Projects

P-1.29 Write a Python program that outputs all possible strings formed by using the characters `'c'`, `'a'`, `'t'`, `'d'`, `'o'`, and `'g'` exactly once.

P-1.30 Write a Python program that can take a positive integer greater than 2 as input and write out the number of times one must repeatedly divide this number by 2 before getting a value less than 2.

P-1.31 Write a Python program that can "make change." Your program should take two numbers as input, one that is a monetary amount charged and the other that is a monetary amount given. It should then return the number of each kind of bill and coin to give back as change for the difference between the amount given and the amount charged. The values assigned to the bills and coins can be based on the monetary system of any current or former government. Try to design your program so that it returns as few bills and coins as possible.

P-1.32 Write a Python program that can simulate a simple calculator, using the console as the exclusive input and output device. That is, each input to the calculator, be it a number, like `12.34` or `1034`, or an operator, like `+` or `=`, can be done on a separate line. After each such input, you should output to the Python console what would be displayed on your calculator.

P-1.33 Write a Python program that simulates a handheld calculator. Your program should process input from the Python console representing buttons that are "pushed," and then output the contents of the screen after each operation is performed. Minimally, your calculator should be able to process the basic arithmetic operations and a reset/clear operation.

P-1.34 A common punishment for school children is to write out a sentence multiple times. Write a Python stand-alone program that will write out the following sentence one hundred times: "I will never spam my friends again." Your program should number each of the sentences and it should make eight different random-looking typos.

P-1.35 The ***birthday paradox*** says that the probability that two people in a room will have the same birthday is more than half, provided n, the number of people in the room, is more than 23. This property is not really a paradox, but many people find it surprising. Design a Python program that can test this paradox by a series of experiments on randomly generated birthdays, which test this paradox for $n = 5, 10, 15, 20, \ldots, 100$.

P-1.36 Write a Python program that inputs a list of words, separated by whitespace, and outputs how many times each word appears in the list. You need not worry about efficiency at this point, however, as this topic is something that will be addressed later in this book.

5.3 Dynamic Arrays and Amortization

When creating a low-level array in a computer system, the precise size of that array must be explicitly declared in order for the system to properly allocate a consecutive piece of memory for its storage. For example, Figure 5.11 displays an array of 12 bytes that might be stored in memory locations 2146 through 2157.

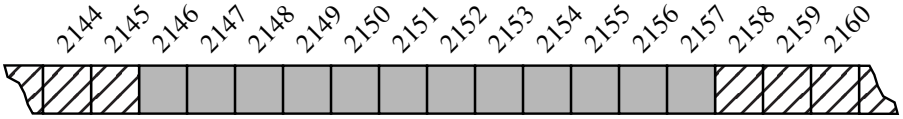

Figure 5.11: An array of 12 bytes allocated in memory locations 2146 through 2157.

Because the system might dedicate neighboring memory locations to store other data, the capacity of an array cannot trivially be increased by expanding into subsequent cells. In the context of representing a Python `tuple` or `str` instance, this constraint is no problem. Instances of those classes are immutable, so the correct size for an underlying array can be fixed when the object is instantiated.

Python's `list` class presents a more interesting abstraction. Although a list has a particular length when constructed, the class allows us to add elements to the list, with no apparent limit on the overall capacity of the list. To provide this abstraction, Python relies on an algorithmic sleight of hand known as a ***dynamic array***.

The first key to providing the semantics of a dynamic array is that a list instance maintains an underlying array that often has greater capacity than the current length of the list. For example, while a user may have created a list with five elements, the system may have reserved an underlying array capable of storing eight object references (rather than only five). This extra capacity makes it easy to append a new element to the list by using the next available cell of the array.

If a user continues to append elements to a list, any reserved capacity will eventually be exhausted. In that case, the class requests a new, larger array from the system, and initializes the new array so that its prefix matches that of the existing smaller array. At that point in time, the old array is no longer needed, so it is reclaimed by the system. Intuitively, this strategy is much like that of the hermit crab, which moves into a larger shell when it outgrows its previous one.

We give empirical evidence that Python's `list` class is based upon such a strategy. The source code for our experiment is displayed in Code Fragment 5.1, and a sample output of that program is given in Code Fragment 5.2. We rely on a function named `getsizeof` that is available from the `sys` module. This function reports the number of bytes that are being used to store an object in Python. For a list, it reports the number of bytes devoted to the array and other instance variables of the list, but *not* any space devoted to elements referenced by the list.

Python's os Module

To provide a Python implementation of a recursive algorithm for computing disk usage, we rely on Python's os module, which provides robust tools for interacting with the operating system during the execution of a program. This is an extensive library, but we will only need the following four functions:

- **os.path.getsize(path)**
  Return the immediate disk usage (measured in bytes) for the file or directory that is identified by the string path (e.g., `/user/rt/courses`).
- **os.path.isdir(path)**
  Return True if entry designated by string path is a directory; False otherwise.
- **os.listdir(path)**
  Return a list of strings that are the names of all entries within a directory designated by string path. In our sample file system, if the parameter is `/user/rt/courses`, this returns the list `['cs016', 'cs252']`.
- **os.path.join(path, filename)**
  Compose the path string and filename string using an appropriate operating system separator between the two (e.g., the / character for a Unix/Linux system, and the \ character for Windows). Return the string that represents the full path to the file.

Python Implementation

With use of the os module, we now convert the algorithm from Code Fragment 4.4 into the Python implementation of Code Fragment 4.5.

```
1  import os
2
3  def disk_usage(path):
4    """Return the number of bytes used by a file/folder and any descendents."""
5    total = os.path.getsize(path)                # account for direct usage
6    if os.path.isdir(path):                      # if this is a directory,
7      for filename in os.listdir(path):          # then for each child:
8        childpath = os.path.join(path, filename) # compose full path to child
9        total += disk_usage(childpath)           # add child's usage to total
10
11   print ('{0:<7}'.format(total), path)         # descriptive output (optional)
12   return total                                 # return the grand total
```

Code Fragment 4.5: A recursive function for reporting disk usage of a file system.

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 goals 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 structures, 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 stored, 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 "duck typing" comes from an adage attributed to 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 classes 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.

P-2.35 Write a set of Python classes that can simulate an Internet application in which one party, Alice, is periodically creating a set of packets that she wants to send to Bob. An Internet process is continually checking if Alice has any packets to send, and if so, it delivers them to Bob's computer, and Bob is periodically checking if his computer has a packet from Alice, and, if so, he reads and deletes it.

P-2.36 Write a Python program to simulate an ecosystem containing two types of creatures, ***bears*** and ***fish***. The ecosystem consists of a river, which is modeled as a relatively large list. Each element of the list should be a Bear object, a Fish object, or None. In each time step, based on a random process, each animal either attempts to move into an adjacent list location or stay where it is. If two animals of the same type are about to collide in the same cell, then they stay where they are, but they create a new instance of that type of animal, which is placed in a random empty (i.e., previously None) location in the list. If a bear and a fish collide, however, then the fish dies (i.e., it disappears).

P-2.37 Write a simulator, as in the previous project, but add a Boolean gender field and a floating-point strength field to each animal, using an Animal class as a base class. If two animals of the same type try to collide, then they only create a new instance of that type of animal if they are of different genders. Otherwise, if two animals of the same type and gender try to collide, then only the one of larger strength survives.

P-2.38 Write a Python program that simulates a system that supports the functions of an e-book reader. You should include methods for users of your system to "buy" new books, view their list of purchased books, and read their purchased books. Your system should use actual books, which have expired copyrights and are available on the Internet, to populate your set of available books for users of your system to "purchase" and read.

P-2.39 Develop an inheritance hierarchy based upon a Polygon class that has abstract methods area( ) and perimeter( ). Implement classes Triangle, Quadrilateral, Pentagon, Hexagon, and Octagon that extend this base class, with the obvious meanings for the area( ) and perimeter( ) methods. Also implement classes, IsoscelesTriangle, EquilateralTriangle, Rectangle, and Square, that have the appropriate inheritance relationships. Finally, write a simple program that allows users to create polygons of the various types and input their geometric dimensions, and the program then outputs their area and perimeter. For extra effort, allow users to input polygons by specifying their vertex coordinates and be able to test if two such polygons are similar.

Reference Counts

Within the state of every Python object is an integer known as its ***reference count***. This is the count of how many references to the object exist anywhere in the system. Every time a reference is assigned to this object, its reference count is incremented, and every time one of those references is reassigned to something else, the reference count for the former object is decremented. The maintenance of a reference count for each object adds $O(1)$ space per object, and the increments and decrements to the count add $O(1)$ additional computation time per such operations.

The Python interpreter allows a running program to examine an object's reference count. Within the sys module there is a function named getrefcount that returns an integer equal to the reference count for the object sent as a parameter. It is worth noting that because the formal parameter of that function is assigned to the actual parameter sent by the caller, there is temporarily one additional reference to that object in the local namespace of the function at the time the count is reported.

The advantage of having a reference count for each object is that if an object's count is ever decremented to zero, that object cannot possibly be a live object and therefore the system can immediately deallocate the object (or place it in a queue of objects that are ready to be deallocated).

Cycle Detection

Although it is clear that an object with a reference count of zero cannot be a live object, it is important to recognize that an object with a nonzero reference count need not qualify as live. There may exist a group of objects that have references to each other, even though none of those objects are reachable from a root object.

For example, a running Python program may have an identifier, data, that is a reference to a sequence implemented using a doubly linked list. In this case, the list referenced by data is a root object, the header and trailer nodes that are stored as attributes of the list are live objects, as are all the intermediate nodes of the list that are indirectly referenced and all the elements that are referenced as elements of those nodes. If the identifier, data, were to go out of scope, or to be reassigned to some other object, the reference count for the list instance may go to zero and be garbage collected, but the reference counts for all of the nodes would remain nonzero, stopping them from being garbage collected by the simple rule above.

Every so often, in particular when the available space in the memory heap is becoming scarce, the Python interpreter uses a more advanced form of garbage collection to reclaim objects that are unreachable, despite their nonzero reference counts. There are different algorithms for implementing cycle detection. (The mechanics of garbage collection in Python are abstracted in the gc module, and may vary depending on the implementation of the interpreter.) A classic algorithm for garbage collection is the ***mark-sweep algorithm***, which we next discuss.

5.2.1 Referential Arrays

As another motivating example, assume that we want a medical information system to keep track of the patients currently assigned to beds in a certain hospital. If we assume that the hospital has 200 beds, and conveniently that those beds are numbered from 0 to 199, we might consider using an array-based structure to maintain the names of the patients currently assigned to those beds. For example, in Python we might use a list of names, such as:

```
['Rene', 'Joseph', 'Janet', 'Jonas', 'Helen', 'Virginia', ... ]
```

To represent such a list with an array, Python must adhere to the requirement that each cell of the array use the same number of bytes. Yet the elements are strings, and strings naturally have different lengths. Python could attempt to reserve enough space for each cell to hold the *maximum* length string (not just of currently stored strings, but of any string we might ever want to store), but that would be wasteful.

Instead, Python represents a list or tuple instance using an internal storage mechanism of an array of object ***references***. At the lowest level, what is stored is a consecutive sequence of memory addresses at which the elements of the sequence reside. A high-level diagram of such a list is shown in Figure 5.4.

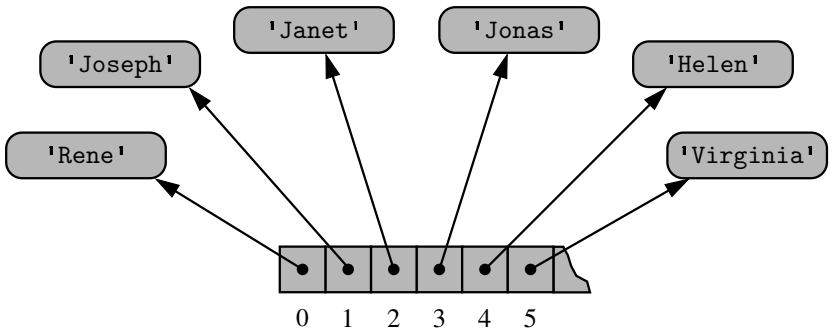

Figure 5.4: An array storing references to strings.

Although the relative size of the individual elements may vary, the number of bits used to store the memory address of each element is fixed (e.g., 64-bits per address). In this way, Python can support constant-time access to a list or tuple element based on its index.

In Figure 5.4, we characterize a list of strings that are the names of the patients in a hospital. It is more likely that a medical information system would manage more comprehensive information on each patient, perhaps represented as an instance of a Patient class. From the perspective of the list implementation, the same principle applies: The list will simply keep a sequence of references to those objects. Note as well that a reference to the **None** object can be used as an element of the list to represent an empty bed in the hospital.

15.1.1 Memory Allocation

With Python, all objects are stored in a pool of memory, known as the ***memory heap*** or ***Python heap*** (which should not be confused with the "heap" data structure presented in Chapter 9). When a command such as

```
w = Widget( )
```

is executed, assuming Widget is the name of a class, a new instance of the class is created and stored somewhere within the memory heap. The Python interpreter is responsible for negotiating the use of space with the operating system and for managing the use of the memory heap when executing a Python program.

The storage available in the memory heap is divided into ***blocks***, which are contiguous array-like "chunks" of memory that may be of variable or fixed sizes. The system must be implemented so that it can quickly allocate memory for new objects. One popular method is to keep contiguous "holes" of available free memory in a linked list, called the ***free list***. The links joining these holes are stored inside the holes themselves, since their memory is not being used. As memory is allocated and deallocated, the collection of holes in the free lists changes, with the unused memory being separated into disjoint holes divided by blocks of used memory. This separation of unused memory into separate holes is known as ***fragmentation***. The problem is that it becomes more difficult to find large continuous chunks of memory, when needed, even though an equivalent amount of memory may be unused (yet fragmented). Therefore, we would like to minimize fragmentation as much as possible.

There are two kinds of fragmentation that can occur. ***Internal fragmentation*** occurs when a portion of an allocated memory block is unused. For example, a program may request an array of size 1000, but only use the first 100 cells of this array. There is not much that a run-time environment can do to reduce internal fragmentation. ***External fragmentation***, on the other hand, occurs when there is a significant amount of unused memory between several contiguous blocks of allocated memory. Since the run-time environment has control over where to allocate memory when it is requested, the run-time environment should allocate memory in a way to try to reduce external fragmentation as much as reasonably possible.

Several heuristics have been suggested for allocating memory from the heap so as to minimize external fragmentation. The ***best-fit algorithm*** searches the entire free list to find the hole whose size is closest to the amount of memory being requested. The ***first-fit algorithm*** searches from the beginning of the free list for the first hole that is large enough. The ***next-fit algorithm*** is similar, in that it also searches the free list for the first hole that is large enough, but it begins its search from where it left off previously, viewing the free list as a circularly linked list (Section 7.2). The ***worst-fit algorithm*** searches the free list to find the largest hole of available memory, which might be done faster than a search of the entire free list

10.1 Maps and Dictionaries

Python's **dict** class is arguably the most significant data structure in the language. It represents an abstraction known as a ***dictionary*** in which unique ***keys*** are mapped to associated ***values***. Because of the relationship they express between keys and values, dictionaries are commonly known as ***associative arrays*** or ***maps***. In this book, we use the term *dictionary* when specifically discussing Python's dict class, and the term *map* when discussing the more general notion of the abstract data type.

As a simple example, Figure 10.1 illustrates a map from the names of countries to their associated units of currency.

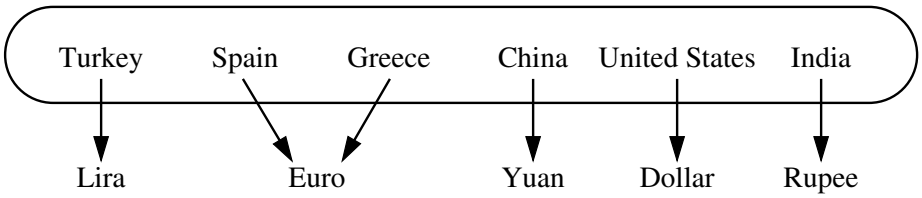

Figure 10.1: A map from countries (the keys) to their units of currency (the values).

We note that the keys (the country names) are assumed to be unique, but the values (the currency units) are not necessarily unique. For example, we note that Spain and Greece both use the euro for currency. Maps use an array-like syntax for indexing, such as currency['Greece'] to access a value associated with a given key or currency['Greece'] = 'Drachma' to remap it to a new value. Unlike a standard array, indices for a map need not be consecutive nor even numeric. Common applications of maps include the following.

- A university's information system relies on some form of a student ID as a key that is mapped to that student's associated record (such as the student's name, address, and course grades) serving as the value.
- The domain-name system (DNS) maps a host name, such as www.wiley.com, to an Internet-Protocol (IP) address, such as 208.215.179.146.
- A social media site typically relies on a (nonnumeric) username as a key that can be efficiently mapped to a particular user's associated information.
- A computer graphics system may map a color name, such as 'turquoise', to the triple of numbers that describes the color's RGB (red-green-blue) representation, such as (64,224,208).
- Python uses a dictionary to represent each namespace, mapping an identifying string, such as 'pi', to an associated object, such as 3.14159.

In this chapter and the next we demonstrate that a map may be implemented so that a search for a key, and its associated value, can be performed very efficiently, thereby supporting fast lookup in such applications.