CHAPTER 1

Abstract Data Types

The foundation of computer science is based on the study of algorithms. An ***algorithm*** is a sequence of clear and precise step-by-step instructions for solving a problem in a finite amount of time. Algorithms are implemented by translating the step-by-step instructions into a ***computer program*** that can be executed by a computer. This translation process is called ***computer programming*** or simply ***programming***. Computer programs are constructed using a ***programming language*** appropriate to the problem. While programming is an important part of computer science, computer science is not the study of programming. Nor is it about learning a particular programming language. Instead, programming and programming languages are tools used by computer scientists to solve problems.

1.1 Introduction

Data items are represented within a computer as a sequence of binary digits. These sequences can appear very similar but have different meanings since computers can store and manipulate different types of data. For example, the binary sequence `01001100110010110101110011011100` could be a string of characters, an integer value, or a real value. To distinguish between the different types of data, the term ***type*** is often used to refer to a collection of values and the term ***data type*** to refer to a given type along with a collection of operations for manipulating values of the given type.

Programming languages commonly provide data types as part of the language itself. These data types, known as ***primitives***, come in two categories: simple and complex. The ***simple data types*** consist of values that are in the most basic form and cannot be decomposed into smaller parts. Integer and real types, for example, consist of single numeric values. The ***complex data types***, on the other hand, are constructed of multiple components consisting of simple types or other complex types. In Python, objects, strings, lists, and dictionaries, which can contain multiple values, are all examples of complex types. The primitive types provided by a language may not be sufficient for solving large complex problems. Thus, most languages allow for the construction of additional data types, known as ***user-defined types*** since they are defined by the programmer and not the language. Some of these data types can themselves be very complex.

1.1.1 Abstractions

To help manage complex problems and complex data types, computer scientists typically work with abstractions. An ***abstraction*** is a mechanism for separating the properties of an object and restricting the focus to those relevant in the current context. The user of the abstraction does not have to understand all of the details in order to utilize the object, but only those relevant to the current task or problem.

Two common types of abstractions encountered in computer science are procedural, or functional, abstraction and data abstraction. ***Procedural abstraction*** is the use of a function or method knowing what it does but ignoring how it's accomplished. Consider the mathematical square root function which you have probably used at some point. You know the function will compute the square root of a given number, but do you know how the square root is computed? Does it matter if you know how it is computed, or is simply knowing how to correctly use the function sufficient? ***Data abstraction*** is the separation of the properties of a data type (its values and operations) from the implementation of that data type. You have used strings in Python many times. But do you know how they are implemented? That is, do you know how the data is structured internally or how the various operations are implemented?

Typically, abstractions of complex problems occur in layers, with each higher layer adding more abstraction than the previous. Consider the problem of representing integer values on computers and performing arithmetic operations on those values. Figure 1.1 illustrates the common levels of abstractions used with integer arithmetic. At the lowest level is the hardware with little to no abstraction since it includes binary representations of the values and logic circuits for performing the arithmetic. Hardware designers would deal with integer arithmetic at this level and be concerned with its correct implementation. A higher level of abstraction for integer values and arithmetic is provided through assembly language, which involves working with binary values and individual instructions corresponding to the underlying hardware. Compiler writers and assembly language programmers would work with integer arithmetic at this level and must ensure the proper selection of assembly language instructions to compute a given mathematical expression. For example, suppose we wish to compute $x = a + b - 5$. At the assembly language level, this expression must be split into multiple instructions for loading the values from memory, storing them into registers, and then performing each arithmetic operation separately, as shown in the following psuedocode:

```
loadFromMem( R1, 'a' )
loadFromMem( R2, 'b' )
```

```
add R0, R1, R2
sub R0, R0, 5
storeToMem( R0, 'x' )
```

To avoid this level of complexity, high-level programming languages add another layer of abstraction above the assembly language level. This abstraction is provided through a primitive data type for storing integer values and a set of well-defined operations that can be performed on those values. By providing this level of abstraction, programmers can work with variables storing decimal values and specify mathematical expressions in a more familiar notation ($x = a + b - 5$) than is possible with assembly language instructions. Thus, a programmer does not need to know the assembly language instructions required to evaluate a mathematical expression or understand the hardware implementation in order to use integer arithmetic in a computer program.

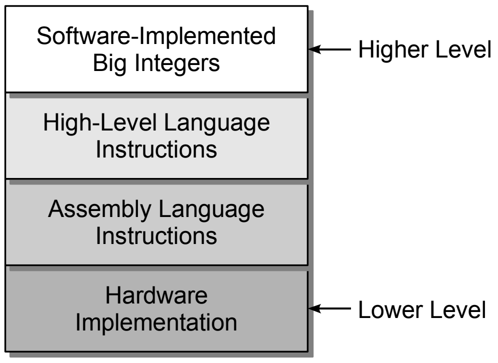

Software-Implemented Big Integers
Higher Level
High-Level Language Instructions
Assembly Language Instructions
Hardware Implementation
Lower Level

Figure 1.1: Levels of abstraction used with integer arithmetic.

One problem with the integer arithmetic provided by most high-level languages and in computer hardware is that it works with values of a limited size. On 32-bit architecture computers, for example, signed integer values are limited to the range $-2^{31} \ldots (2^{31} - 1)$. What if we need larger values? In this case, we can provide long or "big integers" implemented in software to allow values of unlimited size. This would involve storing the individual digits and implementing functions or methods for performing the various arithmetic operations. The implementation of the operations would use the primitive data types and instructions provided by the high-level language. Software libraries that provide big integer implementations are available for most common programming languages. Python, however, actually provides software-implemented big integers as part of the language itself.

1.1.2 Abstract Data Types

An ***abstract data type*** (or ***ADT***) is a programmer-defined data type that specifies a set of data values and a collection of well-defined operations that can be performed on those values. Abstract data types are defined independent of their implementation, allowing us to focus on the use of the new data type instead of how it's implemented. This separation is typically enforced by requiring interaction with the abstract data type through an ***interface*** or defined set of operations. This is known as ***information hiding***. By hiding the implementation details and requiring ADTs to be accessed through an interface, we can work with an abstraction and focus on what functionality the ADT provides instead of how that functionality is implemented.

Abstract data types can be viewed like black boxes as illustrated in Figure 1.2. User programs interact with instances of the ADT by invoking one of the several operations defined by its interface. The set of operations can be grouped into four categories:

- ***Constructors***: Create and initialize new instances of the ADT.
- ***Accessors***: Return data contained in an instance without modifying it.
- ***Mutators***: Modify the contents of an ADT instance.
- ***Iterators***: Process individual data components sequentially.

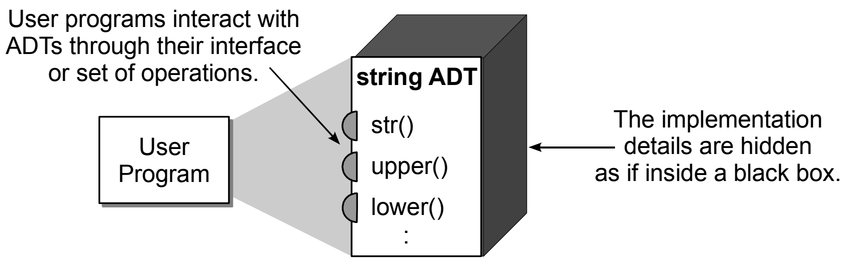

User programs interact with ADTs through their interface or set of operations.
User Program
string ADT
str()
upper()
lower()
:
The implementation details are hidden as if inside a black box.

Figure 1.2: Separating the ADT definition from its implementation.

The implementation of the various operations are hidden inside the black box, the contents of which we do not have to know in order to utilize the ADT. There are several advantages of working with abstract data types and focusing on the "what" instead of the "how."

- *We can focus on solving the problem at hand instead of getting bogged down in the implementation details.* For example, suppose we need to extract a collection of values from a file on disk and store them for later use in our program. If we focus on the implementation details, then we have to worry about what type of storage structure to use, how it should be used, and whether it is the most efficient choice.

- *We can reduce logical errors that can occur from accidental misuse of storage structures and data types by preventing direct access to the implementation.* If we used a list to store the collection of values in the previous example, there is the opportunity to accidentally modify its contents in a part of our code where it was not intended. This type of logical error can be difficult to track down. By using ADTs and requiring access via the interface, we have fewer access points to debug.

- *The implementation of the abstract data type can be changed without having to modify the program code that uses the ADT.* There are many times when we discover the initial implementation of an ADT is not the most efficient or we need the data organized in a different way. Suppose our initial approach to the previous problem of storing a collection of values is to simply append new values to the end of the list. What happens if we later decide the items should be arranged in a different order than simply appending them to the end? If we are accessing the list directly, then we will have to modify our code at every point where values are added and make sure they are not rearranged in other places. By requiring access via the interface, we can easily "swap out" the black box with a new implementation with no impact on code segments that use the ADT.

- *It's easier to manage and divide larger programs into smaller modules, allowing different members of a team to work on the separate modules.* Large programming projects are commonly developed by teams of programmers in which the workload is divided among the members. By working with ADTs and agreeing on their definition, the team can better ensure the individual modules will work together when all the pieces are combined. Using our previous example, if each member of the team directly accessed the list storing the collection of values, they may inadvertently organize the data in different ways or modify the list in some unexpected way. When the various modules are combined, the results may be unpredictable.

1.1.3 Data Structures

Working with abstract data types, which separate the definition from the implementation, is advantageous in solving problems and writing programs. At some point, however, we must provide a concrete implementation in order for the program to execute. ADTs provided in language libraries, like Python, are implemented by the maintainers of the library. When you define and create your own abstract data types, you must eventually provide an implementation. The choices you make in implementing your ADT can affect its functionality and efficiency.

Abstract data types can be simple or complex. A ***simple ADT*** is composed of a single or several individually named data fields such as those used to represent a date or rational number. The ***complex ADTs*** are composed of a collection of data values such as the Python list or dictionary. Complex abstract data types are implemented using a particular ***data structure***, which is the physical representation of how data is organized and manipulated. Data structures can be characterized by how they store and organize the individual data elements and what operations are available for accessing and manipulating the data.

There are many common data structures, including arrays, linked lists, stacks, queues, and trees, to name a few. All data structures store a collection of values, but differ in how they organize the individual data items and by what operations can be applied to manage the collection. The choice of a particular data structure depends on the ADT and the problem at hand. Some data structures are better suited to particular problems. For example, the queue structure is perfect for implementing a printer queue, while the B-Tree is the better choice for a database index. No matter which data structure we use to implement an ADT, by keeping the implementation separate from the definition, we can use an abstract data type within our program and later change to a different implementation, as needed, without having to modify our existing code.

1.1.4 General Definitions

There are many different terms used in computer science. Some of these can have different meanings among the various textbooks and programming languages. To aide the reader and to avoid confusion, we define some of the common terms we will be using throughout the text.

A ***collection*** is a group of values with no implied organization or relationship between the individual values. Sometimes we may restrict the elements to a specific data type such as a collection of integers or floating-point values.

A ***container*** is any data structure or abstract data type that stores and organizes a collection. The individual values of the collection are known as ***elements*** of the container and a container with no elements is said to be ***empty***. The organization or arrangement of the elements can vary from one container to the next as can the operations available for accessing the elements. Python provides a number of built-in containers, which include strings, tuples, lists, dictionaries, and sets.

A ***sequence*** is a container in which the elements are arranged in linear order from front to back, with each element accessible by position. Throughout the text, we assume that access to the individual elements based on their position within the linear order is provided using the subscript operator. Python provides two immutable sequences, strings and tuples, and one mutable sequence, the list. In the next chapter, we introduce the array structure, which is also a commonly used mutable sequence.

A ***sorted sequence*** is one in which the position of the elements is based on a prescribed relationship between each element and its successor. For example, we can create a sorted sequence of integers in which the elements are arranged in ascending or increasing order from smallest to largest value.

In computer science, the term list is commonly used to refer to any collection with a linear ordering. The ordering is such that every element in the collection, except the first one, has a unique predecessor and every element, except the last one, has a unique successor. By this definition, a sequence is a list, but a list is not necessarily a sequence since there is no requirement that a list provide access to the elements by position. Python, unfortunately, uses the same name for its built-in mutable sequence type, which in other languages would be called an array list or vector abstract data type. To avoid confusion, we will use the term ***list*** to refer to the data type provided by Python and use the terms ***general list*** or ***list structure*** when referring to the more general list structure as defined earlier.

1.2 The Date Abstract Data Type

An abstract data type is defined by specifying the domain of the data elements that compose the ADT and the set of operations that can be performed on that domain. The definition should provide a clear description of the ADT including both its domain and each of its operations as only those operations specified can be performed on an instance of the ADT. Next, we provide the definition of a simple abstract data type for representing a date in the proleptic Gregorian calendar.

1.2.1 Defining the ADT

The Gregorian calendar was introduced in the year 1582 by Pope Gregory XIII to replace the Julian calendar. The new calendar corrected for the miscalculation of the lunar year and introduced the leap year. The official first date of the Gregorian calendar is Friday, October 15, 1582. The proleptic Gregorian calendar is an extension for accommodating earlier dates with the first date on November 24, 4713 BC. This extension simplifies the handling of dates across older calendars and its use can be found in many software applications.

Define **Date ADT**

A *date* represents a single day in the proleptic Gregorian calendar in which the first day starts on November 24, 4713 BC.

- `Date( month, day, year )`: Creates a new `Date` instance initialized to the given Gregorian date which must be valid. Year 1 BC and earlier are indicated by negative year components.
- `day()`: Returns the Gregorian day number of this date.
- `month()`: Returns the Gregorian month number of this date.
- `year()`: Returns the Gregorian year of this date.
- `monthName()`: Returns the Gregorian month name of this date.
- `dayOfWeek()`: Returns the day of the week as a number between 0 and 6 with 0 representing Monday and 6 representing Sunday.
- `numDays( otherDate )`: Returns the number of days as a positive integer between this date and the `otherDate`.
- `isLeapYear()`: Determines if this date falls in a leap year and returns the appropriate boolean value.

- `advanceBy( days )`: Advances the date by the given number of days. The date is incremented if `days` is positive and decremented if `days` is negative. The date is capped to November 24, 4714 BC, if necessary.
- *comparable* `( otherDate )`: Compares this date to the `otherDate` to determine their logical ordering. This comparison can be done using any of the logical operators `<, <=, >, >=, ==, !=`.
- *toString* `()`: Returns a string representing the Gregorian date in the format `mm/dd/yyyy`. Implemented as the Python operator that is automatically called via the `str()` constructor.

The abstract data types defined in the text will be implemented as Python classes. When defining an ADT, we specify the ADT operations as method prototypes. The class constructor, which is used to create an instance of the ADT, is indicated by the name of the class used in the implementation.

Python allows classes to define or overload various operators that can be used more naturally in a program without having to call a method by name. We define all ADT operations as named methods, but implement some of them as operators when appropriate instead of using the named method. The ADT operations that will be implemented as Python operators are indicated in italicized text and a brief comment is provided in the ADT definition indicating the corresponding operator. This approach allows us to focus on the general ADT specification that can be easily translated to other languages if the need arises but also allows us to take advantage of Python's simple syntax in various sample programs.

1.2.2 Using the ADT

To illustrate the use of the Date ADT, consider the program in Listing 1.1, which processes a collection of birth dates. The dates are extracted from standard input and examined. Those dates that indicate the individual is at least 21 years of age based on a target date are printed to standard output. The user is continuously prompted to enter a birth date until zero is entered for the month.

This simple example illustrates an advantage of working with an abstraction by focusing on what functionality the ADT provides instead of how that functionality is implemented. By hiding the implementation details, we can use an ADT independent of its implementation. In fact, the choice of implementation for the Date ADT will have no effect on the instructions in our example program.

NOTE

Class Definitions. Classes are the foundation of object-oriented programing languages and they provide a convenient mechanism for defining and implementing abstract data types. A review of Python classes is provided in Appendix D.

Listing 1.1 The `checkdates.py` program.

```python
# Extracts a collection of birth dates from the user and determines
# if each individual is at least 21 years of age.
from date import Date

def main():
   # Date before which a person must have been born to be 21 or older.
  bornBefore = Date(6, 1, 1988)

   # Extract birth dates from the user and determine if 21 or older.
  date = promptAndExtractDate()
  while date is not None :
    if date <= bornBefore :
      print( "Is at least 21 years of age: ", date )
    date = promptAndExtractDate()

# Prompts for and extracts the Gregorian date components. Returns a
# Date object or None when the user has finished entering dates.
def promptAndExtractDate():
  print( "Enter a birth date." )
  month = int( input("month (0 to quit): ") )
  if month == 0 :
    return None
  else :
    day = int( input("day: ") )
    year = int( input("year: ") )
    return Date( month, day, year )

# Call the main routine.
main()
```

1.2.3 Preconditions and Postconditions

In defining the operations, we must include a specification of required inputs and the resulting output, if any. In addition, we must specify the preconditions and postconditions for each operation. A ***precondition*** indicates the condition or state of the ADT instance and inputs before the operation can be performed. A ***postcondition*** indicates the result or ending state of the ADT instance after the operation is performed. The precondition is assumed to be true while the postcondition is a guarantee as long as the preconditions are met. Attempting to perform an operation in which the precondition is not satisfied should be flagged as an error. Consider the use of the `pop(i)` method for removing a value from a list. When this method is called, the precondition states the supplied index must be within the legal range. Upon successful completion of the operation, the postcondition guarantees the item has been removed from the list. If an invalid index, one that is out of the legal range, is passed to the `pop()` method, an exception is raised.

All operations have at least one precondition, which is that the ADT instance has to have been previously initialized. In an object-oriented language, this precondition is automatically verified since an object must be created and initialized via the constructor before any operation can be used. Other than the initialization requirement, an operation may not have any other preconditions. It all depends on the type of ADT and the respective operation. Likewise, some operations may not have a postcondition, as is the case for simple access methods, which simply return a value without modifying the ADT instance itself. Throughout the text, we do not explicitly state the precondition and postcondition as such, but they are easily identified from the description of the ADT operations.

When implementing abstract data types, it's important that we ensure the proper execution of the various operations by verifying any stated preconditions. The appropriate mechanism when testing preconditions for abstract data types is to test the precondition and raise an exception when the precondition fails. You then allow the user of the ADT to decide how they wish to handle the error, either catch it or allow the program to abort.

Python, like many other object-oriented programming languages, raises an exception when an error occurs. An ***exception*** is an event that can be triggered and optionally handled during program execution. When an exception is raised indicating an error, the program can contain code to catch and gracefully handle the exception; otherwise, the program will abort. Python also provides the `assert` statement, which can be used to raise an `AssertionError` exception. The `assert` statement is used to state what we assume to be true at a given point in the program. If the assertion fails, Python automatically raises an `AssertionError` and aborts the program, unless the exception is caught.

Throughout the text, we use the `assert` statement to test the preconditions when implementing abstract data types. This allows us to focus on the implementation of the ADTs instead of having to spend time selecting the proper exception to raise or creating new exceptions for use with our ADTs. For more information on exceptions and assertions, refer to Appendix C.

1.2.4 Implementing the ADT

After defining the ADT, we need to provide an implementation in an appropriate language. In our case, we will always use Python and class definitions, but any programming language could be used. A partial implementation of the `Date` class is provided in Listing 1.2, with the implementation of some methods left as exercises.

Date Representations

There are two common approaches to storing a date in an object. One approach stores the three components—month, day, and year—as three separate fields. With this format, it is easy to access the individual components, but it's difficult to compare two dates or to compute the number of days between two dates since the number of days in a month varies from month to month. The second approach stores the date as an integer value representing the Julian day, which is the number of days elapsed since the initial date of November 24, 4713 BC (using the Gregorian calendar notation). Given a Julian day number, we can compute any of the three Gregorian components and simply subtract the two integer values to determine which occurs first or how many days separate the two dates. We are going to use the latter approach as it is very common for storing dates in computer applications and provides for an easy implementation.

Listing 1.2 Partial implementation of the `date.py` module.

```
# Implements a proleptic Gregorian calendar date as a Julian day number.

class Date :
   # Creates an object instance for the specified Gregorian date.
  def __init__( self, month, day, year ):
    self._julianDay = 0
    assert self._isValidGregorian( month, day, year ), \
           "Invalid Gregorian date."

     # The first line of the equation, T = (M - 14) / 12, has to be changed
     # since Python's implementation of integer division is not the same
     # as the mathematical definition.
    tmp = 0
    if month < 3 :
      tmp = -1
    self._julianDay = day - 32075 + \
             (1461 * (year + 4800 + tmp) // 4) + \
             (367 * (month - 2 - tmp * 12) // 12) - \
             (3 * ((year + 4900 + tmp) // 100) // 4)

   # Extracts the appropriate Gregorian date component.
  def month( self ):
    return (self._toGregorian())[0]  # returning M from (M, d, y)

  def day( self ):
    return (self._toGregorian())[1]  # returning D from (m, D, y)

  def year( self ):
    return (self._toGregorian())[2]  # returning Y from (m, d, Y)

   # Returns day of the week as an int between 0 (Mon) and 6 (Sun).
  def dayOfWeek( self ):
    month, day, year = self._toGregorian()
    if month < 3 :
      month = month + 12
      year = year - 1
    return ((13 * month + 3) // 5 + day + \
           year + year // 4 - year // 100 + year // 400) % 7

   # Returns the date as a string in Gregorian format.
  def __str__( self ):
    month, day, year = self._toGregorian()
    return "%02d/%02d/%04d" % (month, day, year)

   # Logically compares the two dates.
  def __eq__( self, otherDate ):
    return self._julianDay == otherDate._julianDay

```

(Listing Continued)

Listing 1.2 Continued ...

```
49    def __lt__( self, otherDate ):
50      return self._julianDay < otherDate._julianDay
51
52    def __le__( self, otherDate ):
53      return self._julianDay <= otherDate._julianDay
54
55    # The remaining methods are to be included at this point.
56    # ......
57
58    # Returns the Gregorian date as a tuple: (month, day, year).
59    def _toGregorian( self ):
60      A = self._julianDay + 68569
61      B = 4 * A // 146097
62      A = A - (146097 * B + 3) // 4
63      year = 4000 * (A + 1) // 1461001
64      A = A - (1461 * year // 4) + 31
65      month = 80 * A // 2447
66      day = A - (2447 * month // 80)
67      A = month // 11
68      month = month + 2 - (12 * A)
69      year = 100 * (B - 49) + year + A
70      return month, day, year
```

Constructing the Date

We begin our discussion of the implementation with the constructor, which is shown in lines 5–19 of Listing 1.2. The Date ADT will need only a single attribute to store the Julian day representing the given Gregorian date. To convert a Gregorian date to a Julian day number, we use the following formula[1] where day 0 corresponds to November 24, 4713 BC and all operations involve integer arithmetic.

```
T = (M - 14) / 12
jday = D - 32075 + (1461 * (Y + 4800 + T) / 4) +
                   (367 * (M - 2 - T * 12) / 12) -
                   (3 * ((Y + 4900 + T) / 100) / 4)
```

Before attempting to convert the Gregorian date to a Julian day, we need to verify it's a valid date. This is necessary since the precondition states the supplied Gregorian date must be valid. The `_isValidGregorian()` helper method is used to verify the validity of the given Gregorian date. This helper method, the implementation of which is left as an exercise, tests the supplied Gregorian date components and returns the appropriate boolean value. If a valid date is supplied to the constructor, it is converted to the equivalent Julian day using the equation provided earlier. Note the statements in lines 13–15. The equation for converting a Gregorian date to a Julian day number uses integer arithmetic, but

[1]Seidelmann, P. Kenneth (ed.) (1992). *Explanatory Supplement to the Astronomical Almanac*, Chapter 12, pp. 604—606, University Science Books.

NOTE

Comments. Class definitions and methods should be properly commented to aide the user in knowing what the class and/or methods do. To conserve space, however, classes and methods presented in this book do not routinely include these comments since the surrounding text provides a full explanation.

the equation line `T = (M - 14) / 12` produces an incorrect result in Python due to its implementation of integer division, which is not the same as the mathematical definition. By definition, the result of the integer division `-11/12` is 0, but Python computes this as $\lfloor -11/12.0 \rfloor$ resulting in -1. Thus, we had to modify the first line of the equation to produce the correct Julian day when the month component is greater than 2.

CAUTION

Protected Attributes and Methods. Python does not provide a technique to protect attributes and helper methods in order to prevent their use outside the class definition. In this text, we use identifier names, which begin with a single underscore to flag those attributes and methods that should be considered protected and rely on the user of the class to not attempt a direct access.

The Gregorian Date

To access the Gregorian date components the Julian day must be converted back to Gregorian. This conversion is needed in several of the ADT operations. Instead of duplicating the formula each time it's needed, we create a helper method to handle the conversion as illustrated in lines 59–70 of Listing 1.2.

The `_toGregorian()` method returns a tuple containing the day, month, and year components. As with the conversion from Gregorian to Julian, integer arithmetic operations are used throughout the conversion formula. By returning a tuple, we can call the helper method and use the appropriate component from the tuple for the given Gregorian component access method, as illustrated in lines 22–29.

The `dayOfWeek()` method, shown in lines 32–38, also uses the `_toGregorian()` conversion helper method. We determine the day of the week based on the Gregorian components using a simple formula that returns an integer value between 0 and 6, where 0 represents Monday, 1 represents Tuesday, and so on.

The ***toString*** operation defined by the ADT is implemented in lines 41–43 by overloading Python's `__str__` method. It creates a string representation of a date in Gregorian format. This can be done using the string format operator and supplying the values returned from the conversion helper method. By using Python's `__str__` method, Python automatically calls this method on the object when you attempt to print or convert an object to a string as in the following example:

```
firstDay = Date( 9, 1, 2006 )
print( firstDay )
```

Comparing Date Objects

We can logically compare two `Date` instances to determine their calendar order. When using a Julian day to represent the dates, the date comparison is as simple as comparing the two integer values and returning the appropriate boolean value based on the result of that comparison. The "comparable" ADT operation is implemented using Python's logical comparison operators as shown in lines 46–53 of Listing 1.2. By implementing the methods for the logical comparison operators, instances of the class become ***comparable*** objects. That is, the objects can be compared against each other to produce a logical ordering.

You will notice that we implemented only three of the logical comparison operators. The reason for this is that starting with Python version 3, Python will automatically swap the operands and call the appropriate reflective method when necessary. For example, if we use the expression `a > b` with `Date` objects in our program, Python will automatically swap the operands and call `b < a` instead since the `__lt__` method is defined but not `__gt__`. It will do the same for `a >= b` and `a <= b`. When testing for equality, Python will automatically invert the result when only one of the equality operators (`==` or `!=`) is defined. Thus, we need only define one operator from each of the following pairs to achieve the full range of logical comparisons: `<` or `>`, `<=` or `>=`, and `==` or `!=`. For more information on overloading operators, refer to Appendix D.

TIP

Overloading Operators. User-defined classes can implement methods to define many of the standard Python operators such as `+`, `*`, `%`, and `==`, as well as the standard named operators such as `in` and `not in`. This allows for a more natural use of the objects instead of having to call specific named methods. It can be tempting to define operators for every class you create, but you should limit the definition of operator methods for classes where the specific operator has a meaningful purpose.

1.3 Bags

The Date ADT provided an example of a simple abstract data type. To illustrate the design and implementation of a complex abstract data type, we define the Bag ADT. A ***bag*** is a simple container like a shopping bag that can be used to store a collection of items. The bag container restricts access to the individual items by only defining operations for adding and removing individual items, for determining if an item is in the bag, and for traversing over the collection of items.

1.3.1 The Bag Abstract Data Type

There are several variations of the Bag ADT with the one described here being a simple bag. A grab bag is similar to the simple bag but the items are removed from the bag at random. Another common variation is the counting bag, which includes an operation that returns the number of occurrences in the bag of a given item. Implementations of the grab bag and counting bag are left as exercises.

Define **Bag ADT**

A *bag* is a container that stores a collection in which duplicate values are allowed. The items, each of which is individually stored, have no particular order but they must be comparable.

- `Bag()`: Creates a bag that is initially empty.
- *length*`()`: Returns the number of items stored in the bag. Accessed using the `len()` function.
- *contains*`( item )`: Determines if the given target item is stored in the bag and returns the appropriate boolean value. Accessed using the `in` operator.
- `add( item )`: Adds the given item to the bag.
- `remove( item )`: Removes and returns an occurrence of `item` from the bag. An exception is raised if the element is not in the bag.
- *iterator*`()`: Creates and returns an iterator that can be used to iterate over the collection of items.

You may have noticed our definition of the Bag ADT does not include an operation to convert the container to a string. We could include such an operation, but creating a string for a large collection is time consuming and requires a large amount of memory. Such an operation can be beneficial when debugging a program that uses an instance of the Bag ADT. Thus, it's not uncommon to include the `__str__` operator method for debugging purposes, but it would not typically be used in production software. We will usually omit the inclusion of a `__str__` operator method in the definition of our abstract data types, except in those cases where it's meaningful, but you may want to include one temporarily for debugging purposes.

Examples

Given the abstract definition of the Bag ADT, we can create and use a bag without knowing how it is actually implemented. Consider the following simple example, which creates a bag and asks the user to guess one of the values it contains.

```
myBag = Bag()
myBag.add( 19 )
myBag.add( 74 )
myBag.add( 23 )
myBag.add( 19 )
myBag.add( 12 )

value = int( input("Guess a value contained in the bag.") )
if value in myBag:
  print( "The bag contains the value", value )
else :
  print( "The bag does not contain the value", value )
```

Next, consider the `checkdates.py` sample program from the previous section where we extracted birth dates from the user and determined which ones were for individuals who were at least 21 years of age. Suppose we want to keep the collection of birth dates for later use. It wouldn't make sense to require the user to re-enter the dates multiple times. Instead, we can store the birth dates in a bag as they are entered and access them later, as many times as needed. The Bag ADT is a perfect container for storing objects when the position or order of a specific item does not matter. The following is a new version of the main routine for our birth date checking program from Listing 1.1:

```
#pgm: checkdates2.py (modified main() from checkdates.py)
from linearbag import Bag
from date import Date

def main():
  bornBefore = Date( 6, 1, 1988 )
  bag = Bag()

   # Extract dates from the user and place them in the bag.
  date = promptAndExtractDate()
  while date is not None :
    bag.add( date )
    date = promptAndExtractDate()

   # Iterate over the bag and check the age.
  for date in bag :
    if date <= bornBefore :
      print( "Is at least 21 years of age: ", date )
```

Why a Bag ADT?

You may be wondering, why do we need the Bag ADT when we could simply use the list to store the items? For a small program and a small collection of data, using a list would be appropriate. When working with large programs and multiple team members, however, abstract data types provide several advantages as described earlier in Section 1.1.2. By working with the abstraction of a bag, we can: a) focus on solving the problem at hand instead of worrying about the implementation of the container, b) reduce the chance of introducing errors from misuse of the list since it provides additional operations that are not appropriate for a bag, c) provide better coordination between different modules and designers, and d) easily swap out our current implementation of the Bag ADT for a different, possibly more efficient, version later.

1.3.2 Selecting a Data Structure

The implementation of a complex abstract data type typically requires the use of a data structure for organizing and managing the collection of data items. There are many different structures from which to choose. So how do we know which to use? We have to evaluate the suitability of a data structure for implementing a given abstract data type, which we base on the following criteria:

1. *Does the data structure provide for the storage requirements as specified by the domain of the ADT?* Abstract data types are defined to work with a specific domain of data values. The data structure we choose must be capable of storing all possible values in that domain, taking into consideration any restrictions or limitations placed on the individual items.

2. *Does the data structure provide the necessary data access and manipulation functionality to fully implement the ADT?* The functionality of an abstract data type is provided through its defined set of operations. The data structure must allow for a full and correct implementation of the ADT without having to violate the abstraction principle by exposing the implementation details to the user.

3. *Does the data structure lend itself to an efficient implementation of the operations?* An important goal in the implementation of an abstract data type is to provide an efficient solution. Some data structures allow for a more efficient implementation than others, but not every data structure is suitable for implementing every ADT. Efficiency considerations can help to select the best structure from among multiple candidates.

There may be multiple data structures suitable for implementing a given abstract data type, but we attempt to select the best possible based on the context in which the ADT will be used. To accommodate different contexts, language libraries will commonly provide several implementations of some ADTs, allowing the programmer to choose the most appropriate. Following this approach, we introduce a number of abstract data types throughout the text and present multiple implementations as new data structures are introduced.

The efficiency of an implementation is based on complexity analysis, which is not introduced until later in Chapter 3. Thus, we postpone consideration of the efficiency of an implementation in selecting a data structure until that time. In the meantime, we only consider the suitability of a data structure based on the storage and functional requirements of the abstract data type.

We now turn our attention to selecting a data structure for implementing the Bag ADT. The possible candidates at this point include the list and dictionary structures. The list can store any type of comparable object, including duplicates. Each item is stored individually, including duplicates, which means the reference to each individual object is stored and later accessible when needed. This satisfies the storage requirements of the Bag ADT, making the list a candidate structure for its implementation.

The dictionary stores key/value pairs in which the key component must be comparable and unique. To use the dictionary in implementing the Bag ADT, we must have a way to store duplicate items as required by the definition of the abstract data type. To accomplish this, each unique item can be stored in the key part of the key/value pair and a counter can be stored in the value part. The counter would be used to indicate the number of occurrences of the corresponding item in the bag. When a duplicate item is added, the counter is incremented; when a duplicate is removed, the counter is decremented.

Both the list and dictionary structures could be used to implement the Bag ADT. For the simple version of the bag, however, the list is a better choice since the dictionary would require twice as much space to store the contents of the bag in the case where most of the items are unique. The dictionary is an excellent choice for the implementation of the counting bag variation of the ADT.

Having chosen the list, we must ensure it provides the means to implement the complete set of bag operations. When implementing an ADT, we must use the functionality provided by the underlying data structure. Sometimes, an ADT operation is identical to one already provided by the data structure. In this case, the implementation can be quite simple and may consist of a single call to the corresponding operation of the structure, while in other cases, we have to use multiple operations provided by the structure. To help verify a correct implementation of the Bag ADT using the list, we can outline how each bag operation will be implemented:

- An empty bag can be represented by an empty list.
- The size of the bag can be determined by the size of the list.
- Determining if the bag contains a specific item can be done using the equivalent list operation.
- When a new item is added to the bag, it can be appended to the end of the list since there is no specific ordering of the items in a bag.
- Removing an item from the bag can also be handled by the equivalent list operation.
- The items in a list can be traversed using a `for` loop and Python provides for user-defined iterators that be used with a bag.

From this itemized list, we see that each Bag ADT operation can be implemented using the available functionality of the list. Thus, the list is suitable for implementing the bag.

1.3.3 List-Based Implementation

The implementation of the Bag ADT using a list is shown in Listing 1.3. The constructor defines a single data field, which is initialized to an empty list. This corresponds to the definition of the constructor for the Bag ADT in which the container is initially created empty. A sample instance of the `Bag` class created from the example `checkdates2.py` program provided earlier is illustrated in Figure 1.3.

Listing 1.3 The `linearbag.py` module.

```python
# Implements the Bag ADT container using a Python list.
class Bag :
   # Constructs an empty bag.
  def __init__( self ):
    self._theItems = list()

   # Returns the number of items in the bag.
  def __len__( self ):
    return len( self._theItems )

   # Determines if an item is contained in the bag.
  def __contains__( self, item ):
    return item in self._theItems

   # Adds a new item to the bag.
  def add( self, item ):
    self._theItems.append( item )

   # Removes and returns an instance of the item from the bag.
  def remove( self, item ):
    assert item in self._theItems, "The item must be in the bag."
    ndx = self._theItems.index( item )
    return self._theItems.pop( ndx )

   # Returns an iterator for traversing the list of items.
  def __iter__( self, item ):
    ......
```

Most of the implementation details follow the specifics discussed in the previous section. There are some additional details, however. First, the ADT definition of the `remove()` operation specifies the precondition that the item must exist in the bag in order to be removed. Thus, we must first assert that condition and verify the existence of the item. Second, we need to provide an iteration mechanism that allows us to iterate over the individual items in the bag. We delay

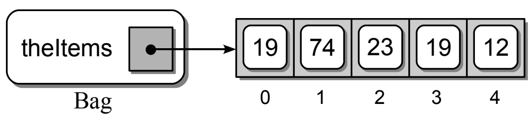
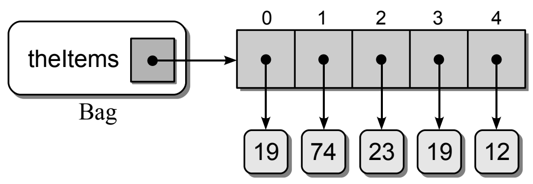


Figure 1.3: Sample instance of the `Bag` class implemented using a list.

the implementation of this operation until the next section where we discuss the creation and use of iterators in Python.

A list stores references to objects and technically would be illustrated as shown in the figure to the right. To conserve space and reduce the clutter that can result in some figures, however, we illustrate objects in the text as boxes with rounded edges and show them stored directly within the list structure. Variables will be illustrated as square boxes with a bullet in the middle and the name of the variable printed nearby.



1.4 Iterators

Traversals are very common operations, especially on containers. A traversal iterates over the entire collection, providing access to each individual element. Traversals can be used for a number of operations, including searching for a specific item or printing an entire collection.

Python's container types—strings, tuples, lists, and dictionaries—can be traversed using the `for` loop construct. For our user-defined abstract data types, we can add methods that perform specific traversal operations when necessary. For example, if we wanted to save every item contained in a bag to a text file, we could add a `saveElements()` method that traverses over the vector and writes each value to a file. But this would limit the format of the resulting text file to that specified in the new method. In addition to saving the items, perhaps we would like to simply print the items to the screen in a specific way. To perform the latter, we would have to add yet another operation to our ADT.

Not all abstract data types should provide a traversal operation, but it is appropriate for most container types. Thus, we need a way to allow generic traversals to be performed. One way would be to provide the user with access to the underlying data structure used to implement the ADT. But this would violate the abstraction principle and defeat the purpose of defining new abstract data types.

Python, like many of today's object-oriented languages, provides a built-in iterator construct that can be used to perform traversals on user-defined ADTs. An ***iterator*** is an object that provides a mechanism for performing generic traversals through a container without having to expose the underlying implementation. Iterators are used with Python's `for` loop construct to provide a traversal mechanism for both built-in and user-defined containers. Consider the code segment from the `checkdates2.py` program in Section 1.3 that uses the `for` loop to traverse the collection of dates:

```
 # Iterate over the bag and check the ages.
for date in bag :
  if date <= bornBefore :
    print( "Is at least 21 years of age: ", date )
```

1.4.1 Designing an Iterator

To use Python's traversal mechanism with our own abstract data types, we must define an iterator class, which is a class in Python containing two special methods, `__iter__` and `__next__`. Iterator classes are commonly defined in the same module as the corresponding container class.

The implementation of the `_BagIterator` class is shown in Listing 1.4. The constructor defines two data fields. One is an alias to the list used to store the items in the bag, and the other is a loop index variable that will be used to iterate over that list. The loop variable is initialized to zero in order to start from the beginning of the list. The `__iter__` method simply returns a reference to the object itself and is always implemented to do so.

Listing 1.4 The _BagIterator class, which is part of the `linearbag.py` module.

```python
# An iterator for the Bag ADT implemented as a Python list.
class _BagIterator :
  def __init__( self, theList ):
    self._bagItems = theList
    self._curItem = 0

  def __iter__( self ):
    return self

  def __next__( self ):
    if self._curItem < len( self._bagItems ) :
      item = self._bagItems[ self._curItem ]
      self._curItem += 1
      return item
    else :
      raise StopIteration
```

The `__next__` method is called to return the next item in the container. The method first saves a reference to the current item indicated by the loop variable. The loop variable is then incremented by one to prepare it for the next invocation of the `__next__` method. If there are no additional items, the method must raise a `StopIteration` exception that flags the `for` loop to terminate. Finally, we must add an `__iter__` method to our `Bag` class, as shown here:

```python
def __iter__( self ):
  return _BagIterator( self._theItems )
```

This method, which is responsible for creating and returning an instance of the `_BagIterator` class, is automatically called at the beginning of the `for` loop to create an iterator object for use with the loop construct.

1.4.2 Using Iterators

With the definition of the _BagIterator class and the modifications to the Bag class, we can now use Python's for loop with a Bag instance. When the for loop

```
for item in bag :
  print( item )
```

is executed, Python automatically calls the __iter__ method on the bag object to create an iterator object. Figure 1.4 illustrates the state of the _BagIterator object immediately after being created. Notice the _bagItems field of the iterator object references _theItems field of the bag object. This reference was assigned by the constructor when the _BagIterator object was created.

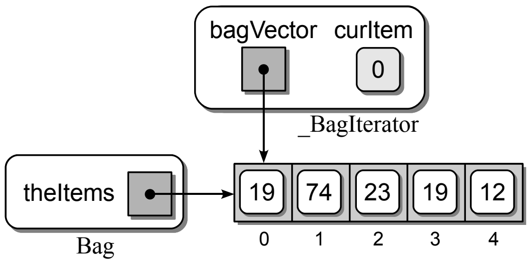


Figure 1.4: The Bag and _BagIterator objects before the first loop iteration.

The for loop then automatically calls the __next__ method on the iterator object to access the next item in the container. The state of the iterator object changes with the _curItem field having been incremented by one. This process continues until a StopIteration exception is raised by the __next__ method when the items have been exhausted as indicated by the _curItem. After all of the items have been processed, the iteration is terminated and execution continues with the next statement following the loop. The following code segment illustrates how Python actually performs the iteration when a for loop is used with an instance of the Bag class:

```
 # Create a BagIterator object for myBag.
iterator = myBag.__iter__()

 # Repeat the while loop until break is called.
while True :
  try:
     # Get the next item from the bag. If there are no
     # more items, the StopIteration exception is raised.
    item = iterator.__next__()
     # Perform the body of the for loop.
    print( item )

   # Catch the exception and break from the loop when we are done.
  except StopIteration:
    break
```

1.5 Application: Student Records

Most computer applications are written to process and manipulate data that is stored external to the program. Data is commonly extracted from files stored on disk, from databases, and even from remote sites through web services. For example, suppose we have a collection of records stored on disk that contain information related to students at Smalltown College. We have been assigned the task to extract this information and produce a report similar to the following in which the records are sorted by identification number.

```
                    LIST OF STUDENTS

ID     NAME                       CLASS       GPA
-----  -------------------------  ----------  ----
10015  Smith, John                Sophomore   3.01
10167  Jones, Wendy               Junior      2.85
10175  Smith, Jane                Senior      3.92
10188  Wales, Sam                 Senior      3.25
10200  Roberts, Sally             Freshman    4.00
10208  Green, Patrick             Freshman    3.95
10226  Nelson, Amy                Sophomore   2.95
10334  Roberts, Jane              Senior      3.81
10387  Taylor, Susan              Sophomore   2.15
10400  Logan, Mark                Junior      3.33
10485  Brown, Jessica             Sophomore   2.91
--------------------------------------------------
Number of students: 11
```

Our contact in the Registrar's office, who assigned the task, has provided some information about the data. We know each record contains five pieces of information for an individual student: (1) the student's id number represented as an integer; (2) their first and last names, which are strings; (3) an integer classification code in the range $[1 \ldots 4]$ that indicates if the student is a freshman, sophomore, junior, or senior; and (4) their current grade point average represented as a floating-point value. What we have not been told, however, is how the data is stored on disk. It could be stored in a plain text file, in a binary file, or even in a database. In addition, if the data is stored in a text or binary file, we will need to know how the data is formatted in the file, and if it's in a relational database, we will need to know the type and the structure of the database.

1.5.1 Designing a Solution

Even though we have not yet been told the type of file or the format used to store the data, we can begin designing and implementing a solution by working with an abstraction of the input source. No matter the source or format of the data, the extraction of data records from external storage requires similar steps: open a connection, extract the individual records, then close the connection. To aide in our effort, we define a Student File Reader ADT to represent the extraction of data from an external file or database. In computer programming, an object used to input data into a program is sometimes referred to as a ***reader*** while an object used to output data is referred to as a ***writer***.

Define **Student File Reader ADT**

A *student file reader* is used to extract student records from external storage. The five data components of the individual records are extracted and stored in a storage object specific for this collection of student records.

- `StudentFileReader( filename )`: Creates a student reader instance for extracting student records from the given file. The type and format of the file is dependent on the specific implementation.
- `open()`: Opens a connection to the input source and prepares it for extracting student records. If a connection cannot be opened, an exception is raised.
- `close()`: Closes the connection to the input source. If the connection is not currently open, an exception is raised.
- `fetchRecord()`: Extracts the next student record from the input source and returns a reference to a storage object containing the data. `None` is returned when there are no additional records to be extracted. An exception is raised if the connection to the input source was previously closed.
- `fetchAll()`: The same as `fetchRecord()`, but extracts all student records (or those remaining) from the input source and returns them in a Python list.

Creating the Report

The program in Listing 1.5 uses the Student File Reader ADT to produce the sample report illustrated earlier. The program extracts the student records from the input source, sorts the records by student identification number, and produces the report. This program illustrates some of the advantages of applying abstraction to problem solving by focusing on the “what” instead of the “how.”

By using the Student File Reader ADT, we are able to design a solution and construct a program for the problem at hand without knowing exactly how the data is stored in the external source. We import the `StudentFileReader` class from the `studentfile.py` module, which we assume will be an implementation of the ADT that handles the actual data extraction. Further, if we want to use this same program with a data file having a different format, the only modifications required will be to indicate a different module in the `import` statement and possibly a change to the filename specified by the constant variable `FILE_NAME`.

The `studentreport.py` program consists of two functions: `printReport()` and `main()`. The main routine uses an instance of the ADT to connect to the external source in order to extract the student records into a list. The list of records is then

Listing 1.5 The `studentreport.py` program.

```
# Produces a student report from data extracted from an external source.
from studentfile import StudentFileReader

# Name of the file to open.
FILE_NAME = "students.txt"

def main():
  # Extract the student records from the given text file.
  reader = StudentFileReader( FILE_NAME )
  reader.open()
  studentList = reader.fetchAll()
  reader.close()

  # Sort the list by id number. Each object is passed to the lambda
  # expression which returns the idNum field of the object.
  studentList.sort( key = lambda rec: rec.idNum )

  # Print the student report.
  printReport( studentList )

# Prints the student report.
def printReport( theList ):
  # The class names associated with the class codes.
  classNames = ( None, "Freshman", "Sophomore", "Junior", "Senior" )

  # Print the header.
  print( "LIST OF STUDENTS".center(50) )
  print( "" )
  print( "%-5s  %-25s  %-10s  %-4s" % ('ID', 'NAME', 'CLASS', 'GPA' ) )
  print( "%5s  %25s  %10s  %4s" % ('-' * 5, '-' * 25, '-' * 10, '-' * 4))
  # Print the body.
  for record in theList :
    print( "%5d  %-25s  %-10s  %4.2f" % \
                  (record.idNum, \
                   record.lastName + ', ' + record.firstName,
                   classNames[record.classCode], record.gpa) )
  # Add a footer.
  print( "-" * 50 )
  print( "Number of students:", len(theList) )

# Executes the main routine.
main()
```

sorted in ascending order based on the student identification number. The actual report is produced by passing the sorted list to the `printReport()` function.

Storage Class

When the data for an individual student is extracted from the input file, it will need to be saved in a storage object that can be added to a list in order to first sort and then print the records. We could use tuples to store the records, but we avoid the use of tuples when storing structured data since it's better practice to use classes with named fields. Thus, we define the `StudentRecord` class

```
class StudentRecord :
  def __init__( self ):
    self.idNum = 0
    self.firstName = None
    self.lastName = None
    self.classCode = 0
    self.gpa = 0.0
```

to store the data related to an individual student. You may notice there is only a constructor with no additional methods. This is a complete class as defined and represents a ***storage class***. The constructor is all that's needed to define the two data fields for storing the two component values.

Storage classes should be defined within the same module as the class with which they will be used. For this application, the `StudentRecord` class is defined at the end of the `studentfile.py` module. Some storage classes may be intended for internal use by a specific class and not meant to be accessed from outside the module. In those cases, the name of the storage class will begin with a single underscore, which flags it as being private to the module in which it's defined. The `StudentRecord` class, however, has not been defined as being private to the module since instances of the storage class are not confined to the ADT but instead are returned to the client code by methods of the `StudentFileReader` class. The storage class can be imported along with the `StudentFileReader` class when needed.

You will note the data fields in the storage class are public (by our notation) since their names do not begin with an underscore as they have been in other classes presented earlier. The reason we do not include a restrictive interface for accessing the data fields is that storage objects are meant to be used exclusively for storing data and not as an instance of some abstract data type. Given their limited use, we access the data fields directly as needed.

1.5.2 Implementation

The implementation of the Student File Reader ADT does not require a data structure since it does not store data but instead extracts data from an external source. The ADT has to be implemented to extract data based on the format in which the data is stored. For this example, we are going to extract the data from

CAUTION

⚠ **Python Tuples.** The tuple can be used to store structured data, with each element corresponding to an individual data field. This is not good practice, however, since the elements are not named and you would have to remember what piece of data is stored in each element. A better practice is to use objects with named data fields. In this book, we limit the use of tuples for returning multiple values from methods and functions.

a text file in which the records are listed one after the other. The five fields of the record are each stored on a separate line. The first line contains the id number, the second and third contain the first and last names, the fourth line contains the classification code, and the grade point average follows on the fifth line. The following text block illustrates the format for a file containing two records:

```
10015
John
Smith
2
3.01
10334
Jane
Roberts
4
3.81
```

Listing 1.6 provides the implementation of the ADT for extracting the records from the text file in the given format. The constructor simply initializes an instance of the class by creating two attributes, one to store the name the text file and the other to store a reference to the file object after it's opened. The `open()` method is responsible for opening the input file using the name saved in the constructor. The resulting file object is saved in the `inputFile` attribute so it can be used in the other methods. After the records are extracted, the file is closed by calling the `close()` method.

Listing 1.6 The `studentfile.py` module.

```
# Implementation of the StudentFileReader ADT using a text file as the
# input source in which each field is stored on a separate line.

class StudentFileReader :
   # Create a new student reader instance.
  def __init__( self, inputSrc ):
    self._inputSrc = inputSrc
    self._inputFile = None

   # Open a connection to the input file.
  def open( self ):
    self._inputFile = open( self._inputSrc, "r" )

   # Close the connection to the input file.
  def close( self ):
    self._inputFile.close()
    self._inputFile = None

   # Extract all student records and store them in a list.
  def fetchAll( self ):
    theRecords = list()
    student = self.fetchRecord()
```

(Listing Continued)

Listing 1.6 Continued . . .

```
23     while student != None :
24       theRecords.append( student )
25       student = self.fetchRecord()
26     return theRecords
27
28    # Extract the next student record from the file.
29   def fetchRecord( self ):
30      # Read the first line of the record.
31     line = self._inputFile.readline()
32     if line == "" :
33       return None
34
35      # If there is another record, create a storage object and fill it.
36     student = StudentRecord()
37     student.idNum = int( line )
38     student.firstName = self._inputFile.readline().rstrip()
39     student.lastName = self._inputFile.readline().rstrip()
40     student.classCode = int( self._inputFile.readline() )
41     student.gpa = float( self._inputFile.readline() )
42     return student
43
44 # Storage class used for an individual student record.
45 class StudentRecord :
46   def __init__( self ):
47     self.idNum = 0
48     self.firstName = None
49     self.lastName = None
50     self.classCode = 0
51     self.gpa = 0.0
```

The `fetchAll()` method, at lines 20–26, is a simple event-controlled loop that builds and returns a list of `StudentRecord` objects. This is done by repeatedly calling the `fetchRecord()` method. Thus, the actual extraction of a record from the text file is handled by the `fetchRecord()` method, as shown in lines 29–42. To extract the student records from a file in which the data is stored in a different format, we need only modify this method to accommodate the new format.

The Student File Reader ADT provides a framework that can be used to extract any type of records from a text file. The only change required would be in the `fetchRecord()` method to create the appropriate storage object and to extract the data from the file in the given format.

Exercises

1.1 Complete the partial implementation of the `Date` class by implementing the remaining methods: `monthName()`, `isLeapYear()`, `numDays()`, `advanceBy()`,