1.1 Python Overview

Building data structures and algorithms requires that we communicate detailed instructions to a computer. An excellent way to perform such communications is using a high-level computer language, such as Python. The Python programming language was originally developed by Guido van Rossum in the early 1990s, and has since become a prominently used language in industry and education. The second major version of the language, Python 2, was released in 2000, and the third major version, Python 3, released in 2008. We note that there are significant incompatibilities between Python 2 and Python 3. *This book is based on Python 3 (more specifically, Python 3.1 or later).* The latest version of the language is freely available at `www.python.org`, along with documentation and tutorials.

In this chapter, we provide an overview of the Python programming language, and we continue this discussion in the next chapter, focusing on object-oriented principles. We assume that readers of this book have prior programming experience, although not necessarily using Python. This book does not provide a complete description of the Python language (there are numerous language references for that purpose), but it does introduce all aspects of the language that are used in code fragments later in this book.

1.1.1 The Python Interpreter

Python is formally an ***interpreted*** language. Commands are executed through a piece of software known as the ***Python interpreter***. The interpreter receives a command, evaluates that command, and reports the result of the command. While the interpreter can be used interactively (especially when debugging), a programmer typically defines a series of commands in advance and saves those commands in a plain text file known as ***source code*** or a ***script***. For Python, source code is conventionally stored in a file named with the `.py` suffix (e.g., `demo.py`).

On most operating systems, the Python interpreter can be started by typing `python` from the command line. By default, the interpreter starts in interactive mode with a clean workspace. Commands from a predefined script saved in a file (e.g., `demo.py`) are executed by invoking the interpreter with the filename as an argument (e.g., `python demo.py`), or using an additional `-i` flag in order to execute a script and then enter interactive mode (e.g., `python -i demo.py`).

Many ***integrated development environments*** (IDEs) provide richer software development platforms for Python, including one named IDLE that is included with the standard Python distribution. IDLE provides an embedded text-editor with support for displaying and editing Python code, and a basic debugger, allowing step-by-step execution of a program while examining key variable values.

Preface

The design and analysis of efficient data structures has long been recognized as a vital subject in computing and is part of the core curriculum of computer science and computer engineering undergraduate degrees. *Data Structures and Algorithms in Python* provides an introduction to data structures and algorithms, including their design, analysis, and implementation. This book is designed for use in a beginning-level data structures course, or in an intermediate-level introduction to algorithms course. We discuss its use for such courses in more detail later in this preface.

To promote the development of robust and reusable software, we have tried to take a consistent object-oriented viewpoint throughout this text. One of the main ideas of the object-oriented approach is that data should be presented as being encapsulated with the methods that access and modify them. That is, rather than simply viewing data as a collection of bytes and addresses, we think of data objects as instances of an ***abstract data type*** (***ADT***), which includes a repertoire of methods for performing operations on data objects of this type. We then emphasize that there may be several different implementation strategies for a particular ADT, and explore the relative pros and cons of these choices. We provide complete Python implementations for almost all data structures and algorithms discussed, and we introduce important object-oriented ***design patterns*** as means to organize those implementations into reusable components.

Desired outcomes for readers of our book include that:

- They have knowledge of the most common abstractions for data collections (e.g., stacks, queues, lists, trees, maps).
- They understand algorithmic strategies for producing efficient realizations of common data structures.
- They can analyze algorithmic performance, both theoretically and experimentally, and recognize common trade-offs between competing strategies.
- They can wisely use existing data structures and algorithms found in modern programming language libraries.
- They have experience working with concrete implementations for most foundational data structures and algorithms.
- They can apply data structures and algorithms to solve complex problems.

In support of the last goal, we present many example applications of data structures throughout the book, including the processing of file systems, matching of tags in structured formats such as HTML, simple cryptography, text frequency analysis, automated geometric layout, Huffman coding, DNA sequence alignment, and search engine indexing.

Contents and Organization

The chapters for this book are organized to provide a pedagogical path that starts with the basics of Python programming and object-oriented design. We then add foundational techniques like algorithm analysis and recursion. In the main portion of the book, we present fundamental data structures and algorithms, concluding with a discussion of memory management (that is, the architectural underpinnings of data structures). Specifically, the chapters for this book are organized as follows:

1. **Python Primer**
2. **Object-Oriented Programming**
3. **Algorithm Analysis**
4. **Recursion**
5. **Array-Based Sequences**
6. **Stacks, Queues, and Deques**
7. **Linked Lists**
8. **Trees**
9. **Priority Queues**
10. **Maps, Hash Tables, and Skip Lists**
11. **Search Trees**
12. **Sorting and Selection**
13. **Text Processing**
14. **Graph Algorithms**
15. **Memory Management and B-Trees**

A. **Character Strings in Python**

B. **Useful Mathematical Facts**

A more detailed table of contents follows this preface, beginning on page xi.

Prerequisites

We assume that the reader is at least vaguely familiar with a high-level programming language, such as C, C++, Python, or Java, and that he or she understands the main constructs from such a high-level language, including:

- Variables and expressions.
- Decision structures (such as if-statements and switch-statements).
- Iteration structures (for loops and while loops).
- Functions (whether stand-alone or object-oriented methods).

For readers who are familiar with these concepts, but not with how they are expressed in Python, we provide a primer on the Python language in Chapter 1. Still, this book is primarily a data structures book, not a Python book; hence, it does not give a comprehensive treatment of Python.

Book Features

This book is based upon the book *Data Structures and Algorithms in Java* by Goodrich and Tamassia, and the related *Data Structures and Algorithms in C++* by Goodrich, Tamassia, and Mount. However, this book is not simply a translation of those other books to Python. In adapting the material for this book, we have significantly redesigned the organization and content of the book as follows:

- The code base has been entirely redesigned to take advantage of the features of Python, such as use of generators for iterating elements of a collection.
- Many algorithms that were presented as pseudo-code in the Java and C++ versions are directly presented as complete Python code.
- In general, ADTs are defined to have consistent interface with Python's built-in data types and those in Python's collections module.
- Chapter 5 provides an in-depth exploration of the dynamic array-based underpinnings of Python's built-in list, tuple, and str classes. New Appendix A serves as an additional reference regarding the functionality of the str class.
- Over 450 illustrations have been created or revised.
- New and revised exercises bring the overall total number to 750.

Online Resources

This book is accompanied by an extensive set of online resources, which can be found at the following Web site:

www.wiley.com/college/goodrich

Students are encouraged to use this site along with the book, to help with exercises and increase understanding of the subject. Instructors are likewise welcome to use the site to help plan, organize, and present their course materials. Included on this Web site is a collection of educational aids that augment the topics of this book, for both students and instructors. Because of their added value, some of these online resources are password protected.

For all readers, and especially for students, we include the following resources:

- All the Python source code presented in this book.
- PDF handouts of Powerpoint slides (four-per-page) provided to instructors.
- A database of hints to ***all*** exercises, indexed by problem number.

For instructors using this book, we include the following additional teaching aids:

- Solutions to hundreds of the book's exercises.
- Color versions of all figures and illustrations from the book.
- Slides in Powerpoint and PDF (one-per-page) format.

The slides are fully editable, so as to allow an instructor using this book full freedom in customizing his or her presentations. All the online resources are provided at no extra charge to any instructor adopting this book for his or her course.

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.

2.2.2 Pseudo-Code

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

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

2.2.3 Coding Style and Documentation

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

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

The main principles that we adopt are as follows:

- Python code blocks are typically indented by 4 spaces. However, to avoid having our code fragments overrun the book's margins, we use 2 spaces for each level of indentation. It is strongly recommended that tabs be avoided, as tabs are displayed with differing widths across systems, and tabs and spaces are not viewed as identical by the Python interpreter. Many Python-aware editors will automatically replace tabs with an appropriate number of spaces.

2.4.3 Abstract Base Classes

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

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

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

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

As a tangible example, the collections.Sequence abstract base class defines behaviors common to Python's list, str, and tuple classes, as sequences that support element access via an integer index. More so, the collections.Sequence class provides concrete implementations of methods, count, index, and __contains__ that can be inherited by any class that provides concrete implementations of both __len__ and __getitem__. For the purpose of illustration, we provide a sample implementation of such a Sequence abstract base class in Code Fragment 2.14.

Sequence Types: The list, tuple, and str Classes

The **list**, **tuple**, and **str** classes are ***sequence*** types in Python, representing a collection of values in which the order is significant. The list class is the most general, representing a sequence of arbitrary objects (akin to an "array" in other languages). The tuple class is an ***immutable*** version of the list class, benefiting from a streamlined internal representation. The str class is specially designed for representing an immutable sequence of text characters. We note that Python does not have a separate class for characters; they are just strings with length one.

The list Class

A **list** instance stores a sequence of objects. A list is a ***referential*** structure, as it technically stores a sequence of ***references*** to its elements (see Figure 1.4). Elements of a list may be arbitrary objects (including the None object). Lists are ***array-based*** sequences and are ***zero-indexed***, thus a list of length n has elements indexed from 0 to $n-1$ inclusive. Lists are perhaps the most used container type in Python and they will be extremely central to our study of data structures and algorithms. They have many valuable behaviors, including the ability to dynamically expand and contract their capacities as needed. In this chapter, we will discuss only the most basic properties of lists. We revisit the inner working of all of Python's sequence types as the focus of Chapter 5.

Python uses the characters [ ] as delimiters for a list literal, with [ ] itself being an empty list. As another example, `['red', 'green', 'blue']` is a list containing three string instances. The contents of a list literal need not be expressed as literals; if identifiers a and b have been established, then syntax [a, b] is legitimate.

The list( ) constructor produces an empty list by default. However, the constructor will accept any parameter that is of an ***iterable*** type. We will discuss iteration further in Section 1.8, but examples of iterable types include all of the standard container types (e.g., strings, list, tuples, sets, dictionaries). For example, the syntax list(`'hello'`) produces a list of individual characters, `['h', 'e', 'l', 'l', 'o']`. Because an existing list is itself iterable, the syntax backup = list(data) can be used to construct a new list instance referencing the same contents as the original.

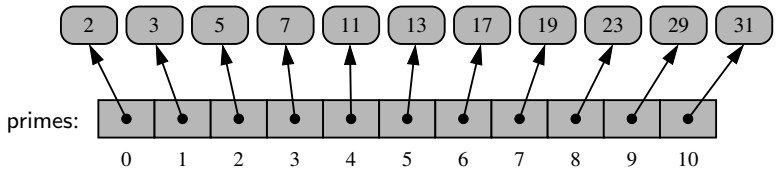

Figure 1.4: Python's internal representation of a list of integers, instantiated as prime = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]. The implicit indices of the elements are shown below each entry.

The set and frozenset Classes

Python's **set** class represents the mathematical notion of a set, namely a collection of elements, without duplicates, and without an inherent order to those elements. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set. This is based on a data structure known as a ***hash table*** (which will be the primary topic of Chapter 10). However, there are two important restrictions due to the algorithmic underpinnings. The first is that the set does not maintain the elements in any particular order. The second is that only instances of ***immutable*** types can be added to a Python set. Therefore, objects such as integers, floating-point numbers, and character strings are eligible to be elements of a set. It is possible to maintain a set of tuples, but not a set of lists or a set of sets, as lists and sets are mutable. The **frozenset** class is an immutable form of the set type, so it is legal to have a set of frozensets.

Python uses curly braces { and } as delimiters for a set, for example, as {17} or {'red', 'green', 'blue'}. The exception to this rule is that { } does not represent an empty set; for historical reasons, it represents an empty dictionary (see next paragraph). Instead, the constructor syntax set( ) produces an empty set. If an iterable parameter is sent to the constructor, then the set of distinct elements is produced. For example, set('hello') produces {'h', 'e', 'l', 'o'}.

The dict Class

Python's **dict** class represents a ***dictionary***, or ***mapping***, from a set of distinct ***keys*** to associated ***values***. For example, a dictionary might map from unique student ID numbers, to larger student records (such as the student's name, address, and course grades). Python implements a dict using an almost identical approach to that of a set, but with storage of the associated values.

A dictionary literal also uses curly braces, and because dictionaries were introduced in Python prior to sets, the literal form { } produces an empty dictionary. A nonempty dictionary is expressed using a comma-separated series of key:value pairs. For example, the dictionary {'ga' : 'Irish', 'de' : 'German'} maps 'ga' to 'Irish' and 'de' to 'German'.

The constructor for the dict class accepts an existing mapping as a parameter, in which case it creates a new dictionary with identical associations as the existing one. Alternatively, the constructor accepts a sequence of key-value pairs as a parameter, as in dict(pairs) with pairs = [('ga', 'Irish'), ('de', 'German')].

It is worth noting that top-level commands with the module source code are executed when the module is first imported, almost as if the module were its own script. There is a special construct for embedding commands within the module that will be executed if the module is directly invoked as a script, but not when the module is imported from another script. Such commands should be placed in a body of a conditional statement of the following form,

```
if __name__ == '__main__':
```

Using our hypothetical `utility.py` module as an example, such commands will be executed if the interpreter is started with a command `python utility.py`, but not when the utility module is imported into another context. This approach is often used to embed what are known as ***unit tests*** within the module; we will discuss unit testing further in Section 2.2.4.

1.11.1 Existing Modules

Table 1.7 provides a summary of a few available modules that are relevant to a study of data structures. We have already discussed the math module briefly. In the remainder of this section, we highlight another module that is particularly important for some of the data structures and algorithms that we will study later in this book.

Existing Modules	
Module Name	**Description**
array	Provides compact array storage for primitive types.
collections	Defines additional data structures and abstract base classes involving collections of objects.
copy	Defines general functions for making copies of objects.
heapq	Provides heap-based priority queue functions (see Section 9.3.7).
math	Defines common mathematical constants and functions.
os	Provides support for interactions with the operating system.
random	Provides random number generation.
re	Provides support for processing regular expressions.
sys	Provides additional level of interaction with the Python interpreter.
time	Provides support for measuring time, or delaying a program.

Table 1.7: Some existing Python modules relevant to data structures and algorithms.

Pseudo-Random Number Generation

Python's random module provides the ability to generate pseudo-random numbers, that is, numbers that are statistically random (but not necessarily truly random). A ***pseudo-random number generator*** uses a deterministic formula to generate the