierarchy is the organization of various exception types in Python. We introduced many of those classes in Section 1.7, but did not discuss their relationship with each other. Figure 2-5 illustrates a (small) portion of that hierarchy. The BaseException class is the root of the entire hierarchy, while the more specific Exception class includes most of the error types that we have discussed. Programmers are welcome to define their own special exception classes to denote errors that may occur in the context of their application. Those user-defined exception types should be declared as subclasses of Exception.

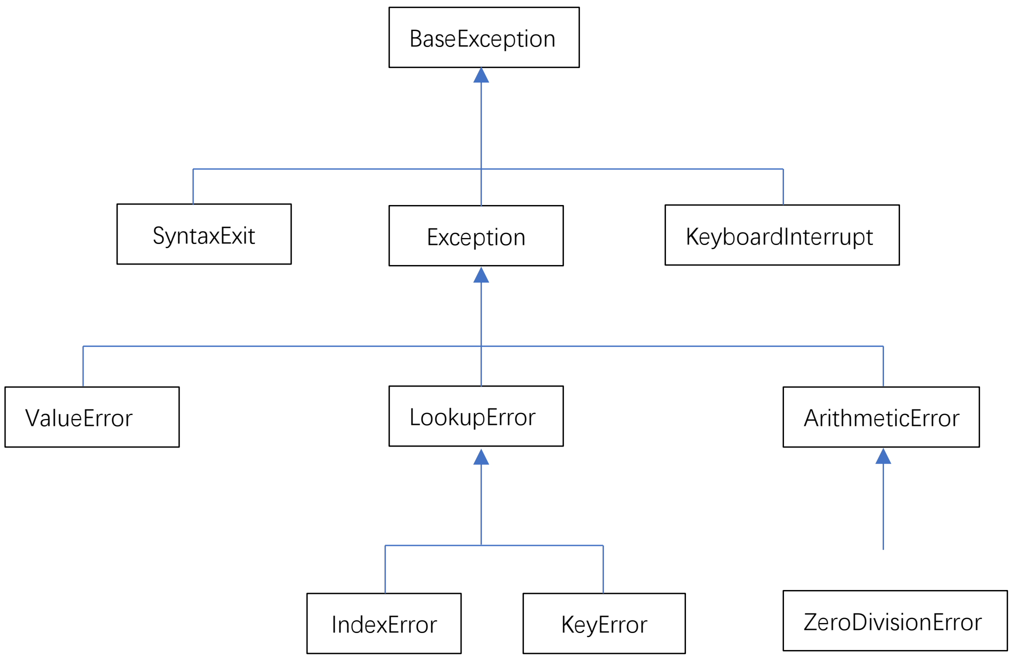


Figure 2-5 A portion of Python's hierarchy of exception types

2.4.1 Extending the CreditCard Class

To demonstrate the mechanisms for inheritance in Python, we revisit the CreditCard class in Section 2.3, implementing a subclass that, for lack of a better name, we name PredatoryCreditCard. The new class will differ from the original in two ways: (1) if an attempted charge is rejected because it would have exceeded the credit limit, a $5 fee will be charged, and (2) there will be a mechanism for accessing a monthly interest charge on the outstanding balance, based upon an Annual Percentage Rate (APR) specified as a constructor parameter.

In accomplishing this goal, we demonstrate the techniques of specialization and extension. To charge a fee for an invalid charge attempt, we override the existing charge method, thereby specializing it to provide the new functionality (although the new version takes advantage of a call to the overridden version). To provide support for charging interest, we extend the class with a new method named process_month.

Figure 2-6 provides an overview of our use of inheritance in designing the new PredatoryCreditCard class, and Code Fragment 2.7 gives a complete Python implementation of that class.

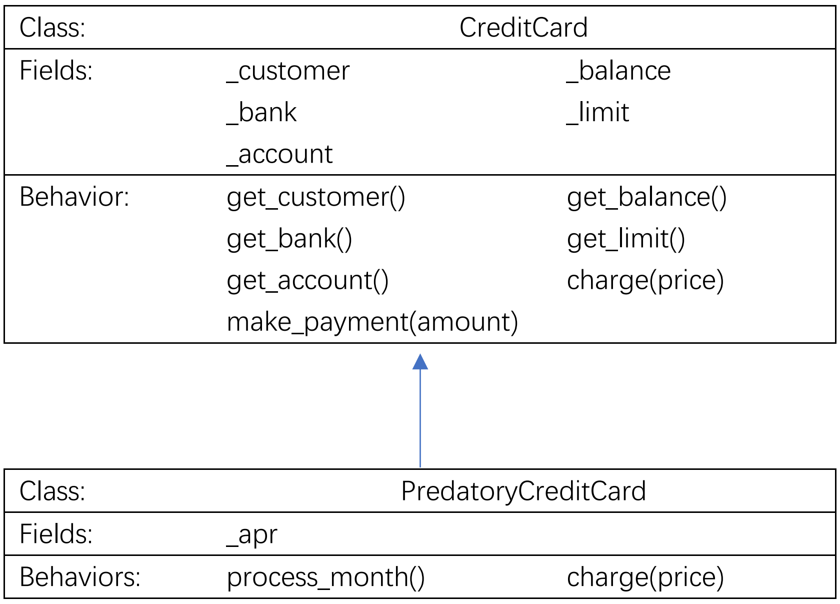
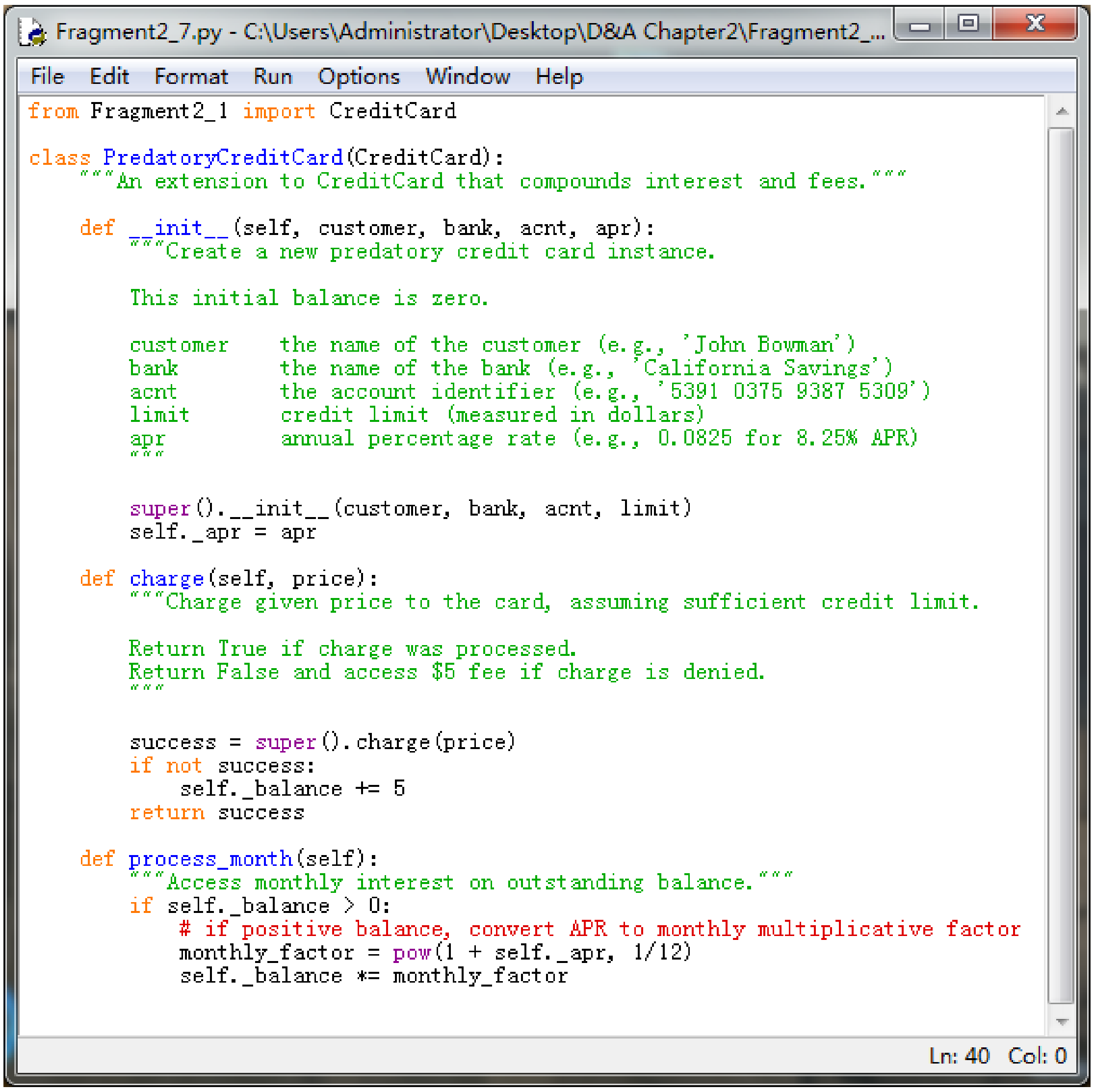

Class: CreditCard
Fields: _customer _balance
_bank _limit
_account
Behavior: get_customer() get_balance()
get_bank() get_limit()
get_account() charge(price)
make_payment(amount)
Class: PredatoryCreditCard
Fields: _apr
Behaviors: process_month() charge(price)

Figure 2-6 Diagram of an inheritance relationship.


Fragment2_7.py - C:\Users\Administrator\Desktop\D&A Chapter2\Fragment2_...
File Edit Format Run Options Window Help
Ln: 40 Col: 0

```python
from Fragment2_1 import CreditCard

class PredatoryCreditCard(CreditCard):
    """An extension to CreditCard that compounds interest and fees."""

    def __init__(self, customer, bank, acnt, apr):
        """Create a new predatory credit card instance.

        This initial balance is zero.

        customer    the name of the customer (e.g., 'John Bowman')
        bank        the name of the bank (e.g., 'California Savings')
        acnt        the account identifier (e.g., '5391 0375 9387 5309')
        limit       credit limit (measured in dollars)
        apr         annual percentage rate (e.g., 0.0825 for 8.25% APR)
        """

        super().__init__(customer, bank, acnt, limit)
        self._apr = apr

    def charge(self, price):
        """Charge given price to the card, assuming sufficient credit limit.

        Return True if charge was processed.
        Return False and access $5 fee if charge is denied.
        """

        success = super().charge(price)
        if not success:
            self._balance += 5
        return success

    def process_month(self):
        """Access monthly interest on outstanding balance."""
        if self._balance > 0:
            # if positive balance, convert APR to monthly multiplicative factor
            monthly_factor = pow(1 + self._apr, 1/12)
            self._balance *= monthly_factor
```

Code Fragment 2.7 A subclass of CreditCard that accesses interest and fees.

To indicate that the new class inherits from the existing CreditCard class, our definition begins with the syntax, class PredatoryCreditCard(CreditCard). The body of the new class provides three member functions: __init__, charge, and process_month. The __init__ constructor serves a very similar role to the original CreditCard constructor, except that our new class, there is an extra parameter to specify the annual percentage rate. The body of our new constructor relies upon making a call to the inherited constructor to perform most of the initialization (in fact, everything other than the recording of the percentage rate). The mechanism for calling the inherited constructor relies on the syntax, super(). Specifically, at line 18 the command

super().__init__(customer, bank, acnt, limit)

Calls the __init__ method that was inherited from the CreditCard superclass. Note well that this method only accepts four parameters. We record the APR value in a new field named _apr.

In similar fashion, our PredatoryCreditCard class provides a new implementation of the charge method that overrides the inherited method. Yet, our implementation of the new method relies on a call to the inherited method, with syntax super().charge(price) at line 28. The return value of that call designates whether the charge was successful. We examine that return value to decide whether to assess a fee, and in return that value to the caller of method, so that the new version of charge has a similar outward interface as the original.

The process_month method is a new behavior, so there is no inherited version upon which to rely. In our model, this method should be invoked by the bank, once each month, to add new interest charges to the customer's balance. The most challenging aspect in implementing this method is making sure we have working knowledge of how an annual percentage rate translates to a monthly rate. We do not simply divide the annual rate by twelve to get a monthly rate (that would be too predatory, as it would result in a higher APR than advertised). The correct computation is to take the twelve-root of 1 + self._apr, and use that as a multiplicative factor. For example, if the APR is 0.0825 (representing 8.25%), we compute $\sqrt[12]{1.0825} \approx 1.006628$, and therefore charge 0.6628% interest per month. In this way, each $100 of debt will amass $8.25 of compounded interest in a year.

Protected Members

Our PredatoryCreditCard subclass directly accesses the data member self._balance, which was established by the parent CreditCard class. The underscored name, by convention, suggest that this is a nonpublic member, so we might ask if it is okay that we access it in this fashion. While general users of the class should not be doing so, our subclass has a somewhat privileged relationship with the superclass. Several object-oriented languages (e.g., Java, C++) draw a distinction for nonpublic members, allowing declarations of **protected** or **private** access modes. Members that are declared as protected are accessible to subclass, but not the general public, while members that are declared as private are not accessible to either. In this respect, we are using _balance as if it were protected (but not private).

Python does not support formal access control, but names beginning with a single underscore are conventionally akin to protected, while names beginning with a double underscore (other than special methods) are akin to private. In choosing to use protected data, we have created a dependency in that our PredatoryCreditCard class might be compromised if the author of the CreditCard class were to change the internal design. Note that we could have relied upon the public get_balance() method to retrieve the current balance within the process_month method. But the current design of the CreditCard class does not afford an effective way for a subclass to change the balance, other than by direct manipulation of the data member. It may be tempting to use charge to add fees or interest to the balance. However, that method does not allow the balance to go above the customer's credit limit, even though a bank would presumably let interest compound beyond the credit limit, if warranted. If we were to redesign the original CreditCard class, we might add a nonpublic method, _set_balance, that could be used by subclasses to affect a change without directly accessing the data member _balance.

2.4.2 Hierarchy of Number Progressions

As a second example of the use of inheritance, we develop a hierarchy of classes for iterating numeric progressions. A numeric progression is a sequence of numbers, where each number depends on one or more of the previous numbers. For example, an **arithmetic progression** determines the next number by adding a fixed constant to the previous value, and a **geometric progression** determines the next number by multiplying the previous value by a fixed constant. In general, a progression requires a first value, and a way of identifying a new value based on one or more previous values.

To maximize reusability of code, we develop a hierarchy of classes stemming from a general base class that we name Progression (see Figure 2-7). Technically, the Progression class produces the progression of whole numbers: 0, 1, 2, 3 ⋯ However, this class is designed to serve as the base class for other progression types, providing as much common functionality as possible, and thereby minimizing the burden on the subclass.

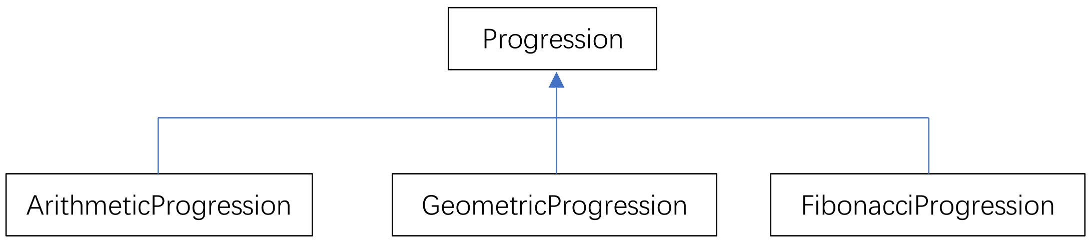


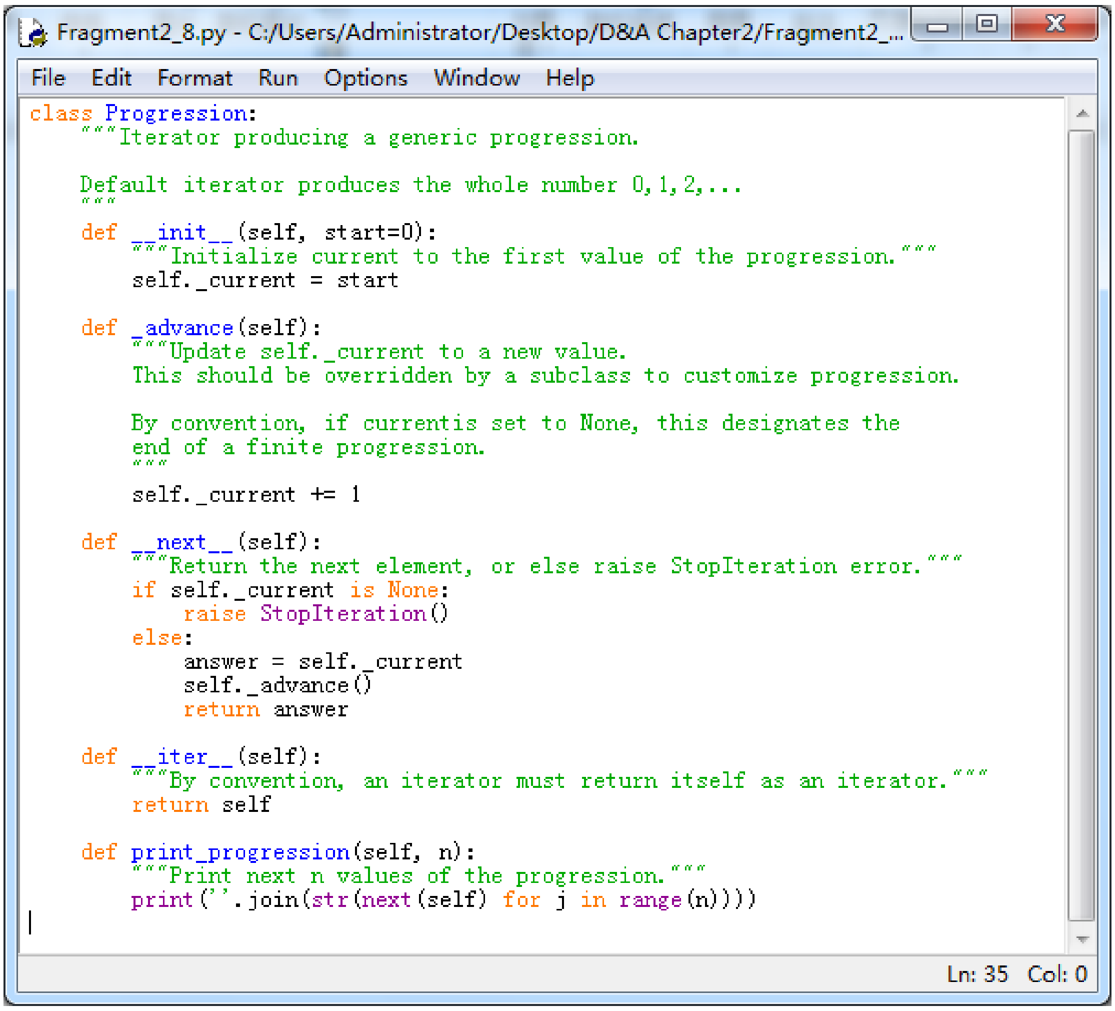
Figure 2-7 Our hierarchy of progression classes

Our implementation of the basic Progression class is provided in Code Fragment 2.8. The constructor for this class accepts a starting value for the progression (0 by default), and initializes a data member, self._current, to that value.

The Progression class implements the conventions of a Python iterator (see Section 2.3.4), namely the special __next__ and __iter__ methods. If a user of the class creates a progression as seq = Progression(), each call to next(seq) will return a subsequence element of the progression sequence. It would also be possible to use a for-loop syntax, for value in seq:, although we note that our default progression is defined as an infinite sequence.

To better separate the mechanics of the iterator convention from the core logic of advancing the progression, our framework relies on a nonpublic method named _advance to update the value of the self._current field. In the default implementation, _advance adds one to the current value, but our intent is that subclasses will override _advance to provide a different rule for computing the next entry.

For convenience, the Progression class also provides a utility method, named print_progression, that displays the next n values of the progression.


Fragment2_8.py - C:/Users/Administrator/Desktop/D&A Chapter2/Fragment2_...
File Edit Format Run Options Window Help
Ln: 35 Col: 0

```python
class Progression:
    """Iterator producing a generic progression.

    Default iterator produces the whole number 0,1,2,...
    """
    def __init__(self, start=0):
        """Initialize current to the first value of the progression."""
        self._current = start

    def _advance(self):
        """Update self._current to a new value.
        This should be overridden by a subclass to customize progression.

        By convention, if currentis set to None, this designates the
        end of a finite progression.
        """
        self._current += 1

    def __next__(self):
        """Return the next element, or else raise StopIteration error."""
        if self._current is None:
            raise StopIteration()
        else:
            answer = self._current
            self._advance()
            return answer

    def __iter__(self):
        """By convention, an iterator must return itself as an iterator."""
        return self

    def print_progression(self, n):
        """Print next n values of the progression."""
        print(''.join(str(next(self) for j in range(n))))
```

Code Fragment 2.8 A general numeric progression class

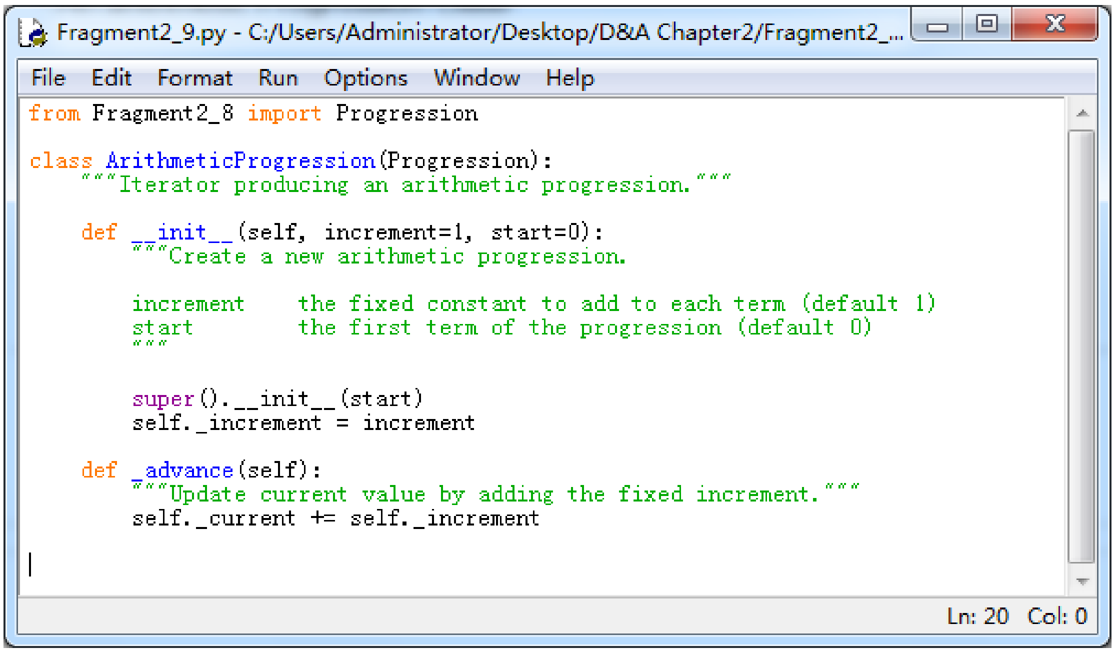
An arithmetic Progression Class

Our first example of a specialized progression is an arithmetic progression. While the default progression increases its value by one in each step, an arithmetic progression adds a fixed constant to one term of the progression to produce the next. For example, using an increment of 4 for an arithmetic progression that starts at 0 results in the sequence 0,4,8,12,⋯

Code Fragment 2.9 presents our implementation of an ArithmeticProgression class, which relies on Progression as its base class. The constructor for this new class accepts both an increment value and a starting value as parameters, although default values for each are provides. By our convention, ArithmeticProgression(4) produces the sequence 0,4,8,12,⋯, and ArithmeticProgression(4,1) produces the sequence 1,5,9,13,⋯

The body of the ArithmeticProgression constructor calls the super constructor to initialize the _current data member to the desired start value. Then it directly establishes the new _increment data member for the arithmetic progression. The only remaining detail in our implementation is to override the _advance method so as to add the increment to the current value.

```
>>>
for i in ArithmeticProgression(4):
	print(i)

0
4
8
12
16Traceback (most recent call last):
  File "<pyshell#16>", line 2, in <module>
	print(i)
KeyboardInterrupt
>>>
for i in ArithmeticProgression(4,1):
	print(i)

1
5
9
13
```


Fragment2_9.py - C:/Users/Administrator/Desktop/D&A Chapter2/Fragment2_...
File Edit Format Run Options Window Help
Ln: 20 Col: 0

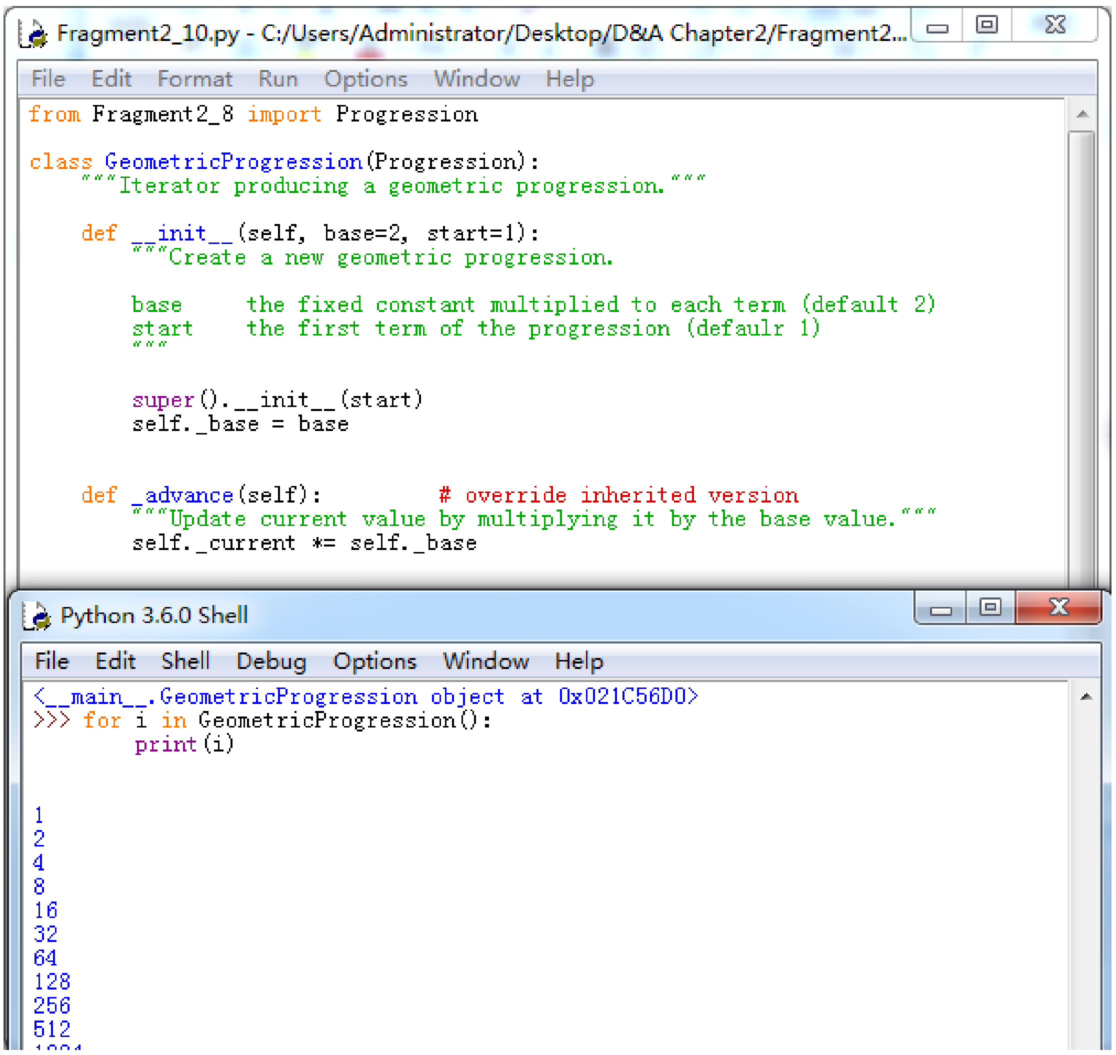
```python
from Fragment2_8 import Progression

class ArithmeticProgression(Progression):
    """Iterator producing an arithmetic progression."""

    def __init__(self, increment=1, start=0):
        """Create a new arithmetic progression.

        increment    the fixed constant to add to each term (default 1)
        start        the first term of the progression (default 0)
        """

        super().__init__(start)
        self._increment = increment

    def _advance(self):
        """Update current value by adding the fixed increment."""
        self._current += self._increment

```

Code Fragment 2.9 A class that produces an arithmetic progression.

A geometric Progression Class

Our second example of a specialized progression is a geometric progression, in which each value is produced by multiplying the preceding value by a fixed constant, known as the **base** of the geometric progression. The starting point of a geometric progression is traditionally 1, rather than 0, because multiplying 0 by any factor result in 0. As an example, a geometric progression with base 2 proceeds as 1, 2, 4, 8, 16, ⋯

Code Fragment 2.10 presents our implementation of a GeometricProgression class. The constructor uses 2 as a default base and 1 as a default starting value, but either of those can be varied using optional parameter.


Fragment2_10.py - C:/Users/Administrator/Desktop/D&A Chapter2/Fragment2...
File Edit Format Run Options Window Help
Python 3.6.0 Shell
File Edit Shell Debug Options Window Help

```python
from Fragment2_8 import Progression

class GeometricProgression(Progression):
    """Iterator producing a geometric progression."""

    def __init__(self, base=2, start=1):
        """Create a new geometric progression.

        base     the fixed constant multiplied to each term (default 2)
        start    the first term of the progression (defaulr 1)
        """

        super().__init__(start)
        self._base = base


    def _advance(self):          # override inherited version
        """Update current value by multiplying it by the base value."""
        self._current *= self._base
```

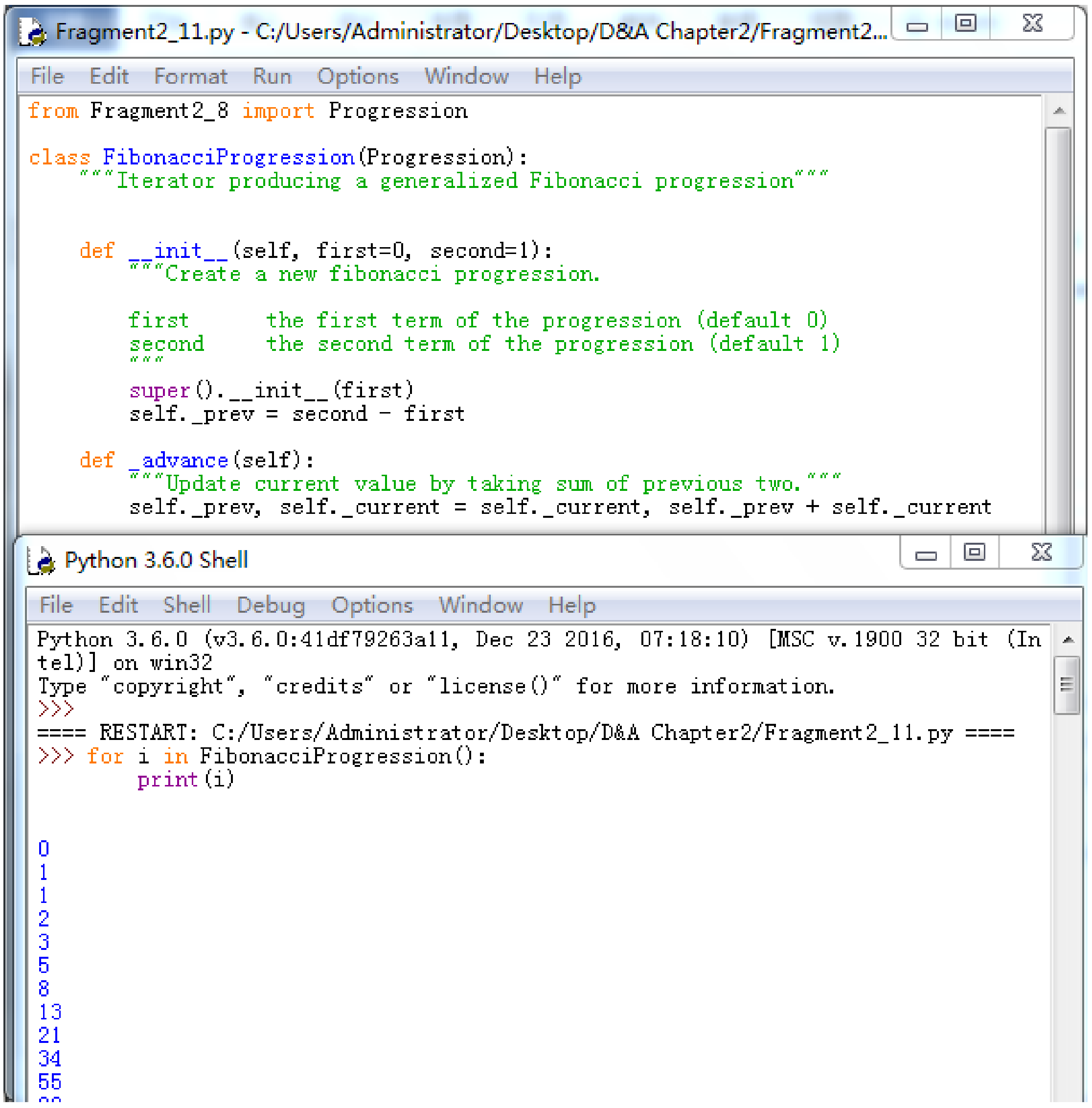
```
<__main__.GeometricProgression object at 0x021C56D0>
>>> for i in GeometricProgression():
        print(i)


1
2
4
8
16
32
64
128
256
512
```

Code Fragment 2.10 A class that produces a geometric progression.

A Fibonacci Progression Class

As our final example, we demonstrate how to use out progression framework to produce a **Fibonacci Progression**. We originally discussed the Fibonacci series in Section 1.8 in the context of generators. Each value of a Fibonacci series is the sum of the two most recent values. To begin the series, the first two values are conventionally 0 and 1, leading to the Fibonacci series 0, 1, 1, 2, 3, 5, 8, ⋯ More generally, such a series can be generated from any two starting values. For example, if we start with values 4 and 6, the series proceeds as 4, 6, 10, 16, 26, 42, ⋯

We use our progression framework to define a new FibonacciProgression class, as shown in Code Fragment 2.11. This class is markedly different from those for the arithmetic and geometric progressions because we cannot determine the next value of a Fibonacci series solely from the current one. We must maintain knowledge of the two most recent values. The base Progression class already provides storage of the most recent value as the _current data member. Our FibonacciProgression class introduces a new member, named _prev, to store the value that proceeded the current one.


Fragment2_11.py - C:/Users/Administrator/Desktop/D&A Chapter2/Fragment2...
File Edit Format Run Options Window Help
Python 3.6.0 Shell
File Edit Shell Debug Options Window Help

```
from Fragment2_8 import Progression

class FibonacciProgression(Progression):
    """Iterator producing a generalized Fibonacci progression"""


    def __init__(self, first=0, second=1):
        """Create a new fibonacci progression.

        first      the first term of the progression (default 0)
        second     the second term of the progression (default 1)
        """
        super().__init__(first)
        self._prev = second - first

    def _advance(self):
        """Update current value by taking sum of previous two."""
        self._prev, self._current = self._current, self._prev + self._current
```

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) [MSC v.1900 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
==== RESTART: C:/Users/Administrator/Desktop/D&A Chapter2/Fragment2_11.py ====
>>> for i in FibonacciProgression():
        print(i)


0
1
1
2
3
5
8
13
21
34
55
```

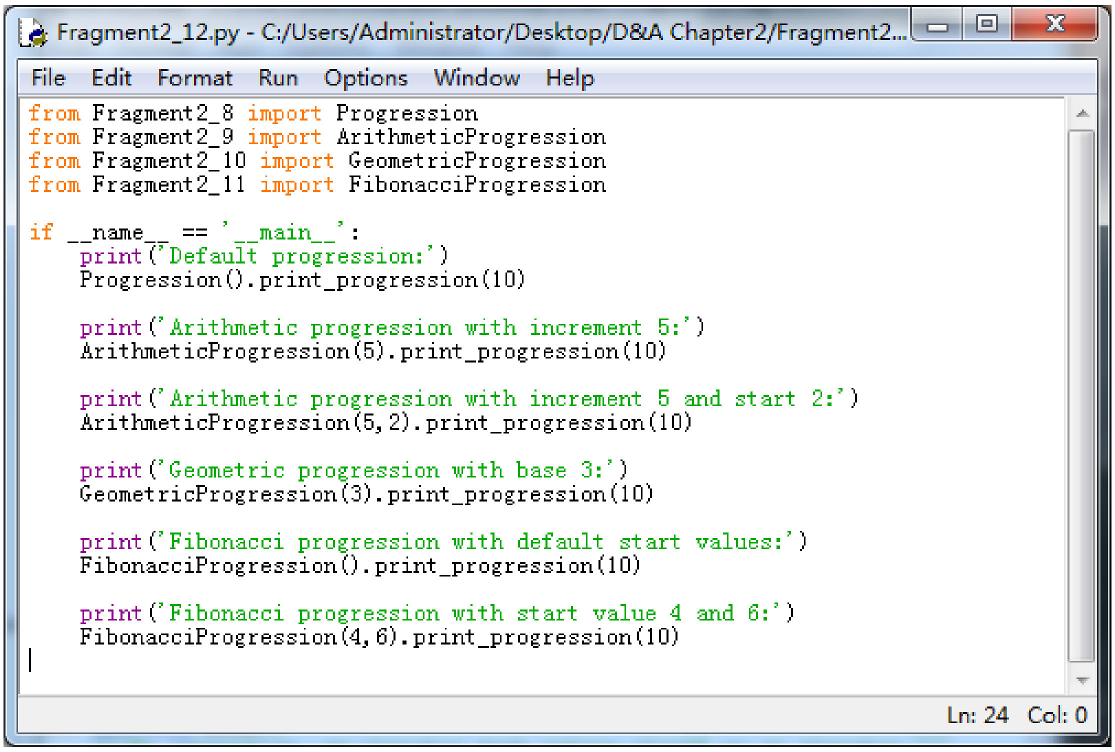
Code Fragment 2.11 A class that produces a Fibonacci progression

With both previous values stored, the implementation of _advance is relatively straightforward. (We use a simultaneous assignment similar to that in Section 1.9.3- Simultaneous Assignments.) However, the question arises as how to initialize the previous value in the constructor. The desired first and second values are provided as parameters to the constructor. The first should be stored as _current so that it becomes the first one that is reported. Looking ahead, once the first value is reported, we will do an assignment to set the new current value (which will be the second value reported), equal to the first value plus the "previous ". By initializing the previous value to (second - first), the initial advancement will set the new current value to first + (second - first) = second, as desired.

Testing Our Progressions

To complete our presentation, Code Fragment 2.12 provides a unit test for all of our progression classes, and the output of the test are shown as follows:

```
>>>
Default progression:
0 1 2 3 4 5 6 7 8 9
Arithmetic progression with increment 5:
0 5 10 15 20 25 30 35 40 45
Arithmetic progression with increment 5 and start 2:
2 7 12 17 22 27 32 37 42 47
Geometric progression with base 3:
1 3 9 27 81 243 729 2187 6561 19683
Fibonacci progression with default start values:
0 1 1 2 3 5 8 13 21 34
Fibonacci progression with start value 4 and 6:
4 6 10 16 26 42 68 110 178 288
>>>
```


Fragment2_12.py - C:/Users/Administrator/Desktop/D&A Chapter2/Fragment2...
File Edit Format Run Options Window Help
Ln: 24 Col: 0

```python
from Fragment2_8 import Progression
from Fragment2_9 import ArithmeticProgression
from Fragment2_10 import GeometricProgression
from Fragment2_11 import FibonacciProgression

if __name__ == '__main__':
    print('Default progression:')
    Progression().print_progression(10)

    print('Arithmetic progression with increment 5:')
    ArithmeticProgression(5).print_progression(10)

    print('Arithmetic progression with increment 5 and start 2:')
    ArithmeticProgression(5,2).print_progression(10)

    print('Geometric progression with base 3:')
    GeometricProgression(3).print_progression(10)

    print('Fibonacci progression with default start values:')
    FibonacciProgression().print_progression(10)

    print('Fibonacci progression with start value 4 and 6:')
    FibonacciProgression(4,6).print_progression(10)
```

Code Fragment 2.12 Unit tests for our progression classes.

2.4.3 Abstract Base Classes

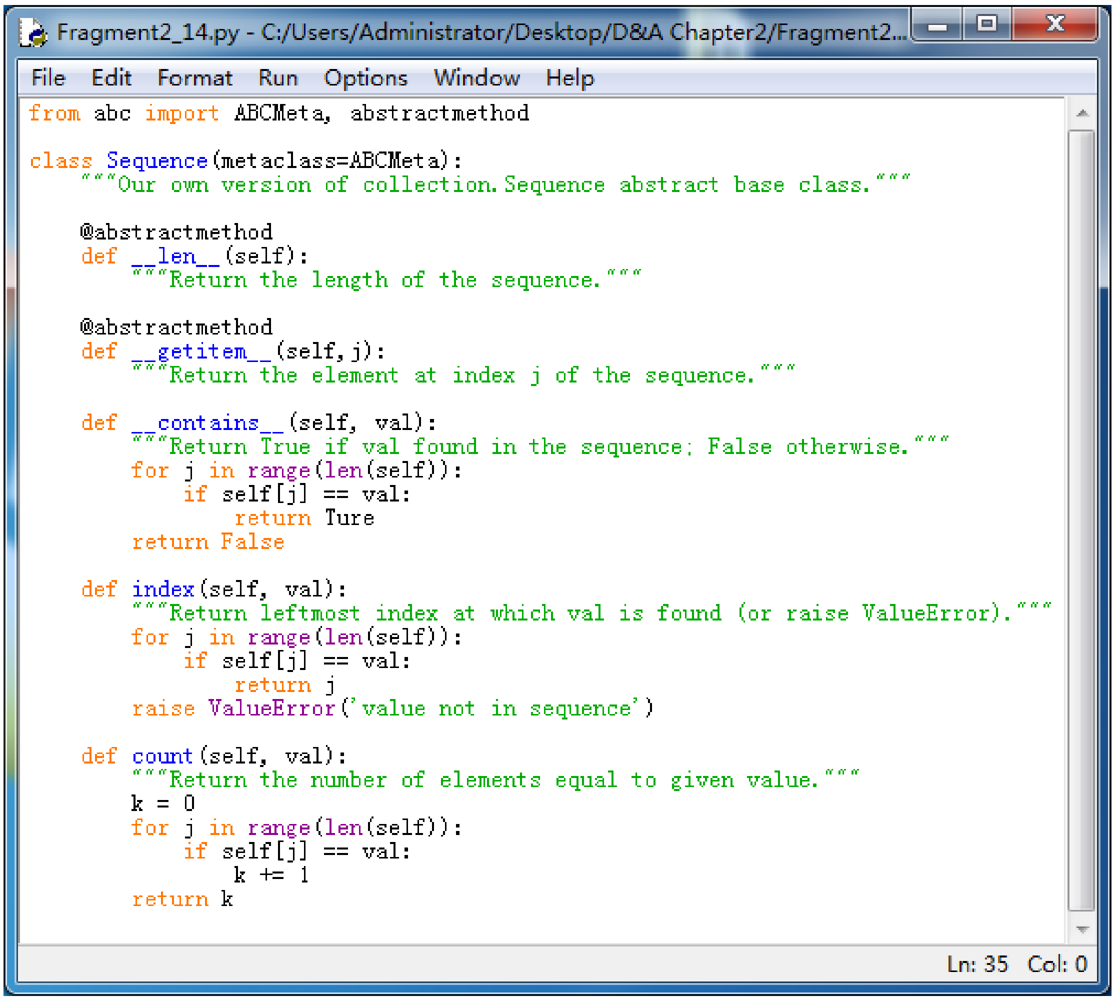
When defining a group of classes as part of an inheritance hierarchy, one technique for avoiding repetition of the code is to design a base class with common functionality that can be inherited by other classes that need it. As an example, the hierarchy from Section 2.4.2 includes a Progression, which serves as a base class for three distinct subclasses: ArithmeticProgression, GeometricProgression and FibonacciProgression. Although it is possible to create an instance of the Progression base class, there is little value in doing so because its behavior is simply a special case of an ArithmeticProgression with increment 1. The real purpose of the Progression class was to centralize the implementations of behaviors that other progressions needed, thereby streamlining the code that is relegated to those subclasses.

In classic object-oriented terminology, we say a class is an **abstract base class** if its only purpose is to serve as a base class through inheritance. More formally, an abstract base class is one that cannot be directly instantiated, while a **concrete class** is one that can be instantiated. By this definition, our Progression class is technically concrete, although we essentially designed it as an abstract base class.

In statically typed languages such as Java and C++, an abstract base class serves as a formal type that may guarantee one or more **abstract methods**. This provides support for polymorphism, as a variable may have an abstract base class as its declared type, even though it refers to an instance of a concrete subclass. Because there are no declared types in Python, this kind of polymorphism can be accomplished without the need for a unifying abstract base class. For this reason, there is not as strong a tradition of defining abstract base classes in Python, although Python's abc module provides support for defining a formal abstract base class.

Our reason for focusing on abstract base classes in our study of data structures is that Python's collection module provides several abstract base classes that assist when defining custom data structures that share a common interface with some of Python's built-in data structures. These rely on an object-oriented software design pattern known as the **template method pattern**. The template method pattern is when an abstract base class provides concrete behaviors that rely upon calls to other abstract behaviors. In that way, as soon as a subclass provides definitions for the missing abstract behaviors, the inherited concrete behaviors are well defined.

As a tangible example, the collection.Sequence abstract base class defines behaviors common to Python's list, str, and tuple classes, as sequences that support element access via an integer index. More so, the collection.Sequence class provides concrete implementations of methods, count, index, and __contains__ that can be inherited by any class that provides concrete implementations of both __len__ and __getitem__. For the purpose of illustration, we provide a sample implementation of such a Sequence abstract base class in Code Fragment 2.13 (Fragment2_14).


Fragment2_14.py - C:/Users/Administrator/Desktop/D&A Chapter2/Fragment2...
File Edit Format Run Options Window Help
Ln: 35 Col: 0

```
from abc import ABCMeta, abstractmethod

class Sequence(metaclass=ABCMeta):
    """Our own version of collection.Sequence abstract base class."""

    @abstractmethod
    def __len__(self):
        """Return the length of the sequence."""

    @abstractmethod
    def __getitem__(self,j):
        """Return the element at index j of the sequence."""

    def __contains__(self, val):
        """Return True if val found in the sequence; False otherwise."""
        for j in range(len(self)):
            if self[j] == val:
                return Ture
        return False

    def index(self, val):
        """Return leftmost index at which val is found (or raise ValueError)."""
        for j in range(len(self)):
            if self[j] == val:
                return j
        raise ValueError('value not in sequence')

    def count(self, val):
        """Return the number of elements equal to given value."""
        k = 0
        for j in range(len(self)):
            if self[j] == val:
                k += 1
        return k
```

Code Fragment 2.13 An abstract base class akin to collection.Sequence

This implementation relies on two advanced Python techniques. The first is that we declare the ABCMeta class of the abc module as a metaclass of our Sequence class. A metaclass is different from a superclass, in that it provides a templete for the class definition itself. Specifically, the ABCMeta declaration assures that the constructor for the class raises an error.

The second advanced technique is the use of the @abstractmethod decorator immediately before the __len__ and __getitem__ methods are declared. That declares these two particular methods to be abstract, meaning that we do not provide an implementation within our Sequence base class, but that we expect any concrete subclasses to support those two methods. Python enforces this expectation, by disallowing instantiation for any subclass that does not override the abstract methods with concrete implementations.

The rest of the Sequence class definition provides tangible implementations for other behavior, under the assumption that the abstract __len__ and __getitem__ methods will exist in a concrete subclass. If you carefully examine the source code, the implementations of methods __contains__, index, and count do not rely on any assumption about the self instances, other than that syntax len(self) and self[j] are supported (by special methods __len__ and __getitem__, respectively). Support for iteration is automatic as well, as described in Section 2.3.4.

In the remainder of this book, we omit the formality of using the abc module. If we need an "abstract" base class, we simply document the expectation that subclasses provide assumed functionality without technical declaration of the methods as abstract. But we will make use of the wonderful abstract base classes that are defined within the collections module (such as Sequence). To use such a class, we need only rely on standard inheritance techniques.

For example, our Range class, from Code Fragment 2.6 of Section 2.3.5, is an example of a class that supports the __len__ and __getitem__ methods. But that class does not support methods count or index. Had we originally declared it with Sequence as a superclass, then it would also inherit the count and index methods. The syntax for such a declaration would begin as:

Class Range(collection.Sequence):

Finally, we emphasize that if a subclass provides its own implementation of an inherited behaviors from a base class, the new definition overrides the inherited one. This technique can be used when we have the ability to provide a more efficient implementation for a behavior than is achieved by the generic approach. As an example, the general implementation of __contains__ for a sequence is based on a loop used to search for the desired value. For our Range class, there is an opportunity for a more efficient determination of containment. For example, it is evident that the expression, 100000 in Range(0, 2000000, 100), should evaluate to True, even without examining the individual elements of the range, because the range starts with zero, has an increment of 100, and goes until two million; it must include 100000, as that is a multiple of 100 that is between the start and stop values. Exercise C-2.27 explores the goal of providing an implementation of Range.__contains__ that avoids the use of a (time-consuming) loop.

2.5 Namespaces and Object-Orientation

A **namespace** is an abstraction that manages all of the identifiers that are defined in a particular scope, mapping each name to its associated value. In Python, functions, classes, and modules are all first-class object, and so the "value" associated with an identifier in a namespace may in fact be a function, class, or module.

In Section 1.10 we explored Python's use of namespaces to manage identifiers that are defined with global scope, versus those defined within the local scope of a function call. In this section, we discuss the important role of namespaces in Python's management of object-orientation.

2.5.1 Instance and Class Namespaces

We begin by exploring what is known as the **instance namespace**, which manages attributes specific to an individual object. For example, each instance of our CreditCard class maintains a distinct balance, a distinct account number, a distinct credit limit, and so on (even though some instances may coincidentally have equivalent balances, or equivalent credit limit). Each credit card will have a dedicated instance namespace to manage such values.

There is a separate **class namespace** for each class that has been defined. This namespace is used to manage members that are to be shared by all instances of a class, or used without reference to any particular instance. For example, the make_payment method of the CreditCard class from Section 2.3 is not stored independently by each instance of that class. That member function from Code Fragment 2.1 and Code Fragment 2.2, the CreditCard class namespace includes the functions: __init__, get_customer, get_bank, get_account, get_balance, get_limit, charge and make_payment. Our PredatoryCreditCard class has its own namespace, containing the three methods we defined for that subclass: __init__, charge, and process_month.

Figure 2-8 provides a portrayal of three such namespaces: a class namespace containing methods of the CreditCard class, another class namespace with methods of the PredatoryCreditCard class, and finally a single instance namespace for a sample instance of the PredatoryCreditCard class. We note that there are two different definitions of a function named charge, one is in the CreditCard class, and then overriding method in the PredatoryCreditCard class. In similar fashion, there are two distinct __init__ implementations. However, process_month is a name that is only defined within the scope of the PredatoryCreditCard class. The instance namespace includes all data members for the instance (including the _apr member that is established by the PredatoryCreditCard constructor).

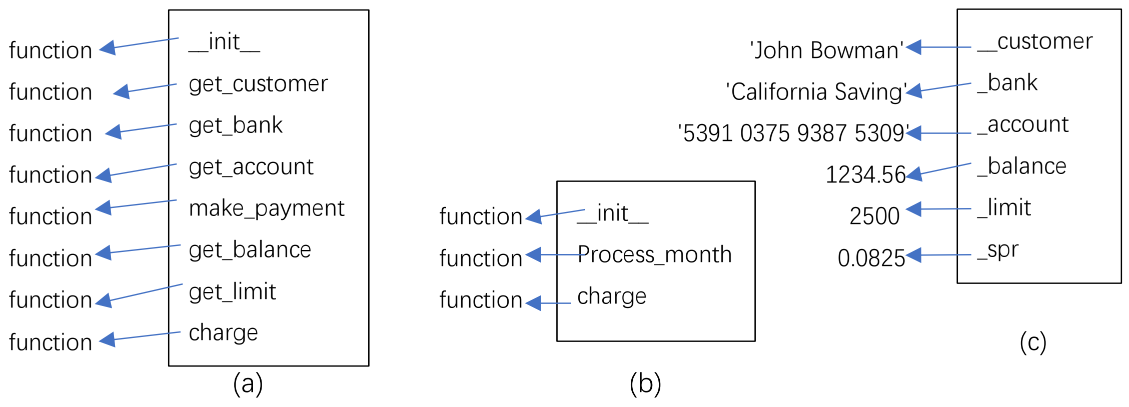


Figure 2-8 Conceptual view of three namespaces: (a) the class namespace for CreditCard; (b) the class namespace for PredatoryCreditCard; (c) the instance namespace for a Predatory CreditCard object

How Entries Are Established in a Namespace

It is important to understand why a member such as _balance resides in a credit card's instance namespace, while a member such as make_payment resides in the class namespace. The balance is established within the __init__ method when a new credit card instance is constructed. The original assignment uses the syntax, self._balance = 0, where self is an identifier for the newly constructed instance. The use of self as a qualifier for self._balance is such as an assignment causes the _balance identifier to be added directly to the instance namespace.

When inheritance is used, there is still a single instance namespace per object. For example, when an instance of the PredatoryCreditCard class is constructed, the _apr attribute as well as attributes such as _balance and _limit all reside in that instance's namespace, because all are assigned using a qualified syntax, such as self._apr.

A class namespace includes all declarations that are made directly within the body of the class definition. For example, our CreditCard class definition included the following structure:

```
class CreditCard:
    def make_payment(self, amount):
        ...
```

Because the make_payment function is declared within the scope of the CreditCard class, that function becomes associated within the name make_payment within the CreditCard class namespace. Although member functions are the most typical types of entries that are declared in a class namespace, we next discuss how other types of data values, or even other classes can be declared within a class namespace.

Class Data Members

A class-level data member is often used when there is some value, such as a constant, that is to be shared by all instances of a class. In such a case, it would be unnecessarily wasteful to have each instance store that value in its instance namespace. As an example, we revisit the PredatoryCreditCard introduced in Section 2.4.1. This class accesses a $5 fee if an attempted charge is denied because of the credit limit. Our choice of $5 for the fee was somewhat arbitrary, and our coding style would be better if we used a named variable rather than embedding the literal value in our code. Often, the amount of such a fee is determined by the bank's policy and does not vary for each customer. In that case, we would define and use a class data member as follows:

```
class PredatoryCreditCard(CreditCard):
    OVERLIMIT_FEE = 5
    def charge(self, price):
        success = super().charge(price)
        if not success:
            self._balance += predatorycreditcard.overlimit_fee
        return success
```

The data member, OVERLIMIT_FEE, is entered into the PredatoryCreditCard class nameapace because that assignment takes place within the immediate scope of the class definition, and without any qualifying identifier.

Nested Classes

It is also possible to nest one class definition within the scope of another class. This is a useful construct, which we will exploit several times in this book in the implementation of data structures. This can be done by using a syntax such as

```
class A:
    class B:
```

In this case, class B is the nested class. The identifier B is entered into the namespace of class A associated with the newly defined class. We note that this technique is unrelated to the concept of inheritance, as class B does not inherit from class A.

Nesting one class in the scope of another makes clear that the nested class exists for support of the outer class. Furthermore, it can help reduce potential name conflicts, because it allows for a similarly named class to exist to in another context. For example, we will later introduce a data structure known as a **linked list** and will define a nested node class to store the individual components of the list. We will also introduce a data structure known as a **tree** that depends upon its own nested node class. These two structures rely on different node definitions, and by nesting those within the respective container classes, we avoid ambiguity.

Another advantage of one class being nested as a member of another is that it allows for a more advanced form of inheritance in which a subclass of the outer class overrides the definition of its nested class. We will make use of that technique in Section 11.2.1 when specializing the nodes of a tree structure.

Dictionaries and the __slots__ Declaration

By default, Python represents each namespace with an instance of the built-in dict class (see Section 1.2.3) that maps identifying names in that scope to the associated object. While a dictionary structure supports relatively efficient name lookups, it requires additional memory usage beyond the raw data that it stores (we will explore the data structure used to implement dictionaries in Chapter 10).

Python provides a more direct mechanism for representing instance namespaces that avoids the use of an auxiliary dictionary. To use the streamlined representation for all instances of a class, that class definition must provide a class-level member named __slot__ that is assigned to a fixed sequence of strings that serve as names for instance variables. For example, with our CreditCard class, we would declare the following:

```
class CreditCard:
    __slot__ = '_customer',   '_bank',   '_account', '_balance', '_limit'
```

In this example, the righthand side of the assignment is technically a tuple (see discussion of automatic packing of tuples in Section 1.9.3).

When inheritance is used, if the base class declares __slot__, a subclass must also declare __slots__ to avoid creation of instance dictionaries. The declaration in the subclass should only include names of supplemental methods that are newly introduced. For example, our PredatoryCreditCard declaration would include the following declaration:

```
Class PredatoryCreditCard(CreditCard):
    __slot__ = '_apr'
```

We could choose to use the __slots__ declaration to streamline every class in this book. However, we do not do so because such rigor would be atypical for Python programs. With that said, there are a few classes in this book for which we expect to have a large number of instances, each representing a lightweight construct. For example, when discussing nested classes, we suggest linked lists and trees as data structures that are often comprised of a large number of individual nodes. To promote greater efficiency in memory usage, we will use an explicit __slot__ declaration in any nested classes for which we expect many instances.

2.5.2 Name Resolution and Dynamic Dispatch

In the previous section, we discussed various namespaces, and the mechanism for establishing entries in those namespaces. In this section, we examine the process that is used when retrieving a name in Python's object-oriented framework. When the dot operator syntax is used to access an existing member, such as obj.foo, the Python interpreter begins a name resolution process, described as follows:

1. The instance namespace is searched; if the desired name is found, its associated value is used.
2. Otherwise the class namespace, for the class to which the instance belongs, is searched; if the name is found, its associated value is used.
3. If the name was not found in the immediate class namespace, the search continues upward through the inheritance hierarchy, checking the class namespace for each ancestor (commonly by checking the superclass class, then its superclass class, and so on). The first time the name is found, its associate value is used.
4. If the name has still not been found, an AttributeError is raised.

As a tangible example, let us assure that mycard identifies an instance of the PredatoryCreditCard class. Consider the following possible usage patterns.

i. My_card._balance (or equivalently, self._balance from within a method body): The _balance method is found within the instance namespace for mycard.
ii. My_card.process_month(): the search begins in the instance namespace, but the name process_month is not found in that namespace. As a result, the PredatoryCreditCard class namespace is searched; in this case, the name is found and that method is called.
iii. My_card.make_payment(200): the search for the name, make_payment, fails in the instance namespace and in the PredatoryCreditCard namespace. The name is resolved in the namespace for superclass CreditCard and thus the inherited method is called.
iv. Mycard.charge(50): the search for name charge fails in the instance namespace. The next namespace checked is for the PredatoryCreditCard class, because that is the true type of the instance. There is a definition for a charge function in that class, and so that is the one that is called.

In the last case shown, notice that the existence of a charge function in the PredatoryCreditCard class has the effect of **overriding** the version of that function that exists in the CreditCard namespace. In traditional object-oriented terminology, Python uses what is known as **dynamic dispatch** (or **dynamic binding**) to determine, at run-time, which implementation of a function to call based upon the type of the object upon which is invoked. This is in contrast to some languages that use **static dispatching**, making a compile-time decision as to which version of a function to call, based upon the declared type of a variable.

2.6 Shallow and Deep Copying

In Chapter 1, we emphasized that an assignment statement foo = bar makes the name foo an **alias** for the object identifier as bar. In this section, we consider the task of making a copy of an object, rather than an alias. This is necessary in applications when we want to subsequently modify either the original or the copy in an independent manner.

Consider a scenario in which we manage various lists of colors, with each color represented by an instance of a presumed color class. We let identifier warmtones denote an existing list of such colors (e.g., oranges, browns). In this application, we wish to create a new list named palette, which is a copy of the warmtones list. However, we want to subsequently be able to add additional colors to palette, or to modify or remove some of the existing colors, without affecting the contents of warmtones. If we were to execute the command

```
palette = warmtones
```

this create an alias, as shown in Figure 2-9. No new list is created; instead, the new identifier palette references the original list.

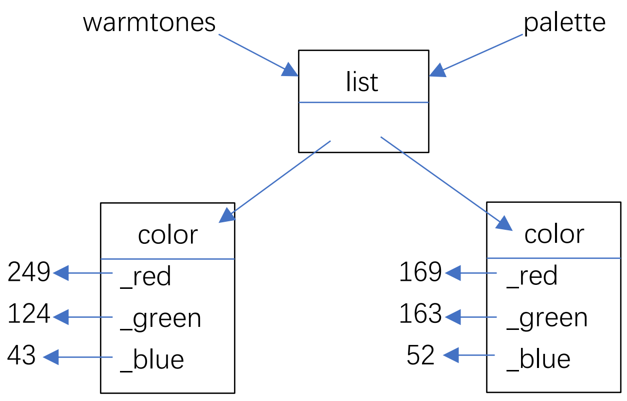


Figure 2-9 Two aliases for the same list of colors.

Unfortunately, this does not meet our desired criteria, because if we subsequently add or remove colors from "palette", we modify the list identified as warmtones.

We can instead create a new instance of the list class by using syntax:

```
palette = list(warmtones)
```

In this case, we explicitly call the list constructor, sending the first list as a parameter. This causes a new list to be created, as shown in Figure 2-10; however, it is what is known as a **shallow copy**. The new list is initialized so that its contents are precisely the same as the original sequence. However, Python's lists are **referential** (see Section 1.2.3), and so the new list represents a sequence of references to the same elements as in the first.

This is a better situation than our first attempt, as we can legitimately add or remove elements from palette without affecting warmtones. However, if we edit a color instance from the palette list, we effectively change the contents of warmtones. Although palette and warmtones are distinct lists, there remains indirect aliasing, for example, with palette[0] and warmtones[0] as aliases for the same color instance.

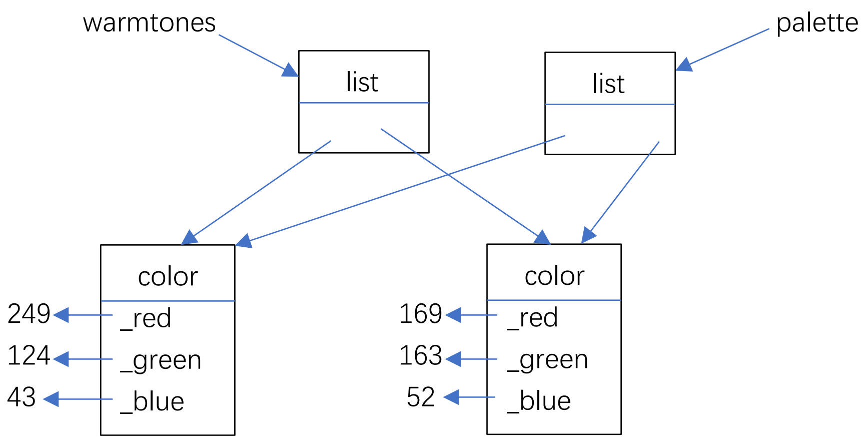


Figure 2-10 A shallow copy of a list of colors.

We prefer that palette be what is known as a deep copy of warmtones. In a deep copy, the new copy references its own copies of those objects referenced by the original version. (see Figure 2-11.)

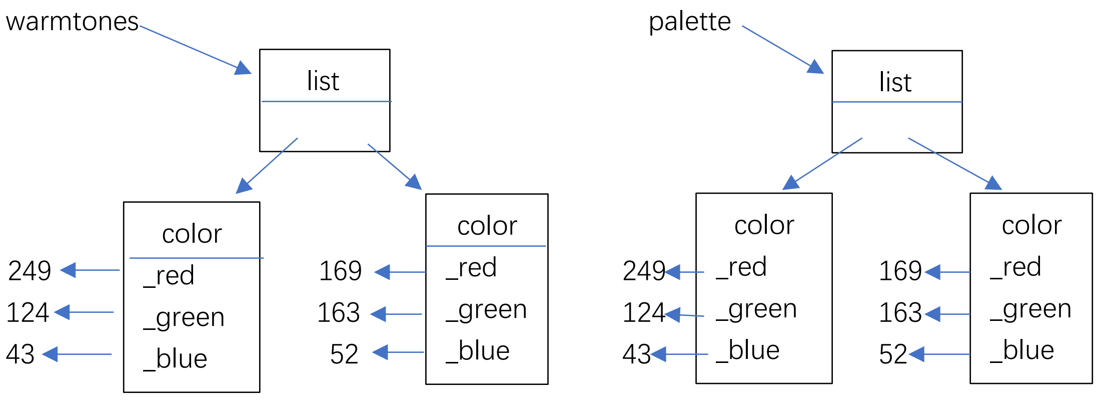


Figure 2-11 A deep copy of a list of colors.

Python's copy Module

To create a deep copy, we could populate our list by explicitly making copies of the original color instances, but this requires that we know how to make copies of colors (rather than aliasing). Python provides a very convenient module, named copy, that can produce both shallow copies and deep copies of arbitrary objects.

This module supports two functions: the copy function creates a shallow copy of its argument, and the deepcopy function creates a deep copy of its argument. After importing the module, we may create a deep copy for our example, as shown in Figure 2-11, using the command:

```
palette = copy.deepcopy(warmtones)
```

2.7 Exercises