Principles
of
Computer Science

Michael C Horsch

Course readings for
CMPT 145

Third Edition, January, 2018
Second Edition, July, 2017
First Edition, January, 2017

Preface

This book is about Computer Science. Let's make it very clear right from the start that Computer Science is more than mere programming. Computer Science is the study of computationally effective compositions. Composing a program is the essence of problem solving. Once we have mastered the basics of programming, we study algorithms, and the data structures that make algorithms effective. All of this can be done without any computers at all, but Computer Science is a lot more fun because we have computers to demonstrate the concepts that we might otherwise experience only in our imaginations. The fact that we have computers makes computer science pre-eminently practical, and that's an endless source of motivation.

This book is not a traditional text book. It's a collection of topical readings, arranged into a single document. The readings are deliberately short, so that readers are not discouraged by long reading assignments. In particular, this is not a book about Python, even though it uses Python. It doesn't cover every detail that a deeply knowledgeable expert would know, because there are other books and future courses for that kind of thing.

This book has a number of very strong influences. Primarily, the course material developed for the predecessor course, CMPT115, developed by Mark Eramian and Ian McQuillan. Several chapters were inspired by the work of Robert Sedgewick and Kevin Wayne; their introductory text, *Introduction to Programming in Python*, and their *Algorithms* text. I borrowed ideas shamelessly from both. My teaching career started as a teaching assistant for a brand new course and work-in-progress that turned into *The Schematics of Computation* by Vincent Manis and James J. Little. Less directly, *Structure and Interpretation of Computer Programs*, by Harold Abelson and Gerald Sussman, was also a source of inspiration. This book is only a shadow of their accomplishments. Chapter 12 has drawn a lot of inspiration from the classic *The Elements of Programming Style*, by Kernighan and Plauger.

I'd like to acknowledge the following people who have contributed to this effort. First and foremost, Mark Eramian and Jeff Long, for long discussions about how to change the way Computer Science is taught in first year at uSask, and for their inspired and dedicated work on those other two new courses in the program. Brittany Chan worked tirelessly to help bring into reality the ideas that the three of us came up with. I'd also like to thank Mark for contributing a ton of advice and

Contents

Part I   Appendices

1 — Computing with Python

Learning Objectives

After studying this chapter, a student should be able to:

- Apply lists to solve computational problems
- Make use of the `random` module
- Apply dictionaries to keep data organized
- To write code using lists and dictionaries to solve problems stated as informal English descriptions.

In this first chapter, we'll do a bit of review of lists, dictionaries, objects, and modules. The purpose is to give students a sense of where they should be, as they are starting this course.

1.1 Two example problems, with solutions

Python lists have a very high degree of utility, and it would be rare to do much programming without using them. We'll spend a significant amount of time working with lists, so it's good that we start with a review.

In Python, a list is a compound data type used to organize data into linear sequences. To access to a data value in a list, you need to know its *index*, i.e., its position in the list, represented by an integer. The first value or item in a list has index 0 (zero), and the last item's index is one less than the the number of items in the list. For example, if a list contains 10 items, the index of the last item is 9.

1.1.1 Sorting a list

To get things started, consider the problem of sorting a list. Specifically, we assume that we have a list named `unsorted`, which has numeric data in it, and we want to create a new list, named `sorted`, containing all of the values in `unsorted` but in increasing order. For example, if we start with the

list `unsorted` = `[3, 1, 2]`, after sorting, the list `sorted` would have the values `[1, 2, 3]`. The short Python program below accomplishes this task.

```python
1  unsorted = [3, 2, 5, 7, 6, 8, 0, 1, 2, 8, 2]
2  sorted = list()
3
4  while len(unsorted) > 0:
5      out = min(unsorted)
6      unsorted.remove(out)
7      sorted.append(out)
8
9  print(sorted)
```

Notice that Python lists can be created by using a literal expression, e.g. Line 1, or by using the function `list()`; on Line 2, we create an empty list, by giving the function no arguments. The assignment statements on Lines 1 and 2 also demonstrate that lists can have names; more formally, we say that the variables `unsorted` and `sorted` each refer to a specific list. On Lines 4 and 5, we see that Python has functions that can be applied to lists: `len()` returns the number of values in the list, and `min()` returns the smallest value in the list. On Lines 6 and 7, we see that a list is actually an object, having a method called `remove()` that removes a given value from the list, and `append()` which puts a given value after all the current values in the list.

The program itself is an algorithm that can be summed up as follows: *Keep on removing the smallest value from* `unsorted` *and adding it to end of* `sorted` *until there are no more values left in* `unsorted`. There are a number of questions you might think about while considering this program. Is it a good algorithm? By what standards can we consider an algorithm to be *good*? Does it actually work? How would we be certain?

There are a number of questions you might think about while considering the use of Python in this program. What does it mean that a variable *refers* to a value? What's the difference between an *object* and a *value*? Why is `len()` a function, but `append()` is a method? And why is one in blue text colour, but not the other?

You may be aware that Python has a built-in function whose name is `sorted()`, which takes a list argument and produces a new list with all the values in ascending order. While it is almost obviously easier to use a built-in function than to write your own code, is it always *better*? And what happens to the function `sorted()` when a program creates a variable of exactly the same name?

1.1.2 Counting prime numbers

The question we want to answer here is this: given a positive integer n, how many prime numbers are there that are less than or equal to n? A *prime number* is an integer greater than 1, that cannot be divided evenly (i.e., without a remainder) by any other integer except 1 and itself. For example, 2, 3, 5, and 7 are prime, but 4, 6, 8, and 9 are not. So if we pick $n = 11$, the prime numbers are $\{2,3,5,7,11\}$, so in total, there are 5 primes less than or equal to 11.

There is no nice simple formula for this, so we have to calculate it by counting. One way to count primes is to step up from 2 to n, asking whether each number is prime or not. The Python script below accomplishes this more or less directly.

```python
1  n = 100000   # end of range of numbers to check for primes
2
```

```
3   count = 0
4   for i in range(2, n + 1):
5       no_factors_found = True   # assume prime until disproven
6       f = 2
7       # check if i is prime by checking remainders
8       while no_factors_found and f < i:
9           if i % f == 0:
10              no_factors_found = False
11          f += 1
12
13      if no_factors_found:
14          count += 1
15
16  print("# Prime numbers between 2 and " + str(n) + ":", count)
```

In this program we made use of the range() function, and the remainder operator %. Before going on, you might consider the same kinds of questions as before: Is it a good algorithm? Does it actually work?

Another way to count primes makes use of a list that records whether or not a number is prime. The list records Boolean values, one for each integer from 0 to *n* (though we won't be looking at 0 or 1, which are not prime by definition). The list will start with all entries True, i.e., the list will assert that until proven otherwise, all the numbers are possibly prime. We'll walk up through the list, starting at 2. We know that 2 is prime, so we'll keep the list entry as True. But all the multiples of 2, namely 4, 6, 8, etc., are all not prime, so we'll jump through the list recording the value False for all the multiples of 2. In this way, we've marked the numbers we've definitely proven to be not prime, and we're left with numbers that might still possibly be prime. This technique is implemented below:

```
1   n = 100000   # end of range of numbers to check for primes
2
3   still_is_prime = (n+1)*[True] # assume prime until disproven
4
5   for i in range(2, n):
6       if still_is_prime[i]:
7           # mark multiples of i as not prime
8           for j in range(2, n//i + 1):
9               still_is_prime[i * j] = False
10
11  # now, every possible prime is a definite prime
12  count = sum([1 for v in still_is_prime[2:] if v])
13
14  print("# Prime numbers between 2 and " + str(n) + ":", count)
```

In this example, we created a list by multiplication; on Line 3, we created a list by joining together *n* + 1 copies of the list [True]. Lists can also be *concatenated* or joined together using +, e.g. [1,3,5] + [2,4,6].

We made use of a very important aspect of lists, namely the use of indexing (Lines 6, 9).

Notice Line 12, which uses a slice to drop the first two elements: still_is_prime[2:], as

well as a comprehension, to form a list containing one value 1 for each `True` in the slice. The Python function `sum()` adds up all the values in a given list; we used it instead of a for-loop to count how many `True` values are in the list.

The program itself is an implementation algorithm named after an ancient Greek mathematician. This algorithm, known as "The Sieve of Eratosthenes," can be summed up as follows:

> Sift the Twos and Sift the Threes,
> The Sieve of Eratosthenes.
> When the multiples sublime,
> The numbers that remain are Prime.
> – Anonymous

The algorithm was known millennia before computers were invented.

Before you skip ahead to the next section, you might consider whether Eratosthenes' algorithm is better than the previous algorithm, and how you might make that assessment one way or the other.

1.2 A grab-bag of other listy stuff

A list comprehension is a very useful way to build a list, and is reminiscent of mathematical set theory. Consider the set of all the even integers in the range from 1 to 9. In Python, you can create this list a couple of ways:

```
>>> [x for x in range(1,10) if x % 2 == 0]
[2, 4, 6, 8]
>>> [2*x for x in range(1,5)]
[2, 4, 6, 8]
```

If you stare closely enough at these Python expressions, you might see the resemblance to mathematical set notation, as follows:

$$\{x \mid x \in \{1,...,9\}, \ x \bmod 2 = 0\}$$

$$\{2x \mid x \in \{1,...,4\}\}$$

It's important to say that lists are not, strictly speaking, identical to sets. A list can contain repeated values, but a set, strictly speaking, does not have repeated values.

Lists can contain anything at all, even though we've only used integers. A list can contain all of the same kind of thing, e.g., a list containing only integers, or different kinds of things, like a list containing numbers and strings.

Python lets you extract parts of a list using a technique called *slicing*, as we saw earlier. For example, the expression `L[2:5]` creates a new list consisting of the third through fifth items in a previous list L. Note that the first index in a slice is inclusive, but the second is not inclusive. The expression `L[0:len(L):2]` creates a new list consisting of all the items in the list L stored at even indices (in English, one way to say this is "every other item," but this is slightly ambiguous). Slicing is so useful that it can be applied to any kind of sequence, including tuples, and strings.

The Python for-loop allows a program to step through a list in two ways. The simplest is to use each item directly, such as:

```
L = [1,3,5] + list(range(2,7,2))
total = 0
for item in L:
    total = total + item
```

Here, the variable `item` is bound to each item in L in sequence. A for-loop like this can be very fast in Python. But if you're summing a list, as in the above example, it's not as fast as the function `sum()`.

The word `in` can be used as a way to test if a given value is stored in a given list. For example:

```
L = [1,3,5] + list(range(2,7,2))
if 5 in L:
    print('Yes')
else:
    print('No')
```

The expression 5 `in` L is a Boolean expression that is `True` if L contains the value 5, and `False` otherwise. You can combine `in` with `not`: The expression 5 `not in` L is a Boolean expression that is `False` if L contains the value 5, and `True` otherwise.

A list is mutable, which means that we can change its contents, as follows:

```
L[2] = 33
```

We can add items to the list using methods `extend()` and `append()`, as in the following examples:

```
L = [1,3,5]
L.extend(list(range(2,7,2)))
L.append(77)
```

In this example, we are using the list L as an object, by giving the name of the object L, the dot, and then something that looks like a function call. The method `extend()` adds a sequence of items to the end of the list; the method `append()` adds a single item to the end of the list. Other important methods of this sort include `pop()`, `remove()`, and `index()`, among many others.

List objects have lots of methods, which may be useful in various ways. One of the most important things to learn about Python is that it is very common for programmers to look up the details about any particular function, object, or method. Some you will learn simply by re-use. Others you will need to look up quickly to remind yourself how to use them. There is a standard website for Python information: `https://docs.python.org/3/`, which should be one of the first places you check.

It's very common for a list to contain other lists. For example:

```
L = [[1],[2,3],[4,5,6],[7,8,9,10]]
print(L[2], L[3][1])
```

In the above example, L is a list containing 4 sub-lists (we say sub-list only to help distinguish which list we're talking about; sub-lists are completely capable lists in every respect). The first sub-list of L has exactly one item. The print statement displays the third sub-list, i.e., `[4,5,6]`, followed by the value 8. Indexing lists-of-lists requires two pairs of `[]`. In the example, L`[3]` refers to the whole sub-list `[7,8,9,10]`; a second set of `[]` allows us to index into the sublist, so L`[3][1]` is the way to access the second item in the fourth sublist of L.

1.3 Dictionaries

A Python dictionary is a collection of *key-value pairs*. A *key* must be an immutable value (e.g., an integer, string, or tuple), and a key must be unique, i.e. any particular key can appear at most once in the dictionary. A dictionary key is roughly analogous to a list index. To access a data value in a list, you need to know its *index*, i.e., its position in the list. To access a data value in a dictionary, you need to know its *key*. In a given dictionary, the keys have to be comparable to each other; usually this means the keys all have to be the same data type, e.g., all strings, or all integers.

A dictionary can be created in two ways: using a literal, or using the Python function `dict()`.

```python
database = {}   # an empty dictionary literal
db2 = dict()    # creates an empty dictionary

# a dictionary literal with 2 key-value pairs
assignment1 = {'Grade' : 33, 'Out of' : 35}

# calling dict() with a list of tuples
assignment2 = dict([('Grade',28), ('Out of',40)])
```

Note that dictionary literals use the curly brackets {}, and the colon character : to separate literal key-value pairs. The `dict()` function is very versatile; it can create an empty dictionary, or if given a list of tuples, it will create a dictionary using the first value in each tuple as the key, and the second value of each tuple as the corresponding value.

We use dictionaries to store the data values; the key is simply the information we use to access the data values. For example:

```python
print('You got', assignment1['Grade'], 'on A1')

assignment2['Grade'] += 3 # make a correction to the grade
```

The syntax for accessing the dictionary uses the square brackets [] just like lists do, using the key for a key-value pair, rather than an index.

You can add a new key-value pair to a dictionary using a simple assignment statement, as follows:

```python
assignment2['Bonus'] = 3 # a new key-value pair is created
```

Take care to remember that you can't add a new value to a list this way!

You can check if a dictionary has a given key using the `in` operator:

```python
if 'Bonus' in assignment1:
    print('You got',assignment1['Grade']+assignment1['Bonus'],'on A1')
```

And you can iterate over the keys in a dictionary using a for-loop:

```python
for k in assignment2:
    assignment2[k] += 3 # make a correction to each value
```

Lists store data values in an explicit sequence, but dictionaries do not. The sequential aspect of a list is crucial. A dictionary's crucial property is association, and there is no standard order for a dictionary's data. If you print a dictionary twice, you might see the data in different order. This is normal and expected, though at this point in your study, unexplained. Consider it Python's business to guarantee that you can always access the data quickly by its key, but it is not Python's

business to ensure any particular order of the data. This idea will become clearer in later chapters (esp. Chapter 22).

Dictionaries generally represent associations: key-value pairs. You could use dictionaries to represent data mappings, i.e., translating one data value to another. For example, you could represent a mapping between months and integers:

```
month_to_number = {'January':1,'February':2,'March':3, # and the rest
```

You can also use dictionaries to store records, a collection of organized data values associated with a single topic or purpose. For example, we can store information about an assignment as a record, as we saw above:

```
assignment1 = {'Grade' : 33, 'Out of' : 35, 'Completed':'8 January'}
```

If you had a list of records, you could call it a database.

Lists can contain dictionaries as values, and dictionaries can contain lists as values (but not as keys, because lists are mutable; a key must be immutable).

1.3.1 An example program

Suppose you were doing some analysis of data from some experiment you had designed. Your data is stored in a text file, and the file consists of 3719 rows (lines), each with 11 comma-separated columns of numeric data. You might write a script to read this data as follows:

```python
1  # open the file
2  fh = open('datafile.txt')
3
4  # read the file
5  data = []
6  for line in fh:
7      # strip the junk off the end of the line,
8      # and split the line into pieces
9      row = [float(d) for d in line.rstrip().split(',')]
10     data.append(row)
11
12 # close the file
13 fh.close()
```

You might need to calculate some statistics on the data stored in any given column. One way to accomplish this task is to write a function that takes a list of data, and returns a record containing the maximum, minimum and average of the data in the list, as below:

```python
1  def calc_stats(col):
2      '''Calculate min, ave, max of a column of data'''
3      stats = {}
4      stats['minimum'] = min(col)
5      stats['maximum'] = max(col)
6      stats['average'] = sum(col)/len(col)
7      return stats
```

Notice how the function creates and populates a dictionary, which serves as a record of the statistics for the column. This function is better for using a dictionary, for several reasons. If you wanted to calculate additional statistics (e.g., median, standard deviation), you could simply add a new key-value pair to the dictionary. The use of a dictionary to return a bunch of values is better than using a list, because a string like `'average'` on line 6, is more intuitive than an index, like 2.

1.4 Modules

The basic Python language provides a lot of tools that are useful for programmers already. *Modules* allow programmers to share functions (and other computational abstractions, which we will discuss later) that they have written so that other people can use them in their own programs.

Modules are stored in separate files from our own programs. Thus, in order to use the functions and objects defined by a module, we have to tell our own program to look in those files and read those definitions. We do this using a Python keyword called `import`.

In general, the syntax for importing modules is: `import` *x* as *y*. Here, *x* must be the name of a module, and *y* must be a valid variable name (often an abbreviation for *x*). This technique creates an *object* called *y* that contains the functions defined in *x*. We'll say more about objects later. There are other ways to use the `import` command, but we will use this one exclusively, just so that there is no unnecessary confusion.

As an example, we will use the `random` module:

```python
import random as rand

rval = rand.randint(0,10)
rlist = [rand.random() for i in range(10)]
```

Here, `rval` is chosen randomly from the integers from 0 to 10; be very careful to note that the range of `randint()` is inclusive on both sides, unlike the Python function `range()`, which is exclusive on one side! The variable `rlist` contains 10 random floating point values between 0 and 1, which is also exclusive on one side.

1.5 Three Example Problems

In this section, we'll describe three example problems that will set the stage for several later topics. The problems themselves are interesting and have many applications and extensions.

1.5.1 The Gambler's Ruin Problem

In science, and in other human endeavours, we occasionally run into phenomena that we cannot explain with a simple formula. When the behaviour of the phenomenon is too complicated, or unknown, we often turn to models we call "stochastic," which basically means that some amount of randomness is used in the model. The randomness is used as a replacement for a mechanism that is either too complex or too obscure to model directly.

One very simple example of a stochastic model is called "The Gambler's Ruin." In this scenario, we imagine a gambler playing some kind of betting game. The game itself is not important, as we are not presently interested in modelling the game or the gambler's strategies. The game gives the gambler a 50-50 chance of winning each time the gambler plays. If the gambler wins a game, he earns 1 unit of currency, and if he loses, he pays 1 unit of currency. The gambler starts with a given

quantity of currency; we'll call this the stake; for sake of clarity, let's say the stake is 20 units. The gambler also has a goal: to earn a known quantity of the currency; let's imagine the goal is to reach 100 units. If the gambler reaches his goal, he stops gambling, as a success. If the gambler reaches zero units of currency, he also stops gambling (the house does not allow gambling on credit), as a failure.

One question we want to answer is this: what is the probability that the gambler will reach the goal? The second question is: success or failure, how many bets, on average, does the gambler place before stopping?

This problem has no easy formula to give us an answer, so we'll have to write a program to play the game a number of times, and use the data generated to estimate the answers to our questions. It's a form of simulation called "Monte Carlo Simulation" which is very common in applications of statistics to science. Without a computer, this kind of simulation would be very difficult to carry out.

1.5.2 The Coupon Collector Problem

Another scenario in which Monte Carlo simulation is very useful is called the "Coupon Collector". In this scenario, we imagine that someone is trying to collect coupons from some kind of advertising or customer loyalty program. There are 100 different kinds of coupons, and the collector gets any one of these coupons at random (presumably every time she visits the store, or the website, but it really doesn't matter). The chance of obtaining any of the coupons is equal, i.e., 1 in 100, and the collector will almost certainly collect more than one copy of some of the coupons.

The question we want to answer is: how many coupons will the collector have in total by the time she's gathered one of every type? It's at least 100, but because repeats are allowed, it is probably higher than that. Allowing repeats is sometimes called "sampling with replacement." Because it's a stochastic process, we'll want to run the simulation a number of times, and take an average as the estimate.

This problem may seem somewhat trivial, but this model can be applied in science to determine if a phenomenon is random or not.

1.5.3 Self-Avoiding Random Walks

The Gambler's Ruin problem can be interpreted as a one dimensional random walk, as follows. Imagine starting at a point somewhere between two ends of a line segment, and randomly choosing to walk one step forward or back. In this interpretation, success is when you reach the one end, and failure is when you reach the other. In this interpretation, the one dimensional random walker is allowed to go back and forth, and is allowed to re-occupy positions multiple times. The probability of success depends on where the walker starts.

The self-avoiding random walk problem is similar, but has a few important differences. The walker starts in the middle of a large room, and can take a step at random in any of the 4 cardinal directions (forward, back, left, right), subject to a constraint: the walker is not allowed to step anywhere he has already stepped. In other words, we have to keep track of where the walker has been. If a random step is chosen that would lead the walker into a position previously occupied, that step is discarded, and a new random step is chosen. If the walker reaches the "wall" of the room, it is called a success, and if the walker gets caught in a position where there are no legal steps that can be made, it is a failure.

The question we want to answer is this: what is the probability of failure?

The Simple Picture of Variables
References, the Heap, and the Stack
Mutable vs Immutable Data
References and functions
References and abstraction

2 — References

> ## Learning Objectives
>
> After studying this chapter, a student should be able to:
>
> - To explain what a reference is.
> - To explain how Python associates variables and references.
> - To explain how Python uses the heap to store values, and frames to store variables and references.
> - To explain how Python deals with expressions involving mutable and immutable values.

2.1 The Simple Picture of Variables

In order to make sense of the material in later chapters, we need to revisit and clarify a concept that we briefly mentioned in the introductory course. In most of the introductory course, we needed only to concern ourselves with two main ideas about variables. We needed to understand that a variable has a name, and that it has a data value associated with it. The first thing we said about variables was that "[v]ariables are a way of giving names to data." Later on we said that variable assignment "is used in Python to *assign* a variable name to a value."

For example, consider the following very simple script:

```
x = 3
y = x + 1
```

A very simple, but imprecise, understanding of this script might involve a table of variable names and data values, something like the following:

x	3
y	4

The table suggests that Python keeps track of variables and their values directly. This understanding was good enough for almost everything we needed to do until now, but we need to emphasize that *this is not really how Python works.* It's a convenient abstraction of the detailed truth.

The topic of this chapter completes our understanding of variables, which requires a third idea, namely, the *means by which* a variable is associated with a data value. This is an important topic, because we can make use of this concept to build much more interesting tools to organize data, which we will start in Chapter 11 and continue in Chapter 17. In this chapter we have to nail down the concept.

References are used in every programming language to some degree or other. For example, Java uses references for its objects. In C and C++, special variables called pointers allow programmers to store and manipulate references. In Python, every value has a reference to it, and every variable stores a reference. These are all the same basic concept, and it will come up again and again as a source of confusion until you master it. It's not hard to grasp the basics, but it's subtle, and has unexpected consequences.

2.2   References, the Heap, and the Stack

Python has two regions of computer memory that get used when a Python program is executed. One of those regions is called the *heap* (the other region is called the *call stack*, and it will come up towards the end of the section). All data used by a Python program gets stored on the heap: numbers, strings, lists, dictionaries, and everything else one could construct (as we will see in Chapter 19). To keep track of the data on the heap, Python records the *location* or *address* of each piece of data. Every piece of data, including numbers and strings and lists and everything else, has an address, and this is a computer memory address. An address is nothing more than a number, an index into the computer memory that Python has access to.

In Python, we can't really do very much with addresses directly, and Python is designed to let us think about addresses much more abstractly, like an arrow drawn to the data value. The arrow represents the fact that Python knows the exact address, and lets us ignore exact addresses. We will call this arrow a *reference*. It is important to understand that a reference is a *value*; it's a value we visualize as an arrow to a data value.

Very many useful programs can be written without needing to understand references. In simple programs, we can simply imagine that Python (somehow) replaces variable names with the data values they label.

The truth about variables is a little more subtle than the simple story. We described a reference as being like an arrow that points to the data. With this metaphor in mind, we can say that a variable is the place that the arrow starts. Every variable starts an arrow, and the arrow ends at a data value. We say that the variable labels, or *refers to*, the data.

An example will help make this analogy more concrete. Consider the following sequence of simple statements:

```
x = 3
y = x + 1
```

In the first line we have a literal value, 3. When Python sees a literal value or expression, it automatically creates a data value on the heap, and remembers the reference. The assignment statement causes the reference to the data value 3 to be connected to the variable x via the reference (an arrow). On the second line, the right hand side is an expression involving the variable x. When

Figure 2.1: Variables refer to values using addresses or references, represented here by an arrow.
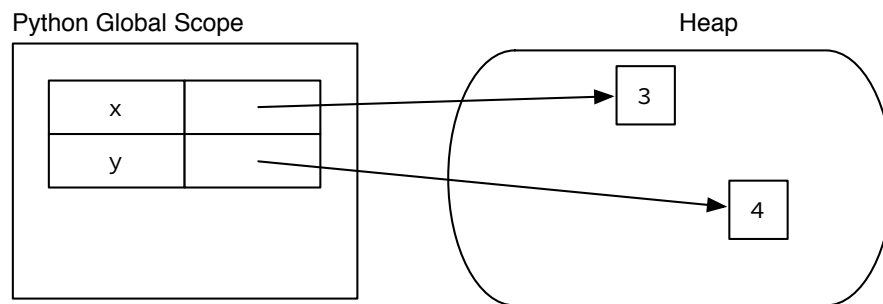
Python sees a variable in an expression like this, Python follows the arrow from the variable name to the data value, which is the number 3 in this example. The expression $x + 1$ creates a new data value, which is also stored on the heap, and the assignment on line 2 causes the variable y to refer to the data value 4. See Figure 2.1 for a diagram of the situation. In pictures and figures, we use arrows to represent the idea of reference. A box that stores a reference always shows an arrow leaving from inside the box to some data value on the heap.

Python associates variables with data values in a table called a *frame*. We can think of a frame as having 2 columns: a column for the variable names, and a second column to store references. When we create a variable in Python, the variable's name is entered into the table, and the reference to its initial value is stored beside it. In Figure 2.1, we see a frame consisting of two rows, listing variables *x* and *y*, and representing references by arrows in a diagram. Remember that a reference is a value, a location in the heap; Python actually records the location in the frame. But when we think about locations and references, we abstract them as an arrow; the arrow starts inside the frame to remind us that it is a value, and ends at a data value in the heap.

When a variable appears in an expression in a Python program, for example x + 1, Python looks for the variable's name in the frame, and if it is there, follows the variable's reference to the value on the heap; Python uses the value in the expression. If a variable appears in an expression but has not been created yet, Python will raise a NameError, which is basically Python's way of saying *that name is not in the frame yet*. When a variable appears on the left-side of an assignment statement, for example, y = x + 1, Python draws a new arrow from the variable's name in the frame to the data value implied by the expression on the right hand of the assignment statement; any previous reference that might have been stored in the frame is replaced. This paragraph should be memorized.

Python creates several frames to manage all the variables and their references. There is a global frame that contains all the variables you create when you write a script outside of any functions. Python also creates a frame each time a function is called, which contains all the parameters and variables created by the function as it runs. The frame that Python creates for a function call gets placed on something called the *call stack*. When a function is called, a frame for the function is pushed on to the top of the stack; when the function returns, the function's frame is removed from the top of the stack. We'll say more about stacks in Chapter 8, and Chapter 16.

Figure 2.2: The assignment statement y = x copies the reference, not the value. Both x and y refer to the same value.
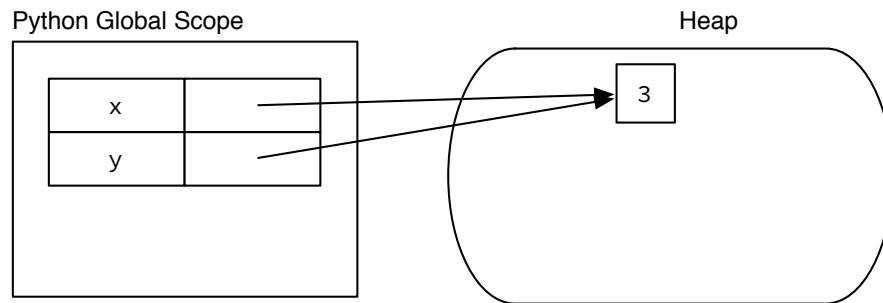
There are a few details that we need to clear up. We need to clarify how Python handles the following:

```
x = 3
y = x
```

The first line is easy enough: the value 3 is stored on the heap, and the variable x stores the reference to the value in the heap. That's what we mean when we say x refers to the value 3. The second line refers to a variable called y, and also stores a reference to a data value. Here, Python copies the *reference to the value 3*; Python does not copy the actual value. In other words, no new data value has been created; but two variables refer to the same data value 3; x and y refer to the same data value. A picture of the situation is found in Figure 2.2.

2.3 Mutable vs Immutable Data

Numbers, strings, tuples, and Boolean values are *immutable*: they cannot be changed. When you add two numbers together, say $3+4$, you're not changing the numbers, you're creating a new value, 7. Python knows how to create values. The result is a new data value, with a new reference into the heap. If you are using a variable to accumulate a sum, as in the following

```
total = 0
for v in a_list:
    total = total + v
```

the variable total initially refers to the value 0 which is on the heap. Every time the loop adds a value to total, a new value is created on the heap, and the assignment changes what the variable total refers to. Python takes care of this and we hardly notice.

But now consider a *mutable* data type, like a list or a dictionary. When we create a list, we are creating a new value; it's a compound data value, but still a value, and it's on the heap. Mutable data values can be changed. That means if you extend a list, add or remove elements, the list actually changes; however, the reference to the list does not change. Consider the following example:

```
copy = []
for v in a_list:
```

Figure 2.3: Data values are not stored in lists. Lists store references to data values.
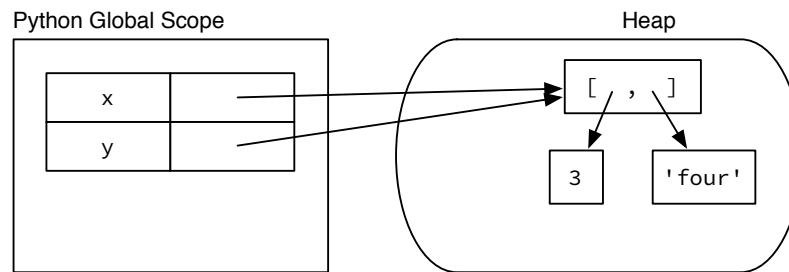

```
      copy.append(v)
```

The variable `copy` never changes; it refers to the same list object the whole time. But that list grows in size to accommodate all the items in `a_list`.

It's important to understand that containers like lists and dictionaries don't actually store data values either. For example, a value like 3 is never stored in a list. Rather, data values are stored in the heap (as usual), and only the references are stored in the list. Suppose we have the following sequence of Python code:

```
x = []
y = x
x.append(3)
y.append('four')
```

Since data values are not stored in lists, the picture looks like Figure 2.3.

One thing to watch out for is the fact that Python will not usually display references for the kinds of data types we've been using, like numbers, lists and dictionaries. If you print a list to the screen, Python will display only the data values, not the references. This can be confusing at first, but it's actually sensible. No one really wants to see the addresses, because there is no way to use that information in Python.

The approach Python takes, wherein every variable stores a reference, is very versatile. It allows us to do complicated things in Python with very little effort. When we talk about lists of lists, we never actually store a list inside another list. Lists are stored in the heap; the list does not store values, but it does store references. A list-of-lists is really a list of references, and those references simply refer to other lists. We can store references in any kind of container: lists, dictionaries, and tuples. We can have a list of dictionaries, which is really a list of dictionary references. We can have a dictionary that refers to other dictionaries, or lists, or anything else for that matter. A variable can refer to any kind of data value: integers, lists, dictionaries, whatever. It's the data itself that knows its type; not the variable.

2.4 References and functions

We have already stated that all values are stored on the heap, that variables refer to values by their location (which we represent with arrows in a diagram), and that assignment statements cause

variables to refer to values. We can now complete the picture by discussing how functions figure into the picture.

When we define a function, we are implicitly telling Python what variables the function uses when it is called. Python can read the function *definition*, and determine where variables are created. This is important, because as we said earlier, a function *call* causes a new frame to be created, and the variables in the scope of the function are part of this new frame. We have to be careful to remember that a function's parameters are also variables, and they are also part of the frame that gets created when a function is called. When the function is called, the function's parameters get their initial value by an implicit assignment using the arguments in the function call.

For example, consider the following:

```
x = 5

def fun(a, b):
    x = a + b
    return x * 2

y = fun(x, x + 1)
```

A quick scan of this code reveals that the script uses variables *x* and *y* in the global scope, and the function has two parameters, *a* and *b*, and a local variable *x*. The function definition tells us what its frame will look like, but the definition does not create a frame; it is only when the function is called that the frame is created. The diagram below shows the situation just after the global variable *x* is initialized.
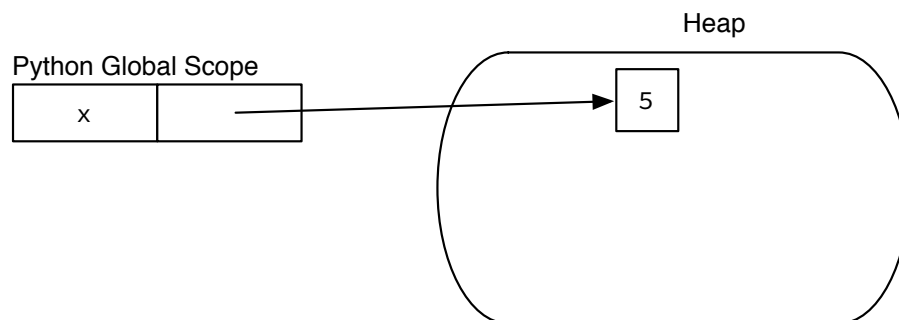
When `fun` is called on the last line of the above code, a frame is created, pushed on the stack, and parameters are initialized. Specifically, the parameter *a* is assigned to the value 5, and the parameter *b* is assigned to the value 6. The first line of the function initializes the local variable *x*, which has the same name as the global variable, but it is in a different frame, so it is distinct. We say that the local variable *shadows* the global variable. The diagram below shows the situation when `fun`'s local variable *x* is initialized, just before returning.
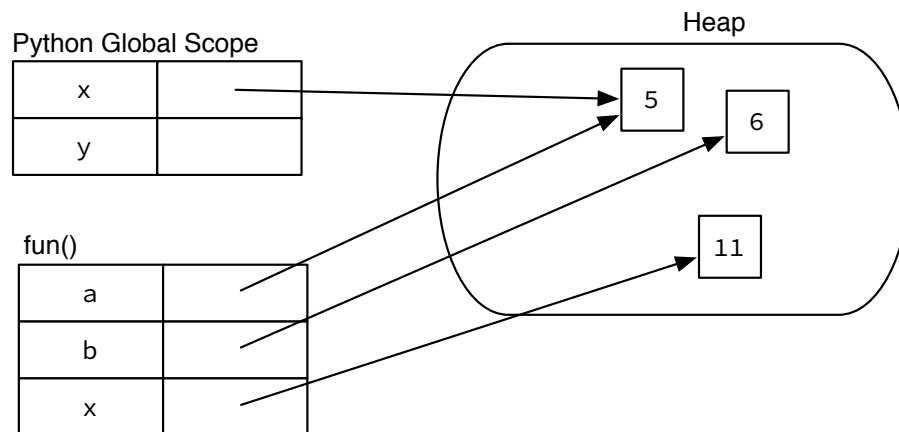
After the function returns, the value *y* is initialized with the return value. This is an assignment, so *y* refers to the value created by the function. The diagram below shows the situation when the global variable *y* is initialized, just after the function returned.
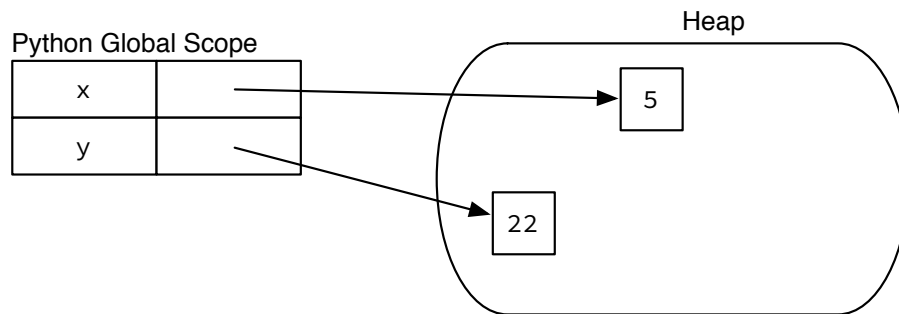
Notice that the frame for the function disappeared; it was popped from the stack after the function has returned.

2.5 References and abstraction

Most of the time, we'll be happy to say that a variable "stores" a given value, even though it is more precise to say that a variable *refers* to a given value. When it is convenient and helpful, we will gloss over the notion of reference entirely; that's a detail that doesn't always need to be made precise. However, there are times when the notion of reference is crucial, and it is at those times when we have to be precise. There are computer scientists who say that abstraction is the fundamental skill, and it is this fluidity between precision and imprecision that they are talking about. For a beginner, it can get confusing; be as precise as you can until you master the concepts.

3 — Software Design Goals

Learning Objectives

After studying this chapter, a student should be able to:

- Define design goals of correctness, and efficiency.
- Define implementation goals of robustness, adaptability, and reusability.
- Explain why the creation of correct program components is important in the production of high-quality software.

3.1 Design Goals

We write software to accomplish a task. Sometimes the task is "get credit in a course," and sometimes it's "do something fun." But often, we will write software because we want the computer to do some work we need to get done. Writing software can take a long time and a lot of effort. We invest the time to write software for fairly specific reasons:

- The computer can do the work much faster than if we do it ourselves by hand.
- The work will be done more than once.

Either of the above is good enough reason; they don't both have to be true.

It would be a pointless exercise to write a program if:

- The program didn't give the right answer.
- The program gave an answer too late to be useful.
- Writing the program was more expensive than getting the answer some other way.

Given the reasons we write software, it's important to be concerned with the *correctness* and *efficiency* of our software.

Definition 3.1   Software is *correct* if it does everything it is intended to do, and nothing that it isn't supposed to do.

A program that is not correct is said to have errors (though sometimes we call those errors "bugs" to make them seem less serious). Very often, a program is partially correct: it gets the right answer some of the time, but maybe not always. For example, a program that simply displays the text "Today is Monday!" is correct on Mondays, but incorrect all the remaining days of the week. As another example, consider a function that returns the square root of any number: if it returns a useful approximation for every non-negative number, it's doing what it should, but if it goes into an infinite loop when the input is negative, it is doing something it shouldn't be doing.

Definition 3.2   Software *efficiency* is a relative measure of the degree to which software consumes computational resources, including time and memory.

A program that is not efficient may be using an algorithm that takes more time than a better algorithm, or it may be using more memory than it needs to. On modern computers, any given program may be efficient enough if the task is small, or if the program won't be reused. But once the tasks start getting big, inefficient programs may give an answer too late to be useful. On modern computers, programs usually are running alongside other programs (multi-tasking), e.g., your web browser and your video editor may be running at the same time. Programs that do not make efficient use of resources will interfere with other programs by using up resources that ought to be shared.

Correctness and efficiency are our design goals. There are quantitative measurements that we can use to assess them. We will discuss these topics in more detail in later chapters.

3.2 Implementation Goals

If software were art, then we'd finish writing the software and it would never change after we're done. But it's not art: software changes every time someone thinks of a way to make it better. People who work with software rarely get to admire it; they are busy trying to get it to do more! So it's also important for software to have other attributes, related to the goals of software developers.

Definition 3.3   Software is *robust* if it behaves well even when things go wrong.

Even if a program is correct, things can go wrong if the user enters bad data, or if a network connection is lost, or any other kind of external interaction behaves unexpectedly. To assess *robustness*, we consider what kinds of things might go wrong that are beyond the control of the program. A program that is not robust is called *brittle*. Brittle software crashes without warning, causing data loss or other inconveniences. If a program still has errors, a robust program might be able to recognize that an error had occurred, and recover from the error without causing the program to crash. Even if a crash is unavoidable, it would be useful for a program to do something helpful, like saving a document, or reporting that an error had occurred with a helpful error message.

Definition 3.4   Software is *adaptable* if small changes in behaviour of the software can be achieved by making small changes in the software.

For example, suppose you've written a program to calculate the average of 10 values in a list or array.

An adaptable program would not require a lot of changes to calculate the average of the numbers in a list or array of any size. If you had to do a major rewrite of the code, the code you started with was not very adaptable.

The reality of writing software is that software is never truly finished. There's just a point at which someone stops working on it. Somebody, sometime (possibly you, tomorrow, when you realize you misunderstood something yesterday — oops!), will find errors, or other short-comings. Somebody may make changes to the requirements you thought were finalized, and to meet those changes, you'll have to adapt the code you've already written. Somebody, possibly you, will realize that the program you wrote 4 months ago to solve problem X can be modified to solve related problem Y. If you design your software with these kinds of possibilities in mind, making the changes will be easier. If you give no consideration to adaptability when you write programs, you are simply making someone else's life, possibly your own, much more difficult.

> **Definition 3.5**   Software is *reusable* if it can be used more than once.

We already make use of a lot of reusable software. Every programming language provides tools to help the programmer that can be used in any project. Think of Python functions like `print()`, or `range()`; think of all the math functions; think of all the modules! This kind of reusability does not happen by accident. Software designers deliberately design these components to be reusable.

Robustness, adaptability, and reusability are our implementation goals. They are qualitative: there's currently no way to measure them scientifically. For example, software can be more, or less, robust, but we don't have a precise way to measure it.

3.3 Example

Consider the short Python program below (which we studied in Chapter 1):

```python
unsorted = [3, 2, 5, 7, 6, 8, 0, 1, 2, 8, 2]
sorted = list()

while len(unsorted) > 0:
    out = min(unsorted)
    unsorted.remove(out)
    sorted.append(out)

print(sorted)
```

It creates a list of numeric values whose values are in increasing order. The algorithm can be summed up as follows: *Keep on removing the smallest value from* `unsorted` *and adding it to end of* `sorted` *until there are no more values left in* `unsorted`. This is one way to interpret the notion of *sorting* a list. Let's consider the algorithm in light of the design and implementation goals from the previous sections.

In terms of assessing or demonstrating *correctness*, we have to give some reason to believe that the code does what it is supposed to do. The English language summary (repeated above) lends credibility to the correctness of the script, though we must be careful, since a summary omits detail, and it might be the details themselves that are wrong. We could try to highlight the principle upon which the algorithm seems to be based. This requires some insight. For example, when you remove

a smallest element *x* from a list, the next time you remove a smallest element, say *y*, it will be larger than *x*, and so it should come after *x* in the sorted list. Mathematical tools (such as mathematical induction) can be applied to address the question of correctness. However, at this point in your study of computer science, the main tool you currently have to establish correctness is testing. We will have much to say about testing in a later chapter.

To assess *efficiency*, we have to be able to measure the quantity of resources used by the program, and compare it to other programs or algorithms. It might be tempting to make a vague statement: taking one element out of a list and putting it into another list seems more or less efficient. You may recall that finding the smallest element of an unordered list is a form of linear search, and to find the smallest, linear search may have to look at every element. The fact is that this algorithm is not terribly inefficient, but it is not as good some of the algorithms we studied in the previous course. In a later chapter, we will have much to say about how to analyze algorithms and establish their costs in a scientific way, so if you were hesitant to assert strongly that the algorithm is efficient, that is probably a natural feeling to have right now.

To assess *robustness*, we consider what kinds of things might go wrong that are beyond the control of the program. Because there is no user input, no file I/O, and no other interaction with the outside world at all, the behaviour of the program is entirely deterministic, i.e., the behaviour of the code is determined by the code itself. As a result, we can call the program robust, even though we did nothing special to make it so.

In terms of *adaptability*, we have to consider what kinds of changes we might imagine making to a program that sorts lists. For one, we might want to change the order of the sorted list to be decreasing; we could do this by changing `min` to `max`. We might want to sort lists of non-numeric values; fortunately, Python assists us here a lot: the function `min` (and `max`) can be applied to lists containing numeric values, or string values, as long as all the values are the same type. We would not be able to use the algorithm as it is to sort more complicated values, say employee records. From this discussion, we might conclude that the program is somewhat adaptable, but limited. We will see, in later chapters, how to get past limitations that seem to prevent programs like this from being applied to complicated values.

Finally, *reusability*. Could this short script be reused in other programs, and if so, how? The algorithm sorts a list, and does not make any assumptions about the size of the list, so it will not be a problem to apply the algorithm to larger lists of numbers. The algorithm depends on the variables `unsorted` and `sorted`, which is a bit of a problem. Right now, the only way to reuse this program is copy-paste, which is well-known to be a bad strategy for code reuse. The program would be a lot more reusable if the algorithm were encapsulated as a function, and saved as part of a module; that is, of course, why we have built-in Python functions like `sorted()`. It would be fair to say that the example program is not very reusable at all! To restate the theme of this chapter, later on in this course we will see principles to make our programs more reusable.

3.4   How to Use the Design and Implementation Goals Effectively

One of the major learning objectives of this course is to develop the intellectual tools to achieve the goals we've outlined above. As a student in the course, you should be keeping these goals in mind as you work. You should see them as concepts that describe ways that you could improve code that you are writing. You can, and should, ask questions such as:

- How do I know this program works correctly?

- Are there situations that might break this code?
- Are there ways to make this program more efficient?
- Have I made coding choices that reduce future adaptability?
- Which parts of this program might be useful in another project?

You shouldn't be overly concerned about achieving perfect code that satisfies all of our design goals in your first draft, and you shouldn't use them as excuses to avoid making progress on your work. You should keep them in mind, and learn to recognize when doing something now will pay off tomorrow. Perfection is not required; try to be mindful of these questions, and try to gain experience in making these judgments. You'll get lots of examples in this course.

We'll spend some time in the course discussing testing so that your confidence in your software's correctness can be high. We'll discuss techniques to measure efficiency of algorithms, so that you can make comparisons scientifically. Many of the topics in the course give concrete examples of reusable and adaptable software.

3.5 Pedagogical Goals

Presumably you're taking this course to learn to write software, so we have to make room for pedagogical purposes that are often at odds with practical purposes. Students write software to gain experience writing software, even if the software they end up with is not completely correct, and even if they take longer than an experienced practitioner would. Experience is a very important pedagogical goal for students! Taking time to gain experience is necessary preparation, so take time to make mistakes, and to learn to fix them. It's not a waste. It's what you're here to learn!

It's also important for professionals to take time to learn new things, so the pedagogical goals don't end immediately after you finish a course or graduate from a program. Another important aspect of this course, and all university courses, is that you learn enough fundamental concepts that you are prepared for a lifetime in which changes happen fast, and you have to learn new things all the time. One of the things you'll have to learn is how to distinguish a principle that will remain useful throughout your career, from a detail that might change as technology changes. For example, the Python language is a detail; software efficiency is a principle.

4 — Software Development Processes

Learning Objectives

After studying this chapter, a student should be able to:

- Describe the difference between software development and software design.
- Describe the waterfall model of software development.
- Describe the iterative method of software development.
- Describe the test driven design (TDD) strategy.
- Apply TDD to the design of simple Python functions.
- Employ the stepwise refinement design process to simple problems.
- Discuss the importance of version control.
- Explain the purpose of prototyping.

Producing quality software is expensive in terms of time and money. Software design processes are methodologies to help keep the cost of developing software down. Even though this is an introductory level course, it will be worthwhile to consider your own time as a resource that you want to spend wisely. If you are not following a methodology, and if you are not monitoring your time, and if you have no responsibilities to anyone for how you use your time, you are probably wasting large amounts of time.

In this chapter, we'll discuss some practical strategies for students to apply when building software. The topics of software development and design could easily occupy entire courses, and it is not our goal to learn everything about them. The purpose of this chapter is to give students tools to apply when building software for small to moderately-sized projects, including undergraduate assignments and projects.

4.1 Software development processes

Software *development* is usually defined to include the whole life-cycle of a software system, beginning from initial intent, and concluding with deployment and maintenance. Within that cycle, there is a activity called software *design*, which is the process used to plan the software system.

4.1.1 Waterfall model

A very basic methodology is is called the *waterfall model*. The waterfall model outlines all the activities relevant to software development, giving them a sequential order. It's called *waterfall* because *sequential* is a seriously boring word, even if it is completely appropriate here. The waterfall model outlines the following stages:

1. Requirements. At this stage, you figure out what the software is needed for, and what needs to be done. The result of this stage is usually a document indicating what the software must do. The larger the project, the more important this stage is. It is essential that all parties (team members, clients, management) agree on the requirements. We'll say more about requirements as we go.

2. Design. At this stage, the developer(s) come up with a plan describing how to meet the requirements. This is called the design. The design may not include much code, but should present a clear set of objectives and approaches for building the software. The design can be approached as the high level aspects of iterative refinement, without fully completing the application. The output is a specification document, which can be analyzed and scrutinized. The more time spent at this stage, the fewer surprises will come up during the implementation.

3. Implementation. This stage is all about writing the code for the system. Start with the low level functional units, and build upwards. Part of the implementation phase includes testing of the software. The implementation is complete when the design plan is completed.

4. System Testing and Verification. At this stage, the application is checked, to ensure that it meets the requirements.

5. Maintenance. The application has to be delivered, either to a client, to an app store, or put on some kind of storage medium for brick-and-mortar sales. Releasing software into the world is just the beginning of the work! Software to install the application might need to be written. Updates and upgrades need to be planned. Bug reports need to be collected, and bugs need to be fixed.

The waterfall model is straight-forward, and it names all the major tasks in a software development project. The "waterfall" part suggests that there's no going back to a previous stage, which is probably not realistic. Moving along sequentially will work for fairly small projects, but it is not very effective for very large projects. The main problem with the waterfall model is that requirements change very quickly. Another big problem is that for very large projects, simply agreeing all the requirements in advance may be infeasible.

Despite its weaknesses, it is helpful to consider the waterfall model applied to your homework assignments. It gives names to tasks that you didn't even realize you are doing. When you read an assignment question, you are reading an informal requirements document; when you ask questions about it, you are clarifying the requirements. When you think about how to solve the problem, you are doing design and implementation; perhaps you are not in the habit of separating these two tasks, though. Before you hand your program in, you're doing system testing and verification. The waterfall model can help you monitor how much progress you are making, and it can help you to be

objective about your efforts.

4.1.2 Iterative and incremental model

The *iterative and incremental model* of development is based on the tasks identified in the waterfall model, but is slightly more flexible. Rather than figuring out all the requirements in advance of all development, the iterative model starts with a small subset of the full requirements, and applies the waterfall model to it, repeating steps 1-4 (above), as needed. With a small system built, you can get feedback from your client, and more requirements can be added. The system gets larger incrementally, and when the full set of requirements are documented and achieved, the process can enter the maintenance phase of the waterfall model.

The *iterative and incremental model* allows a large project to be constructed incrementally, and getting feedback from the client or customer is useful sooner than later. The drawback of the incremental model is that the requirements can still change, and without careful planning, requirements added later may be inconsistent with earlier versions of the system.

It may be valuable for students to apply the incremental model to assignments and projects. The key thing would be to understand the requirements well-enough that you can iteratively add requirements by the time it is due, so that you can obtain credit for your work. The risk is that you might iteratively step away from a good product, and not have enough time to correct your development trajectory. Still, students benefit from the advice of always having something working, even if it is not complete; testing as you go, rather than all at once, and don't try to build the whole project all at once.

4.1.3 Requirements and Specifications

One of the key tasks in the software development process is establishing the requirements. Software requirements indicate, as precisely as possible, what the application or script is supposed to do. For example, every programming question you've seen in a computer science course starts with a description of what you're supposed to do; your instructors have provided the requirements for you. Outside of homework, gathering the requirements for a project is a normal part of accomplishing the task.

This task almost always starts with an English (or other natural human language) description of the requirements. Requirement specifications written in human natural languages are often slightly vague, and may leave out some information. Even professionals start out with a vague idea about what they are trying to do, and they have to work on it to make it more precise. When a client or a colleague comes to you with an idea, moving from vague statement to precise specification is a key part of the collaboration. Sometimes your colleague or client will make contradictory requests, without realizing it. Sometimes they won't even know what they want.

After the requirements are written, the application needs to be designed. We'll have more to say about this task in the next section. A design document should include specifications for functions or modules (and other abstractions well see in this course, such as objects and ADTs). These are sometimes called specifications. A specification is a precise statement about what a function or module should do, including arguments, inputs, outputs. A specification document may suggest an algorithm at a high level of abstraction, but the specification is a document that one uses to assess the implementation, so it should not be expressed in code. A specification is the precursor to writing code.

Again, the natural starting point for specifications is to describe them in natural human language.

We can try to be fairly careful, but specifications written in English will always be vague unless they are ponderous and impenetrable. Human natural language is not always the right language for specifications. For one thing, specifications written in natural languages are difficult to reason about. It is far more appropriate to use mathematics to write specifications. The necessary math to describe computer science specifications is the language of sets, relations, logics, proofs, and mathematical induction. These topics are taught in second year courses; for now, we have to make do with English.

For courses and assignments, you will find requirements and specifications of varying levels of precision. As you no doubt have observed during your computer science courses, written specifications for assignments do not answer all the questions you may have, and sometimes have to be corrected because they contain mistakes. For students, it is easiest if instructors give very precise and thorough specifications. But part of the task of problem solving is to be able to refine a vague specification, and make it precise. Don't flinch when you have to do this (but you may grumble about it, of course, that's normal).

4.2 Software Design Strategies

Software development includes the task of designing the software system, or application. In the previous course, we said only a little about software design, leaning heavily on the concept of abstraction and stepwise refinement as a way to move from idea to Python script. Stepwise refinement is the process of moving from an abstract description of a computational task to a more refined description by adding detail. We usually start with a pseudo-code description at a very high level, and in the process of adding detail, the task begins to look more and more like a script or a program.

A very powerful approach to software design is called *test-driven design* (TDD). It can be applied the waterfall model or the incremental model during the implementation stage, that is, after you have a plan for what functions and modules you want to build. The main idea is that test code is written first, before any code is written to provide the functionality. This seems counter-productive, but it is powerful for a number of reasons. First, if you can't write a test case for a function you are thinking about writing, you don't understand it well enough to start. Second, with careful design of your test cases, you can know exactly when your code satisfies the requirements; you won't be writing extra code. Third, TDD keeps developers from writing a lot of code all at once, and adding multiple errors, that need time to be debugged. The time you spend alternating between test code and application code is more than made up for in reduced debugging time.

The TDD process is based on three principles:

1. Test first: Do not write a line of code until you have written a failing test for it.
2. Do not write more of a test than is necessary to cause the test to fail.
3. Do not write more code than is necessary to pass the failed test.

The idea is that you develop the function and the tests incrementally, starting with a trivial test that will fail because you haven't written the function yet. Then you write just enough of the function to pass the trivial test. When it does, you add another test, which must fail because your function doesn't do anything yet. Then you write enough code in the function to pass the test. Keep adding to the testing, incrementally, and keep adding to the function, to satisfy the tests.

Test-driven development can be directly applied to projects of small to moderate sizes. To apply TDD strategies, you must have a clear requirements document already in place. You also have to have a design plan, identifying, at a high level, the functions needed to achieve the requirements;

each function must have a specification, written fairly precisely. With this in hand, the TDD approach can be applied to the implementation stage of the development process. Start with the low-level functions (the ones that do not call other functions in your design). Apply the TDD strategy to the implementation of this function, starting with trivial tests, and moving towards completion When you've tested every part of the specification, and your implementation passes all those tests without error, you move on to the next function.

4.3   Version Control

Software development and design is difficult even for experts. There are design decisions to be made before they can be implemented; it is not rare for decisions to be reversed some time in the future. You've probably experienced the problem yourself. You had a program almost working, then you added or changed some code, and made it worse. Now your are faced with a choice: you can undo your changes and try something else, or you keep those changes, and make more. It's common for both options to feel risky, because of the effort you'll need in the future if you make the wrong choice now. You could make a backup copy of your code in its current state, so that if you had to, you could come back to it and try something else. And if you have done this several times, you are faced with another problem: which of the multiple backup versions do you revert to?

Version control is the name for software tools that allow content providers to manage their project content and all their backups in an organized way. Each backup is called a *version*, which is maintained by the version control software. The entire history of all versions of the project is stored carefully, and each version can be documented with a note that summarizes why the version was created. Version control allows a content provider to switch to any stored version at any time, without searching through folders of backup files. Multiple versions can be compared, to see differences in the content. Version control allows teams to collaborate, and helps them keep consistent versions of the project as they work on different parts of the project at the same time.

Version control tools provide a lot of functionality, but we'll focus on the basics. When you start a new project, you *initialize* version control for that project. Every time you accomplish part of the project, you document what you did with a short note, and you tell the version control to store the current state of your project as a *version*. Version control can help you navigate the history of the versions you stored; you'll want your short notes to be descriptive and helpful. You can use version control to compare two different versions already stored, or the difference between the current state with any previous version. If you ever need to return to a previous version, you identify which version you want, and tell the version control to restore that version.

The value of learning version control is that you can safely record the entire history of every assignment question or project, as you build it. If your project is under version control, you can make experimental changes, and discard them with a single button click, getting back to your previous version. If you are using the incremental and iterative development model, you can earmark every feature you add to the project, and if necessary, roll-back to a previous version. You can add or delete code without hesitation, because you can always call it back at any time, if you have been using version control to record your version history.

4.4   Prototypes

A software prototype is an application intended to demonstrate some subset of the requirements of a software project. Prototypes are valuable to the process of software development because they

allow the exploration of design decisions (stage 2 of the waterfall model), before committing to them in an implementation (stage 3). They also allow developers to get feedback from the clients before the final application is complete. Software professionals have wide variety of strategies for using prototypes. A prototype can evolve into a final product by a process similar to the iterative and incremental model discussed above. Sometimes a prototype is developed quickly with the intent to discard it before the implementation of the final project. For this strategy to be effective, a prototype has to be developed very quickly, and the developers have to learn enough from the effort to help them make better design choices for the final implementation.

The value of the prototype concept to us in this course is the idea that it is not foolish to try a few ideas quickly, discover some hidden implications of those ideas, and then throw them away completely to try something new. Students typically commit themselves to a single path of development, and are naturally quite reluctant to make drastic decisions like deleting whole pages of code. With the knowledge that prototyping is a real strategy, and the confidence that version control has the complete history of your work, you should learn to throw code away and try something else.

4.5   Summary

There is still a lot more to say about software development and design processes, and those things must be left to more advanced classes. Take to heart the fact that software development is difficult because it is so easy to make mistakes, and you need to allocate time to managing those mistakes deliberately. If you assume that everything will go well, that no errors will occur, that your understanding of the requirements is perfect, or that they are perfectly stated, you will find yourself underestimating the time it takes you to complete your work. By monitoring your progress, by allocating time for things to go wrong, and by careful planning, you can make far better use of your time. By using version control, you have the freedom to try out ideas, without committing to them, and without making a total mess of your project with ad hoc copies and code that's been commented out. You don't need your development and design plans to be perfect at this point; you simply need to practice using them. This course is intended to teach you to develop software; the answers to assignments are not as important as the process by which you answered them.

5 — Procedural Abstraction

Learning Objectives

After studying this chapter, a student should be able to:

- Define the concept of procedural abstraction.
- Give an example of procedural abstraction, and explain how it helps meet design and implementation goals.
- Describe the concept of a function interface, and its components.
- Write interface documentation for simple functions.
- Describe the role of procedural abstraction as part of the implementation process.
- Describe the role of procedural abstraction as part of the design process.

5.1 Procedural Abstraction

Computer science uses abstraction to help manage the complexity of the software we try to build. Without some strategy to manage complexity, software design is prohibitively difficult. Computer scientists in the 1970s were very alarmed that their ambitions for complex software were beyond their ability to produce. There was a concerted research effort to invent concrete techniques for abstraction for the software design process that would allow software developers to create the kinds of software we see today. We will look at two very specific forms of abstraction: *procedural abstraction* in this chapter, and *data abstraction* in Chapter 6. Both of these concepts are useful to help achieve the design and implementation goals described in Chapter 3.

When we define a function, we are creating a *procedural abstraction*. A function encapsulates an algorithm, i.e., a sequence of instructions that achieve a purpose. More importantly, a function definition separates the implementation (its instructions) from the function's use (its function call). Procedural abstraction allows us to look at software from two different perspectives: implementation, i.e., *how to do it*, and purpose, i.e., *what it's for*.
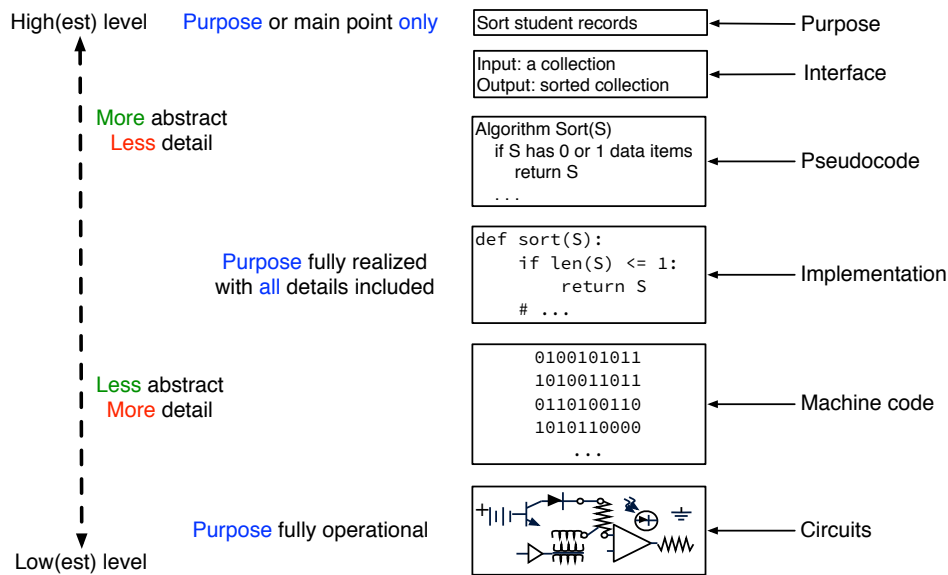
Figure 5.1: Abstraction is a continuum, with many levels. Computer Science can be studied entirely in the top half of the continuum, but it's only a practical science because of the efforts made in the levels represented in the bottom half of the diagram.

5.1.1 The benefits of procedural abstraction

Procedural abstraction enhances *reusability*: a function containing encapsulated instructions can be called multiple times without copying the instructions. Copying blocks of code carries the risk of copying errors that you have not detected yet. Your work increases if you are forced to debug multiple copies of the same code. On the other hand, re-using a function multiple times may cause the same error to happen multiple times, but if you fix the function, you've fixed it once-and-for-all, which is far better for you!

Procedural abstraction enhances *adaptability*: changes to the encapsulated code can be made often without affecting any other part of a program. This also facilitates *efficiency* improvements indirectly: encapsulated code can be improved, or even replaced with something much more efficient, and every call to this function benefits from the improvement! It enhances *correctness*: a function or procedure can be the target of focussed testing; this can often be accomplished without having to test the whole program all at once. It allows us to use variable names in the scope of a function without conflicting with other variables of the same name in other functions. Most importantly, once written, we can use the function by remembering its purpose, without having to keep the implementation in mind.

5.2 Multiple levels of procedural abstraction

Figure 5.1 gives a visual representation of the different levels of procedural abstraction we work with in Computer Science. The highest level of abstraction is a statement of purpose, and the lowest level of abstraction is the purpose fully realized as behaviour on a computer. It is possible to consider even lower levels, e.g., the quantum behaviour of sub-atomic particles as they interact in the computer,

but the author's diagramming skills were not up to the challenge.

Between these two extremes is a continuum, and the diagram gives some key points in this continuum. You'll (hopefully) recognize an incomplete snippet of Python code near the middle of the continuum; the precise location is not really important, only that it's between the two extremes. The activity of software design usually operates in the top levels of the continuum, although it is not uncommon for developers to have to work at the lower levels too. In this course, we're likely to stay higher up in the continuum; the lower levels are just as important, but we can't learn everything all at once. There are courses that will teach you more about the lower levels later on.

Another important position highlighted in the continuum diagram is labelled *interface*. The *function interface* describes the function's input-output relationships; this is sometimes called a *specification*. The function interface establishes the requirements for correctly using the function, so that it achieves its purpose, and therefore plays a critical role in design and documentation.

5.3 Describing an Interface

The documentation of the interface for a function should provide enough information for someone to use the function without needing to study the implementation. There are lots of ways to document an interface, and many software development teams design their own standards and requirements. In this course, we will adopt the following practice:

> **Practice 5.1**   Every function's interface documentation will consist of four parts, as follows.
>
> **Purpose:** A brief statement of the purpose of the function, in plain language.
>
> **Pre-conditions:** A description of every parameter for the function, including assumptions or restrictions on their values.
>
> **Post-conditions:** a description of any effect produced by the function that could be detected after the function returns, e.g., changes to data defined outside the function itself, or any kind of I/O; optionally omitted if the function has no effects.
>
> **Return:** a description of the value returned; optionally omitted, if a function returns no value.
>
> ∎

Whether you write your function in Python, pseudocode, or any other programming language, these are the aspects that professionals will want to see in your documentation. Here's how our convention looks in Python:

```
 1  def sqrt(x):
 2      """
 3  Purpose:
 4      Approximate the square root of a given number, x.
 5  Pre-conditions:
 6      x: A non-negative number
 7  Post-conditions:
 8      (none)
 9  Return:
10      a number y such that y*y is close to x.
11      Note: if x is negative, the value None is returned
12      """
```

```
13        if x < 0:
14            return None
15        else:
16            y = 1.0
17            while abs(x - y*y) > 0.001:
18                y = (y + x/y)/2.0
19            return y
```

The pre-condition gives a restriction on the value for the parameter, *x*, indicating that it must be greater than or equal to zero. In the above example, we wrote (none) under the heading *post-conditions* to indicate that the function does not have any effect apart from returning a value. A function with no effect other than returning a value is sometimes called a *pure function*. Notice that the return value is described informally, but very carefully, and in a way that allows the return value to be checked.

In Python, it is convention to put the interface documentation in the doc-string of the function. This will allow anyone to access your interface documentation in Python's interactive mode, as in the following example:

```
>>> help(sqrt)
Help on function sqrt:

sqrt(x)
    Purpose:
        Approximate the square root of a given number, x.
    Pre-conditions:
        x: A non-negative number
    Post-conditions:
        (none)
    Return:
        a number y such that y*y is close to x.
        Note: if x is negative, the value None is returned
```

It's important to note that the format of our documentation is a convention, and you will be expected to use it (unless instructed otherwise, e.g., in certain examination questions). Every computer science course will have different expectations, and every software development project team will have different formats. Our conventions are established by the goal of emphasizing what's important about documenting any interface. Once you get in the habit of documenting your designs, the format really isn't important enough to argue about. In other words, the format is a detail.

5.3.1   Documenting the interface after defining the function

In a course such as this one, it is very common for assignments and tests to ask students to design a function to achieve some purpose, either to practice some concept, or to demonstrate their mastery of a concept. What often ends up happening is that the assignment question usually does a pretty good job of describing the interface of the function. It is also quite natural for students to implement the function as the primary intellectual activity, and then, as an after thought, and perhaps a bit resentfully, go through the trouble of documenting the interface. To this, the author says "Fair enough!" But remember that assignments and tests are unusually unrealistic. We have to encourage

professional practices in artificial environments so that you can bring those practices with you in more natural, less constrained environments. Be diligent in documenting your interfaces, and learn to do it right away (not at the last moment), and learn to do it quickly. This is a practice that will help you a lot.

5.4 As part of the implementation process

One of the ways you can use procedural abstraction is by taking some code that you have already written, and making it the body of a (new) function. For example, we could start with the code from Chapter 1 1, and we might simply wrap a function definition around it:

```python
def selection_sort():
    """
    Purpose:
        Displays the list [0, 1, 2, 2, 2, 3, 5, 6, 7, 8, 8]
        to the console.
    Pre-conditions:
        none
    Post-conditions:
        A list appears on the console.
    Return:
        none
    """
    unsorted = [3, 2, 5, 7, 6, 8, 0, 1, 2, 8, 2]
    sorted = list()
    while len(unsorted) > 0:
        out = min(unsorted)
        unsorted.remove(out)
        sorted.append(out)
    print(sorted)


selection_sort()
```

Armed with this function, we now have the power to sort that one specific list of integers as many times as we like! The interface to this function is very simple: no inputs or return, and a very specific effect, namely some output appears on the console.

There is good reason to doubt the value of this implementation. There is no need for a function with that specific behaviour. The reason it's not very useful is that it makes too many decisions. The function decides which list to sort, and it decides how to sort it, and it decides what to do with the resulting sorted list.

With a few more edits, we can generalize the function and make it more useful. First, we can add a parameter to the interface, so that the function's input is the list to be sorted. This allows the decision about which list to sort to be made outside the function. Second, we can return the sorted list, which allows the decision about what to do with it to be made outside the function as well.

```
1   def selection_sort(unsorted):
2       """
3       Purpose:
4           To order the val ues in increasing order.
5       Pre-conditions:
6           :param unsorted: a list of numbers
7       Post-conditions:
8           All values are removed from unsorted.
9       Return:
10          a list of all the values from unsorted,
11          but in increasing order.
12      """
13      sorted = list()
14      while len(unsorted) > 0:
15          out = min(unsorted)
16          unsorted.remove(out)
17          sorted.append(out)
18      return sorted
19
20
21  data = [3, 2, 5, 7, 6, 8, 0, 1, 2, 8, 2]
22  data_sorted = selection_sort(data)
23  print(data)
```

The interface documentation is careful to describe a pretty significant side-effect of the function: namely that the function empties out the original list! In effect, this function's interface tells us that "You can't sort your data and keep it, too." For that reason it's not a great function; but this example demonstrates the value and importance of documenting post-conditions.

There are several lessons we can learn from this example. First, making a function by encapsulating code that you've already written is a perfectly normal implementation activity. In fact, it is so normal that many interactive development environments (IDEs, such as PyCharm) have *refactoring tools* that assist developers in making these kinds[1] of changes. Second, the example emphasizes procedural abstraction as a technique for generalization; a function is more general if more of the decisions about what should happen inside the function are made outside of the function. For example, a function that returns the sorted list is more general than a function that displays the sorted list to the console. Third, it suggests a very simple problem solving strategy: when faced with a computational problem, try to solve a very concrete instance of the problem, just by writing a script. When the script works, use procedural abstraction to make a general tool out of it.

5.5   As part of the design process

In the previous course we introduced a simple design process called *iterative refinement*. Starting with a very abstract statement of the purpose of the application, the iterative refinement process keeps adding detail to abstract statements, either by adding information to a statement, or by breaking an

---

[1] There is a whole application menu devoted to refactoring tools in most IDEs. Check it out in PyCharm.

abstract statement into a sequence of somewhat more detailed statements. Procedural abstraction plays a role in this process, by allowing a designer to express a purpose in terms of a function that has not been written yet. Each abstract statement in the iterative refinement process could be a function call to a function that has not been implemented yet. The power of procedural abstraction comes into play because you can talk very intelligently about a computation in terms of *what it does* without having an implementation. In other words, you can design an application by starting with the names and purposes of a few very high level functions, without having to implement them. You can design the interfaces of these functions by deciding how they will work together, without having to decide how they work.

> **Principle 5.1**   The process of defining an interface is an essential part of any design process. ∎

To design an interface, even if we're not exactly sure what the algorithm should be, we start by writing a statement of purpose, written as clearly as possible. Then we can add pre-conditions, post-conditions, and a description of what the return value should be, again, as carefully as possible, without ever worrying about *how it works*.

The interface should provide enough information to allow a software developer to implement the function or procedure, without requiring any further information about how it would be used. It should also provide enough information to allow a software developer to know how to use the function properly, without studying the encapsulated instructions.

Good interface documentation takes a moment or two to write down, but it serves a number of purposes beyond communication. First, it is a good indicator for understanding the problem. If you cannot write down the interface, then you do not understand what you are trying to do, and you need to think a bit more carefully. That's not to say that you need to have an algorithm. It's perfectly normal to know what you have to do without yet knowing how you're going to do it. But if you don't understand what you're trying to do well enough to write down an interface, you really don't understand it well enough to even start thinking about algorithms!

Second, designing an interface helps you focus on a single task. You can make a number of decisions about your function just at the interface level, and these will guide the rest of your problem-solving. Even if you haven't worked out the interface completely, working on it will help. For this reason you should treat the interface as part of the design, and part of the problem-solving, even when it is tempting to see it as time-wasting work instructors force you do before you hand something in.

6 — Abstract Data Types

Learning Objectives

After studying this chapter, a student should be able to:

- Describe the concept of data abstraction.
- Give an example of data abstraction, and explain how it helps meet design and implementation goals.
- Define the concept of data type in terms of values and operations, and give examples.
- Define the concept of data structure in terms of values and structure, and give examples.
- Define the concept of abstract data type, in terms of what is hidden, and what is exposed, and give examples.
- Identify the data components and behaviours of multiple abstract data types.
- Implement a coherent abstract data type, with loose coupling between components and behaviours.

6.1 Data Types

A data type is a fancy word for yet another concept you are familiar with. Think about numbers. There are integer numbers, real numbers, and, in computer science, floating-point numbers. Numbers are just values with a specific literal appearance: mostly digits. We know that numbers can be added, multiplied, squared, cube-rooted, etc; these are operations for numbers. There may be some operations that don't always make sense, such as dividing a number by zero.

> **Definition 6.1**   A *data type* consists of a set of values, and some operations on those values.

Integers are a data type. Floating-point numbers are a data type. They are different data types, though they have some values and operations in common. Booleans are a data type. There are just two values in that data set, and the logical operators (AND, OR, NOT) are the operations.

Data types can be built into a programming language. In fact, most languages have integers, floating-point numbers, and Booleans available for use. The operations provided by the language provide the only means to manipulate these data types.

Data types can also be added to a programming language by writing programs. This turns out to be a very significant part of programming. The basic idea is to figure out what's special about the computational problem you're trying to solve, and design (or use) a data type that helps you write the program that solves it. We'll see many examples of this during the course.

6.2 Data Structures

A data structure is a special kind of a data type.

> **Definition 6.2**   A *data structure* is a data type with compound data values that are organized (or *structured*) in a particular way.

Python strings and lists are data structures: the values are compound, i.e., a collection of some kind, and there is a sequential organization imposed on the values. A Python dictionary is a data structure, too. It organizes the data in a clever way to make indexing efficient; we'll have more to say about this organization in a later chapter.

Data structures, being a special kind of data type, also have operations. You are aware of some of the list operations, at least: indexing, append, remove, to name a few. In this course, we'll learn about several typical data structures that are frequently used to solve computational problems. We'll learn what they are, how to use them, and how you can implement them.

The fact that data structures deal with compound data is important. For example, a list contains a sequence of values. In a textbook, or in a Python program, we represent a list using text symbols, e.g., `[1, 2, 3]`. But inside the computer the list has to be stored in some kind of computer-based organization, so that the operations can do their job. This organization is what makes a data structure what it is.

6.3 Abstract Data Types (ADTs)

Data abstraction is based on the idea that we can look at compound data from three different perspectives: its purpose, its operations, and the implementation of its structure.

You are already acquainted with Python's lists and dictionaries data structures. For example, you should know that you can store data sequentially in a list. That's the purpose of a list, after all.

You also learned a few of the operations for lists, and can probably use them in your own programs. Lists allow random access (i.e, using indexing), and sequential access (i.e., using for-loops and comprehensions), and dictionaries allow lookup by key (i.e., using key-based indexing); these examples are just a few of their operations. But you have probably never seen the code for those operations, and you probably only have a rough idea about how the data is actually stored. For many purposes, a rough idea is good enough!

Keep in mind that the operations are are procedural abstractions, in the sense described in the previous chapter. We can understand the operations described by their interface, without seeing the implementation of the operations.

Earlier we said that lists were data structures, and that is true. Underlying all the Python operations, lists are organized in a way that the computer understands. That makes them a data

structure. But even now, we haven't seen any of the details about the organization. In this sense, lists are abstract. The details are hidden, and we do not need access to these hidden details to use them, because we have the interface to all the operations for lists.

Lists and dictionaries are typical examples of abstract data types (ADTs). They are abstract in the sense that we use them without having seen the Python implementation of the operations, nor the organization. We do know, however, that Python lists and dictionaries allow us to store collections of data; this is their purpose.

> **Definition 6.3**   An *abstract data type* (ADT) is a data type that encapsulates data (using code) to prevent almost all access to it, except through carefully designed operations.

There are many benefits to restricting access to the data. If the operations for accessing the data are correctly defined, then the data stored cannot be corrupted by using them appropriately. In other words, correctness is enhanced by restricting access to data. Robustness is improved, as the operations can be designed to manage unexpected situations. Adaptability is improved, as the operations can be standardized (like the operations on Python's lists).

Unrestricted access to data leads to software that is not robust, not adaptable, and presents very grave danger of that data being corrupted. In Python, it is difficult to imagine data corruption being much of a problem, but it's very common, especially in languages like the C programming language, which was designed for speed, not safety.

There are two crucial points here. The first is application independence: ADTs allow software designers to create a concrete data structure, and provide useful operations for it, independently of any consideration about who is going to use it, and what their overall purpose is. The second is implementation independence: ADTs allow software designers to obtain all the benefits of data organization in their application through high-level operations. Code that uses ADT operations is easier to understand.

The cost of using an ADT is that you have to read some documentation very carefully, and be sure that you are using the ADT and its operations properly.

6.4 Examples of Abstract Data Types

In this section, we'll present two very simple examples of ADTs. Neither are deeply useful in most circumstances, but they meet all the requirements for a complete ADT. The lesson to take away is that an ADT hides data, and allows access to it only through its operations. The examples also demonstrate how ADTs can be implemented in languages that do not have classes or objects. ADTs are completely general, and can be implemented in any language. While we'll show you how to use classes and objects to implement ADTs later, we'll use the first part of the course to demonstrate that you don't need objects and classes to do it.

The technique we'll use for the first few chapters is that our ADTs will consist of procedures and functions whose purpose is to create or modify a given data structure. We'll always have a procedure to create an empty or initial value, and every remaining operation of the ADT will have this data structure provided as its first argument.

> **Principle 6.1**   When using an ADT, refrain from accessing the data in the data structure directly: always use the ADT operations, and never take advantage of any knowledge that you might have

of the data structure itself.   ∎

This principle will help improve program correctness, adaptability, and reusability. It take practice and discipline, and more importantly, time, which always seems in short supply.

Python was designed to give the programmer maximum flexibility, so in Python there's no way to force programmers to follow the ADT discipline. But in other languages, it is possible for a language to restrict the way programmers use an ADT, so that a program can only use the operations.

Our ADTs will always be implemented in a Python script, or *module*, which defines the operations. The ADT implementation will typically only define the operations, and do nothing else; it will just be a collection of function definitions. When we want to use an ADT that we've designed and implemented, we'll `import` the ADT module. Any script that uses an ADT will be called the *application*, or sometimes, the *client*.

6.4.1 Registry

A registry is a data structure that records a Boolean value for a given number of events. You might be thinking it sounds a lot like a list or array, which is true. But a registry will only store Boolean values, whereas a list can store anything.

A registry has four operations:

- `Registry.create(n, bval)` creates a new registry, giving it exactly *n* elements, each initialized to the Boolean value `bval`. The new registry is returned.
- `Registry.set(reg, i)` stores the value `True` in the given registry `reg` at index *i*.
- `Registry.reset(reg, i)` stores the value `False` in the given registry `reg` at index *i*.
- `Registry.is_registered(reg, i)` returns the value stored in the given registry `reg` at index *i*.

The full implementation of the Registry ADT can be found in Appendix A.

6.4.2 Statistics ADT

The purpose of this ADT is to compute the *mean* and *variance* of any numeric data set. The mean (also known as *average*) gives us an idea of what the typical data value is like. Another way to think about the mean is as a way to describe the data set's central tendency; data values are spread out around the mean value. The average is calculated by summing all the data values, and dividing by the number of data values. The variance tells us how much the data values vary from each other. If it's close to zero, then all the data values are very close together (on a number line); if the variance is large, then the data set has values that are quite spread out. Mathematically, the "variance" is defined as the average of all the squared differences between each data value and the mean value.

If the final sentence in the previous paragraph seems confusing to you, that's okay, because English is not the right language for it. There are two immediately better languages: mathematics, and a programming language (any one of them). We use math to describe a calculation precisely, and maybe to re-express it one way or another by symbolic derivations. We use a programming language to *implement* the calculation described by the mathematics.

The mean of *n* numbers is traditionally given the symbol μ, and the formula for the mean is simply written as follows:

$$\mu = \frac{\sum_{i=1}^{n} a_i}{n}$$

The $\sum$ symbol indicates a summation, i.e., a repeated addition of the term to its left. In fact, a mathematical summation is very similar to the Python function `sum()` (see below). You've probably written Python code to compute a sum or an average many times. The variance of those n numbers is traditionally given the symbol σ^2. The formula for variance is as follows:

$$\sigma^2 = \frac{\sum_{i=1}^{n}(\mu - a_i)^2}{n}$$

The square root of the variance is called the standard deviation (whose symbol is simply σ). These formulae can be expressed in Python as follows:

```python
mu = sum(data)/len(data)
sigsq = sum([(mu - x)**2 for x in data])/len(data)
```

Now that we understand the computations necessary to calculate mean and variance, we're going to hide the calculations behind an ADT that can be re-used without memorizing formulae.

A Statistics ADT has four operations:

- `Statistics.create()` creates a statistics record. The mean and variance are undefined until data is added. The new record is returned.
- `Statistics.add(stat, val)` adds the data value `val` to the given statistics record `stat`.
- `Statistics.mean(stat)` returns the mean of the data previously added to the given statistics record `stat`.
- `Statistics.var(stat)` returns the variance of the data previously added to the given statistics record `stat`.

The full implementation of the Statistics ADT can be found in Appendix A.

6.4.3 Example

In the following example, we use the Registry ADT to keep track of the outcomes of simulated die rolls. We're also using the Statistics ADT to find the average number of die rolls until we see the first repeated number.

```python
import random as random
import Registry as Registry
import Statistics as Statistics

sides = 6
stats = Statistics.create()
trials = 100

for i in range(trials):
    die_record = Registry.create(sides, False)
    repeat_observed = False
    roll_count = 0
    while not repeat_observed:
        roll = random.randrange(6)
        roll_count += 1
        if Registry.is_registered(die_record,roll):
```

```
                repeat_observed = True
           else:
                Registry.set(die_record, roll)
     Statistics.add(stats, roll_count)

print("Rolled the", sides, "sided die", Statistics.mean(stats),
      "times (on average) to observe a repeat roll.")
```

For a 6-sided die, the average number of rolls to get the first repeat is about 3.8.

6.4.4 Discussion

We noted above that a Registry seems to be nothing more than a list of Boolean values. Since you are already familiar with lists, and Boolean values, you might well question the utility of the Registry ADT.

You could replace each of the operations with Python code using lists. But think about the mistakes you could make with a list that you can't make with a Registry. A registry only stores Boolean values. It would be a mistake to store any other value in a registry, and the registry interface simply does not allow that kind of error to happen. The operations also go some distance towards documenting the code: the operations are descriptive of the purpose of the code, and using plain old Python list operations is less descriptive.

Any application using a Statistics record could also be rewritten to use a couple of Python variables, and a line or two of code to compute an average of a data set. It would be slightly more effort to replace the calculation of the variance; for one thing, you'd have to look up the formula, which you may not know. Then you'd have to check that your calculations are correct. Using the Statistics record is easier.

If you dig around the code (Appendix A), you can discover how the Statistics record works. You might have thought that it stores all the data in a list, and then calculates the mean and variance from that list. But the code shows that it's possible to implement a Statistics record so that you don't have to store all the data; you can calculate the mean and variance from 3 variables, without storing the whole data set inside the record. This can save a ton of space, especially for large data sets! If you were not using a Statistics ADT, you would have to design a solution to calculate the mean and variance. Using the Statistics ADT, that work was done by someone else for you to use. This is the true value of ADTs: someone can make some clever design decisions and you just need to know how to use them.

Another advantage to using an ADT is that once you implement it, and provide an interface for each of its operations, it can be re-used in any program that needs this functionality. Furthermore, as long as the operation interfaces do not change, you could improve the ADT by using a different organization, or a different algorithm, without having to change any of the software that uses the ADT, because all of the changes are hidden behind the interface.

The investment that you make in using and designing ADTs pays off more when your programs get bigger. If you don't practice using ADTs when programs are small, you will not have the experience to use them when you really need them!

In general, using ADTs directly addresses several of design and implementation goals from Chapter 3. ADTs enhance correctness, because data cannot be changed except through well-defined operations. These operations can be tested thoroughly. ADTs are also highly re-usable, and adaptable.

As a final comment, remember that these two examples are just examples to demonstrate a

concept. In other words, the examples we used in this chapter are details to demonstrate the principles of abstract data types. If you think ordinary Python code is more direct and simpler than these two examples, that's an acceptable point of view, because our example applications were fairly small. There are more interesting ADTs to study in the coming chapters, and their benefits will be a little more convincing.

7 — Software Testing

Learning Objectives

After studying this chapter, a student should be able to:

- Explain the importance of test coverage.
- Compare the differences between black box testing and white box testing.
- Explain why the creation of correct program components is important in the production of high-quality software.
- Apply a variety of strategies to the testing and debugging of simple programs.
- Construct, execute and debug programs using a modern IDE and associated tools such as unit testing tools and visual debuggers.
- Construct and debug programs using the standard libraries available with a chosen programming language.

Testing is a very broad and important topic. Testing processes range from the work that programmers do with their own code during development, to what a project team does before releasing a product. There are advanced courses in software engineering that delve deeply into the topic of testing. This chapter will give you some tools that will help you develop software for courses and projects before you get to those more advanced courses. There are practices that will help you complete your work on time, and with confidence.

Software errors occur because normal human behaviour includes making mistakes. Any plan to deliver software (including such mundane software as assignments) that does not include a substantial amount of time devoted to testing and debugging is naive and unprofessional. Hopefully you will see testing as an activity that you do because you don't know what bugs lie still hidden in your code, waiting to make their presence known to you at the least convenient time.

It's very difficult to see a mistake in code even when you are sure there is one. No doubt you've experienced this, but it's important to understand that it's a difficulty for programmers at all levels of experience. Optical illusions are evidence of human brains imposing a belief on what appears to be

true, contrary to actual truth. Your human brain actively tries to make the world fit to your interests, hiding the truth about your code behind what your brain wants to believe about your code. You can gain experience, and learn to see somewhat more carefully, but the best and only strategy against human error is good testing.

7.1   Software error horror stories

There are some pretty serious consequences for software applications that get released with undetected errors. Errors can be harmful to people, or may cause damage to equipment or the environment.

In the 1980s, a medical linear accelerator known as **Therac-25** was designed to administer radiation therapy to cancer patients; this model was the first machine of its kind to be completely controlled by software. Due to a combination of factors, including the presence of software errors in the input module, and the inadequate administrative process for reporting accidents, several patients received massive radiation overdoses, leading to disability, and at least 3 deaths.

In 1996, ESA's $370M **Ariane 5 rocket** disintegrated 37 seconds after its first launch. The direct cause of the failure was a software error in which a very large value produced by an accelerometer was rejected by the autopilot software as spurious, when in fact it was accurate. As a result, the autopilot software halted abnormally (a software system crash), the rocket veered out of control and disintegrated.

In 1998, NASA launched their $125M **Mars Climate Orbiter**. The software used on Earth by scientists to control the orbiter remotely sent control signals to the orbiter using imperial units (lbs, feet, Fahrenheit, etc), but the orbiter itself was programmed to receive those signals in metric units (grams, meters, Celsius, etc). The mismatched expectations caused the orbiter to fail to establish a stable orbit around Mars, and it crashed.

Software errors are can also be damaging to scientific reputations. Researchers at the **Scripps Institute** borrowed some software for the analysis of protein crystal structure from another lab, and published five significant papers in very highly ranked journals. Then they discovered that the software had one function whose output flipped the sign of some numeric values, invalidating their entire analysis. The team had to issue public retractions for their papers in these highly respected journals.

The cost of software errors you make in assignments and course projects will hopefully not be measured by lives lost, or equipment ruined. The costs to you will be measured in your own time spent working. The more time it takes for you to find and fix the errors, the more likely it will be that you won't be able to complete your work.

7.2   Terminology

A *software fault* occurs when the software does not do what it is supposed to do, or when it does something that it is not supposed to do. A fault is behaviour that can be observed; it's a symptom, an indication that something went wrong. As a symptom, a fault has a cause, which is often called an *error*. The error can be as simple as a line of code with a typographical mistake, or it can be as difficult as a module whose implementation does not solve the problem it was intended to solve. A error may not be observable directly, and it may cause a cascade of observable faults, one after the other. A error can also be the cause of a fault that happens quite far away from the error.

The informal definition of suggests that we know what the program is supposed to do. A *specification* is a written statement or document describing what the program is supposed to do. In a

software development project, the specification document is written, perhaps in consultation with a client, before the project begins. In our first year courses, we present assignment questions (and exam questions) as clearly as we can; these are specifications. We try to be clear so that you know what to do, and so that you know when your program doesn't answer the question. But usually, the world is not so helpful. Developing a correct specification document is just as time consuming as developing a program, and just as important.

A *test case* is a specific input for the program or function, or perhaps the conditions under which the code is run. A test case consists of the input values (or other context) to try, the expected or right answer, and the reason or purpose to run the test. In the previous course, we recommended use of a standard form for test cases:

Input(s):	Description of inputs.
Output(s):	Description of expected outputs.
Reason:	Reason for test case.

An *oracle* is any method that can tell us whether we have the right answer for a given test case. For example, the square-root button on a calculator app can serve as an oracle for the `sqrt()` function. This kind of oracle provides the right answer. An oracle can also be a technique to check whether an answer is correct. For example, it is possible to check if x is the square root of y by checking that $x^2 = y$; you don't need to know the square root of y to check. An oracle can be external to the software, or it can be a function within the software, or it can be a function written entirely for the purpose of testing another function. For some test cases, the oracle may be an expert who has provided expert information, which developers use as a standard. The specification document cannot be an oracle except for very simple, almost trivial, programs; a specification is an abstraction of all the oracles for all the functionality of a program.

An important testing concept is *test coverage*, which is an objective measurement of the extent of the testing that has been done. Typically, test coverage is measured in terms of lines of code, number of branches, number of functions, or number of modules tested. Naturally, more coverage is better, but as you are aware, you don't have infinite time. Your goal should be to get the most confidence out of the testing you can afford to do.

Software can be tested on a number of different levels. When testing is done on individual functions or methods, we call it *unit testing*. When testing is done on functions or modules working together, it is called *integration testing*. Finally, when completed applications are tested, it's called *system testing*

7.3   Testing as error discovery

We test our software so that the faults in the software are exposed. Once exposed, we can attempt to fix them, but the process of fixing faults, i.e., debugging, is separate. It is very important to reject the idea of testing as the process that proves your software has no faults. As we tried to suggest in an earlier section, we are too easily mislead by our own psychology; we can easily fail to find faults by inadequate testing processes, which proves nothing.

> **Principle 7.1**   Testing is the active process of error discovery.   ∎

A good attitude for testing is a scientific one. In science, we never try to prove that our hypothesis is true. Scientists design experiments that attempt to reveal that the hypothesis is false; a negative

result means that the hypothesis was not disproved, but this does not imply that the hypothesis is true. If, after vigorous experimentation, the hypothesis cannot be shown to be false, we tentatively increase our confidence that it is true.

Software testing has to be just as vigorous to prove the presence of faults. The hypothesis we make during testing is that software is correct; testing tries to prove the presence of errors. Vigorous testing might not reveal any errors, but we can never conclude from a failure to find errors that our software is error-free. The strongest conclusion we can draw is that faults may be present that have not been discovered yet.

When we write software, we hope that we can keep the number of faults small, and that the software faults that do happen can be fixed relatively easily. However, we must always expect errors, and we must be dedicated to finding and eliminating them. This takes a lot of time, and cannot be considered an after-thought; you are not done coding until you are done testing and debugging your code. It's so important, I'll highlight the advice.

> **Practice 7.1**   Testing and debugging has to be part of your development plan.   ■

7.4   Exhaustive testing is impossible

All software development projects have a budget for testing. There are developers whose main job on a project is to plan and implement the testing needed to be sure that the product meets the specification. And every developer performs testing on the work they contribute to a project, before it goes to the testers. Even your time budget for homework assignments has to include time for testing and debugging.

In the previous course, we covered some strategies for testing, to provide a starting point. We presented the concept of *black-box testing*, where test-cases were designed based entirely on the interface of a function (inputs, outputs, and effects); or more generally, on the purpose of the function. The term *black-box* implies that the test-cases were designed without looking at the code that is being tested. Similarly, we presented the concept of *white-box testing*, where test-cases were designed based entirely on the lines of code that make up the function or program. White-box test cases are designed to force execution of certain lines of code, or certain paths through the code, such as checking complex conditionals or nested loops.

Most functions or programs can be applied to infinite combinations of inputs, and you can't try them all. Many complex functions or programs will have a very large number of possible paths through the code, and it may be impossible to attempt to identify them all, much less test them all. Even if you test all the paths through your current program, there may be lines of code or paths through your program that you did not write because you misunderstood the problem. In other words, 100% coverage with white-box tests may not be enough!

The law of diminishing returns applies to testing: each additional test adds confidence, but the increase in confidence due to an additional test case gets smaller as you add test cases; the cost to implement each test is roughly constant. The goal is to find a set of test cases that is small enough that it is not a waste of effort, but is still effective at finding the most faults. Finding good test cases is at least as hard as writing code.

7.5 Designing test cases

You can't rely solely on black-box testing or white-box testing, but you can begin your test case development with a mix of cases coming from both white-box and black-box points of view.

To address the the problem of a seemingly infinite range of test cases, we introduce the idea of test case *equivalence classes*. In general, an equivalence class is a set of things that are equivalent from a certain point of view. For example, we can consider the set of even integers an equivalence class, from the point of view of looking at whether a number divided by 2 leaves a remainder or not.

A set of test cases is called an equivalence class if either of the following is true:

- They all cause same path through the code (white-box).
- The result on any test case in the set is the same on all the others (black-box).

In other words, if we can identify a finite number of equivalence classes, we can reduce the infinite combinations of possible inputs to a finite set of examples from the equivalence classes. All we need is one or two test cases from the various equivalence classes, and we can cover an infinite range of possibilities. The number and description of equivalence classes for testing depends on the code you are trying to test.

Equivalence classes express a range of test values, for example:

- Numbers: E.g., positive, zero, negative, even, odd.
- Strings: E.g., non-empty strings with no blanks; non-empty strings with blanks; empty strings.
- Lists: E.g., empty; singleton; all the same; increasing order; decreasing order; even size; odd size.

Of course, these examples show the kinds of equivalence classes you might start thinking about. If your program or function has multiple inputs, equivalence classes will be combinations of inputs. For white-box testing, check conditions using <, >=, etc. Useful equivalence classes for testing a loop include the range of inputs that cause a loop to do exactly one iteration, exactly zero iterations, and more than one iterations.

Another important concept in test case development is *boundary cases* (also called *edge cases*). Equivalence classes express a range of possible inputs, and the ranges will be mutually exclusive. More importantly, the ranges will meet at the boundaries of the equivalence classes. Values on or near the boundaries are very important test cases, and you should include test cases at the boundary, and as close as you can on both sides of every boundary.

For the kinds of programs you will write as a student, you can make certain assumptions about how your program will be executed. For example, we have tended to assume that inputs to functions will be of the correct type, and that functions should guard against invalid or implausible values within that type. This is a good way to avoid lots of unhelpful complexity in a first year course, but it it not sufficient for professionally developed software. Your testing should include unexpected values or types, even if they crash your program. In first year, it's okay for your Python program to crash when given values of the wrong type; but professional developers should be sure that their programs don't silently accept values of the wrong type and succeed by doing something wrong. You will learn techniques to recover from errors due to incorrect type in later courses.

It's fairly easy to think of test cases for checking that the program does what it should. Make sure you consider test cases for checking that the program does not do what it should not do.

7.6 Writing test scripts

A test script (or test driver) is a program that tests other programs. In the previous course, a test case was written in the form of an if-statement that compared the output of a function to the expected output. Our test cases only reported faults, and remained silent on successful tests. We also suggested that a test case could be represented by a dictionary, and that a suite of test cases could be put into a list. In the terminology of this chapter, we recognize the practice from the previous course as the very simplest form of unit testing, in which you could easily compare the output of a function to the known, correct value. That's not always so easy, especially when the inputs and outputs are complicated, like lists or dictionaries, or the data structures we will see later in the course.

To deal with more sophisticated functions, we have to write more sophisticated test scripts. This may involve writing functions that process the output of function in some way. Of course, any functions you write to help with testing are worthless if they contain errors. Fortunately, these helper functions are going to be simpler than the functions they are testing, and so the testing is also going to be easier.

You will sometimes need to write functions or scripts that initialize some data, and set up a context in which to run the test. This kind of script is called a *test harness*. A test harness may have to initialize the data in a way that simulates or pretends to be the result of some other function in your program. For example, you may be testing a function to check for duplicate values in a dictionary. You could write a dictionary literal, but it would be just as easy to write a script to create the dictionary literal programmatically, i.e., but a sequence of assignment statements.

7.7 Debugging

When your testing reveals a fault, all you really know is that your program did not perform as intended. Knowing you have a fault does not tell you how to fix the error! To find out what your program is *doing wrong*, you need to know what your program is *doing*. Do not assume that an error has a quick or easy fix. It's a natural human tendency to hope, but the professional attitude is more cautious. Errors are *almost always where you are not looking*, so you must continuously challenge your own assumptions about what is causing the faults you've found.

It is very tempting to assume all faults as if they were independently caused. But this is very bad reasoning, and can cause you more trouble, not less. The problem is that you can almost always change your program to prevent any single fault from occurring without actually identifying the error. Such changes may only cover up a error that occurred earlier.

If you have a failed test case, write similar test cases within the equivalence class and at the boundaries, and see if they fail too. Once you are sure that the code makes the same mistake on the equivalence class, you can focus on the parts of the code that are associated with that class. You should always be careful not to make a change to your code too soon, because you could be introducing another error.

Try to come up with a hypothesis for the error that caused the fault. Use your hypothesis about the error to write more test cases for it. If your hypothesis is right, these new test cases should also fail before you fix the error, and succeed after. Do not make any change to your program unless you have a hypothesis, and you have verified that your hypothesis is at least plausible. You should only change the code to fix the error if the testing confirms your hypothesis.

As long as the functions are small, and their intended behaviour is well-understood, you can step through the functions with an interactive debugger, or you can do a very careful reading of the

function, to try to discover the cause of the fault. For faults discovered during integration testing, there may be too much code for you to step through one line at a time using the debugger. For these faults, you need a less time-consuming option than single-stepping line-by-line through your program.

All debuggers allow programmers to set a *break-point*, which is a way of telling the debugger to run the program normally until a chosen line of code, where the execution breaks, and you can inspect the state of your data structures and variables. When you do this inspection, you should be looking for variables whose values are not what they should be, or data structures whose organization is inappropriate, or containing the wrong data. Of course, to use these tools, you need to be quite clear about what the program should be doing; if you are not, you won't be able to tell what's gone wrong!

Another technique called "wolf-fencing" often helps locate where something has gone wrong. The idea is to use print statements throughout your code, to display evidence about what your program is doing. While you can simply blanket your program with print statements, it can also be helpful to use a kind of binary search: put a print statement in the middle of your program, to see if the problem can be detected at that point. If so, you would add another print statement near the middle of the first half of the program; if not, add a print statement near the middle of the final half. Remember, the point is to locate where something has gone wrong, so that you can study a small portion of the code close up, stepping through it with a debugger. An advanced form of this technique is called "logging;" instead of printing information to the console, you'd send the input to a separate data file called a log file. The advantage is that your console is not cluttered by data, and you don't have to write or delete tons of print statements during debugging. The log output always gets sent to the log file; but you don't have to read it all the time. The disadvantage is that log files can get pretty big, and looking through them can be tedious.

Modern languages like Python and Java have runtime errors, which can be helpful in tracking down what has gone wrong. Some older languages are less helpful, and sometimes programs will simply halt abnormally without warning and without any hint about what happened. For these languages (C, C++ to name just two), wolf-fencing, logging, and break-points are essential. Thankfully, most languages in current use have debuggers with more or less the same set of features (breakpoints, stepping, etc). No matter which language you are using, if you want to be productive and respected, learn the associated debugging tool as well.

Don't under-estimate the value of simply reading your code carefully. Software engineering calls this a *code walk-through*, and it's usually done with a group of team members. We're biased against finding errors in our own code; errors based on a mistaken (or unstated) assumption are going to be invisible. Try to explain your code to someone else, even someone who does not understand what you are doing. Use a puppet. "Experience has shown that many of the errors discovered are actually found by the programmer, rather than the other team members, during the narration. In other words, the simple act of reading aloud one's program to an audience seems to be a remarkably effective error-detection technique." [1]

---

[1]Myers, Glenford J. "Chapter 3 - Program Inspections, Walkthroughs, and Reviews". The Art of Software Testing. John Wiley & Sons. © 1979.

8 — Stacks and Queues

Learning Objectives

After studying this chapter, a student should be able to:

- Describe what a queue is, and its basic operations.
- Describe the FIFO protocol.
- Employ a queue as a solution to a software design problem.
- Describe what a stack is, and its basic operations.
- Explain the LIFO protocol.
- Describe common applications for stacks and queues.
- Use stacks and queues in simple algorithms.

In Chapter 6, we defined an ADT called a *registry*. The operations allowed elements of the registry to be set or reset in any order. That kind of "any order" access is typical of Python lists, and of arrays in many languages. The term *random access* is often used to describe the idea that any element of a collection can be accessed quickly and efficiently. It may surprise you to learn that we don't always need access to every element of a collection. It may also surprise you to learn that there are data structures that are useful but which do not allow random access.

In this chapter, we will present two kinds of collections that have limited access to their elements. The ADTs we look at here restrict the access to the data in very specific ways, and these are key to their definition. These ADTs are very common in computer science, and have a wide variety of applications. Functionally, these are very useful tools. Abstractly, they set the foundation for a useful way of designing all kinds of tools.

8.1 Queues: First-In First-Out (FIFO)

A queue is a data structure used to ensure that data is processed in the order it is received. Real-life examples of queues include the order that customers are served at stores and coffee shops, or the

Figure 8.1: Top: An *enqueue* operation adds a new data element (shaded) to the back of the queue. Bottom: A *dequeue* operation removes a data element (shaded) from the front of the queue.
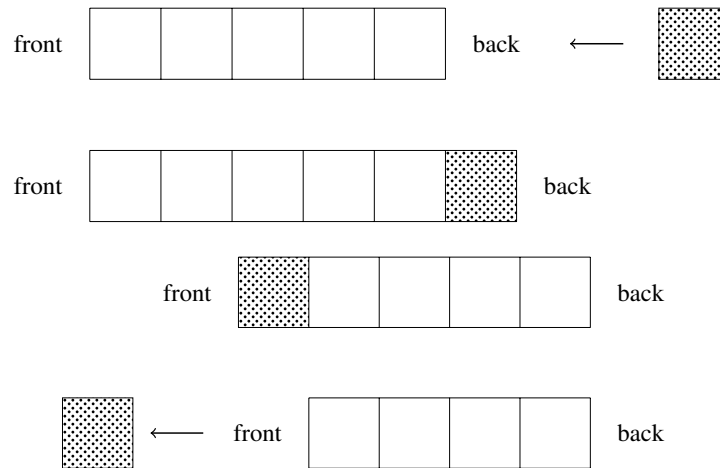
order of cars waiting at a red-light at an intersection. In a coffee shop, customers enter the queue by standing behind the person who was previously last in the queue. The customers in the queue get served in the order that they arrived.

The organization of a queue is its key aspect: items in the queue are processed in the order that they arrived. This behaviour is commonly called *first-in first-out*, and when this behaviour is encoded by rules (such as a computer program), it's called a *protocol*. A queue has a *front* and a *back*. The data at the front has been in the queue the longest amount of time, and the data at the back has been in the queue the shortest.

Definition 8.1   A *queue* (also called a *FIFO queue*) is a compound data structure in which the data values are ordered according to the first-in first-out protocol.

Data in a queue is accessed in limited ways. Every data element is added to the queue at the back. The common name for this operation is *enqueue*. A data element is removed only from the front of the queue. The common name for this operation is *dequeue*.

The primary operations provided for Queue ADTs are as follows:

- `create()` creates an empty queue.
- `is_empty(queue)` returns `True` if the given queue has no data in it.
- `size(queue)` returns the number of data values in the given queue.
- `enqueue(queue, value)` adds a given value to back of the given queue.
- `dequeue(queue)` removes the value at the front of the given queue, and returns it.
- `peek(queue)` returns the value at the front of the given queue, without removing it from the queue.

Below is a simple demonstration of the key operations mentioned. In it, a queue is created, prior to enqueueing the values 1-5. The value 1 is enqueued first, and 5 is enqueued last. All the data is dequeued one element at a time, and displayed. The data is pulled out of the queue in FIFO order:

1 is dequeued first because it was enqueued first; 5 is dequeued last because it was enqueued last. Note how the while loop uses the `is_empty()` operation to control how often values are dequeued.

```python
import Queue as Queue

data = Queue.create()

for v in range(1,6):
    Queue.enqueue(data, v)

while not Queue.is_empty(data):
    d = Queue.dequeue(data)
    print(d)
```

8.2 Queue Application: Buffering communications

Queues are extremely useful, and play a significant role in many applications. We've already briefly mentioned situations in which you might have encountered the FIFO protocol in real life: the coffee shop, for example.

Queues are used as *buffers* to smooth communications. The most obvious example is the transmission of audio or video data, from websites and services such as YouTube, NETFLIX, or SoundCloud.[1] Data transmission rates are highly variable on the internet, and computers themselves can be faster or slower. Queues can be used to moderate the effects of variable rates.

A general model for this application has a producer service sending data to a consumer. If the producer and consumer communicate directly, sending and receiving one data element at a time problems can arise. If the producer sends data faster than the consumer can make use of it, the producer has to wait for the consumer. If the consumer can make use of the data faster than the producer can send it, the consumer has to wait for the producer. If the transmission rate varies, then both the producer and consumer can be waiting at different times.

To smooth out the amount of time either side spends waiting, a buffer is used (either at the consumer side, or both sides). The producer can send data to the consumer's buffer as quickly as possible at first, and the buffer stores it temporarily. Once there is enough data in the buffer, as determined by the application, the consumer can start using the data in the buffer without waiting for it. Then the producer can send data to the buffer in bursts, or a steady rate. As long as there is data in the buffer, the consumer doesn't have to wait.

A buffer is essentially a queue. The data in the buffer is processed in the order it was sent, which is the order the consumer is expecting it. You are probably aware of buffering on video-on-demand services, such as YouTube or NETFLIX. But buffers are an essential component underlying the software that enables all internet communications, not just video. Buffers are used in modern graphics hardware, queuing up video frames to be displayed on your monitor. Even the console I/O that we do in Python is smoothed by input and output buffers. Disk drives, too.

---

[1]These names are trademarks owned by their respective owners.

Figure 8.2: Left: An *push* operation adds a new data element (shaded) to the top of the stack. Right: A *pop* operation removes a data element (shaded) from the top of the queue.
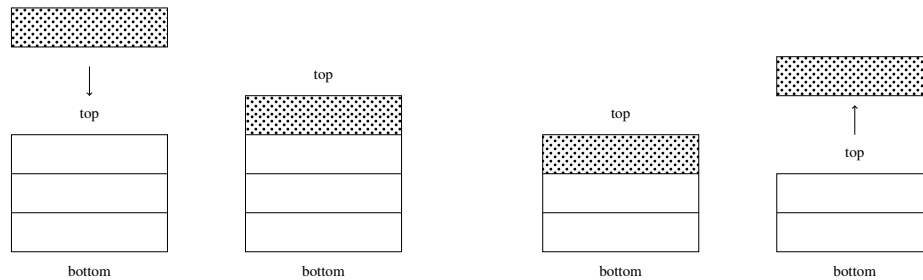
8.3   Stacks: Last-In First-Out (LIFO)

A stack is a data structure used to ensure that the most recent data is processed first. In real life, we tend to accumulate stacks of things like plates, books, exam papers. Other examples of stacks include the layering of sedimentary rocks: the top of the layer is the youngest layer, and the bottom layers is the oldest. These real life examples are over-simplified, since you could lift a stack of books to access one near the bottom, and sedimentary rocks do experience phenomena such as folding and subduction.

We use the expression *last-in first-out* to describe a stack's protocol. It implies that the most recent data is always processed first.

> **Definition 8.2**   A *stack* (also called a *pushdown stack*) is a compound data structure in which the data values are ordered according to the last-in first-out protocol.

Organizationally, a stack has a top, and a bottom. The data at the bottom of the stack has been in the stack for the longest amount of time, and the data at the top of the stack has been there the shortest. Operationally, data is added and removed only at the top of the stack. Adding to a stack is commonly called a *push*, and removing an item from the stack is called a *pop*.

The primary operations provided for Stack ADTs are as follows:

- `create()` creates an empty stack
- `is_empty(stack)` returns `True` if the given stack has no data in it
- `size(stack)` returns the number of data values in the given stack
- `push(stack, value)` adds a given value to top of the given stack
- `pop(stack)` removes the value at the top of the given stack, and returns it
- `peek(stack)` returns the value at the top of the given stack, without removing it from the stack.

Below is a simple demonstration of the key operations mentioned. In it, a stack is created, prior to pushing the values 1-5. The value 1 is pushed first, and 5 is pushed last. All the data is popped one element at a time, and displayed. The data is pulled out of the queue in LIFO order: 5 is dequeued first because it was pushed most recently; 1 is popped last because it was pushed first. Note how the while loop uses the `is_empty()` operation to control how often values are popped.

```
import Stack as Stack

data = Stack.create()

for v in range(1,6):
    Stack.push(data, v)

while not Stack.is_empty(data):
    d = Stack.pop(data)
    print(d)
```

8.4 Stack Application: Backtracking

You've almost certainly encountered stacks of plates and books and other physical items that can be literally stacked.

You have also encountered stacks in your web-browser. When you visit a page, the URL for that page gets pushed on a stack, which we will call the *back-stack*. The *back* button pops the back-stack to re-display the page you the page you were most recently looking at. Most web-browsers also have a *forward* button, which also uses a stack, which we could call the *forward-stack*. When you go back, your current page's URL is pushed onto the forward-stack, and one URL is popped from the back-stack and displayed to you. Likewise, when you use the forward button, the URL of the current page is pushed onto the back-stack, and a URL is popped from the forward-stack and displayed. That's how you can go back and forth through your recent browser history without losing anything.

Stacks are also useful in applications like word processors and other kinds of content creation applications. These applications typically have an *undo* button, and a *re-do* button. These are very similar to the browser buttons back and forward. Every time you modify a document, data describing the modification[2] is pushed on an *undo-stack*. When you use the undo button, the undo-stack is popped, and the application uses the information to undo the action. The modification data can be pushed on a second stack, which is controlled by the re-do action button. The only tricky thing with undo and re-do is encoding the modifications in a way that allows the application to store the changes on the stack, so that they can be performed.

This concept is so common that it has a name: back-tracking, i.e., returning to a previous situation so that you can try something else. We've seen how stacks enable a person to back-track to a previous state in the above examples. The concept of back-tracking is also applicable to the design of algorithms, as we will see in Chapter 25. In a nut-shell, if a program uses back-tracking, it has been programmed to explore a *space* of possible solutions, looking for the right solution that might be in the space. If the algorithm is faced with a number of options, and there's no known way to tell which option is best, back-tracking allows the algorithm to try them all, without getting lost. Back-tracking gets used as an algorithm strategy sometimes when no better algorithm has been discovered, kind of a last resort, or as we will suggest, a first stab at solving tricky problems. A stack is a crucial part of all back-tracking algorithms.

We will see other applications of queues and stacks in Chapter 9.

---

[2] It might be hard to imagine what form this data might take, but whatever it is, it goes on a stack!

9 — Applications of Stacks and Queues

Learning Objectives

After studying this chapter, a student should be able to:

- Employ a stack as a solution to a software design problem.
- Write programs that use stacks and queues.
- Explain the key features of a simple queueing simulation.
- Explain how the queueing simulation makes use of the Queue ADT.
- Explain the key features of bracket checking, using a stack.
- Explain how bracket checking makes use of the Stack ADT.
- Explain the key features of evaluating post-fix expressions, using a stack.
- Explain how evaluating post-fix expressions makes use of the Stack ADT.

9.1 Introduction

In this chapter we demonstrate the use of the Queue and Stack ADTs in algorithms that highlight the importance of their key aspects. The algorithms are demonstrative of techniques in computer science as well.

9.2 Queueing simulation

A recurring theme in computer science is the use of software to answer questions that would be difficult or impossible to answer using analytical techniques. In the case of the simulation in this chapter, analytical techniques are able to demonstrate several of the properties we describe, but studying the algorithm is useful for the technique it demonstrates.

We will be simulating the situation in which customers arrive (say at a coffee shop) and wait in a queue for service. One of the main assumptions in our simulation is that there will be one queue in which all the customers will wait. This keeps things simple, as we won't have to simulate the

decision process of customers faced with multiple queues. The customers arrive, wait in the queue for some amount of time, get served, and then leave the queue. We're interested in knowing how long the customers wait in the queue on average, and how long the queue gets. To answer these questions, we will demonstrate a program that literally simulates arrivals, waiting, service, and finally departure of customers.

It should be pointed out that this kind of simulation has a long history in computer science. Queueing theory, as it is called, starts with the simple model we show here, and extends to models that are much more sophisticated. Understanding the mechanics of queueing is vital to their use in the real world (as in the coffee shop example, but extending to supply chain management for manufacturing, and other applications), as well as the software services in the modern world. Your computer is full of applications of queues that affect your daily experience, from the queues that are used to buffer the multi-tasking operating system on your computer, to the queues that buffer streaming data from games and multi-media services like YouTube and NETFLIX.

Two of the key aspects of the simulation are the customer *arrival* times, and the customer *service* times. In simulations of this kind, it is standard to assume that the customers arrive randomly, but with an assumed arrival rate. While we have used random numbers generated by Python's `random` module previously, we have always done so by assuming a number is drawn from a given range (e.g., $[0, 1]$) with equal probability. We could use the same assumption here, but it's standard to assume that arrival times and service times are not equally likely.

The standard assumption for this simple model is that arrival times are governed by something called a *Poisson process*. The Poisson process puts no upper limit on the time between arrivals; a very long period between arrivals is possible, but very unlikely. The Poisson process describing the arrival times is governed by the *arrival rate*, in arrivals per unit time. We won't be doing a real-time simulation, so the time unit can be arbitrary; it's intuitive in coffee shop simulations to imagine arrivals per minute, but in a NETFLIX simulation, data arrivals might be measured in data packet arrivals per millisecond. When we think in terms of arrivals per unit time, we are using an abstraction.

In the Poisson process describing arrivals, the time between arrivals is random, but described by an *exponential distribution*, in which the time between arrivals is more likely to be shorter than longer. The standard model also assumes that the service time is described by an exponential distribution, governed by an assumed *service rate*. As with arrival times, services times are random, and more likely to be shorter than longer.

In Python, we can ask for a random number from an exponential distribution using the `random` module, using the function `expovariate`, as in the following function:

```python
def sample_time(x):
    '''
    Return a random sample time until an event.
    In the long run, the events will have a rate of x
    events per unit time.

    Preconditions:
        x: the desired arrival rate, per unit time
    Post-conditions:
        none
    Return:
```

```
        a random  sample  time  that  obeys  the  rate  x
    '''
    return  rand.expovariate(x)
```

We call this function with either the service rate or the arrival rate, and we get a random number generated for us that obeys the assumptions about arrivals and service described above. Observe that we're using procedural abstraction here to hide the details about how the samples are obtained; it's a lot of work for a one-line function body, but if we ever decide to change the assumptions about how sample times are generated, we can make those changes in one function, and the rest of the script won't change. That's adaptability!

Let's turn to the mechanics of the queueing model. There will be a single queue, and customers will arrive, enter the queue, and wait for service. When the customer reaches the front of the queue, they are served, which takes some time, before they leave the queue. In our simulation, we will use a FIFO Queue to store arrival times.

```
# customers  wait  in  a  service  queue
service_queue = Queue.create()
```

When a customer leaves the queue, we can check the current time, and calculate the amount of time the customer spent in the queue by subtraction. We will use the Statistics ADT introduced in Chapter 6 to manage the calculation of the average wait time.

```
# we're interested  in  the  average  time  in  the  queue
waiting = Statistics.create()
```

Time is a key aspect of our simulation, and we normally think of time as progressing in uniform steps, like a clock ticking in seconds or milliseconds. But in this kind of simulation, which is called an *event-based simulation*, time advances in leaps to whichever event is determined to be next. To work through a simulation with arrivals and service, we keep track of two times: the time of the next arrival event, and the time of the next service event.

- The next arrival is an event that places a new arrival in the queue.
- The next service is an event that removes a customer out of the queue; it indicates the time that the service for the next customer finishes.

To start the simulation, the next service event is scheduled to occur after the first arrival.

```
# keep  track  of  when  a  new  customer  will  arrive
nextArrival = sample_time(arrival_rate)

# keep  track  of  when  a  service  event  is  complete
nextService = nextArrival + sample_time(service_rate)
```

Our simulation will run for an amount of simulated time, specified by the variable `sim_length`. The simulation will repeatedly advance to whichever event is next. Sometimes the next event is an arrival, so we'll handle the arrival by putting the arrival in the queue.

However, when the service event is next, we'll calculate how long the customer at the front of the queue was waiting, and then schedule a new arrival event. If the queue happens to be empty, then the next service event happens after the arrival of the next customer. On the other hand, if the queue is not empty, the next service event occurs some time after the most recent service event.

```
# simulate the arrival-service process
while nextService < sim_length:

    # handle all arrivals that occur before the service is complete
    while nextArrival < nextService:
        Queue.enqueue(service_queue, nextArrival)
        nextArrival = nextArrival + sample_time(arrival_rate)

    # service time is over; serve the first customer in the queue
    this_arrival = Queue.dequeue(service_queue)

    # how long was the customer waiting?
    waited = nextService - this_arrival
    Statistics.add(waiting, waited)

    # determine when service to the next customer will end
    if Queue.is_empty(service_queue):
        nextService = nextArrival + sample_time(service_rate)
    else:
        nextService = nextService + sample_time(service_rate)

print('Average wait time:', Statistics.mean(waiting))
```

Notice the use of meaningful variables names, and the use of comments. The script could be written without the use of the Queue and Statistics ADTs, but their use makes the code clearer and more concise. It takes effort to produce code like this, even after you know how the algorithm works. The full listing of the queueing simulation can be found in Appendix B.

9.2.1 Comments

The queueing simulation presented here is also known as an M/M/1 queue, which is studied in applied probability courses, as one of the elementary models for the use of probability theory. Since this is not a probability course, we'll say only a few words about what such studies tell us.

First of all, our intuitions tell us that if the service rate is slower than the arrival rate, the queue will get very big. Formal analysis confirms this. If the arrival rate is slower than the service time, then number of customers waiting in the queue depends on the ratio between rates. To clarify this, formal analysis typically uses λ to represent the arrival rate, and μ to represent the service rate. If $\mu \leq \lambda$, the queue can grow arbitrarily large, because customers will arrive faster than they are served. But if $\mu \geq \lambda$, there may be a few bursts of high traffic, but the service can keep the queue moving: in this case, the average number of customers in the queue is $\lambda/(\mu - \lambda)$. For example, if the arrival rate, $\lambda = 1$, and the service rate is $\mu = 2$, the queue will have on average 1 customer in it. Formal analysis also tells us that the average waiting time for customers is $1/(\mu - \lambda)$. So if the arrival rate, $\lambda = 1$, and the service rate is $\mu = 2$, the average waiting time is also 1 time unit. We can set our simulation to verify these formal results empirically.

9.3   Bracket checking

At some point in your study of programming you may have wondered how Python can convert the text you type in an editor and turn it into a thing that the computer does. If you haven't wondered that, perhaps you can take a few seconds now. We'll wait for you.

The process of converting Python code (or any programming language for that matter) into machine code is a complicated process, but we can gain an appreciation for that process by looking at an algorithm for bracket checking. When we write Python expressions that use arithmetic, the expression needs to obey certain rules about the syntax of the expressions. The process of checking syntax is called *syntactic analysis*, or sometimes *parsing*. If the syntax is not correct, Python cannot make sense of the expression, and reports a syntax error.

Bracket checking can be considered are very limited form of syntactic analysis. It demonstrates the use of a stack very effectively; stacks are key to solving the larger problem of more general syntactic analysis. We want out program to recognize the kinds of bracketing we'd expect to see in our Python expressions. For example, we might write an assignment statement using an expression as follows:

```
result = ((3 + 4) * (5 + 6))
```

This example should be accepted as syntactically correct (even though the outer-most brackets are not actually necessary, they are still acceptable). In contrast, the following example should produce a syntax error:

```
result = ((3 + 4 * (5 + 6))
```

because there are more left-brackets than right-brackets.

To simplify the presentation, we will eliminate the arithmetic, and just keep the brackets. This is enough to demonstrate the use of stacks in an understandable, if a bit simplified, application. Without arithmetic, our expressions look a bit weird, and they'll be strings:

```
example1 = '(()())'
example2 = '((())'
example3 = '(())'
```

We want a program to analyze any string consisting simply of brackets, and indicate if every left-bracket has a matching right-bracket. The key to this task is to realize that a right-bracket matches the most recently seen left-bracket. The phrase "most recently seen" is a strong indication that we need to use a stack. And for reasons that we will discuss later, we will also use a queue. We will use a boolean variable called `unmatched_close` to indicate whether we've found a right-bracket without a matching left-bracket, as in the third example above.

```
# create the initial empty containers
chars    = MyQueue.create()
brackets = Stack.create()
unmatched_close = False
```

The first thing we do is put all the brackets into a queue. This simplifies the matter of stepping through the string, because we don't need to keep track of the length of the string, or where we are in the string.

```
# put all the characters in the Queue
for c in example:
    MyQueue.enqueue(chars, c)
```

The bracket checking process works by examining the queue of characters, one at a time. If the current character is a left-bracket, it is pushed onto the stack. If the current character is a right-bracket, it either has a matching left-bracket on the top of the stack, or, if the stack is empty, no matching left-bracket. The program below also ignores every character that is not a bracket; this allows us to process examples with only brackets, or examples of arithmetic expressions.

```
# brackets match if and only if every '(' has
# a corresponding ')'
while not MyQueue.is_empty(chars) and not unmatched_close:
    c = MyQueue.dequeue(chars)
    if c == '(':
        Stack.push(brackets,c)
    elif c == ')' and not Stack.is_empty(brackets):
        Stack.pop(brackets)
    elif c == ')' and Stack.is_empty(brackets):
        unmatched_close = True
    else:
        pass
```

There are two reasons we could have left the while loop above. First, we may have detected an unmatched right-bracket, in which case we can report that fact. Otherwise, the reason we exited the loop was because all the characters in the queue were processed. In that case, the stack could be empty, which means every pair of brackets matched perfectly. However, if the stack is not empty, it means there were more left-brackets than right-brackets, and we should report it.

```
# check how the analysis turned out
if unmatched_close:
    print("Found a ')' with no matching '('")
elif not Stack.is_empty(brackets):
    print("At least one '(' without a matching ')'")
else:
    print('Brackets matched')
```

While simple, this process of stacking up characters is essential to the analysis of text, whether for applications to convert Python programs to machine code, or for processing natural languages (like English) to extract meaning. The programs for those are more complicated, but the bracket-checking program is demonstrative of the principles involved. Text, whether program or prose, is a sequential representation, and keeping parts of it in a stack allows a program to draw local associations, like matching brackets, or noun-verb agreement.

The full listing of the bracket matching can be found in Appendix B.

9.4 Post-fix Arithmetic evaluator

Now that we've demonstrated how to use a stack to check for a limited form of syntactic analysis, we will give a demonstration of how to take syntactically correct expression and come up with the correct value. This is called *evaluation*, and a program that does this is called an *evaluator*.

In order to keep the evaluator simple, and to focus on the use of stacks, we have to restrict the kinds of expressions we will evaluate. Normally we write arithmetic expressions by putting the operators *between* the operands, as in $3+4$. When we have more complicated expressions with several operators in it, we typically use a set of rules called "order of operations" to describe how an expression should be evaluated. For example, the order of operations tells us that multiplication should occur before addition in expressions such as $3+4 \times 5$. We can dispense with the order of operations if we insist on using brackets for every operator, as in the example $((3+4) \times 5)$.

There is another way that we can dispense with memorizing order of operation rules, which requires writing expressions differently. Instead of writing the operator between the operands, we will write the operator *after* both operands. For example, instead of writing $(3+4)$, we would write $(3 \ 4 \ +)$. As another example, instead of writing $((3+4) \times 5)$, we would write $((3 \ 4 \ +) \ 5 \ \times)$. As a final example, instead of writing $((3+4) \times (5+6))$, we would write $((3 \ 4 \ +) \ (5 \ 6 \ +) \times)$. If you take a moment to reflect (we'll wait), you may realize that we actually don't need brackets if we put the operator after its arguments. For example, there is no ambiguity in the expression: $3 \ 4 \ + \ 5 \ 6 \ + \ \times$. This arrangement of operators and operands is called *post-fix* or *reverse Polish* notation.

To evaluate the expression, we scan it left-to-right, and when we find an operator, we perform the operation on the two immediately preceeding operands. The phrase "immediately preceeding" should be a strong indicator of the use of a stack. The key to correct evaluation is that we replace the expression with its value, and continue moving to the right. Here is an example of the process:

$$\underline{3 \ 4 \ +} \ 5 \ 6 \ + \ \times$$
$$7 \ \underline{5 \ 6 \ +} \ \times$$
$$\underline{7 \ 11 \ \times}$$
$$77$$

In the example, $3 \ 4 \ +$ is replaced by 7 (as indicated by underlining), and then $5 \ 6 \ +$ by 11. When we reach the $\times$, there are two values waiting to be used, and we get the final result, 77.

It may be a little confusing to see the operator last, but once you get used to the idea, it's no more trouble than the traditional organization of operations. In the early days of computer technology, before smartphones and highly portable computers, specialized devices called electronic calculators were used to assist in calculations done on paper. The earliest calculators did not have brackets to help users enter their calculations.[1] The first calculators with a stack were used by scientists and engineers, who mentally translated "normal" expressions into post-fix, so they could use their advanced calculators, much to the annoyance of the scientists and engineers who insisted that slide-rules are the only calculating device a so-called "real" scientist should use.

To evaluate post-fix expressions, we start with a string that contains the expression. We assume that at least once space separates numbers and operators from each other. For example:

---

[1]This is because the earliest calculators did not have a stack. The effects of this deficiency can still be observed in the default Calculator app in Microsoft Windows. Go ahead, open it up and enter $3+4 \times 5$ in exactly that order, and be appalled.

```
example = "f 56 + "
```

We will implement the evaluator as a function, with the following interface:

```python
def evaluate(expr):
    """
    Evaluate a postfix expression.
    Pre-conditions:
        expr: a string containing a postfix expression
    Post-Conditions:
        none
    Return:
        the value of the expression
    """
```

As in the bracket checking program, we will use a queue to store the expression, and a stack to assist in the evaluation.

```python
    # create the initial empty data structures
    expression = Queue.create()
    evaluation = Stack.create()
```

The first order of business is to split the string at the spaces, and put the pieces into the queue. Again, using a queue allows us to process the pieces without having to keep track of where we are in the string, or in a list of pieces.

```python
    expr_list = expr.split()
    # put all the items in the Queue
    for c in expr_list:
        Queue.enqueue(expression, c)
```

The function walks through the expression left-to-right, using the expression queue. We dequeue a piece of the string, and look at it.

- If the piece looks like an operator, we pop the evaluation stack twice, perform the operation, and put the result back on the stack.
- If the piece doesn't look like any operator, convert it to a number, and push it onto the stack.

```python
    while not Queue.is_empty(expression):
        c = Queue.dequeue(expression)

        if c == '*':
            v1 = Stack.pop(evaluation)
            v2 = Stack.pop(evaluation)
            Stack.push(evaluation, v1*v2)
        elif c == '/':
            v1 = Stack.pop(evaluation)
            v2 = Stack.pop(evaluation)
```

```
                Stack.push(evaluation, v2/v1)
        elif c == '+':
                v1 = Stack.pop(evaluation)
                v2 = Stack.pop(evaluation)
                Stack.push(evaluation, v1+v2)
        elif c == '-':
                v1 = Stack.pop(evaluation)
                v2 = Stack.pop(evaluation)
                Stack.push(evaluation, v2-v1)
        else:
                Stack.push(evaluation, float(c))

    return Stack.pop(evaluation)
```

The full listing of the post-fix evaluator can be found in Appendix B.

The above code correctly evaluates proper post-fix expressions, and makes appropriate use of computational resources. It makes use of procedural abstraction and ADTs, so it takes advantage of reusable components, and itself is fairly reusable. But it's not very good code, because it is very brittle (i.e., not robust). It assumes that the input string is a perfectly correct post-fix expression. The function will cause a run-time error if the string is anything but perfect. The code is also very repetitive, with very small differences between the four cases that handle the different operators. It's good enough to demonstrate the use of stacks and queues, and to teach the concept of post-fix arithmetic, but we can do better. And we will, in Chapter 12!

10 — List-based Stacks and Queues

Learning Objectives

After studying this chapter, a student should be able to:

- Employ data abstraction to implement a queue.
- Employ data abstraction to implement a stack.
- Explain the concept of an adapter in the context of abstract data types.
- Describe the list-based implementation of stacks and queues.
- Explain some of the reasons that wrapping an ADT around a list can improve correctness and adaptability.

10.1   Why use Queues and Stacks when we have lists?

Python lists are very versatile, and have a lot of uses. They have all the operations we identified as essential for FIFO Queues and LIFO Stacks in the previous chapter, and lots more besides. While such a useful tool is good to have, it is also useful to apply a certain amount of discipline to complex programming tasks, to improve correctness, and increase adaptability and robustness.

In this chapter, we will see how we can implement the Queue and Stack ADTs using Python lists. Essentially, we will be converting a Queue (or Stack) operation into a list method. This is such a common technique in software development that it has its own name. When we take a data structure or ADT *A* and provide operations so that it looks and behaves like ADT *B*, we have built an *adapter*. It is useful because sometimes a programming task requires a data structure that is not available directly in a language or a module. One way to provide the tool is to adapt something similar to provide the required functionality. This chapter gives the very simplest example of that kind of adaptation. The approach generalizes well beyond what we will demonstrate in this chapter.

Another reason to prefer using an ADT adaptation over Python's versatile lists is the way that it enhances correctness. If your program needs LIFO behaviour, for example, using a Queue documents

this fact in the program, and also enforces the LIFO protocol. Should you need to fix, extend or reuse the program, the fact that you used a Queue will prevent you from forgetting a crucial aspect of the algorithm. If you did not use a Queue, but used a Python list directly, it would be much easier to subvert the purpose of the list, and violate the FIFO protocol. That kind of subversion may allow you to make a quick bug fix in the short term, but might introduce a very subtle bug that would be difficult to find in the long term.

The last reason for this chapter is to set the scene for an important lesson in the flexibility of ADTs. An ADT is defined by the interface of its operations, not by the implementation of the operations. In this chapter, we will implement the Queue and Stack ADTs using Python lists. In a later chapter, we will implement the same ADTs using different data structures. The lesson will be that once an ADT's interface is defined, several implementations may be possible. Any application that uses the ADT's interface can use any one of the implementations. So you can prototype your application using a very simple ADT implementation, and then improve it by replacing the simple ADT with a more efficient one. This is the very essence of adaptability and robustness.

10.2 Adapting Python Lists to Implement Queues

The essential nature of a Queue is the FIFO protocol. Python provides two list methods that we will use to implement the Queue ADT.

append() This Python list method takes an item and adds it to the end of the current list. We use this method to implement the enqueue operation.
pop() This Python list method takes an index and removes the item at that index from the list. If we call pop(0) the first element in the list is removed and returned. We use this to implement the dequeue operation.

The list-based implementation simply wraps an ADT around the Python list. The object we are calling a queue is nothing but a list. Anyone who understood this fact could use any of Python list methods on the queue we have implemented. The front of our queue will be at list index 0, and the back of the queue will be at list index $n-1$, where n is the length of the list. The following sections describe the implementation in a little more detail. The complete implementation can be found in Appendix C.

10.2.1 Operation: create()

The create() function returns a new empty queue. In this implementation, a queue is nothing but a list. Python list() function creates a new, empty list, and we are simply using that function.

```python
def create():
    """
    Purpose
        creates an empty queue
    Return
        an empty queue
    """
    return list()
```

10.2.2 Operations: size() and is_empty()

The is_empty() operation returns True if the queue is empty. The size() operation returns the number of elements stored in the queue. Both implementations use Python's len() function directly, because in this implementation, a queue is nothing but a list.

```python
def is_empty(queue):
    """
    Purpose
        checks if the given queue has no data in it
    Pre-conditions:
        queue is a queue created by create()
    Return:
        True if the queue has no data, or false otherwise
    """
    return len(queue) == 0


def size(queue):
    """
    Purpose
        returns the number of data values in the given queue
    Pre-conditions:
        queue: a queue created by create()
    Return:
        The number of data values in the queue
    """
    return len(queue)
```

10.2.3 Operation: enqueue()

The enqueue() operation adds a new data element to the back of the queue. To implement this operation, we used Python's list method append(), which adds a value to the end of the list.

```python
def enqueue(queue, value):
    """
    Purpose
        adds the given data value to the given queue
    Pre-conditions:
        queue: a queue created by create()
        value: data to be added
    Post-condition:
        the value is added to the queue
    Return:
        (none)
    """
    queue.append(value)
```

10.2.4 Operation: dequeue()

The dequeue() operation removes a data element from the front of the queue. To implement this operation, we used Python's list method pop(), which removes and returns a value from the list. Calling pop(0) removes and returns the first value in the list.

```
 1  def dequeue ( queue ):
 2      """
 3      Purpose
 4          removes and returns a data value from the given queue
 5      Pre - conditions :
 6          queue : a queue created by create ()
 7      Post - condition :
 8          the first value is removed from the queue
 9      Return :
10          the first value in the queue
11      """
12      return queue . pop (0)
```

10.2.5 Operation: peek()

The peek() operation returns a reference to the data element at the front of the queue. To implement this operation, we use Python's list indexing. This method returns the value, without removing it.

```
 1  def peek ( queue ):
 2      """
 3      Purpose
 4          returns the value from the front of given queue
 5          without removing it
 6      Pre - conditions :
 7          queue : a queue created by create ()
 8      Post - condition :
 9          None
10      Return :
11          the value at the front of the queue
12      """
13      return queue [0]
```

10.3 Adapting Python Lists to Implement Stacks

The essential nature of a Stack is the LIFO protocol. Python provides two list methods that we will use to implement the Queue ADT.

append() This Python list method takes an item and adds it to the end of the current list. We use this method to implement the push operation.

pop() This Python list method takes an index and removes the item at that index from the list. If we call pop() *without any argument*, the *last* element in the list is removed and returned. We use this to implement the pop operation.

As with the queue implementation, our stack implementation simply wraps an ADT around the Python list. The object we are calling a queue is still just a list. Anyone who understood this fact could use any of Python list methods on the queue we have implemented. The bottom of our stack will be at list index 0, and the top of the stack will be at list index $n - 1$, where n is the length of the list. We could have implemented the stack so that the top of the stack is at index 0. The implementation would be slightly different, but from outside the ADT, no one would be able to tell which end we are using. The following sections describe the implementation in a little more detail. The complete implementation can be found in Appendix C.

10.3.1 Operation: create()

The `create()` function returns a new empty queue. In this implementation, a queue is nothing but a list. Python `list()` function creates a new, empty list, and we are simply using that function.

```python
def create():
    """
    Purpose
        creates an empty stack
    Return
        an empty stack
    """
    return list()
```

10.3.2 Operations: size() and is_empty()

The `is_empty()` operation returns `True` if the queue is empty. The `size()` operation returns the number of elements stored in the queue. Both implementations use Python's `len()` function directly, because in this implementation, a queue is nothing but a list. This implementation is identical to the queue operations of the same name. Using the same name is actually a helpful idea. It's one fewer detail to memorize, and it hints at the idea that there is a core similarity between queues and stacks.

```python
def is_empty(stack):
    """
    Purpose
        checks if the given stack has no data in it
    Pre-conditions:
        stack is a stack created by create()
    Return:
        True if the stack has no data, or false otherwise
    """
    return len(stack) == 0


def size(stack):
    """
    Purpose
        returns the number of data values in the given stack
    Pre-conditions:
```

```
18          stack: a stack created by create()
19      Return:
20          The number of data values in the queue
21      """
22      return len(stack)
```

10.3.3 Operation: push()

The push() operation adds a new data element to the top of the queue. To implement this operation, we used Python's list method append(), which adds a value to the end of the list.

```
1  def push(stack, value):
2      """
3      Purpose
4          adds the given data value to the given stack
5      Pre-conditions:
6          queue: a stack created by create()
7          value: data to be added
8      Post-condition:
9          the value is added to the stack
10     Return:
11         (none)
12     """
13     stack.append(value)
```

10.3.4 Operation: pop()

The pop() operation removes a data element from the top of the stack. To implement this operation, we used Python's list method pop(), which removes and returns a value from the list. Calling pop() without an argument removes and returns the last value in the list.

```
1  def pop(stack):
2      """
3      Purpose
4          removes and returns a data value from the given stack
5      Pre-conditions:
6          stack: a stack created by create()
7      Post-condition:
8          the top value is removed from the stack
9      Return:
10         returns the value at the top of the stack
11     """
12     return stack.pop()
```

10.3.5 Operation: peek()

The peek() operation returns a reference to the data element at the top of the stack. To implement this operation, we use Python's list indexing. This method returns the value, without removing it.

```
1  def peek(stack):
2      """
3      Purpose
4          returns the value from the front of given stack
5          without removing it
6      Pre-conditions:
7          stack: a stack created by create()
8      Post-condition:
9          None
10     Return:
11         the value at the front of the stack
12     """
13     return stack[-1]
```

10.3.6 Summary

The similarity between the queue and stack implementations is very strong. The names given to the key operations differ, but both ADTs have an operation to add to the collection, and an operation to remove from the collection. The list-based implementation simply provides a discipline on the use of Python's lists in an effort to prevent programming errors. For example, using enqueue means you don't have to memorize which end of the list you're using for the back of the queue. The operation takes care of that detail. And if there is a good reason to change which end of the list is used as the back and which is used as the front, that change can be made in the ADT implementation, once and for all.

Keep in mind that the computer does not define up or down, or front or back the way we do in the real world. Gravity, for example, gives us a preferred orientation for real world stacks. But without gravity, the top of a stack is defined only by the push() and pop() operations. Likewise, for a queue, its front (and back) is defined by dequeue() (and enqueue()).

We should understand this technique of adapting a list as having very strong advantages for large software projects and long term investments in software development. For one-off scripts, and scripts that will get re-written almost immediately, going to the trouble of using an adapter like this is probably not worth the effort. Also, if your short-lived script has a bug that you cannot find, adapting that list you are using as a stack or queue might be a way to ensure that the bug is not in your use of lists.

Finally, the general strategy of taking a data structure that you already have, and making it look like something you need, is very powerful for rapid prototyping. For example, imagine if Python did not have dictionaries. We could adapt a Python list to behave like a dictionary by creating an adapter ADT, but it would probably not be very efficient. Once the ADT operations have been designed and implemented, we can replace the list-based implementation with something better, and every piece of code that used the ADT would run more efficiently, without having to change any of the application code.

11 — The Node ADT

> ## Learning Objectives
>
> After studying this chapter, a student should be able to:
>
> - To describe the concept and structure of a node.
> - To explain the operations of the Node ADT.
> - To employ Node ADT operations in Python programs.

11.1   Nodes

In computer science, the term *node* is used when talking about the way data is connected. In this chapter, we'll use the word node as the name of a data structure that allows us to build connected chains containing data. A node is a simple record, implemented as a dictionary, with two fields:

data A data value
next A node (or the value `None`)

As we have seen, data values are not stored in lists or dictionaries. Instead, references to data values and other nodes. For example, we could define the variable `anode` as follows:

```
anode = {'data': 1, 'next': None}
```

The node stores the value 1, and does not refer to any other node.

Because it is a Python dictionary, it's possible to store anything at all in the data field or next field of a node. For that reason, we will introduce an ADT for nodes, so that we can enhance correctness. Hopefully, the interface will prevent a few errors!

The primary operations provided for the node ADTs are as follows:

- `create(data, next)` creates a node to contain the data value and the next value.
- `get_data(node)` returns the contents of the data field

- get_next(node) returns the contents of the next field
- set_data(node, v) sets the contents of the data field to v
- set_next(node, n) sets the contents of the next field to n

Our Python implementation is very simple. For create, we use the *keyword parameter* technique, to provide a default value for the next field, which can be over-ridden by calling create with a keyword argument:

```python
def create(data, next=None):
    """
    Create a new node for the given data.
    Pre-conditions:
        data:  Any data value to be stored in the node
        next:  Another node (or None, by default)
    Post-condition:
        none
    Return:
        the node created
    """
    return {'data':data, 'next':next}
```

The two operations get_data(node) and get_next(node) simply look up the appropriate field in the dictionary.

```python
def get_data(node):
    """
    Retrieve the contents of the data field.
    Pre-conditions:
        node: a node created by create()
    Post-conditions:
        none
    Return
        the data value stored previously in the node
    """
    return node['data']


def get_next(node):
    """
    Retrieve the contents of the next field.
    Pre-conditions:
        node: a node created by create()
    Post-conditions:
        none
    Return
        the value stored previously in the next field
    """
    return node['next']
```

The operation `set_data(node, val)` stores the given value in the node in the data field. Keep in mind that the reference to the value is being stored, not the actual value itself.

```python
def set_data(node, val):
    """
    Set the contents of the data field to val.
    Pre-conditions:
        node: a node created by create()
        val:  a data value to be stored
    Post-conditions:
        stores the new data value, replacing the existing value
    Return
        none
    """
    node['data'] = val
```

The operation `set_next(node, val)` stores the given value in the node in the next field. Keep in mind that the reference to the value is being stored, not the actual value itself. This is especially important for `set_next()`, since its purpose is to store another node (as a reference). As we know, dictionaries only store references, and any value at all could be stored in the next field. We have to practice strong discipline to make sure that the value stored in the next field is either `None` or another node.

```python
def set_next(node, val):
    """
    Set the contents of the next field to val.
    Pre-conditions:
        node: a node created by create()
        val:  a node, or the value None
    Post-conditions:
        stores the new next value, replacing the existing value
    Return
        none
    """
    node['next'] = val
```

11.2 Node chains

With the node ADT, we can create a primitive kind of list-like compound data type, which we will call a node-chain. In many programming languages, node-chains are the main compound data type. The first such language, named LISP (for "LISt Processing"), made this kind of node chain central to its operation, so much so that LISP programs themselves look like lists. We use the term node-chains, because it is descriptive and accurate. However, some text books use the term *linked-list*, which would be fine, except that we want to use that term for a higher level, more abstract concept. Because of their ubiquity, it is instructive to see how node-chains work, as we will need the technique in Chapter 17.

Figure 11.1: A node-chain diagram, corresponding to a sequence of node creations.
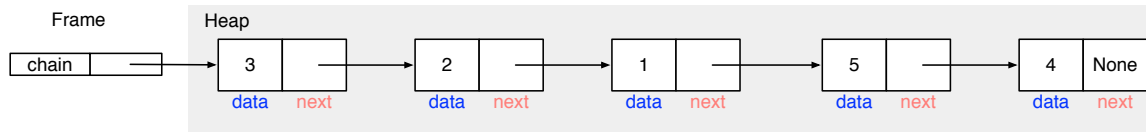
> **Definition 11.1**   A node-chain is defined as follows:
>
> 1. A structure with exactly zero nodes is an *empty node-chain*.
> 2. A structure with exactly one node is a *node-chain* **only if** its `next` field refers to an empty node-chain.
> 3. If *c* is a non-empty node-chain, then a node, *n*, whose `next` field refers to *c* is a *node-chain*.

A node-chain is nothing more than a sequence of nodes, one containing a reference to the next. A node-chain has to have an anchor, or a hook, to attach the first node in the chain. Node chains have an implied directionality, namely in the direction suggested by the `next` field. Each node in the node-chain refers to another node, except the last, which has to refer to an empty node-chain.

We will use the Python value `None` to represent an empty node-chain. Note that this value is not actually a node, as defined above. Trying to use any of the node ADT operations on it will result in a run-time error, so we have to be careful to check whether a given node-chain is empty or not, before we do anything to it. A consequence of the use ofNone to represent an empty chain is that the last node in any node-chain has to have the value `None` stored in the `next` field; this is how we know that there are no more nodes further down the chain.

One way to build a node chain is with a sequence of calls to `create()`:

```
1  # create a chain of nodes with data 3, 2, 1, 5, 4
2
3  chain = node.create(4)
4  chain = node.create(5, chain)
5  chain = node.create(1, chain)
6  chain = node.create(2, chain)
7  chain = node.create(3, chain)
```

The first node created (line 3) stores the data value 4; in this function call, we've only given one argument, and the default value `None`, specified in the definition of the function, is used. At that point in the code, the variable `chain` refers to a `chain` of exactly one node. On line 4, a new node is created, with data value 5, and the current value of `chain` in the next field. It is important to understand that on the left-hand-side of the assignment, the value of the variable `chain` is still a single node, containing the data value 3. Creating the new node on line 3 copies the reference stored in the variable `chain` to the data field in the new node. Once the new node is created, the reference to the new node is stored in the variable `chain`. At this point in the code, the variable `chain` refers to a node containing 5, and there is a reference from this node to another node containing 3. In other words, `chain` refers specifically to the first node in the chain of 2 nodes, but indirectly, through stored references, chain also refers to the whole chain.

A picture of the resulting chain is found in Figure 11.1. Hopefully, you can now appreciate

all the work we had to do to explain references. The end of the code sequence above results in a chain of 5 nodes, with 5 different data values. Another thing to keep in mind is that the diagram in Figure 11.1 is an abstraction of the truth: the variable anchor is in Python's global scope, which we have not bothered to show, and the node values are on the heap. The biggest simplification in the diagram is that the integers are drawn inside the boxes; the true picture would have each number as a value on the heap, and arrows from the `data` fields to the corresponding numbers.

The sequence created the node for the value 4 first, and kept putting the new nodes *in front* of the nodes previously created. You should recognize this as being similar to how a stack operates. Be aware that this is our first example, and that it is possible to put new nodes anywhere in the chain, with a little care.

Writing code for node-chains is instructive. As a first example, here's Python code that will display the even data values in the chain:

```python
1  # Display the even numbers in the chain
2  anode = anchor
3  while anode != None:
4      val = node.get_data(anode)
5      if val % 2 == 0:
6          print(val)
7      anode = node.get_next(anode)
```

In this example, we start by making a copy of the reference stored in the variable `anchor`, and storing it in the variable `anode`. We will use this variable to step through the chain, one node at a time, using a while loop. We must use a while loop for our nodes, because Python has no idea how to step through our chain using a for-loop. The condition for the while loop checks whether we've reached the end of the chain. There are no more nodes to look at if the variable `anode == None`. Notice line 7 especially. This is a key technique! Line 7 replaces the reference stored in `anode` with the reference stored in `anode`'s next field. Visually, this is stepping from one node to another by following the arrow out of the box labelled next.

It's important to point out that we always use a temporary variable to step through a node-chain. While it is possible to use the variable `anchor` for this purpose, it is rarely correct, except for trivial examples. We set up the variable `anchor` to be the anchor of our node-chain. If we change the value of this variable, we change the anchor point for the chain, and we lose the original anchor point. If we step through the node-chain by changing the anchor point, and we reach the end of the node-chain, we have effectively unhooked the whole node-chain from the anchor. It's really important to understand that without any anchor point, we have no way to access the node-chain when we're done, and it is absolutely, totally, and in all other ways, unaccessible.

This is not specific to nodes; any data value in the heap is lost if the reference to it is not stored somewhere. Python will simply sweep through the heap reclaiming the memory used by data not connected somehow to a frame; this process is called *garbage collection*, though it is more precisely analogous to recycling.

While we could (and should) write a lot of code to demonstrate the kinds of programs to manipulate node-chains, we will look at just one more example, specifically, removing a node from the far end of the chain. A node-chain can be arbitrarily long, even though our example chain only has 5 nodes in it. To reach the end of the node-chain, we could step through it one node at a time. We know that we've reached the last node of the chain when the node's next value is `None`. To unhook the last node from the chain, we have to set the next field of the second-last node to `None`. However,

if we actually step to the last node, there is no way to step backwards across a reference (an arrow in our diagram). Another standard technique for working with node-chains is to use two temporary variables, referring to two adjacent nodes in the chain. If we step both nodes through the chain, when one node is the last node, the other will be the second last node.

Here's the example.

```
1   anode = anchor
2   prev = None
3   while node.get_next(anode) is not None:
4       prev = anode
5       anode = node.get_next(anode)
6
7   if prev is not None:
8       node.set_next(prev, None)
9   else:
10      anchor = None
```

The two variables are `anode` and `prev`; `prev` will always be the node that contains a reference to `anode`. On line 3, the while loop condition checks whether `anode` is in fact the last node in the node-chain. One line 4, we advance `prev`, and on line 5, we advance `node`. Lines 4-5 ensure that `prev` is always one node behind `anode`. On line 6, we check whether `prev` is not None. The only way for `prev` to be None is if the first node in the node-chain is the only node in the node-chain, and removing the last node means removing the only node!

12 — Programming practice and style

Learning Objectives

After studying this chapter, a student should be able to:

- To be aware of stylistic concerns in programming, such as the importance of good variable names, program documentation, and clarity of code.
- To describe practices for defensive programming, such as assertions and exceptions.
- To describe different techniques for avoiding or handling floating point equality.
- To describe different techniques for avoiding or handling divide by zero.

12.1 Introduction: Style counts

Overwhelmingly, the focus of first year computer science is on getting the computer to do the thing that the assignment requires them to do. Admittedly, a program that doesn't work is worthless in this context. However, it must be stated with extreme emphasis that a program that cannot be understood by a person, whether that be an instructor, a marker, a colleague, or yourself, is also worthless. In first year, you write a program, hand it in, and then discard it. This pattern is true during your training, but it will absolutely false when you are out there doing real work. As you gain in experience, the exclamation "But it works!" becomes less and less relevant.

Style is important because valuable software will be read by human beings a lot more than you might think. You'll be working with code that someone wrote months or years ago, possibly before you joined the project team. You'll have to revisit your own work countless times as you bring a product to market, or as you prepare your data analyses for your scientific work. A well-written program is far easier to debug than a poorly-written one. And no matter how emphatically you tell yourself "I'll fix it later," *later* rarely turns into *now*.

This chapter is an attempt to inspire the kinds of aesthetics students should be reaching for as they reach the end of their first year of computer science. If you fail to employ the kinds of practices

we will present in this chapter, you are increasing the burden on everyone in the future who may be forced to read your horrible, but functional, code. Don't be that person who pretends that *using obscure language features* is an indicator of *cleverness*, and who thinks *it was hard to write* means *it should be hard to read*. Your software, produced under normal non-examination conditions, should be elegant, thoughtful, stylish, clear, concise, and of course, functional.

12.2   Style Guidelines

This section is going to be driven by examples. Some of these examples are real, in the sense that they were drawn from real programs. In some cases, the examples are contrived to make a point.

> **Practice 12.1**   Choose good variable names.   ∎

You're tired of hearing this, no doubt, but it's so important, that's where we always start. A good variable name is as brief as possible but still conveys the purpose of the variable. A good name reminds the reader of the purpose, so that reading becomes effortless. In contrast, a bad name adds an extra burden to the reader, by forcing them to memorize or deduce the purpose of the variable. Remember, the person being forced to work hard to figure out what your program does may be you yourself!

A good variable name uses words, or short phrases. Here are some examples:

```
clauses = # ...
left_subtree = # ...
number_of_elements = # ...
```

Different programming languages have conventions about how to write multi-word variable names. In Python, the convention is to use `snake_case` for variables, with lower case words separated by the underscore character. In other languages, the conventions may be different. Organizations and project teams may also have a set of conventions they use; use them!

Poor variable names use acronyms, ambiguous abbreviations, or just arbitrary characters. For example:

```
cs = # ...
l = # ...
n = # ...
```

When you're just blasting out code, you may find that using short variable names helps you get your ideas expressed quickly. That's fine; a very common piece of advice for all varieties of writing is to write the first draft quickly, and with minimal care, then go back and do your editing. When you use this strategy, make good use of your computer! Most Integrated Development Envirnoments (IDEs, like PyCharm) have a *Refactoring* menu, which will allow you to make changes to variable names all at once. This tool is slightly more advanced than your word processor's search-and-replace, because the IDE knows the rules of your programming language. The point here is that you can't use the excuse of "It takes too long" to turn poor variable names into good ones.

> **Practice 12.2**   Use short names for loop control variables.   ■

This is an exception to the previous advice. If your variable only has a short lifetime, say as part of a loop, a short name is preferred. Names like `v` or `val` are good in for-loops, and variables like `i` and `j` are good for indexed while-loops. Here's a trivial example:

```
for v in alist:
    # do stuff with v
```

When your loop control variables' names are too long, it actually increases the burden on your reader. For this purpose, long variable names tend to obscure the purpose of the loop:

```
for value_found_in_alist in alist:
    # do stuff with value_found_in_alist
```

You have to read the for-loop a lot more carefully to figure out what the variable is, and what it's ranging over.

> **Practice 12.3**   Avoid using variable names *l* (ELL), *I* (upper case i), and *o* (OH).   ■

This advice is just self-defence. Depending on the font you're using in your editor, the characters *l*, *I*, and *1* may be similar enough to cause confusion. Here's an example:

```
for l in clauses:
    # do stuff with 1
```

The same is true for *o*, *O*, and *0*. The best thing to do avoid these two single letter variable names entirely. Here's a better version:

```
for cl in clauses:
    # do stuff with cl
```

> **Practice 12.4**   Don't over-comment.   ■

Adding comments to your code can be immensely helpful. Keep in mind that as you gain experience coding, you will find the code itself to be increasingly informative on its own. The kinds of comments that would have helped you when you were just learning about for-loops won't be needed now that you've mastered them.

Comments should be about what's *not obvious* in your code. Adding comments about obvious concepts simply adds clutter, and distraction. Here's an exaggerated example of cluttered commenting:

```
i = 0 # list index for summing the list
total = 0 # running total for summing the list
while i < len(some_list):
    total = total + some_list[i]   # add to the running total
    i += 1    # increment i
```

The problem with the comments above is that they concern the individual statements, but say nothing helpful about the larger purpose. The comments in the body of the while loop are superfluous, as they say in English exactly what the Python says. This is not helpful to anyone, and should be avoided. Here's a better version.

```python
# calculate  the  total  cost  from  the  list  of  costs
i = 0
total = 0
while i < len(some_list):
    total = total + some_list[i]
    i += 1
```

Use comments to explain the purpose of a function, or, if the algorithm is tricky, the intuitions about the algorithm. Another thing you should do is include citations to books, articles, or websites if you are using an algorithm that you got from someone else. Assuming this is permitted by your assignment requirements! A reminder will save time repeating the research you did to find it in the first place.

> **Practice 12.5**   Write clearly – don't be too clever.   ■

The following example makes use of Python's list comprehensions. The problem is that the code does too much in one line. Experienced Python programmers encourage the use of comprehensions when they make the code easier to understand, but that's not the case here:

```python
new_cl = [[l for l in c if (-1)*flip*first_lit != l]
          for c in clauses if flip*first_lit not in c]
```

The following rewritten code accomplishes the same calculations.

```python
new_clauses = []
for clause in clauses:
    if flip * first_lit not in clause:
        new_clause = []
        for literal in clause:
            if (-1) * flip * first_lit != literal:
                new_clause.append(literal)
        new_clauses.append(clause)
```

The important point here is not about comprehensions. It's about writing code that does what it says clearly. Code that's too cleverly constructed may contain bugs that are even more cleverly hidden.

> **Practice 12.6**   Break complex expressions into smaller pieces.   ■

The example above is also demonstrative of this advice. In Python, the Python runtime system analyzes the code before it is executed, and finds ways to make your code run quickly on the computer you are using. In fact, all programming language systems do this, sometimes more, and sometimes less. The point is you don't have to worry about making little decisions about what the

computer should do all at once. You can afford to make your program clear, and let the system make instruction-level optimizations.

> **Practice 12.7**   Replace *repetitive expressions* by calls to a *common function*.   ∎

This is the essence of procedural abstraction, about which we've said a lot already. The most egregious demonstrations of repetitive expression come from the use of copy/paste, which is very tempting. But copy/paste is a terrible tool because it duplicates errors, and introduces opportunity to add errors. Here's an example, in which a simple while loop is used to calculate a small number of factorials:

```python
1  a = 10
2  b = 12
3  fa = 1
4  for i in range(1,a):
5      fa = fa * i
6  fb = 1
7  for i in range(1,b):
8      fb = fb * i
9  fab = 1
10 for i in range(1,a+b-1):
11     fab = fa * i
12 print('Mario paths:', fab/(fa*fb))
```

No doubt you've spotted the bug on line 11, which was introduced to demonstrate the kind of bug that gets introduced to code when it is copied/pasted. You may also note that the three loops don't quite calculate factorial, even when corrected. Is this a bug, or is this deliberate? It's far from clear unless you understand the purpose of the calculation on line 12.

It's far better to replace repetitive expressions to calls to a common function.

```python
1  def fact(n):
2      f = 1
3      for i in range(1,n+1):
4          f = f * i
5      return f
6
7  a = 10
8  b = 12
9  print('Mario paths:', fact(a+b-2)/(fact(a-1)*fact(b-1)))
```

With a single function replacing repeated code, any bugs in the algorithm are localized, and constrained (for `factorial`, at least). Also notice how using the function helps to clarify the intention of the calculation: For whatever reason, the expression on line 9 requires calculation of factorials that are one less than a, b and $a+b-1$. It seems deliberate, whereas the previous implementation seemed almost accidental.

12.3 Improving your code

This section is intended to provoke some ideas about how students can improve their code, without changing underlying algorithms, or worrying about efficiency. The suggestions here can improve robustness, but the basic unit of improvement is readability.

> **Practice 12.8**   Turn long conditionals into lookups   ■

In our courses, we presented conditionals long before any of the compound data structures like lists. But those long conditionals that you wrote when you were first learning them are not the only way, nor even a good way, to structure your code. Take for example the following partial example of an conditional about month names:

```python
if month == 1:
    print('Jan')
elif month == 2:
    print('Feb')
# etc, 12 months
```

It doesn't matter what the code does; we're looking at its structure. A long conditional adds a lot of code without actually contributing clarity. And while no one expects the number of months to change, there are applications of long conditionals that might be a little more dynamic.

Using a list to replace a long conditional is consistent with the concept of encapsulation. It essentially converts code (the conditional) into data (a list or dictionary):

```python
month_names = ['Jan', 'Feb', 'Mar', # and the rest
print(month_names[month-1])
```

The long conditional is reduced to a simple list access, and no clarity or purpose is lost. When the look-up is based on data that's not an integer, we can use a dictionary instead:

```python
month_dict = {'Jan':1, 'Feb':2, 'Mar':3, # and the rest
month = month_dict[month_name]
```

> **Practice 12.9**   Factor common statements out of loops and if-statements.   ■

This advice takes advantage of the mathematical nature of algorithms. There are patterns in code that can be simplified, in the same way that expressions in mathematics can be simplified. The first example demonstrates the use of sequence in a conditional:

```python
if x >= 0:
  y = x
  print('y is positive')
else:
  y = x
  print('y is negative')
```

In this example, Notice that y = x is the first thing done in both branches, and that it is independent of the condition $x \geq 0$. For that reason we can revise the code as follows:

```python
y = x
if x >= 0:
  print('y is positive')
else:
  print('y is negative')
```

We moved the common statement so that it occurs before the conditional because it appeared first in sequence. The revision does not change the behaviour of the code. It makes the code more concise, and probably clearer.

Similarly, we can transform the next example by moving the common statement after the conditional:

```python
if x >= y:
  x = x / 2
  print(x)
else:
  x = x * 2
  print(x)
```

Here, the value printed depends on the change that is being made to x, so moving the print statement before the conditional would change its behaviour. But moving it after would not change the behaviour.

```python
if x >= y:
  x = x / 2
else:
  x = x * 2
print(x)
```

A similar thing can be done with statements in loops. For example:

```python
for v in range(10):
  x = 0
  print(v*x)
```

Again, the statement x = 0 does not depend on v, so it is repeated unnecessarily. Here's an obviously better version:

```python
x = 0
for v in range(10):
  print(v*x)
```

Here, we have to move the statement x = 0 before the loop, because the loop depends on the value of x. A symmetric case can be made for moving code from inside a loop so that it is executed after the loop.

The point of this advice is that code has behaviour that can be reasoned about mathematically. The concepts you learned in your math class can be applied, if indirectly, to improving the quality

of your computer program. Sometimes, mathematics is not about math at all, but about creating an environment where certain patterns of reasoning can be taught, so that those patterns of reasoning can be applied outside of mathematics. In other words, *abstraction*.

Practice 12.10   Don't check for end-of-loop inside your loops!   ∎

It's fairly common for novice programmers to worry about doing too much all at the same time. For example, the following code is a counted loop from 0 to N, but contains a conditional that executes statement A when $i = N - 1$.

```
i = 0
while i < N:
     statement B
     if i == N - 1:
         statement A
     i += 1
```

The condition $i = N - 1$ is only true on the very last iteration of the loop. In English, this loop says "Keep doing B, and at the end of the loop, do A."

The main problem with this implementation is that it obscures the task unnecessarily. A far better structure is to realize that we don't need to keep checking if the loop is over. We can simply move statement A outside the loop, as follows:

```
i = 0
while i < N:
     statement B
     i += 1
statement A
```

As an added bonus, the cost the body of the loop is reduced. The initial implementation checked for the end of the loop twice for every value of i, but the improved version only checks once per value.

Practice 12.11   Don't use conditionals when a Boolean expression will do.   ∎

Because Boolean and relational expressions are required for conditionals and loops, it is common for novice programmers to believe implicitly that the only way to use them is in the context of a conditional or a loop. One of the key insights in a novice programmer's progression towards expertise is the realization that Boolean expressions can be used anywhere that any other kind of expression can be used. For example, consider the following function:

```
def function(a, p):
  if a == 3 or p < 0.005:
    return True
  else:
    return False
```

It returns a Boolean value, based on a Boolean expression. A far more concise implementation of this function simply uses the value of the expression as a return value:

```
def function(a, p):
  return a == 3 or p < 0.005
```

Boolean expressions represent calculations that evaluate to Boolean values, which can be assigned to a variable name, returned from a function, or used as arguments to other functions. Boolean expressions are not second-class citizens in computer science; they are absolutely equal in status to any other kind of expression. They simply evaluate to a different data type.

Practice 12.12   Don't compare a Boolean expression to a Boolean literal. ∎

When novices first learn about if-statements and loops, they see lots of examples of conditions comparing numbers using relational operators. It is natural to over-generalize and use the same pattern when dealing with Booleans. For example, consider the following if-statement:

```
if Stack.is_empty(astack) == True:
    print('Stack is empty')
```

This condition can be simplified by recognizing that the expression `Stack.is_empty(astack)` is a Boolean value that does not need to be compared to a literal in the context of an if-statement. We can use the expression all by itself to replace a comparison to `True`:

```
if Stack.is_empty(astack):
    print('Stack is empty')
```

This simplifies code, making it more readable.

The same kind of transformation can be done when the comparison is the the Boolean `False`, but we need to use the operator `not` as well:

```
while not Stack.is_empty(astack):
    print('Popping', Stack.pop(astack))
```

We can avoid the over-complicated condition that states `Stack.is_empty(astack) == False`.

Practice 12.13   Don't use a while loop to step through a list, even if you need indices! ∎

This advice is specific to Python. Suppose you're implementing a simple linear search through a list of items, and you need to know the index of the item:

```
while i < len(some_list):
    if some_list[i] == value:
        print('Found', value, 'at index', i)
    i += 1
```

The use of a while loop gives us control of the index variable, i. This is the way we taught you to do this kind of thing, but only because we can't teach you every thing all at once. The better way to write a loop like this is to use more Python, specifically, the function `enumerate`, which takes a sequence, and returns a sequence of tuples, namely the index and the value. Here's the same loop, rewritten:

```
for i, v in enumerate(some_list):
    if v == value:
        print('Found', value, 'at index', i)
```

This makes your code more concise, and eliminates common sources of error: forgetting to increment the while-loop counter (or incrementing it incorrectly), or indexing the list incorrectly. In more primitive programming languages, this kind of abstraction is not available, or it's not core to the language.

> **Practice 12.14**   Don't *patch* bad code – *rewrite* it.   ∎

Sometimes there's no way to *nudge* a program to fix it or make it better. While it is contrary to human nature, it is important for programmers to understand that sometimes you have to treat your first implementation as a *prototype*. The thing about prototypes is that they are useful only to help you understand what the algorithmic issues are. In real software development teams, prototypes are deleted and *replaced completely* all the time. Learn when *you personally* need to delete and start again from scratch. Throwing away code seems wasteful, especially when an assignment deadline is looming. But if you start early enough, you can learn a lot from a quick and dirty prototype, and then you can restart the development with a little fore-knowledge of what will be needed. And if you are taking advantage of version control, deleting your code is completely safe. You always have access to your backups!

12.4   Making your program faster

We have already mentioned the concept of efficiency, and we will spend some time to study an analytical technique to determine the efficiency of our algorithms. And as we saw above, sometimes a simple modification to code can improve its speed. This is all well and good, as long as you don't worry about this kind of thing too soon. Before you do any optimization, improve your code for clarity and robustness. If you can improve the clarity of your code, you might be able to find ways of making it more efficient that might not be possible when the program is convoluted or obscure.

> **Practice 12.15**   Make sure your code is correct *before* you make it faster.   ∎

There's no point in speeding up code that doesn't do the right thing. An inefficient implementation that works is far better than an efficient implementation that doesn't work. *Keep your code correct right* when you make it faster. Test your inefficient version throughly, too. Make sure that your test results do not change once you've started optimizing. Finally, make sure your function is *important enough* to make faster. If your function only gets called once, optimizing it may not contribute much value to your program. On the other hand, optimizing a function that gets called a lot will have a far greater effect on your program.

12.5   Defensive Programming

Defensive programming is an attitude that programmers practice when they stop pretending everything is fine. As a novice, assuming the best lets us learn the basic principles of programming, but if

anything can go wrong, as Murphy's Law states, it will. As you gain more experience programming, you will pick up habits that will help you avoid things going wrong, but to get you started, here are a few pieces of advice to help prevent errors.

Fundamentally, a defensive attitude ensures that a program protects itself against incorrect or illegal data. Preconditions and return values need to be checked; conditionals need to include a branch for those events that you assume can't possibly happen (because they will happen).

Many programming languages, including Python, have a tool for checking conditions that might identify if something has gone wrong. The Python command `assert` can be used to check a condition, and if the condition is false, it will halt the program with a run-time error. We will see a number of examples of its use. It is useful in situations where it is not clear what should be done if something has gone wrong.

Pre- and Post-Conditions

We have placed great importance on the specification of a function, to include a description of what values the function accepts as arguments. We assumed that the code making the function call will play nicely, by sending the right kind of information. But we should, when possible, use an assertion to check, as in the following example:

```python
def factorial(n):
    """

    Purpose: Compute the factorial.
    Precondition:
        :param n: a non-negative integer
    Return
        :return: a non-negative integer
    """
    assert n >= 0, 'invalid input to factorial'
    if n == 0:
        return 1
     else:
        return n*factorial(n-1)
```

The assertion checks whether *n* has an appropriate value, and if `factorial()` is ever called with a negative number, the assertion will *fail*, and Python will halt execution of the program, sending the given message to the console as part of the error report:

```
Traceback (most recent call last):
  File "fact.py", line 16, in <module>
    factorial(-5)
  File "fact.py", line 9, in factorial
    assert n >= 0, 'invalid input to factorial'
AssertionError: invalid input to factorial
```

You can use assertions to check for impossible situations outside of functions as well. They are very good for wolf-fencing and test scripts too. For example, here's how we might use an assertion in a test script:

```python
input = [1, 3, -5, 7, 9]
```

```
expected = -5
result = min(input)
assert result==min, 'Expected'+str(expected)+'found'+str(result)
```

The assertion states what should be true, and replaces an if-statement controlling a print statement.

An assertion that fails brings the whole program to a complete halt. This behaviour is fine when the program is in development, but it's a bit less ideal in a completed application. For this reason, Python allows you to turn off assertion checking, so that you don't have to edit the code and delete the assertions yourself. For your own protection, we won't show you how to do that yet.

A more versatile tool that can be used when things go wrong as called an *exception*. An exception is like an assertion, in that it can bring a program to a halt. But unlike an assertion, an exception can be *caught*, which allows programs to recover from some kinds of error situations. We will not say any more about exceptions for now.

> **Practice 12.16**   Don't shadow Python functions.   ■

As we learned in earlier chapters, variables in Python are just names, and almost anything can be attached to a name. This includes functions! A Python function has a name, but the name is simply a variable whose value refers to an encapsulated algorithm. For this reason, Python programmers can make the mistake of creating variables whose name is the same as a function.

```
sum = 0
for val in my_list:
    sum += val
```

Normally, this is an mistake, though sometimes it has no bad consequences, so it may not be as serious as an error. In the above example, the name sum is assigned the value 0, and the value 0 *replaces* its previous value, which is the well-known function capable of adding stuff up. There is nothing special about names in Python; the values are special, though. A function can be called, but an integer cannot be called.

A more subtle problem related to replacing a value is called *shadowing*.

```
def total(my_list):
    sum = 0
    for val in my_list:
        sum += val
    return sum
```

Here, the function total includes a variable named *sum*, which has the same name as the Python function sum. But the scope of the local variable is the function total, not the whole program. In this case, the use of the name sum refers to the local variable, and the Python function is not accessible; it is *shadowed*. The word shadow describes the situation well: the function sum cannot be seen because local variable sum gets in the way.

> **Practice 12.17**   Avoid modifying list structure while iterating over it.   ■

Python lists are mutable, which means they can be changed, either by changing the contents, or by changing the structure. While changing the contents of a list is pretty intuitive, changing its structure requires more care. When you delete or pop an item, the indices of some of the elements may change. For example, if we delete at item at index 5, the item formerly at index 6 is re-indexed, and takes position at index 5. Every item after index 5 is re-indexed. It's important to emphasize that deleting an element behaves *as if* all the items are shifted forward, but in Python, the lists don't actually shift; but their indices do change! It's important to remember this behaviour when you are trying to change the structure of a list using a loop. For example:

```python
a = list(range(10))
i = 0
while i < len(a):
    a.pop(i)
    i += 1
```

Because pop causes elements to be re-indexed, this loop only removes the even numbered items. For this reason, it's probably a good idea to avoid modifying the structure of a list while you are iterating over it. It is far less trouble to create a new list with the structure you want, from the old list.

Practice 12.18   Watch out for equality of floating point.   ∎

Floating point calculations have tiny errors due to the fact that floating point numbers have finite precision. Every calculation introduces a little more error, but some calculations add a lot more error than others. As a result, two floating point values are hardly ever equal, even if the math tells us that real values should be equal.

One of the biggest misconceptions about floating point numbers is that numbers the look simple in decimal notation are simple when stored in a computer. Here's an example:

```python
x = 0.3
y = 0.1 + 0.1 + 0.1
if x == y:
    print('Equal')
else:
    print('Not equal!')
```

In this example, x and y are not equal, because in binary, 0.1 has an infinitely repeating representation, much like $1/3$ is in decimal.

When your values are in normal, every day ranges, you can avoid odd behaviour by using a smaller value as an absolute threshold, like 0.00001:

```python
x = 0.3
y = 0.1 + 0.1 + 0.1
if abs(x - y) < 0.000001:
    print('Equal enough')
else:
    print('Not equal enough!')
```

The key behind this technique is that the threshold has to be smaller than the values you are comparing. If they are not, small but significant differences may be misinterpreted as unimportant using an absolute threshold.

Another technique is to use a threshold whose size is relative to the size of the values you are comparing:

```python
x = 0.3
y = 0.1 + 0.1 + 0.1
if abs(x - y)/max(abs(x),abs(y)) < 0.000001:
    print('Equal enough')
else:
    print('Not equal enough!')
```

Here, we're comparing the relative difference to a threshold, which works, as long as x and y are not both zero.

> **Practice 12.19**   Watch out for division by zero.   ■

The way to handle division by zero depends on your situation. If you have control over the denominator, you can use a conditional. For example, in the previous example about checking equality using a relative threshold:

```python
x = 0.3
y = 0.1 + 0.1 + 0.1

if x == 0 and abs(y) < 0.000001:
    print('Equal enough')
elif y == 0 and abs(x) < 0.000001:
    print('Equal enough')
elif abs(x - y)/max(abs(x),abs(y)) < 0.000001:
    print('Equal enough')
else:
    print('Not equal enough!')
```

This can be done because the case for a zero denominator tells you something important: if both values are zero, you don't need to divide at all to check for equality.

If the division is part of a formula, and the denominator value comes from some other part of the program, you should just let the division go through:

```python
def ratio(x, y):
    """Compute x/y for some reason.
    Preconditions: y != 0
    """
    return x/y
```

You might be tempted to use an assertion here, but division already raises an error when zero is used as a denominator, so checking yourself is extra work. Also, Python uses an exception for divide

by zero, and it's possible that some other part of the program will catch this exception, so using an assertion will interfere with the exception handling.

Finally, if the application domain has special requirements, you may be able to handle a division by zero using those requirements. For example, in some scientific calculations, division by zero can be handled by returning a particular value, maybe even zero:

```python
def ratio(x, y):
    """Compute x/y for some reason.
    If y == 0, ratio returns 0 because SCIENCE.
    Preconditions: x, y are numbers
    """
    if y == 0: return 0
    else: return x/y
```

Obviously, this is not a general solution. But zeroes headed towards a denominator don't always mean trouble.

12.6 How to improve your programming skills

To develop good programming style, you have to read code. Study examples given in class, and in books. Browse Python.org and Stack Overflow. Read as much code as you write. Learn the idioms of the programming language you are using. Always practice coding. Write little programs every day. Review your programs from past work, and try to make them a little better. When you start new assignments, imitate the style of your instructors. You don't have to produce a work of art every time you produce a program, but you can make it a habit to assess the quality of your own work. Keep a journal of the errors that caused you grief, or took a long time to debug.

Learn to use all the functionality of your IDE. Don't wait for a class to teach you these things. There are really valuable tools you can learn just by exploring the IDE menu. Learn to touch type. Stop using the mouse and menu. Use and memorize key-commands for your IDE or your text editor.

13 — Node-based Stacks and Queues

> **Learning Objectives**
>
> After studying this chapter, a student should be able to:
>
> - Describe the node-based implementation of stacks and queues.

13.1 Why use nodes to implement Queues and Stacks?

We started the course by reviewing lists, and we've already seen an implementation of Stacks and Queues based on Python lists.

In this chapter, we will see how we can implement the Queue and Stack ADTs using the node data structure we introduced in the previous chapter. As we mentioned, the node structure is not an esoteric detail. The idea of using a data structure like a node is one of the key advances in computer science. Prior to the invention of nodes, computer scientists could only store collections of data in finite, contiguous ranges of bytes, commonly called arrays. There were no lists before nodes were invented. Finally, it is important to keep mentioning the idea that we can completely change the implementation of an ADT like a stack or a queue, but as long as the interface stays the same, the behaviour of any application using the ADT is unchanged. This is the very essence of adaptability and robustness.

13.2 Using Nodes to Implement Queues

The essential nature of a Queue is the FIFO protocol. A node-based queue will use a node chain to contain the values in the queue, and to maintain the sequential ordering essential to the FIFO protocol. The *front* of the queue will be the first node in the node chain, and the *back* of the queue will be the last node in the node chain. We will use the Node ADT to implement the Queue operations, but from the outside, only the Queue operations will be visible, and the reliance on the Node ADT will not be evident.

We will be using a dictionary as a record to store information relevant to the queue. The fields we will be keeping track of are as follows:

front A reference to the first node in the node chain that stores the queued values; the front of the queue. If the queue is empty, this should be the Python value `None`. This field acts as the anchor for the queue's node chain.

back A reference to the last node in the node chain that stores the queued values; the back of the queue. If the queue is empty, this should be the Python value `None`.

size An integer representing the number of elements currently stored in the queue. This value should be non-negative. A size of zero implies that the queue has not nodes, and no data.

It is important to point out that this extra record completely hides the node-chain of stored data. This record is an extra level of complexity that the ADT programmer needs to maintain. The advantage of that extra complexity is that any application using the Queue ADT need only call the Queue operations, and will not need any code for dealing with node-chains. The Queue operations are responsible for adding new data values to the back of the node-chain, and for removing data values from the front of the node-chain. The application only needs to use `enqueue()` and `dequeue()`.

The following sections describe the implementation in a little more detail. The complete implementation can be found in Appendix D.

13.2.1 Operation: create()

The `create()` function returns a new empty queue. In this implementation, a queue is dictionary used as a record. This operation simply sets all the fields to the values that indicate an empty queue.

```python
def create():
    """
    Purpose
        creates an empty queue
    Return
        an empty queue
    """
    queue = {}
    queue['size'] = 0        # how many elements in the queue
    queue['front'] = None    # the node chain starts here
    queue['back'] = None     # the node chain ends here
```

If the queue is displayed, it will look like a dictionary. Keep in mind that there is no implied sequence for a Python dictionary, and the displayed order of the keys may change depending on which computer you use, or which version of Python you use.

For example, using the `create()` operation in a script such as

```python
import Queue as Queue

aqueue = Queue. create()
```

can be visualized as in Figure 13.1. The diagram is simplified a bit, as the Queue data structure is in reality a dictionary, but the diagram shows the keys in blue, and the values in rectangles. As well, the values are represented as being stored within the Queue, but in reality, references to `None` and `0` are stored in the Queue.

Figure 13.1: The top diagram shows an empty Queue, created on the heap by the `create()` operation. The bottom diagram shows what a queue would look like after several values have been enqueued.
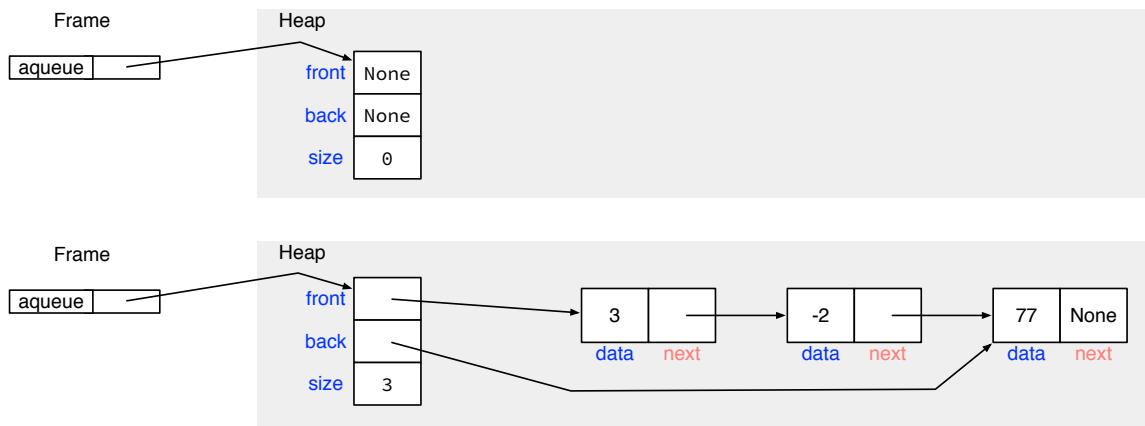
A Queue containing a few values is also shown in Figure 13.1 (the bottom diagram). This demonstrates the kind of consistency that all Queue operations need to maintain. The field `front` refers to the first node in the node-chain; the field `back` refers to the last node in the same node-chain. The field `size` always stores the number of nodes in the node-chain.

13.2.2 Operations: size() and is_empty()

The `is_empty()` operation returns `True` if the queue is empty. The `size()` operation returns the number of elements stored in the queue. Both implementations below simply look up the value of the `size` field in the dictionary. This implies that the operations for enqueueing and dequeueing data have to change the size consistently.

```python
def size(queue):
    """
    Purpose
        returns the number of data values in the given queue
    Pre-conditions:
        queue: a queue created by create()
    Return:
        The number of data values in the queue
    """
    return queue['size']
```

```python
def is_empty(queue):
    """
    Purpose
        checks if the given queue has no data in it
    Pre-conditions:
        queue is a queue created by create()
    Return:
        True if the queue has no data, or false otherwise
    """
    return queue['size'] == 0
```

13.2.3 Operation: peek()

The peek() operation returns a reference to the data value currently at the front of the queue, without removing it from the queue. If the given queue is empty, the operation causes a run-time error, because the assertion will fail.

```python
def peek(queue):
    """
    Purpose
        returns the value from the front of given stack
        without removing it
    Pre-conditions:
        queue: a non-empty queue created by create()
    Post-condition:
        None
    Return:
        the value at the front of the queue
    """
    assert not is_empty(queue), 'peeked into an empty queue'

    first_node = queue['front']
    result = node.get_data(first_node)
    return result
```

13.2.4 Operation: dequeue()

The dequeue() operation removes a data value from the front of the queue. In this implementation, the value is removed from the front of the node chain stored in the Queue record. The dequeue operation has three situations to be handled:

1. If the queue is currently empty, nothing can be dequeued. In this implementation, the Python value None is returned.
2. If the queue has exactly 1 value stored, dequeueing a value would result in an empty queue. This requires special care, since both the first node and the last node in the chain are being affected.
3. If the queue has more than one value stored, then only the first node in the chain is affected; the information about the last node in the chain does not change.
   To remove the first node in the chain, we have to save a reference to the data value stored there, and then change the queue's front field so that it refers to the node after the first node in the chain. By changing the reference of the first node in the chain, we are omitting the node that was previously the front in the chain. This is enough to remove the value and the node from the queue

In both cases where a value can be dequeued, we can store a temporary reference to the node to be removed, and temporarily store the value to be returned. We can also change the queue's front field, by making a copy of the node's next reference. Note that if the node to be removed is the very last one in the chain, its next will be Python value None, and that is the proper value for the first node of an empty chain. We can also decrease the size of the queue.

   We only need to make sure that if we remove the last value in a queue, that the queue's back field is also set to None; if the queue had several values in it, removing the first node would require not change to the queue's back field.

```python
def dequeue(queue):
    """
    Purpose
        removes and returns a data value from the given queue
    Pre-conditions:
        queue: a queue created by create()
    Post-condition:
        the first value is removed from the queue
    Return:
        the first value in the queue, or None
    """
    assert not is_empty(queue), 'dequeued an empty queue'

    prev_first_node = queue['front']
    result = node.get_data(prev_first_node)
    queue['front'] = node.get_next(prev_first_node)
    queue['size'] -= 1
    if queue['size'] == 0:
        queue['back'] = None
    return result
```

Figure 13.2: The effects of a dequeue operation on a non-empty queue. The reference stored in `front` changes, and the `size` decreases by one.
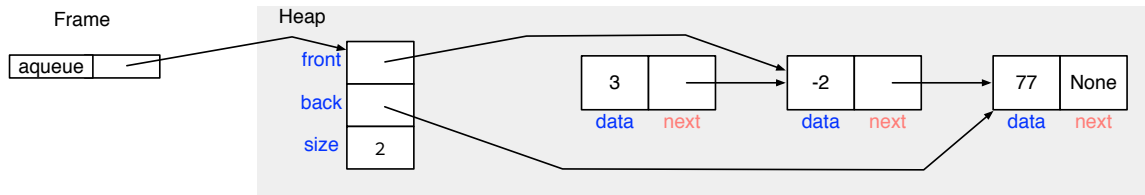
For example, if `dequeue()` were called on the Queue drawn in the bottom half of Figure 13.1, the result would as shown in Figure 13.2. Notice that the reference stored in `front` now indicates the node containing the value -2. The diagram shows that the node containing the value 3 still present in the heap, but nothing in the node chain refers to it, so it is not part of the Queue. If there is no way to access this node from a frame, Python will reclaim the memory it uses for some other purpose at some time in the future.

13.2.5 Operation: enqueue()

The `enqueue()` operation adds a new data value to the back of the queue. In this implementation, a new node is created to store the new value. The new node will always be placed at the end of the chain, that is, the new node will be the last node, but there are two cases that require special care.

1. If the queue is currently empty, the new value will be stored in a new node that is both the first node and the last node in the chain. For that reason, we have to change the Queue record so that `front` and `back` both refer to the same new node.
2. If the queue is not currently empty, then only the last node in the chain is directly affected; the information about the first node in the chain does not change.
   To add a new node at the end of the chain, we can use the reference to `back` directly. To prevent too much confusion, let's use the term *previously last node* to refer to the node that was at the back of the chain prior to a new value being enqueued. When the new node is attached properly, the previously last node should contain a reference to the new node, and the queue's `back` field should refer to the new node.

In both cases, the size of the queue increases by 1.

Figure 13.3: The effects of an enqueue operation on a non-empty queue. The new node is connected to the end of the chain, and the reference stored in `back` changes. The value stored in `size` is increased by one.
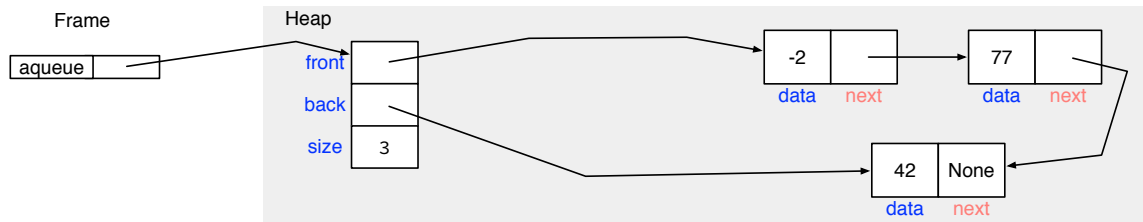

```
def enqueue(queue, value):
    """
    Purpose
        adds the given data value to the given queue
    Pre-conditions:
        queue: a queue created by create()
        value: data to be added
    Post-condition:
        the value is added to the queue
    Return:
        (none)
    """
    new_node = node.create(value, None)

    if is_empty(queue):
        queue['front'] = new_node
        queue['back'] = new_node
    else:
        prev_last_node = queue['back']
        node.set_next(prev_last_node, new_node)
        queue['back'] = new_node

    queue['size'] += 1
```

For example, if `enqueue()` were called on the Queue to add the value 42 to the Queue drawn in Figure 13.2, the result would as shown in Figure 13.3. Notice that the reference stored in `back` now indicates the node containing 42, and the node that used to be at the back now refers to the new node as well.

13.3   Using Nodes to to Implement Stacks

The implementation in this section is very similar to the implementation of node-based queues. We will be sure to highlight the differences.

The essential nature of a Stack is the LIFO protocol. A node-based stack will use a node chain to contain the values in the stack. The front of the node-chain will be the top of the stack. Since the LIFO protocol requires access only to the top of the stack, the stack implementation will be slightly simpler.

As with the queue, earlier, we will be using a dictionary as a record to store information relevant to the stack. The fields we will be keeping track of are as follows:

top   A reference to the first node in the node chain that stores the stacked values. If the stack is empty, this should be the Python value `None`.

size   An integer representing the number of elements currently stored in the stack. This value should be non-negative. A size of zero implies that the stack has no nodes, and no data.

Note that we are not using a reference to the last node in the stack's node chain. It is simply not needed, because no Stack operation needs direct access to the bottom of the stack!

The following sections describe the implementation in a little more detail. The complete implementation can be found in Appendix D.

13.3.1   Operation: create()

The `create()` function returns a new empty stack. In this implementation, a stack is dictionary used as a record. This operation simply sets all the fields to the values that indicate an empty stack.

```
def create ():
    """
    Purpose
        creates an empty stack
    Return
        an empty stack
    """
    stack = {}
    stack['size'] = 0          # how many elements in the stack
    stack['top'] = None        # the node chain starts here
    return stack
```

If the stack is displayed, it will look like a dictionary. Keep in mind that there is no implied sequence for a Python dictionary, and the order of the keys may change depending on which computer you use, or which version of Python you use.

For example, using the `create()` operation in a script such as

```
import Stack as Stack

astack = Stack. create ()
```

can be visualized as in Figure 13.4. The diagram is simplified a bit, as the Stack data structure is in reality a dictionary, but the diagram shows the keys in blue, and the values in rectangles. As well,

Figure 13.4: The top diagram shows an empty Stack, created on the heap by the `create()` operation. The bottom diagram shows what a queue would look like after several values have been pushed.
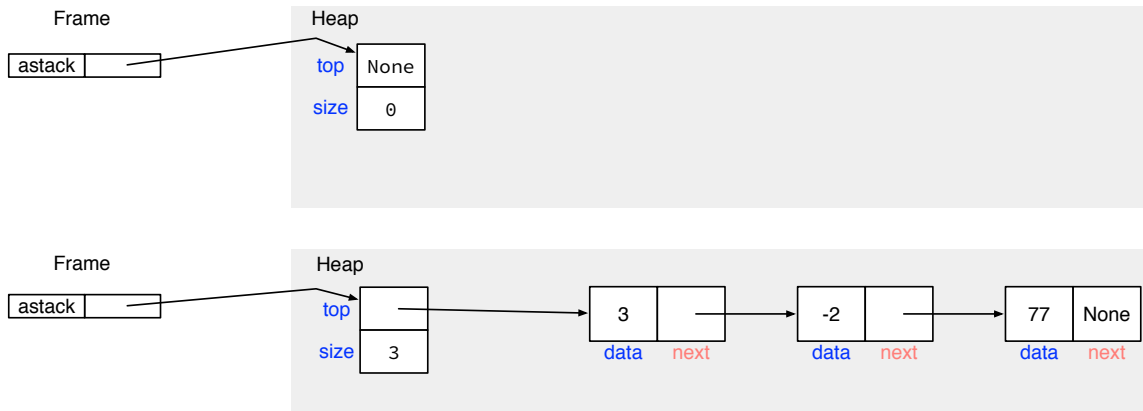
the values are represented as being stored within the Stack, but in reality, references to `None` and `0` are stored in the Stack.

13.3.2 Operations: size() and is_empty()

The `is_empty()` operation returns True if the queue is empty. The `size()` operation returns the number of elements stored in the queue. Both implementations below simply look up the value of the `size` field in the dictionary. This implies that the operations for pushing and popping data have to change the size consistently.

```
def size(stack):
    """
    Purpose
        returns the number of data values in the given queue
    Pre-conditions:
        queue: a queue created by create()
    Return:
        The number of data values in the queue
    """
    return stack['size']
```

```python
def is_empty(stack):
    """
    Purpose
        checks if the given queue has no data in it
    Pre-conditions:
        queue is a queue created by create()
    Return:
        True if the queue has no data, or false otherwise
    """
    return stack['size'] == 0
```

These two operations should look familiar, as they are identical to the Queue operations of the same name, except for the parameter names.

13.3.3 Operation: peek()

The peek() operation returns a reference to the data value currently at the top of the stack, without removing it from the stack. If the given stack is empty, the operation causes a run-time error, because the assertion will fail.

```python
def peek(stack):
    """
    Purpose
        returns the value from the front of given stack
        without removing it
    Pre-conditions:
        stack: a stack created by create()
    Post-condition:
        None
    Return:
        the value at the front of the stack
    """
    assert not is_empty(stack), 'peeked into an empty stack'

    first_node = stack['top']
    result = node.get_data(first_node)
    return result
```

13.3.4 Operation: pop()

The pop() operation removes a data element from the top of the stack. In this implementation, the value is removed from the node at the front of the node chain. This is very similar to the dequeue operation of the node-based queue, but is a little simpler because the stack does not have to maintain the reference to the last node in the chain.

There is basically only one special case to be aware of. In the case that the stack is empty when popped, the operation will cause a run-time error, because the assertion will fail. If the stack is not empty, we make a copy of the reference to the top node, and remember the data stored in that node.

Figure 13.5: The effects of a pop operation on a non-empty stack. The reference stored in `top` changes, and the `size` decreases by one.
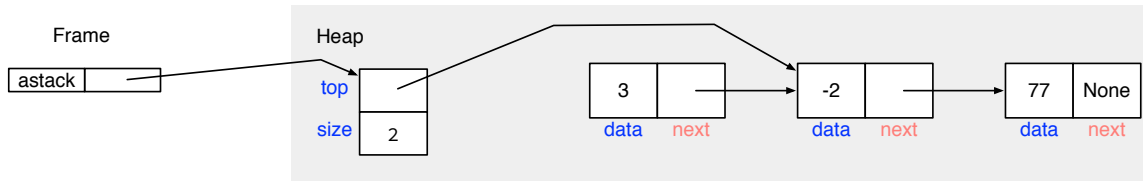
Whatever follows the removed node will be the top of the chain, and we decrease the size of the stack.

```
def pop(stack):
    """

    Purpose
        removes and returns a data value from the given stack
    Pre-conditions:
        stack: a stack created by create()
    Post-condition:
        the first value is removed from the stack
    Return:
        the first value in the stack, or None
    """
    assert not is_empty(stack), 'popped an empty stack'

    prev_first_node = stack['top']
    result = node.get_data(prev_first_node)
    stack['top'] = node.get_next(prev_first_node)
    stack['size'] -= 1
    return result
```

For example, if `pop()` were called on the Stack drawn in the bottom half of Figure 13.4, the result would as shown in Figure 13.5. Notice that the reference stored in `top` now indicates the node containing -2. The diagram shows that the node containing 3 still present in the heap, but nothing in the node chain refers to it, so it is not part of the Stack. If there is no way to access this node from a frame, Python will reclaim the memory it uses for some other purpose at some time in the future.

13.3.5 Operation: push()

The `push()` operation adds a new data element to the top of the queue. In this implementation, a new node is created to store the new value. The new node will always be placed at the front of the chain. Unlike the situation with enqueue, adding a new node to the front of the chain requires no special consideration. A new node is created for the new value, and this node has a next reference to the node that used to be the front of the stack's node chain. A reference to the new node itself is copied to the stack's `top` field. The size of the stack is increased by one, as well.

Figure 13.6: The effects of an push operation on a non-empty stack. A new node is added to the chain, and `top` changes. The `size` increases by one.
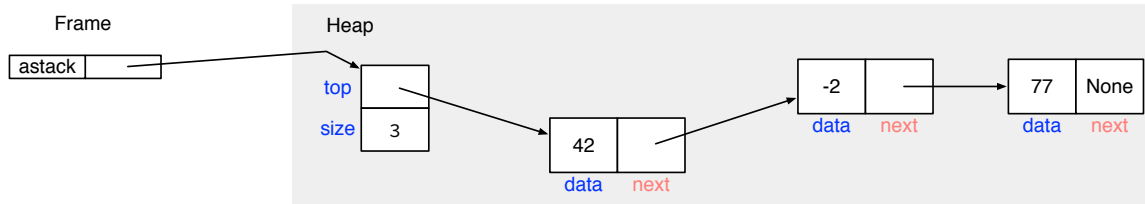

```
def push(stack, value):
    """
    Purpose
        adds the given data value to the given stack
    Pre-conditions:
        stack: a stack created by create()
        value: data to be added
    Post-condition:
        the value is added to the stack
    Return:
        (none)
    """
    new_node = node.create(value, stack['top'])
    stack['top'] = new_node
    stack['size'] += 1
```

For example, if `push()` were called on the Queue to add the value 42 to the Stack drawn in Figure 13.5, the result would as shown in Figure 13.6. Notice that the reference stored in `top` now indicates the node containing 42, and the new node now refers to the node that used to be at the top.

13.4 Summary

The similarity between the queue and stack implementations is very strong. The stack was a bit simpler, because we only had to keep track of one end of the chain.

One of the most important lessons to learn in this chapter is the case-analysis. When a textbook tells you that "the dequeue operation has three cases to handle," it might seem obvious. But coming up with those cases on your own when you're solving any problem is not always so easy. Humans typically focus on a typical case, for example, the case where the node chain has a few nodes in it; some people call this kind of situation a *sunny day* case. Some people will also notice the case of an empty node chain, but hardly any novice programmer will realize that the case of a node chain with exactly one node in it is also a special case for the dequeue operation; these could be called *special* cases, or sometimes *edge* cases. Nothing about Python (or any language) tells us about what the edge cases are, or how to handle them; the only way to discover them is to understand your problem so thoroughly that you've figured them out. It is essential that you get into the habit of trying to

figure out all the cases for your particular problem. The only way to find special cases is to keep asking "What if ...?" For many kinds of computational problems, drawing diagrams and working things out on paper (or a white board) is really the best practice, before you sit down at a computer to start programming.

14 — Linked Lists

> ## Learning Objectives
>
> After studying this chapter, a student should be able to:
>
> - Employ data abstraction to implement a list.
> - Describe the node-based implementation of lists.

14.1 Linked List ADT

We started the course by reviewing lists, suggesting that they are very useful tools that help us solve many programming tasks. But we understand Python's lists as an ADT; that is, we have an understanding of their purpose, and we've used many of their operations, but we have not seen their implementation.

In this chapter, we will see how we can implement an ADT very similar to Python's lists using the Node ADT that we presented in Chapter 11; this kind of implementation is known as a linked-list. Our ADT will be based strongly on the implementation of node-based queues in Chapter 13. We'll describe the ADT operations in some detail, but the implementation will be left as a homework assignment.

The term *linked-list* is common in computer science. Frequently, it is used to refer the to structure we called a *node-chain*. For pedagogical purposes, we decided to help avoid confusion by introducing the term node-chain, which is helpfully descriptive. This allows us to use the term linked-list to name the ADT that is the topic of this chapter. For us, the Linked List is at the same level of abstraction as the Queue and Stack ADTs of the previous chapter. Like the Queue and Stack ADTs, the Linked List ADT will be a record encapsulating a node-chain. Unlike the Queue and Stack ADTs, which provide only a few operations, a Linked List will be far more flexible, providing operations not needed by Stacks and Queues.

The exercise of building a linked list ADT from the node ADT has several purposes. First of all,

it's very good programming experience in general. It is important to point out that many languages use references like Python provides, and the practice allows us to transfer that knowledge to those languages. Second, the practice of using references is necessary, since we will need them again in Chapter 17, for something a little more complicated. Believe it or not, there are more interesting and complicated data structures than stacks, queues and lists that need this concept.

The linked list ADT we develop in this chapter is not how Python lists are implemented. Python lists are even more advanced.

14.2 Linked List Record

Like the Stack and the Queue, a Linked List is a record that refers to a single node-chain. The first node in the node-chain is called the *head* of the list, and the last node in the node-chain is called the *tail* of the list. Be careful to understand that there is exactly one chain, but we have two separate references to nodes on the chain.

head A reference to the first node in the node chain that stores the queued values. If the queue is empty, this should be the Python value None. This field acts as the anchor for the queue's node chain.

tail A reference to the last node in the node chain that stores the queued values. If the queue is empty, this should be the Python value None.

size An integer representing the number of elements currently stored in the queue. This value should be non-negative. A size of zero implies that the queue has no nodes, and no data.

14.3 Linked List Operations

Here we briefly outline a few of the operations.

14.3.1 Operation: create()

The create() function returns a new empty linked list record. This operation simply sets all the fields to the values that indicate an empty list.

```
def create ():
    """
    Purpose
        creates an empty list
    Return
        :return an empty list
    """
    llist = {}
    llist['size'] = 0      # how many elements in the stack
    llist['head'] = None   # node chain starts here
    llist['tail'] = None   # and ends here; initially empty
    return llist
```

This is very similar to the node-based queue in Chapter 13.

For example, using the create() operation in a script such as

Figure 14.1: The top diagram shows an empty List, created on the heap by the `create()` operation. The bottom diagram shows what a linked list would look like after several values have been added to it.
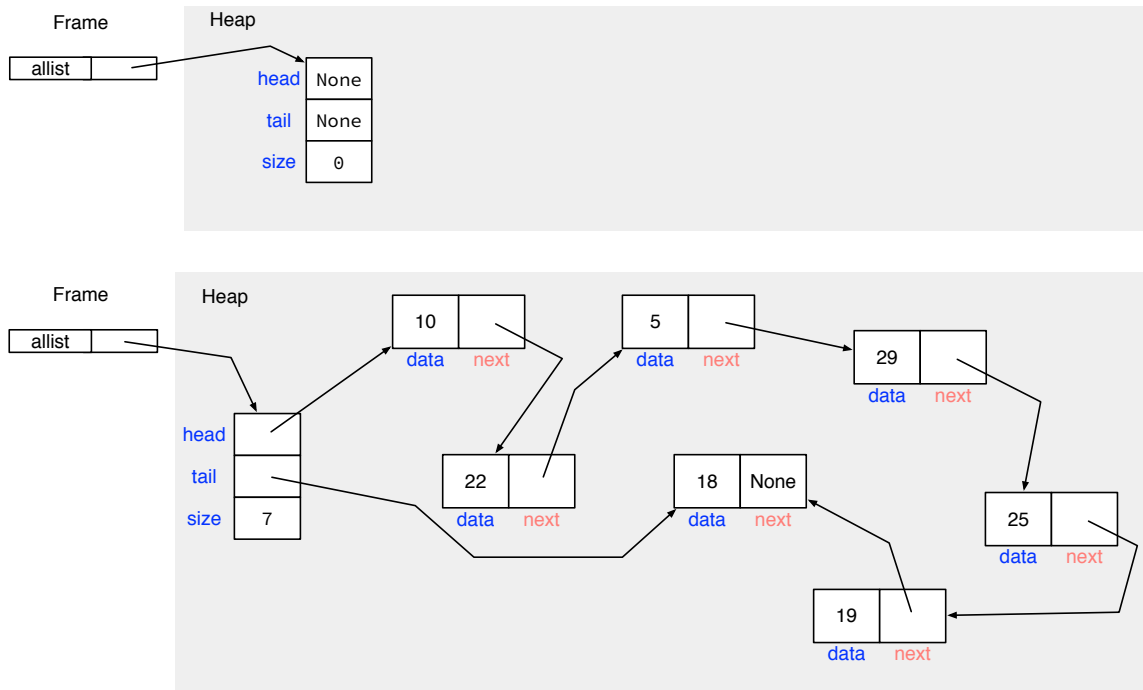

```
import LList as List

allist = List.create()
```

can be visualized as in Figure 14.1. The diagram is simplified a bit, as the Linked List data structure is in reality a dictionary, but the diagram shows the keys in blue, and the values in rectangles. As well, the values are represented as being stored within the Linked List, but in reality, references to None and 0 are stored in the Stack.

14.3.2 Operations: size() and is_empty()

Because queues, stacks and linked lists are all based on the Node ADT, the operations `size()` and `is_empty()` are basically all the same, and therefore do not need to be presented here.

14.3.3 Summary of Linked List Operations

In this section, the Linked List operations are described by their purpose. In the descriptions, no mention of the node-chain is made. However, the operations must be implemented using the node-chain stored in the linked list record, in the same way that the Queue and Stack ADT operations were implemented using node chains.

Several operations use the concept of an index. As with Python lists, the index 0 refers to the first value in the list, and the last valid index for a list of size *n* is $n-1$.

add_to_front(alist, val) Insert the given value `val` into the given Linked List `alist` so that the new value is at the front of the sequence of values.

remove_from_front(alist) Removes and returns the first value in the given Linked List `alist`.

add_to_back(alist, val) Add the given value `val` to the given Linked List `alist` so that the new value is at the end of the sequence of values.

remove_from_back(alist) Removes and returns the last value in the given Linked List `alist`.

get_data_at_index(alist, idx) Return the value stored in `alist` at the index `idx`. This function does not change the sequence; it simply reports what the value is stored at the given index.

set_data_at_index(alist, idx, val) Store `val` into `alist` at the index `idx`. This operation does not change the structure of the list. It simply replaces the value currently stored at `idx` with the given value.

value_is_in(alist, val) Check if the given value `val` is in the given list `alist`. Returns `True` if `val` is anywhere in the sequence, and `False` otherwise.

get_index_of_value(alist, val) Report the index of the given value `val` in the given list `alist`. If `val` appears more than once, the index of the first occurrence is reported. This function returns the tuple (`True, i`) if the given value appears in the list, where *i* is the index of `val`. If the value is not in the list, this function returns `False, None`.

insert_value_at_index(alist, val, idx) Insert `val` into `alist` at index `idx`. This operation changes the structure of the list by adding a new value into the sequence, provided that `idx` is a valid index. For simplicity, we'll assume the index is non-negative, and in the range 0 to *n*, where *n* is the length of the list. If the index given is equal to the size of the list, the new value is added to the end of the sequence.

delete_item_at_index(alist, idx) Delete the value at index `idx` in the given list `alist`. This operation changes the structure of the list, by removing a value. For simplicity, we'll assume that a valid index is non-negative, and in the range 0 to $n-1$, where *n* is the length of the list.

delete_value(alist, val) Delete the value `val` from the given list `alist`. This operation changes the structure of the list, by removing a value. If the given value appears more than once, on the first occurrence is removed. If the given value does not appear in the sequence, the list remains unchanged. The function returns `True` is a value was deleted, or `False` if not.

14.3.4 Implementation details

The implementation of many of the above operations is similar to Stack and Queue operations given in Chapter 13. The operations that are different usually involve an index argument. Unlike Python lists, which can be indexed directly using normal list indexing syntax, e.g., `alist[2]`, the Linked List ADT cannot use indexing syntax. Because the values are stored in a node-chain, index for a Linked list operation tells you how far along the node chain you have to step to find the value indicated by the index.

The operations for indexing require the use of while loops. Python's for-loops are useful for Python lists, but not so much for node-chains.

14.3.5 Perspective

The purpose of this chapter is to reveal one important technique as the basis for the list data structure give to Python programmers in the core Python language.

Functional programming languages, e.g., LISP, Scheme, SML, Haskell, use node chains as their essential compound data structure. Object oriented languages such as C++ and Java have standard modules for Linked lists very similar to what we have seen here. More primitive languages, such as C, do not have lists at all, and one of the "rites of passage" for C programmers was building their own Linked List ADT.

While Python lists are not currently linked lists the way we've seen here, they certainly were linked lists like this very early in Python's development as a language. As more people used Python, greater importance was placed on making the operations as efficient as possible, so a more advanced data structure was introduced. It is very important to realize that a better implementation of the list operations is hidden behind the ADT interface, and applications already written using Python did not need to be rewritten.

There are lots of operations we could have added to the Linked List ADT. For example, in Python we can *slice* a list, e.g. `alist[2:7:2]`. We could write a Linked List operation that does the same thing. The point is that every Python list method is implemented as a function that is in some sense similar to the Linked List operations in this chapter.

15 — Algorithm Analysis

Learning Objectives

After studying this chapter, a student should be able to:

- Explain the value of algorithm analysis.
- Describe the use of abstraction to quantify time and space in analysis.
- Explain the value and purpose of best-case, worst-case, and expected-case analyses.
- Employ best- and worst-case analyses for time and space for algorithms involving loops, conditionals and function calls.
- Explain the relationships between basic polynomial complexity classes.
- Explain what is meant by "best" and "worst" case behavior of an algorithm. [Partial, AL-BA-1]
- Determine informally the time and space complexity of simple algorithms. [Usage, AL-BA-3]

15.1 The importance of understanding costs

When you're using a computer to solve a problem, you should always be aware of the costs of the solution. In this chapter, we'll focus on the costs associated with running a program: namely, the amount of time and memory required by the program while it is running. Concerns about time and memory costs really only come into significance when the programs get larger, when datasets get larger, and when the problems get more difficult to solve.

Being able to assess costs is essential in making design choices: if you have more than one algorithm to solve a given problem, you will want to choose the one that has lower costs. Computers are fast, but they are not infinitely fast. The programs we write in a first-year course may run in milliseconds, but that's not something we should take for granted. The software applications, and Internet services, we use are fast, precisely because designers understood the costs, and figured out

ways to reduce them. A lot of the speed we expect comes from the hardware, but even miraculous hardware cannot always save designers from poor software design choices.

We'll hone our analysis skills on Python programs, but it's crucial to understand that these skills are transferable to any programming language. In fact, we will even be able to apply these skills to algorithms expressed in more abstract ways, like pseudocode and flowcharts. The key point here is that we want to understand the costs of a program while it's still in the design phase, without implementing it first. If we can assess the cost of an algorithm, the analysis will apply to any of the programs we might come up with as an implementation of the algorithm.

Our analysis methods will identify the key costs, and express them in a way that is independent of the speed of any particular computer, independent of the programming language, and independent of minor implementation differences. It's such an important skill that we have to start practicing the basics sooner than later. In fact, this chapter will focus almost entirely on time costs. Once you get the hang of the analyses here, you can easily use the same skills to measure memory costs, or network costs, or almost any other computational resource, using a very similar approach.

15.1.1   Abstraction #1: Time

It may sound odd, but we will not measure time costs in typical time units like seconds, minutes, or hours. To understand why, consider a single program executed on two different computers, say one from 1967, and another from 2017. It's obvious that the program would be doing the same calculations on both computers, and the computers would be doing the calculations at different speeds. We want our analysis to say more about the algorithm than the computer it's running on.

We will measure time indirectly, in terms of computational steps. For us, a step will be any computational action that cannot be broken down into other steps. Expressing the costs in terms of the number of steps allows us to assess the algorithm, independent of the computer running the program. We will say more about what counts as a computational step a bit later, but it will include such things as arithmetic operations, assignment statements, comparisons, etc. There is nothing surprising waiting for us.

We assume that a step will take a constant amount of time. This is not always true, but it's very useful in many cases. One reason is that computer programs typically use relatively small numbers. Any floating point number that can be stored on the heap counts as relatively small. Any integer that is likely to come up in normal calculations counts as relatively small.[1]

We also assume that all the different kinds of steps take roughly the same time on a given computer to do. This is also a bit of an approximation. For example, we are suggesting that adding two numbers is a step, and multiplying two numbers is a step, but multiplication requires addition (think about the algorithm to multiply 2345×6845), so multiplication is obviously a bigger step than addition. For many common purposes, we are content to say that addition and multiplication both count as the same size step because the numbers involved are relatively small, and the time it takes to multiply is so small that time differences between adding and multiplying doesn't really matter. The key assertion here is this: We don't (usually) lose a lot of decision-making power by assuming each step is performed in about the same amount of time.

---

[1] In some languages, the range of numbers that can be stored using the basic language constructs has precise limits. In Python 3, the floating point numbers are limited this way, but integers are not limited at all. So strictly speaking, integers in Python 3 are not always considered small in this way. But normal calculations usually don't involve numbers so big that they represent extraordinary costs.

15.1.2 Abstraction #2: Input size

Consider a program that analyzes some data. It is reasonable to assume that the program would need a larger number of steps to analyze more data. Our accounting for steps has to be flexible enough to allow analysis no matter what amount of data is used.

To do this, we employ a technique called "parameterization." The word "parameterization" means that we give a mathematical name to a quantity that is either unknown in advance, or may change depending on things beyond our control.

Using parameterization, we will assign a mathematical name to the size of the problem; typically, we use N or n. For example, we might want to sort a list of size N. Or we might want to add numbers having N digits. Or maybe we need storage for a video whose length is N seconds. There are also situations where the size of a problem may require multiple parameters, for example, applying an algorithm to a spreadsheet with N rows and M columns.

Our analysis will express the costs of a program in terms of the parameters describing the size of the input. In this way, we have an analysis that does not depend on any given example data set. For example, an algorithm to find the smallest value in a list of N values might require $3N + 4$ steps. The cost is being expressed as a function of the size of the input, and the input size is being expressed as a parameter, N. We'll see how to do this analysis; right now, we need to understand the idea of input size and parameterization.

15.1.3 Abstraction #3: Categories

In biology, scientists study plants and animals, assess their similarities and differences, and use a complicated organizational scheme to express the relationships between species. In computer science, we also study, evaluate, and categorize algorithms. Like biology, our categorization scheme has to be useful for algorithms we already know, and for any that might be invented in the future.

Our categorization scheme will distinguish significant differences, and ignore minor differences. Suppose you are trying to decide between Algorithm A and Algorithm B, both of which solve the same problem. Suppose you determine that Algorithm A takes $3N + 3$ steps, and Algorithm B takes $3N + 4$ steps. Obviously, Algorithm A is one step better, but you are on the right track if you are wondering whether a difference of one step is actually significant.

The categorization system will be based on two ideas:

1. Each category will contain a set of algorithms.
2. Each category will be labelled with a simple mathematical function that describes the algorithms in the set.

We'll say a bit more about the meaning of the categories a bit later.

There are as many categories in our system as there are different kinds of functions, but we'll start with the easy ones. The categories will be labelled as functions expressed in terms of input size. For example, we'll have a category labelled as linear, N, as well as quadratic, N^2. Any constant power of N is a category, too, e.g., N^{15}. We have categories based on exponential and logarithmic functions, e.g., 2^N and $\log_2 N$. We even have a category for functions that don't depend on a size parameter N at all (this is the category labelled with the value 1).

The formal notation for our categories uses something called "Big-O" notation. For example, we

have the category $O(N)$ and the category $O(N \log N)$. A good start to the categories is the following:

$$
\begin{aligned}
O(1) \quad &< \quad O(\log N) < O(N) < O(N \log N) \\
&< \quad O(N^2) < O(N^3) < \ldots < O(N^k) \\
&< \quad O(2^N) < O(N!) < O(N^N)
\end{aligned}
$$

We're using the symbol $<$ to mean "increases more slowly than" in the ordering. The order encodes mathematical facts like this: the function $y = x^2$ increases faster than $y = \log_2 x$. We'll sometimes say "$O(N)$ is lower order than $O(N^2)$," which simply means that $O(N)$ appears to the left of $O(N^2)$. The ranking above uses k to represent any constant expression relative to N. Finally, don't be mislead by the fact that the categories appear on several lines; the whole system doesn't fit on one line!

Since we typically prefer the most efficient algorithms, we want to use ones that fall into categories as far left on the spectrum as possible. The closer to O(1), the more efficient the algorithm. The ordering we provided is not complete though. We left out O($N^2 \log N$), for example, which comes between O(N^2) and O(N^3). You can put any function of N into this list at the right place.

15.1.4   Abstraction #4: Worst case behaviour

When assessing the cost of an algorithm, there are algorithms that always do the same amount of work, no matter what. For example, taking an average: no matter what, taking an average requires looking at all the data. But there are algorithms whose costs depend on the size, but also the arrangement of the data, or some other factor that does not depend on size.

For example, if we are using linear search to check if a value appears in a list, the program will take fewer steps if the value is near the front of the list, and a lot more if the value is near the end of the list. If the value is the first item in the list, the program has to do very little work. If the value does not appear in the list at all, or if the value is very last in the list, the program has to look at all the values in the list.

We say that the worst case for an algorithm is an arrangement of data that forces the algorithm to do the maximum amount of work for a given problem size. Likewise, the best case allows the algorithm to do the minimum amount of work for a given problem size. Be careful not to misunderstand the role of problem size in best and worst cases. Problem size is independent of best and worst cases. To say that the best case for linear search is when the list is empty is completely wrong!

It would be unrealistic to use the best case as being representative of the possible range of costs for linear search. On the other hand, we often use the worst-case as representative of the costs. This identifies a certain amount of pessimism in our analysis, but when there are factors beyond your control, it is better to be prepared for the worst possible outcome, than to plan for only the best outcome. We are often interested in best and worst cases. Sometimes they describe the range of possible behaviours. Sometimes, and especially in asymptotic analysis, we find the best and worst cases fall into the same category, which reassures us that the categorization is informative.

One naturally thinks of using an "average case" analysis to make a reasonable compromise between the best and worst cases. When this is actually possible, this is a good idea. However, average case analyses are notoriously difficult, because of the complexity of the things we ask a computer to do.

15.2 Asymptotic analysis

To put algorithms into categories, we'll be applying a principle called "asymptotic analysis." This means that we'll infer what happens as the problem size gets really big. It will allow us to ignore any effect that is relatively small.

For example, in the previous example with Algorithms A and B, we know that they are different by one step, no matter what N is, but that one step becomes less and less significant as N increases. Asymptotic analysis allows us to put both algorithms into the same category, namely $O(N)$. We say "Asymptotically, Algorithm A and Algorithm B have the same runtime costs." While the term "asymptotic analysis" may sound deep and imposing, applying it boils down to eliminating details that can be ignored; in other words, it's just abstraction with a fancy name! The name serves to guide us in what details can be thrown away.

It's important to clarify what the asymptotic categorization means. Each category is labelled by a function, like N^2. If we take any algorithm in the $O(N^2)$ category, and plot a curve of the number of steps it uses as a function of N, the graph would never increase faster than the N^2 shape, especially when you get to very large values of N. More precisely, for any algorithm A in $O(N^2)$, we could always find a positive number, k, so that the graph of kN^2 is higher then the graph of the costs of A. An algorithm in $O(N^2)$ cannot be pushed into another category by increasing or decreasing the costs by a factor of 2 or 10 or 10 million. An algorithm cannot be pushed into another category by adding or subtracting 10 steps or 10 million steps. It's the basic shape of the function that labels the category that is important.

Technically, there is a formal mathematical meaning to our categorization system, which describes the categories as upper bounds on costs. These technical details are important, but we will leave the technical presentation for future courses.

15.2.1 Basics of Working with Big-O

Remember our approach is to count steps that a computer would perform carrying out an algorithm. We'll use parameterization to express the number of steps in terms of the problem size. We might end up with an expression like $3N^2 + 4N + 18$, which describes the number of steps carried out when the problem size is N. We'll talk about how to count steps in the next section.

There are a number of simple rules that you can use to perform asymptotic analysis using Big-O notation. These rules are all derivable from first principles, but since I haven't shown you those, we'll simply state the rules.

> **Principle 15.1** In asymptotic analysis, the following identities hold for any expressions A and B:
>
> $$\begin{aligned} O(A) + O(B) &= O(A+B) \\ O(A) \times O(B) &= O(A \times B) \\ A \times O(B) &= O(A \times B) \end{aligned}$$
>
> ∎

For example, we can use the first identity to simplify an expression as follows. Starting with the previously mentioned example:

$$O(3N^2 + 4N + 18) = O(3N^2) + O(4N) + O(18)$$

> **Principle 15.2** In asymptotic analysis:
>
> - If A is a constant, $O(A) = O(1)$.
> - If A is a constant, and B is any expression, $O(A \times B) = O(B)$.
>
> ∎

Continuing the example, the three terms we have now are $O(3N^2)$, $O(4N)$, $O(18)$. The constant factors are 3, 4, and 18. If we apply the rule, we get the following:

$$O(3N^2) + O(4N) + O(18) = O(N^2) + O(N) + O(1)$$

This rule does not apply to exponents, which are constants, but not coefficients.

> **Principle 15.3** In asymptotic analysis, if $O(A) < O(B)$, then $O(B) + O(A) = O(B)$ ∎

According to our hierarchy, we know that $O(1)$ and $O(N)$ are lower order than $O(N^2)$, so they can be dropped. We keep the highest order term only. Therefore, we can write:

$$O(N^2) + O(N) + O(1) = O(N^2)$$

Thus, asymptotic analysis tells us that $3N^2 + 4N + 18$ is $O(N^2)$.

We applied the rules in a specific order, but it turns out that the order you use doesn't matter. We could have dropped lower order terms first, then replaced the constant factors.

15.2.2 Counting Steps in Algorithms

At last, we get to the business of the actual analysis, after the long prelude of abstractions. We'll start with a definition of the concept of a computational step. A single step is any computational action that cannot be broken down into other steps; they are the smallest computational actions that we can count by looking at the code. Some authors call these *primitive operations*.

> **Definition 15.1 Step**
> Any one of the following can be considered as a single step:
>
> - A single arithmetic operation ($\times, +, -, /, //, \%$) provided the numbers are not unreasonably large
> - A single Boolean operation (`and`, `or`, `not`)
> - Assigning a value to a variable
> - Comparing two values using a relational operator such as ==, <, etc.
> - Indexing a list element
> - Accessing a dictionary element by its key.
> - A function call (the cost of executing the function is separate)
> - Returning a value from a function

Every simple statement in Python (or any other language for that matter) consists of a number of steps. For example, consider the following statements:

```
a = 3*y+4
b = data[2]*data[3]
c = 3*math.sqrt(data[i+1])
```

The first statement has 3 steps; the second has 4 steps, and the third statement requires 4 steps, not counting the number of steps that `sqrt` does. The whole sequence of statements above requires 11 steps (not counting `sqrt`).

Time for another principle stating the obvious:

> **Principle 15.4   Sequence**
> In asymptotic analysis, the total number of steps for a sequence of statements is the sum of the steps for each individual statement. ∎

For a conditional statement, it would be a mistake to add the costs of all branches together to get a "total".

> **Principle 15.5   Conditionals**
> In asymptotic analysis, analyze each block in a conditional independently. Add the cost of the conditions to each relevant branch. ∎

The best-case uses the branch with the smallest total cost; the worst case uses the branch with the largest total cost. If the conditional has no else block, the best case is an empty block that has zero cost (which is the smallest any cost can get).

The principle of sequential statements can be generalized to a statement about loops:

> **Principle 15.6   Loops**
> In asymptotic analysis, the total cost of a loop is obtained by adding up the costs from every repetition, and including costs involved with controlling the loop. ∎

There is an easy case, and a more complicated case. If the cost of the body of the loop never changes over the repetitions, you can compute the total by multiplication: the number of times the loop body is executed, and the cost of executing the loop body once.

Here's an example of the easy case. Suppose you eat three apples per day for N days. The number of apples you eat per day doesn't change over time. So three apples per day times N days is $3 + 3 + \cdots + 3 = 3(1 + 1 + \cdots + 1) = 3N$ apples. Here, apples represent steps, and the number of days represent repetition.

The more difficult case applies when the cost of the body of the loop changes depending on where you are in the total sequence. If the cost of the body changes, you can't always use the short cut of multiplication. You have to figure out how to calculate the total using addition.

Here's a simple example of the more difficult case. Suppose you eat one apple on the first day, two apples on the second day, three apples on the third day, ..., and N apples on the Nth day. The number of apples eaten on a particular day depends on what day it is.

The total number of apples is $1 + 2 + 3 + 4 + \cdots + N = N(N+1)/2$. This is a formula that we use a lot in computer science; keep it in mind so that when you see it you'll remember it. Eventually, you'll memorize it simply due to repeated use!

Keep in mind that the body of a loop can contain statements, sequences, conditionals and other loops. You have to analyze them separately, first!

> **Principle 15.7   Functions**

> In asymptotic analysis, the cost of executing a function has to be analyzed separately and added
> to the cost of any statement containing a function call.   ∎

Above, we stated that calling a function requires a single step. This step involves sending the data to the function, and starting the function off, but does not include the steps performed by the function itself. The cost of the function itself has to be analyzed separately. That cost is added to the cost of any statement the function call appears in.

15.3 Examples

15.3.1 Step analysis

Consider the following example.

```
1  total = 0
2  for i in range(10):
3      total = total + 1
```

- Line 1: 1 step
- Just line 3, once: 2 steps
- Just line 2, over-all: 10 assignments, 9 additions, so 19 steps
- Line 3 is repeated 10 times, so 30 steps
- Totals: $1 + 19 + 30 = 50$ steps
- Asymptotically, 50 is $O(1)$
- Literally, the problem size here is not a variable.

Consider the following example.

```
1  total = 0
2  for i in range(N):
3      total = total + 1
```

- Line 1: 1 step
- Just line 3, once: 2 steps
- Just line 2, over-all: N assignments, $N-1$ additions, so $2N-1$ steps.
- Line 3 is repeated N times, so $3N$ steps
- Detailed step analysis: $1 + 2N - 1 + 3N = 5N$
- Asymptotic category: $5N$ is $O(N)$.

15.3.2 Method 2: Simplified Steps

Consider the following example.

```
1  total = 0
2  for i in range(10):
3      total = total + 1
```

- Line 1: 1 step, so $O(1)$
- Just line 3, once: 2 steps, so $O(1)$
- Just line 2, over-all: 10 assignments, 9 additions, 19 steps, so $O(1)$

- Line 3 is repeated 10 times, so $10 \times O(1) = O(1)$
- Totals: $O(1) + O(1) + O(1) = O(1 + 1 + 1) = O(1)$
- Literally, the problem size here is not a variable.

Consider the following example.

```
1  total  =  0
2  for  i  in  range(N):
3       total  =  total  +  1
```

- Line 1: 1 step, so $O(1)$
- Just line 3, once: 2 steps, so $O(1)$
- Just line 2, over-all: N assignments, $N - 1$ additions, $2N - 1$ steps, so $O(N)$
- Line 3 is repeated N times, so $N \times O(1) = O(N)$
- Totals: $O(1) + O(N) + O(N) = O(2N + 1) = O(N)$

15.3.3 Logarithmic loops

Consider the following example.

```
1  total  =  0
2  i  =  1
3  while  (i  <  N):
4       total  =  total  +  1
5       i  =  i  *  2
```

Question: How many times does the loop-body execute?
Insight: The values for i are: 1, 2, 4, 8, ...
Answer: We can double i at most $\log_2 N$ times before $i > N$

Lines 1,2,4,5 are all $O(1)$. Lines 4,5 are executed $\log N$ times. The total cost is $O(\log N)$.

15.3.4 Nested loops with independent limits

- Start with the inner-most loop
- Analyze to find the number of steps
- Abstraction: treat the inner-most loop as a statement with the analyzed cost.
- Work outward, one loop at a time.

```
1  total  =  0
2  for  i  in  range(N):
3       for  j  in  range(M):
4            total  =  total  +  1
```

Assume M and N are separate input size parameters. Lines 3-4 are $O(M)$. Repeating the inner loop N times gives a total of $N \times O(M) = O(NM)$.

15.3.5 Nested loops with dependent limits

- Start with the inner-most loop

- Analyze to find the number of steps expressed using dependency
- Abstraction: treat the inner-most loop as a statement with the analyzed cost.
- Sum the total costs of the dependent loop
- Work outward, one loop at a time.

```
1  total = 0
2  for i in range(N):
3       for j in range(i):
4            total = total + 1
```

- In general, lines 3-4 are repeated $O(i)$ times
- Total cost: sum the costs for i from 0 to $N-1$
- Total: $O(0) + O(1) + O(2) + \cdots + O(i) + \cdots + O(N-1)$
  To find an answer for this sum, we need to use the following identity:

$$\sum_{i=1}^{n} i \equiv 1 + 2 + 3 + 4 + \cdots + n = \frac{n(n+1)}{2}$$

  This is the same one that came up in the discussion of Principle 10.7.
- Total: $O(0) + O(1) + O(2) + \cdots + O(N-1) = O(N^2)$

15.4 Warnings and Advice

None of this material is difficult once you get the hang of it. Don't let the novelty get in the way. It's just counting and abstraction.

There are other kinds of costs associated with algorithms which we won't discuss in this chapter. Some algorithms use a lot of memory, so that is a cost. Some applications require network communications, and the cost of communication needs to be considered. Assessing these costs obey the same principles as the costs we cover in this chapter.

The are costs that the approach in this chapter does not apply to. The most obvious of these is the development cost: how much time it takes to bring the algorithm from design (i.e., when you know how the algorithm will work) to implementation (having correct, working code implementing the algorithm). There's also the cost of research: if you have a problem, but no algorithm, you need to come up with one somehow! It's not always easy. There are other costs as well.

In a previous section, we assumed that all steps take the same amount of time, notably, addition and multiplication. For very complicated mathematical algorithms, especially those that might take a very long time to run, scientists sometimes determine costs by counting additions and multiplications separately, precisely because they take different amounts of time. But, that's a pretty special case of analysis, which you can easily learn after you master the basics.

Finally, this chapter is an introduction to analysis of algorithms. This is where we have to stop, but it's not the final word, and there are other forms of analysis that can be applied to answer slightly different kinds of questions.

16 — Recursion

Learning Objectives

After studying this chapter, a student should be able to:

- Derive recursive formulations for algorithms.
- Author, test, and debug recursive Python functions.
- Explain recursion in terms of the delegation metaphor.
- Describe how recursion is implemented in terms of functions calls and stack frames.
- Analyze simple recursive functions for time complexity.
- Analyze simple recursive functions for space complexity, including the use of the system stack.
- Determine whether a recursive or iterative solution is most appropriate for a problem. [Assessment, SDF-AD-6]

16.1 Introduction

Recursion is a form of repetition using function calls instead of loops. The main idea is that a function tells you how to perform a task, and so if you need to perform the task again, you just call the function again. There's a little more to it than that, but that's the basic idea.

Anything that can be done with a loop can also be done with recursion, and vice versa. You may wonder why we bother with recursion, since we already know two kinds of loops. It can be said that a loop is the kind of repetition that derives from a practical engineering design, whereas recursion is the mathematical basis for all repetition. Some algorithms can be written very elegantly and efficiently using recursion, but would be significantly more difficult to write with a loop. There are programming languages, such as LISP, Scheme, Haskell, Prolog, whose entire repertoire of repetition rests on the foundation of recursion. Recursion is not some mathematical curiosity; it's a fundamental aspect of computation.

Throughout this chapter, we will refer to an idea of computing or performing a *task*. A recursive function, like any function, is best designed to do one thing well. In a very real sense, recursive functions are highly focussed on a narrowly defined task. Our examples will start with tasks such as summing a series, or calculating a complex formula. We will talk about performing tasks of different sizes, by which we mean that the task depends on a function parameter that dictates how much work to do. We will sometimes use the words *main task* to refer to the general case of the task we are trying to solve, and the word *subtask* to refer to smaller instance or version of the main task.

16.1.1 Terminology

All recursive functions have the same general structure. At first glance, they are functions containing a conditional (if-else statement, or a variation).

- At least one of the branches of the if-statement gives a simple answer to a very simple task. This is called a *base case*. In general, if you are designing a recursive function to perform some task involving repetition, then the base case is the result of the most basic variation of the task you can imagine.
- The other branch (or branches) transforms the current task into one or more sub-tasks, and then combines the results of the subtask(s) to form the result of the main task. This is called the *recursive case*. The recursive case always makes a function call (usually one, but sometimes more) to the function being defined.

Let's look at an example. Here, we are adding up all the squares of integers from 0 to N, that is, we are calculating $0^2 + 1^2 + 2^2 + 3^2 + \cdots + N^2$:

```
def sumSquares(N):
    if (N <= 0): #  base case
        return 0
    else: #  recursive case
        return (N*N) + sumSquares(N-1)
```

The base case here is characterized by $N = 0$. When $N = 0$, the sum of squares of integers from 0 to 0 is equal to 0, and that is what the function returns, without doing anything else. The base case for a recursive function is almost always so simple that it requires very little problem solving. Usually, common sense tells us the base case without much thought. When writing your own recursive functions, you might be concerned that the base case seems too easy. Don't be. It usually is that easy. We usually write a conditional that checks for the base case explicitly, as above, but as you know, an if-statement can always be reformulated so that one or the other branches is implicit.[1]

The recursive case is characterized by $N > 0$. Here, the recursive case is found in the *else* part of the if-statement. It tells us that we can calculate the required sum by

1. Obtaining the sum of squares from 0 to $N - 1$ using a recursive call
2. Adding N^2 to the sum computed recursively.

In this function, the main task (computing the sum up to N) is solved by first solving a smaller subtask (computing the sum up to $N - 1$) using recursion, and the result is combined using addition.

---

[1]In our base case, we also included $N < 0$, which is a practical design, in case someone calls the function with a negative integer. It gives the *wrong answer* for negative input (because there is no right answer for negatives, but the function returns 0).

For many of the simplest kinds of tasks, there is one base case, and one recursive case. For more complicated kinds of tasks, there may be more than one base case, or more than one recursive case. You really need to understand the problem very well to decide if you need multiple base cases or recursive cases. The number of these cases is determined by the problem you are solving, not the recursive function you are designing.

16.1.2 The Delegation Metaphor

It is helpful to use a metaphor to help reduce confusion. Think of a function call as delegation. That is, no matter what function is involved, a function call is like calling in an assistant, giving him a task, and some room to work, and waiting for him to come back with a result (or the task complete).

With this metaphor, it doesn't matter if you give your assistant the instructions for the same function you are working on, or if you give your assistant instructions for a different function. The assistant takes only the data you give him, and works completely independently. He may create variables with the same names as ones you created, but they are different variables. He may also call an assistant to help with a subtask.

The delegation metaphor is not just a story that sounds nice and easy. It actually reflects the behaviour of the computer when a function is involved, recursive or otherwise. But we'll have more to say about that towards the end of the chapter.

16.1.3 Avoid the rabbit hole

A recursive function need not be confusing. Lots of hay is made about recursion as self-reference, but ignore it if you can. Students are often fooled into believing that they have to understand the whole sequence of function calls that lead to the correct answer. Students whose grasp on the concept of recursion is not solid often refer to the "winding" and "unwinding" of recursion. They think of recursion as doing something "all the way down" and then doing something different "all the way back up." I call this the rabbit hole, and it's not helpful.

Recursion can be understood as a relationship, and humans are good at seeking and discovering these kinds of relationships. For recursion, there are two relationships you need to grasp.

1. The base case. It's typically easy to grasp, and therefore not really worthy of long discussion.
2. For the recursive case, all you need to think about is the relationship between the main task and the immediate subtask. You need to be able to say how the parameters for the main task are related to the parameters for the immediate subtask; this will involve a computation or calculation of some sort. You also need to be able to say how a solution to the subtask is related to the solution of the main task.

That's it. It's not confusing or mysterious. That is not to say that it is easy to change your mind. If all you practice is loops, convincing yourself that recursion is worth learning is the true educational battle.

16.2 How to Design a Recursive Function

You can always get a start on defining a recursive program by typing the following template:

```
def <function name>(<data>)
    if <base case test>:
        <base case task>
```

```
  else:
    Calculate <new params> for smaller, related problem
    <subSolution> = Call <name> recursively on <new params>
    <solution> = Combine <subSolution> with <data>
    return <solution>
```

Then you fill in the blanks. Not all at once, and maybe you will make some revisions as you go. And there will be simplifications you can make when you're done.

Regardless, the design of the recursive function requires the following steps. You don't have to derive them in this order, but you do have to come up with something for all of them:

1. Give your function a good name. Not `flower` or `Yoda` or `function`. The name should reflect the computation it's supposed to do. This step helps you focus. Using a name that does not reflect the purpose of the task makes your design work harder.
2. Decide the input and output for your function (pre-conditions and return value, possibly post-conditions). Your input is the data your function needs to determine what to do, and you need to be able to use this data to distinguish between the base case and the recursive case.
3. Determine the test for the base case. It is very common to have tests like `N == 0` or `N == 1`. But be careful! The right base case test comes from the task, not from Python. If you don't understand the task, you might not have the right base case test. You need to be able to identify the very simplest instance(s) of the task that your function will perform.
4. Write the code for the task that deals only with the base case. Once you have identified the base case, the code you need for the base case is usually very simple. If your function returns a value, then you need a return statement here.
5. Write the code for the recursive case. We can break it down into steps here:

   (a) Code that calculates the inputs for the subtask using the input information about the main task.
   (b) Call the function recursively, using the input for the sub-task. The template stores the result in a variable, but that's to be able to make the steps distinct. Simplification can often eliminate this assignment.
   (c) Combine the result of the sub-task with something about the main task. This is very often a simple step, like addition, or sequence. The combination has to be based on knowledge of the task.
   (d) Return the result, if appropriate.

16.2.1 The recursive case

By far, the recursive case is the part that requires the most thought. There are two key notions to designing the recursive case:

1. You need to be able to say how the parameters for the main task are related to the parameters for the immediate subtask; this will involve a computation or calculation of some sort.
2. You also need to be able to say how a solution to the subtask is related to the solution of the main task.

In our first example, suppose we gave a recursive function to calculate the sum $0^2 + 1^2 + 2^2 + 3^2 + \cdots + N^2$. This was our main task. We can do some grouping to form a subtask, namely $0^2 + 1^2 + 2^2 + 3^2 + \cdots + (N-1)^2$. This is a simpler version of the main task. The subtask's parameter

is one less than the main task's. We can give the two sums each a name, $sumsq(N)$ for the main task, and $sumsq(N-1)$ for the subtask. It's almost too easy to relate the two tasks. If we start with the equations for $sum(N)$ and $sum(N-1)$:

$$\begin{aligned} sumsq(N) &= 0^2 + 1^2 + 2^2 + 3^2 + \cdots + N^2 \\ sumsq(N-1) &= 0^2 + 1^2 + 2^2 + 3^2 + \cdots + (N-1)^2 \\ sumsq(N) &= sumsq(N-1) + N^2 \end{aligned}$$

The third line above is the relationship between the two tasks. Notice how the solution to the subtask, namely $sumsq(N-1)$ is combined with the value N^2, which comes from the main task, using addition.

16.3 Examples

Simple Sum

Consider the task of adding up all the integers from 1 to N, where N is any positive number. In other words, we want to calculate $1 + 2 + 3 + \cdots + N$. This could easily be done with a loop, but let's derive a recursive solution. The word derive is chosen carefully. Many people will tell you that writing a program is an art. However, the tools of mathematics are helpful. We literally will derive a function in the same way that you would derive a formula.

The key idea for this problem is to recognize that it does not matter which order you add the numbers; any ordering gives you the same answer as all the other orderings. One of the many possible ways to transform the task is to group all but the last number into a sum, and then add the last value N to it:

$$1 + 2 + 3 + \cdots + N = (1 + 2 + 3 + \cdots + (N-1)) + N$$

In other words, we have broken the task of adding numbers from 1 to N into 2 steps: first add the numbers 1 to $(N-1)$, and then add N to that. Clearly, adding up the numbers from 1 to $(N-1)$ is a smaller task than adding the number from 1 to N. If we had some way to do that calculation, all we'd have to do is add N to the result. Fortunately, the function we are designing is just such a function.

Let's use the notation $sum(N)$ to represent the sum of integers from 1 to N. We have to be careful to use this notation only for $N > 0$, because otherwise it doesn't make sense. With this notation, we can also say $sum(N-1)$ is the sum of integers from 1 to $N-1$ (as long as $N-1 > 0$). By the property we observed above, we can finally say that $sum(N) = sum(N-1) + N$, as long as $N-1 > 0$. From here, it is a short exercise to write a recursive function in Python.

```python
def sum(N):
    if (N == 1):        # base case
        return 1
    else:               # recursive case
        return N + sum(N-1)
```

The base case comes from the knowledge that $sum(1) = 1$; the recursive case comes from the equation $sum(N) = sum(N-1) + N$.

Notice that we were primarily engaged in understanding the task (summing a bunch of integers), and we used a bit of basic math to describe the properties of the task. When we finished, we translated the math into Python. The recursive function, therefore, describes a mathematical truth about the

task. One only has to understand the language of Python and to understand the nature of addition to see that the program is correct.

We can improve the function by considering the base case test. When given a positive integer value for N, the function works just fine. But if we call this function with the argument 0, or a negative value like -5, the base case test will never be true, since the recursive case always subtracts 1 from any N. This is known as infinite recursion. To avoid this, we can change the base case test to $N \leq 1$. This prevents infinite recursion, but also gives a wrong answer; there is no well-defined answer to the mathematical problem $sum(-5)$, and any answer a program will give is wrong. But a wrong answer is preferable to an infinite loop.

Even or Not?

Here's a slightly different example. We can write a recursive function to determine whether a given positive integer is even or not. There are better ways to do this task, but it's an interesting example for other reasons.

Suppose we are given a positive integer X, and suppose for the sake of example that $X > 2$ (we'll ignore negative numbers). There is a property of even numbers that is extremely useful: if X is an even number, then so is $X - 2$; likewise, if X is not even, then $X - 2$ is also not even. In other words, we have identified a relationship between 2 numbers X and $X - 2$: they are either both even, or both odd.

We can make this relationship work for us in the form of a recursive function. Our function will return the Boolean value true if a given X is even, and false if X is odd. We also know that two is an even number, but one is not even:

```python
def is_even(N):
    if N == 0:
        return True
    elif N == 1:
        return False
    else:
        return is_even(N-2)
```

This example has two base cases and one recursive case. Notice that X is the input to the function, and that the recursive step transforms the task into a subtask about the value $X - 2$. The recursive case decides whether $X - 2$ is even or not, and there is no combination here because the answer for $X - 2$ is the same as the answer for X.

Again, we motivated the function by explaining a relationship between X and $X - 2$. The function describes this relationship in Python. It is a matter of understanding something about numbers, and a little Python to see that the program is correct.

16.4 Functions, recursion, and the call stack

Every programming system that allows the use of functions and function calls does so by means of a special stack that keeps track of details like parameters, arguments, and return values. This stack has many names: the call stack, or the system stack. The stack is allocated to a running program by the computer's operating system, and has a generous amount of room for the complexity of modern software applications. However, an important aspect of the call stack is that it is finite in size, and cannot be extended. The call stack is unlike any of the stacks we have seen in this course for that

reason: we have never encountered a stack that we could not extend, either by making the underlying list a little bigger, or by adding a new node data structure.

The finite size of the call stack implies that the computer could, if too many functions were called, run out of space to use for calling more functions. In some applications, this might be catastrophic for the running program, and data could be lost because of it. Running out of space on the call stack is known as *stack overflow*. Yes, the website is named for a catastrophic event that every developer should attempt to avoid!

In Python, the system stack is carefully monitored, so if the stack gets full, the running program is halted. This is still catastrophic for the program, but not for Python running the program. Python can report that the stack overflowed, and then it can (by default) display an outline of the entire call stack, to help the programmer figure out what happened. In fact, every time one of your programs has a run-time error, Python displays an outline of the call stack.

As we know, a stack is useful for LIFO behaviour, and data gets pushed onto the stack, and popped off the stack. The call stack represents all the functions that have been called since the start of the program. Every time a function is called, some data (called a frame), which we will describe later) is pushed on the call stack. While the function is active, the frame remains on the stack. When the function reaches the return statement, the frame is popped from the stack. The LIFO protocol is crucial here. If function A calls function B, then function B's frame is added after the frame for A was added. Function B must return before function A can continue, so B's frame is popped from the stack, allowing function A to carry on.

A frame, also called a stack frame or activation record, contains all the information the function needs to run. In particular, the frame contains the variables and parameters created by the function call, along with their values. Along with this information, a stack frame also contains information about how to use the return value that the function computes. A function call can appear in any expression, so the stack frame also has to contain information about what was going on when the function was called, so that the computer can go back and carry on immediately after returning. This is usually in the form of an address of the next machine instruction to perform immediately after returning.

The above description should remind you of the delegation metaphor. The delegate's instructions consist of the code in the function. The office that the delegate works in is the stack frame; the building that the office is in is the call stack. The only thing that's not realistic about the delegation metaphor is that no delegates are "called in" to help. Instead, the computer acts as its own delegate, preparing the frame, going off to do the work, and then coming back. It's as if the CEO of the company does all of the work for every delegation. To manage this, the CEO would have to continually make notes to herself, keeping track of what she was doing before going off to the next office for the next part of the job. These notes are part of the information stored in the stack frame.

Functions only work because of the stack. Without a stack, there would be no functions. Recursive functions are not in any way special. They use the stack frame as any other function would. The only difference is that the stack might contain a tower of frames from the same function, each with a slightly different set of variables, and other data. This tower is the "rabbit hole" I mentioned earlier. It exists, and we need to be aware of it, but we should not feel the need to use this information in designing recursive functions. The call stack enables the implementation of recursive functions; it does not enable the design of recursive functions.

16.5 Algorithm analysis for recursive functions

Analyzing recursive functions to determine the worst-case time complexity in terms of Big-O notation is really not much different from analyzing a loop. Since repetition is being facilitated by function calls, we need to be able to count how many times the recursive function will be called for a given input size. We also need to understand the cost of the body of the function (ignoring the cost of the recursive call).

The combination of these two ideas can be treated very informally. Consider the *sum*(N) function from earlier. We know that the recursive step decreases N by 1, until the base case $N = 1$. In other words, the function is called $O(N)$ times recursively, total. The body of the function, is basically $O(1)$, except for the cost of the recursive function call. So it's pretty clear that the *sum*(N) function has worst-case time complexity of $O(N)$.

There is a more rigorous way to analyze the costs. It is possible to formulate the costs precisely as a recursive relationship of costs. And using math that we will not cover in this course, it is possible to establish mathematically the runtime costs. We teach the math needed to do these formal analyses in second year, and these analyses apply not only to situations that seem rather obvious like *sum*(N), but also to much more complicated algorithms where the math is actually the only way to get an answer at all.

Earlier we mentioned the fact that a recursive function call builds a tower of stack frames on the call stack. There is one frame for every function call, and in the case of *sum*(N), there are N stack frames in the tower. Thus we can see that as well as having worst-case time complexity of $O(N)$, the function also uses $O(N)$ frames on the call stack as well. This is the first time that we've talked about the cost of memory in our analyses. In particular, using a loop does not have this kind of memory use: a loop will access variables that are defined in the stack frame, but the loop itself does not create a stack frame every time it is repeated. In other words, a loop does not have a space cost the same way that a function call does.

In many situations, the fact that recursion consumes stack space, whereas a loop does not, is enough of an argument for preferring loops over recursion. It's a good reason, and in normal situations, if you can use a loop instead of recursion, you should. However, there are situations in which we are willing to pay the cost of recursion. We haven't seen any yet, so you'll just have to wait to see them.

For most computational devices, and most programming languages, a few recursive functions will not be greatly detrimental to your application. You should not be afraid to use recursion, especially if a recursive program seems appropriate. Modern computers are generously equipped with memory, though perhaps smartphones and other embedded devices are not so richly endowed as of this writing.

It must also be said that there is a technique, called *tail-call optimization* which can be applied in some programming languages, that does not incur a new stack frame for every recursive call; this technique reuses the stack frame from the previous call of the recursive function. Python does not use this technique, so in Python, recursion always creates new stack frames. But other languages use tail-call optimization, and so can avoid the costs Python pays (along with many other languages like Java or C/C++). The point here is that it is false to conclude that recursion is never a good choice. The value of recursion depends on context.

17 — Trees and Binary Trees

Learning Objectives

After studying this chapter, a student should be able to:

- Describe what a tree is, in terms of its basic organizational principles.
- Explain the terms: node, child, parent, ancestor, descendant, sibling.
- Describe what a binary tree is, in terms of its basic organizational principle.
- Describe basic operations of binary tree ADT.

17.1   Trees organize data hierarchically

In computer science, a tree is way to organize data in a hierarchical manner. This is in contrast to a list, which organizes data in a sequential manner. Hierarchical organizations are very common. Companies, governments, and institutions all use hierarchies to organize their activities.

Hierarchies are also used to give structure to information. For example, a textbook consists of a number of chapters, and each chapter consists of a number of sections. Sections contain paragraphs, and paragraphs contain sentences. A table of contents often reflects this organization, from the chapter level down to the section level.

The file system on your computer is organized hierarchically. Folders contain sub-folders. In a UNIX system, there is a special folder, call root, that contains everything stored on the disk. It is not a coincidence that it is called root; this folder is the root of the tree that contains every folder and every document on your computer's disk.

If a computer application has a graphical "drop-down" menu, you will be familiar with menu items that expand into sub-menus. This is hierarchical as well.

Trees are useful in computer science because they can be used to organize data in ways that no other organization could provide, or because using them allows us to write algorithms that are more efficient. We will see examples of both.

Figure 17.1: A tree with 14 nodes, 7 leaf nodes, and height 4.
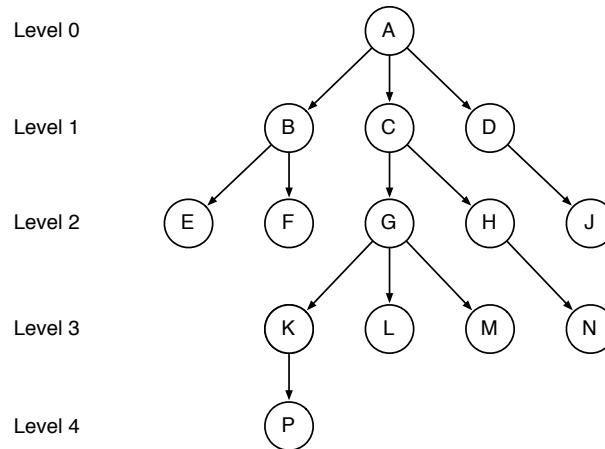
17.2   Informal Terminology

A tree is a collection of data. Each data value is stored in a *node*; this is not a coincidence, as we will define a *tree-node* ADT a little later. In computer science, a tree has a single root. Unlike the root of trees that occur in nature, whose roots have a complicated branching structure, our root is a single node, and in diagrams, we always put the root at the top.

A tree has *branches* connecting nodes. Sometimes the branches are called edges, arcs, or lines. Unlike trees in nature, trees in computer science branch downward; this makes diagrams a bit easier to draw, but also matches the hierarchical organization trees impose on data.

If two nodes are connected by a branch, the node above is called the *parent* of the node below, and the node below is called the *child* of the node above. For example, in Figure 17.1, the node labelled C is a parent of node labelled H; H is a child of C. Sometimes a tree diagram will draw branches with an arrow from parent to child. In general, a parent node may have 1 or more child nodes, however, a child node has exactly one parent. This property is the essence of a tree in computer science. For example, in Figure 17.1, node C has two children, and node H has one child, but no node in the tree has more than one parent.

The only node without any parents is the root; a root is sometimes called an *orphan*, since it has no parent. For example, in Figure 17.1, node A is the root. There may be several nodes that have no children; these nodes are called *leaf* nodes. For example, in Figure 17.1, node F is one of 7 leaf nodes. A leaf is sometimes described as *barren*, since it has no children. If two nodes have the same parent, we call them *siblings*. For example, in Figure 17.1, nodes B, C, and D are siblings. We don't generally extend languages about family relationships much farther than this.

If we start at a node, and follow branches upward in a tree, we find the node's *ancestors*. For example, in Figure 17.1, node G has ancestors G, C, and A. If we start at a node and follow branches downward, we find *descendants*. For example, in Figure 17.1, node G has descendants G, K, L, M, and P. In computer science, a node is usually considered part of its own set of ancestors, and descendants. This doesn't make sense if you think of genealogy, but it makes a kind of mathematical sense: a node is a leaf in its ancestor tree, and a node is the root of its descendant tree.

This last point is worth repeating. Every node can be considered the root of the tree containing

its descendants. As a result, we often call a node a *sub-tree*, even though that is a slight abuse of terminology. A node is not actually a sub-tree, but it is the root of a sub-tree. The tree-ness comes from looking at the way the nodes are organized collectively; the node-ness comes from being connected to other nodes. For example, in Figure 17.1, node G is a node in the tree, but also the root of a sub-tree containing its descendants G, K, L, M, and P.

In computer science, we talk about empty things like empty lists. An empty tree is simply a tree with no nodes. The diagram of an empty tree looks like a drawing of nothing; if you look carefully enough, you will find empty trees drawn everywhere. A tree with exactly one node in it has a root and a leaf, the same node.

17.3   Technical definitions

The previous section established intuitions, but it's time to start being precise. The first thing is a precise, and recursive definition for the term *tree*.

> **Definition 17.1**   A tree is defined as follows:
>
> 1. A structure with exactly zero nodes is an *empty tree*.
> 2. A structure with exactly one node is a *tree*.
> 3. If $t_1, \ldots, t_k$ are non-empty trees, then the structure, s, whose children are the roots of $t_1, \ldots, t_k$ is a *tree*.

The definition is recursive, and this foreshadows the fact that most algorithms that make use of trees will be recursive. In fact, trees are the first real reason we need to learn recursion; most of the examples of recursion we have seen could have been accomplished with a loop. Not so with trees! Most of the computation we will do on trees will not only use recursion, but would be exceedingly difficult to do without recursion. The definition not only defines trees precisely, but it gives the basic organization of every algorithm for trees: the base cases, and the recursive case.

A *path* is a sequence of nodes connected by branches, with no repeated branches allowed. For example, in Figure 17.1, the path between node F and node J goes through nodes F, B, A, D, and J. The *length* of the path is the number of branches in the path. For example, in Figure 17.1, the path between node F and node J has 4 branches. The number of nodes in the path is always one more than the length. There is exactly one path from any node to any other node in the tree. In particular, there is one path from the root to any other node; this path always moves down the tree.

The term *level* describes where a node is in terms of the length of the path from the root to the node. The root node is at level 0, since the path from the root to the root has zero branches. The root's immediate children (if there are any), are at level 1. We usually draw tree diagrams so that all the nodes at the same level are in a row in the diagram. For example, in Figure 17.1, node F is at level 2, and node P is at level 4.

A tree may have many nodes at several levels, but there is always at least one node at the maximum level. For example, in Figure 17.1, node P is at the maximum level 4. The *height* of a tree is always one more than the maximum level of any node in the tree. Another way to put it is that level counts branches, and height counts nodes; The height of a tree is the number of nodes on the longest path from the root to any leaf node in the tree. The height of an empty tree is defined to be zero. For example, in Figure 17.1, the tree's height is 5.

The *ancestors* of a node are nodes on the path from the node to the root. In computer science, we usually say that the ancestors of a node form a *set*, which gives us a mathematical language to

Figure 17.2: A binary tree with 11 nodes, 5 leaf nodes, and height 5.
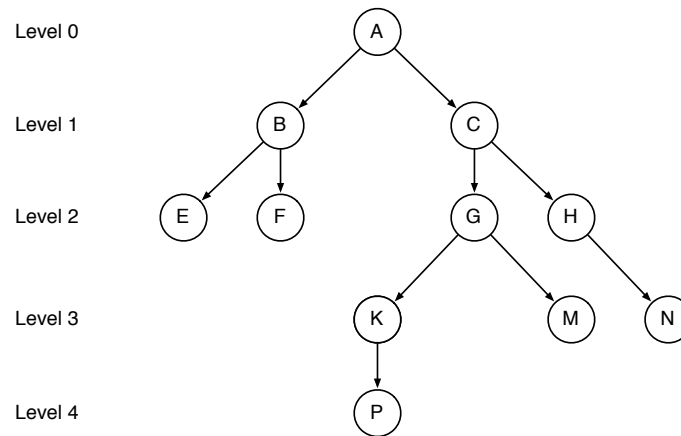
talk about elements and membership, and other useful concepts. So, according to the definition, a node is an element in its own ancestor set. A node *u* is the *descendant* of a node *v* if *v* is an ancestor of *u*. So a node s an element in its own descendant set. It may seem strange to include a node as one of its own ancestors, but it simplifies the definitions a lot.

A *sub-tree* is a tree in every sense of the above definition, but it may be part of a larger tree. If *u* is a node in a tree, the sub-tree rooted at *u* consists of *u* and all of its descendants. Sometimes we abuse terminology a little and call *u* a subtree, even though we know it's a node; the phrase "subtree rooted at" always adds syllables to a sentence but does not always increase clarity.

17.4 Binary trees

A *binary* tree is a special kind of tree with no node having more than 2 children. In part 3 of Definition 17.1, a tree *s* can have no more than *k* children, and for a binary tree, $k = 2$. For example, Figure 17.2 shows a binary tree. Since there are at most 2 children, we can label them *left* and *right*. Sometimes we say *left sub-tree*, or *left node*. In a diagram, we do put the left node to the left of the right node. Either or both of the subtrees could be empty.

A *complete* binary tree is a binary tree that has exactly two children for every node, except for nodes at the maximum level, where the nodes are barren. For example, Figure 17.3 shows a complete binary tree. An empty tree is not complete, because it has no nodes, and therefore no barren nodes. A binary tree with exactly one node (both a leaf and a root) is complete. A complete binary tree has 2^l nodes at level *l*. Go ahead, count them: in Figure 17.3, there are 4 nodes at level 2. One of the most important aspects of a tree is the relationship between the number of nodes in a tree, and the tree's height. In total, a complete binary tree whose height is *h* contains $2^h - 1$ nodes. In Figure 17.3, the height is 4, and there are 15 nodes total. This fact implies that a complete binary tree with *n* nodes has height $\log_2(n+1)$. These facts are going to be useful for us later.

Figure 17.3: A complete binary tree with 15 nodes, 8 leaf nodes, and height 3.
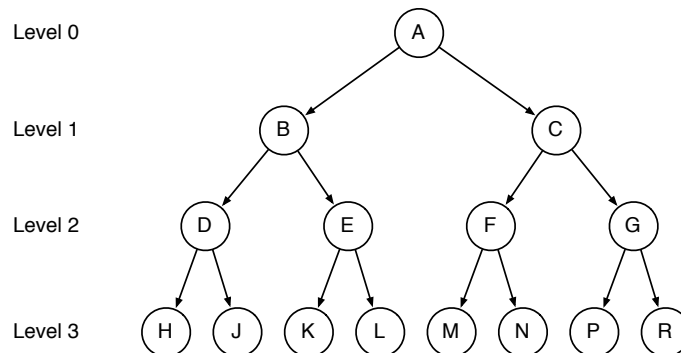
17.5 Binary Treenode ADT

In Chapter 13 we introduced the Node ADT, and we've used it a lot. Perhaps you were skeptical of the utility of nodes, and perhaps you were dubious of the claim that working with nodes was good practice. But we shall see that our work with nodes gives us good background to introduce the ADT that will allow us to create trees.

A `treenode` is a simple record, implemented in Python as a dictionary with three fields:

data A data value
left A `treenode` (or the value `None`)
right A `treenode` (or the value `None`)

The primary operations provided for the `treenode` ADTs are as follows:

- `create(data, left=None, right=None)` creates a `treenode` to contain the data value and the given left and right values. If left and right are not given, the vlaue `None` is used by default.
- `get_data(treenode)` returns the contents of the data field of the given `treenode`
- `get_left(treenode)` returns the contents of the left field of the given `treenode`
- `get_right(treenode)` returns the contents of the right field of the given `treenode`
- `set_data(treenode, v)` given a `treenode`, sets the contents of the data field to v
- `set_left(treenode, n)` given a `treenode`, sets the contents of the left field to n
- `set_right(treenode, n)` given a `treenode`, sets the contents of the right field to n

A `treenode` is very similar to our nodes from Chapter 13. A node has just one possible reference to another node. A `treenode` has two possible links to other `treenodes`. The definition of the operations is absolutely predictable, given our prior work with nodes. You can find the implementation in the Appendix E.

Hopefully you can already predict some of the consequences of the similarity between nodes and treenodes. The first consequence is the fact that a node-chain is a very special kind of tree. We can imagine chain-like trees, where every treenode has at most one child; it has a root, and exactly one leaf. We call this kind of tree *degenerate* because it is not very tree-like. The height of a degenerate tree with n nodes is exactly n. Compare this to a complete binary tree with n nodes, whose height is $\log_2(n+1)$.

Figure 17.4: A binary tree constructed using the treenode ADT.
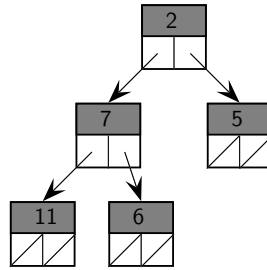
17.6 Trees from treenodes

Just as we did in Chapter 13, we can use the ADT to construct hierarchically organized data structure. We will call this a `primitive binary tree`, though this is not an official computer science term; it is analogous to the term *node-chain*. As we saw in Chapter 8, where we used node-chains to implement Queues and Stacks, we will use primitive binary trees to implement higher-level concepts in later chapters. First, we'll just get the hang of the primitive operations.

Consider the tree in Figure 17.4. There are many ways we could build this tree using the treenode ADT, and here's just one of them:

```
root = treenode.create(2)
a = treenode.create(7)
b = treenode.create(5)
c = treenode.create(11)
d = treenode.create(6)

treenode.set_left(root, a)
treenode.set_right(root, b)
treenode.set_left(a, c)
treenode.set_right(a, d)
```

In the above code, five nodes are created, and at first each node is its own one-node tree. Because each tree is assigned to a Python variable, we can construct the tree by connecting the treenodes by name. This is just one way to construct the tree in the figure. Here is another:

```
root = treenode.create(2,
          treenode.create(7,
              treenode.create(11),
              treenode.create(6)),
          treenode.create(5))
```

In this construction, we're building the tree without naming the subtrees at all. If you read the code carefully, you'll see that the level of the tree is suggested by the indentation.

Perhaps you are thinking that building trees using the treenode ADT is tedious and possibly error prone. You'd be right! We'll only use the ADT to take care of the basics. Just as we implemented

Queues and Stacks using the Node ADT, we'll use the treenode ADT to implement other kinds of hierarchical data structures.

18 — Algorithms on Binary Trees

Learning Objectives

After studying this chapter, a student should be able to:

- Describe pre-order, in-order, and post-order traversals, and demonstrate their behaviour in example binary trees.
- Employ a binary tree as a solution to a software design problem.

18.1 Sequences and Tree Traversals

We emphasized in Chapter 8 that linear data structures like lists, stacks, and queues, impose a linear sequence on the data they store. Unlike linear data structures, a tree does not impose a unique or natural ordering on the data it stores. While we could argue that top-to-bottom, left-to-right is a good sequence for data stored in a tree, it is not the only possibility. There are, in fact, four distinct sequences for the data stored in trees. The first is quite a bit different from the remaining three.

We care about sequences, because we have to write programs for binary trees. Our Python programs are sequential, in the sense that they tell the computer to do one thing at a time. When faced with a problem that requires the use of data stored in binary trees, we have to decide how to handle the fact that a treenode may have 2 children. Fortunately, there are 4 options which will form the basis of many programs that deal with data stored in binary trees. First we'll define the sequences, and the property that characterizes them. After that, we'll show four template functions that demonstrate the sequences. These functions are known as "traversals." Following that, a few examples of programs that use one of the traversals as the basis of the algorithm.

18.1.1 Breadth-first sequence

We already have the notion of treenode levels, as being the distance of a treenode from the root. Informally, the *breadth-first* sequence works top-to-bottom, left-to-right, as mentioned just previously.

Figure 18.1: A binary tree. (a) Breadth-first sequence: A,B,C,D,E,F,G,H,K,M,N,P; (b) Pre-order sequence: A,B,E,F,C,G,K,P,M,H,N; (c) In-order sequence: E,B,F,A,P,K,G,M,C,H,N; (d) Post-order sequence: E,F,B,P,K,M,G,N,H,C,A;
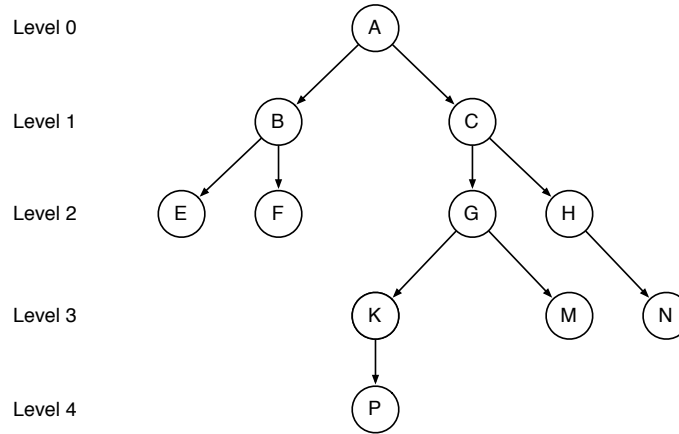
Formally, we say say that the breadth-first sequence arranges the treenodes so that all the nodes at level i appear in the sequence after the nodes at level $i-1$ and before the nodes at level $i+1$.

For example, given the binary tree in Figure 18.1, the breadth-first sequence of nodes is as follows: A,B,C,D,E,F,G,H,K,M,N,P.

18.1.2   Pre-order sequence

The next three sequences are all similar, in that they refer to treenodes and their subtrees.

The pre-order sequence organizes the nodes so that the data at the root appears before the data at its subtrees. For example, given the binary tree in Figure 18.1, the pre–order sequence of nodes is as follows: A,B,E,F,C,G,K,P,M,H,N. To make sense of this organization, choose any node in the tree, say the node with data value C. It is in the subtree of A, so it appears after A in the pre-order sequence. But it is the root of its own subtree, so in the pre-order sequence, it appears before every node below it in the tree. It is before G and H, as well as P and M, for example.

18.1.3   In-order sequence

The in-order sequence organizes the nodes so that the data at the root appears after the data at its left subtree, but before the data in its right subtree. For example, given the binary tree in Figure 18.1, the in-order sequence of nodes is as follows: E,B,F,A,P,K,G,M,C,H,N. To make sense of this organization, choose any node in the tree, say the node with data value C. It is in the right subtree of A, so it appears after A in the in-order sequence. But it is the root of its own subtree, so in the in-order sequence, it appears after every node below it in its left subtree, e.g., C appears after K. In the in-order sequence it appears before anything in its right subtree, e.g., N.

18.1.4   Post-order sequence

The post-order sequence organizes the nodes so that the data at the root appears after the data at its subtrees. For example, given the binary tree in Figure 18.1, the post-order sequence of nodes is as

follows: E,F,B,P,K,M,G,N,H,C,A. To make sense of this organization, choose any node in the tree, say the node with data value C. It is in the subtree of A, so it appears before A in the post-order sequence. But it is the root of its own subtree, so in the post-order sequence, it appears after every node below it in the tree. It is after G and H, as well as P and M, for example.

Take care not to draw the wrong conclusion about pre-order and post-order sequences. The post-order sequence still obeys the left-to-right convention, and this implies that the post-order is not simply the reverse of the pre-order sequence.

18.2 Tree Traversals

In this section, we'll give simple programs that demonstrate the traversals. Keep in mind that the programs we are showing are not inherently useful for what they do. In fact, the programs we'll see simply display the tree data to the console. Clearly there are more interesting things to do with data than display. However, these programs demonstrate the application of the sequences discussed in the previous section. We'll use the phrase "process the data" to indicate in an abstract way that some calculations or computations should be done, and those will depend on the application. We'll simply display the data, for now. When you need an algorithm to accomplish something in a binary tree, chances are very high that you'll base the algorithm on one of these traversals, and you won't be displaying data, you'll be doing something more interesting.

We'll start with pre-, in-, and post-order traversals. These are simple recursive algorithms, and quite easy to implement. We'll introduce the breadth-first traversal last. While it is the easiest one to describe in words (top-to-bottom, left-to-right), it is the hardest one to write a program for. This fact tells us something about human language, and the value of diagrams.

18.2.1 Pre-order traversal

The pre-order sequence organizes the nodes so that the data at the root appears before the data at its subtrees. Therefore, the pre-order traversal processes the data at the root before processing any data at its subtrees. Taking this idea, we can write a recursive algorithm for a pre-order traversal:

1. Process the root of the subtree.
2. Recursively process the left-subtree in pre-order sequence
3. Recursively process the right-subtree in pre-order sequence

In Python, we'll be a little more careful, by including a base case, as follows:

```python
def pre_order(tnode):
    """
    Process the treenode in pre-order sequence.
    """
    if tnode is None:
        return
    else:
        print(treenode.get_data(tnode), end=" ")
        pre_order(treenode.get_left(tnode))
        pre_order(treenode.get_right(tnode))
```

Here, the processing happens at line 8, which happens before the two recursive calls.

On tests and exams and job-interviews, you might be called upon to demonstrate your knowledge of the pre-order sequence without the aid of a computer. In this case, it pays to have a strategy. Use this one for guaranteed success:

1. Write down the root.
2. Draw a box for left and right subtrees.
3. Fill in each box recursively: left, then right.

18.2.2   In-order traversal

The in-order sequence organizes the nodes so that the data at the root appears after the data at its left subtree, but before the data in its right subtree. Therefore, the in-order traversal processes the data at the root after it processes the left subtree, but before processing the data at its right subtree. Taking this idea, we can write a recursive algorithm for an in-order traversal:

1. Recursively process the left-subtree in in-order sequence
2. Process the root of the subtree.
3. Recursively process the right-subtree in in-order sequence

In Python, we'll be a little more careful, by including a base case, as follows:

```
 1  def in_order(tnode):
 2      """
 3      Process the treenode in in-order sequence.
 4      """
 5      if tnode is None:
 6          return
 7      else:
 8          in_order(treenode.get_left(tnode))
 9          print(treenode.get_data(tnode), end=" ")
10          in_order(treenode.get_right(tnode))
```

Here, the processing happens at line 9, which happens after finishing the left subtree, and before starting the right subtree.

On tests and exams and job-interviews, you might be called upon to demonstrate your knowledge of the in-order sequence without the aid of a computer. In this case, it pays to have a strategy. Use this one for guaranteed success:

1. Draw a box for left and right subtrees.
2. Write down the root between the two boxes.
3. Fill in each box recursively: left, then right.

18.2.3   Post-order traversal

The post-order sequence organizes the nodes so that the data at the root appears after the data at its subtrees. Therefore, the post-order traversal processes the data at the root after processing any data at its subtrees. Taking this idea, we can write a recursive algorithm for a post-order traversal:

1. Recursively process the left-subtree in post-order sequence
2. Recursively process the right-subtree in post-order sequence
3. Process the root of the subtree.

In Python, we'll be a little more careful, by including a base case, as follows:

```python
def post_order(tnode):
    """
    Process the treenode in post-order sequence.
    """
    if tnode is None:
        return
    else:
        post_order(treenode.get_left(tnode))
        post_order(treenode.get_right(tnode))
        print(treenode.get_data(tnode), end=" ")
```

Here, the processing happens at line 10, which happens before the two recursive calls.

On tests and exams and job-interviews, you might be called upon to demonstrate your knowledge of the post-order sequence without the aid of a computer. In this case, it pays to have a strategy. Use this one for guaranteed success:

1. Draw a box for left and right subtrees.
2. Write down the root after the boxes
3. Fill in each box recursively: left, then right.

18.2.4 Breadth-first traversal

The breadth-first sequence arranges the treenodes so that all the nodes at level i appear in the sequence after the nodes at level $i-1$ and before the nodes at level $i+1$. Therefore, the breadth-first traversal processes the all the data (from left to right) at level i before it processes any data at level $i+1$. Unlike the previous traversals, the breadth-first traversal is not recursive.

The key to the breadth-first traversal is to understand that if a treenode in the tree is at level i, its children (if there are any) are at level $i+1$. We have to schedule the processing of a node's level $i+1$ children after all the other nodes at level i are processed. Fortunately, we have a nice tool to do this kind of scheduling: a FIFO queue. To see how this would work, let's assume that all the treenodes at level i are already in the queue. If we look at any level i node already in the queue, we can enqueue its children, and by the FIFO protocol, the children are added after everything currently in the queue. In other words, the children at level i are guaranteed to appear after all the level i treenodes.

For example, consider the tree in Figure 18.1. Suppose that the treenodes B, C are in a FIFO queue. If we enqueue B's children, the treenodes E and F will be in the queue after C.

The breadth first traversal uses a FIFO queue to store treenodes, starting with the root. The algorithm dequeues a treenode, processes it, and then enqueues its children, ignoring empty trees. The complete algorithm is given below:

```python
def breadth_first_order(tnode):
    explore = Queue.create()
    Queue.enqueue(explore, tnode)

    while size(explore) > 0:
        current = Queue.dequeue(explore)
        print(tn.get_data(current), end=" ")
```

```
8          left = tn.get_left(current)
9          if left is not None:
10             Queue.enqueue(explore, left)
11         right = tn.get_right(current)
12         if right is not None:
13             Queue.enqueue(explore, right)
```

18.3   Algorithms based on Tree Traversals

The functions given in the previous section are perhaps useful for displaying the contents of a binary tree, during debugging, for example, but not for much else. However, traversals are commonly the foundation of other algorithms that do more interesting tasks. In this section, we'll see some examples. You'll notice a few very important common features of these algorithms.

The recursive definition of a tree, given in the previous chapter, provides guidance in the design of algorithms for trees. They'll almost always be recursive, and even if recursion is not needed, recursion will always be a good choice Second, the base case(s) will frequently be either an empty tree or a tree with no children (a leaf). Third, the recursive cases can be designed by assuming that the function returns the correct result for subtrees; we'll only have to design the calculations that combine the results from the subtrees.

18.3.1   Calculating the height of a binary tree

The height of a tree is the number of nodes on the longest path from the root to any leaf node in the tree. When we see a tree in a diagram, we can almost always identify the deepest leaf nodes, and so the height of a tree is not very difficult to calculate. On the other hand, when a computer is asked to find the height of a tree, starting at the root, it's not at all obvious where the deepest leaf node is. For that reason, a function to find the height of a tree has to try all possibilities.

The key to designing a recursive function for trees is to assume that the function will work on a treenode's subtrees, and then focus on designing the combination step. Let's assume that our tree t has two subtrees, the left lt and the right, rt. We will assume that our function will correctly calculate the height of lt and rt, leaving us the job of using this information to calculate the height of t. Either the two subtrees have equal height, or one of them will have a greater height. In either case, the height of t involves one more treenode on the path than one either subtree. In Python, we have the following function:

```python
def height(tnode):
    """
    Calculate the height of the tree rooted at tnode.
    """
    if tnode is None:
        return 0
    else:
        lh = height(treenode.get_left(tnode))
        rh = height(treenode.get_right(tnode))
        return 1 + max(lh, rh)
```

The recursive calls for the left and right subtrees precede the calculation of the maximum, so we can view this as being based on a post-order traversal.

18.3.2 Counting the number of nodes in a tree

To count the number of nodes in a tree, we will use the same recursive approach as above. The base case, an empty tree, has 0 nodes in it, by definition. For the recursive case, we observe that we can count the number of nodes in the left subtree recursively, and the number of nodes in the right subtree recursively. The total number of nodes in the tree is one more than the sum. In Python:

```python
def count(tnode):
    """
    Count the number of nodes in the tree.
    """
    if tnode is None:
        return 0
    else:
        return (1 + count(treenode.get_left(tnode))
                  + count(treenode.get_right(tnode)))
```

As written, the `count()` function most closely resembles a post-order traversal, since the additions cannot be performed until the second recursive call returns. But it resembles all the recursive traversals to a significant degree.

18.3.3 Searching for a value in the tree

We use trees to organize data hierarchically. At this point, you may well wonder at the utility of using trees, since we've not seen much that's special about trees so far. It will come, in the next chapter! But for now, let's assume that, like an unordered list, a data value could be anywhere in the a tree. The question is how to go about answering whether or not a given value is in the tree or not.

There are four cases. First, we can check if the value is stored in the root of the given tree. Second, it could be in the left subtree; or it could be in the right subtree, which is the third case. For both of these cases, we will use the power of recursion. Finally, the value we're looking for may not be in the tree at all. This last case is a bit subtle. We can only say for sure that the value is not in the tree if we look at the whole tree, and have no success finding it. Of course, if the tree happens to be an empty tree, it will contain nothing, and in particular it will not contain the value we are looking for. In Python:

```python
def member(tnode, val):
    """
    True if val appears somewhere in the tree;
    False otherwise.
    """
    if tnode == None:
        return False
    else:
        return (treenode.get_data(tnode) == val
                  or member(treenode.get_left(tnode), val)
                  or member(treenode.get_right(tnode), val))
```

Note the use of the Boolean operator `or` in the return statement. It might be tempting to write something like the following:

```python
def member(tnode, val):
    """
    True if val appears somewhere in the tree;
    False otherwise.
    """
    if tnode == None:
        return False
    elif treenode.get_data(tnode) == val:
        return True
    elif member(treenode.get_left(tnode), val):
        return True
    elif member(treenode.get_right(tnode), val):
        return True
    else:
        return False
```

This is a correct function, but it says in eight lines of Python what the previous version says using `or` and 4 lines. Note that most programming languages, Python included, define the `or` operation so that it does not perform unneeded computations. Logically, the expression `True or x` evaluates to `True`, no matter what x is. For this reason, in an expression `y or x`, if y is True, the expression x is never evaluated; it would only be evaluated if y were False.

18.3.4   Generalizations about functions on trees

We can make a few generalizations about the examples we've seen. First, almost all functions that have to do with trees are recursive. They are all more or less similar to the recursive traversals. Second, the structure of recursive functions on trees closely mirrors the formal definition of a tree. This is not an accident. The formal definition of a mathematical object, like a tree, identifies very precisely the salient and valuable features of the object. The strategy for designing a recursive tree function on tree t is to assume that the function will return the correct answer for the subtrees, lt and rt, and to focus on the combination step; taking the information returned by the recursive calls, and using it to formulate the correct answer for t. This is not an accidental strategy. It's based on a fundamental property called mathematical induction.

Introduction
Defining classes and instantiating objects
Simple examples
Encapsulation and access control
Perspective

19 — Objects and Classes

Learning Objectives

After studying this chapter, a student should be able to:

- Explain the differences between Procedural and Object Oriented Programming (OOP).
- Explain the difference between a object and a record.
- Explain the difference between a class and an object.
- Explain what attributes and methods are in terms of object oriented programming.
- Define simple classes, including data and methods, in Python.

19.1 Introduction

The programs and ADTs that we've studied have been based on a programming paradigm known as *procedural programming*. This approach to software emphasizes the technique of using functions (also known as procedures) to encapsulate algorithms. We've seen this approach in the design of our ADTs, which are almost exclusively built using records, which we realize in Python using dictionaries. We use ADTs to encapsulate data, and to provide operations to manipulate the data in legitimate and well-understood ways. Less advanced languages will use less advanced tools for procedural programming. For example, the C Programming Language does not have dictionaries, but uses a tool called a `struct` for its records. However, in the procedural programming paradigm, the data abstraction (dictionaries or structs) is enabled by procedures that operate on the data. Since this approach is the only one we've presented so far, it would be natural to conclude that this is the only way to design software.

One of the benefits of this procedural approach is that it is minimalistic. Almost every programming language has these tools, and so you can apply the design techniques no matter which language you end up being required to use for any given project. Another benefit is pedagogical: we can teach procedural ADT design without introducing any new language features, and therefore can focus on

concepts, not details. The procedural programming paradigm arose in the 1970s, when software engineering was in its infancy, and it was seen as a great advance over what had been the common practice before hand: completely unstructured and undesigned software (rather like what you'd see modern students do if we didn't teach things like functions and dictionaries).

In this chapter, we will introduce the object oriented programming paradigm, which will allow us to keep the concept of ADTs, but give us some extra flexibility. We'll see some important advantages over procedural programming, but we will pay a price: we have to learn new programming language details. Many if not all of the object oriented programming concepts we learn using Python will carry over to other object oriented languages, like Java or C++, but they may vary slightly in detail and appearance.

The primary concept of object oriented programming is that we will combine the ideas of data abstraction and procedural abstraction, so that a single entity, which we call an *object*, encapsulates data and the procedures needed to operate on the data. This will allow us to design ADTs that store data and its operations in the same entity. Object oriented programming languages, like Python, Java and C++, have language features that make this idea as easy as possible, but also provide extra functionality that we would not be able to replicate very easily using the procedural programming paradigm. The object oriented programming paradigm is by far the most common approach to the design of very large software systems.

An object consists of data stored in the object ("encapsulated"), in much the same way that a record stores data, as well as functions that operate on the data. The data in an object are called *attributes*, and the functions are called *methods*; this terminology encourages the idea that an object is an entity with a behaviour and some importance, as opposed to a container with stuff in it.

It's important to understand that an object is a value, in the same way that a record (in the form of a dictionary, as was our practice in our study of ADTs) is a value. In previous chapters, we've talked about objects somewhat informally, but we've tried to keep the use of the word consistent with the meaning we take here. An object is stored in the heap, and we can use variables to refer to objects. References to objects can be passed to functions as arguments, and can be returned from functions as return values. We can store references to objects in lists and dictionaries.

A *class* is a formal description of attributes and methods contained by an object. An object is created according to a class description; we say that the object is an *instance* or an *instantiation* of a class. When we say that an object is a value, we also have to say that its type is the class that it instantiates. Many objects can be instantiated from the same class, and the objects will have the same methods, but can have different attributes.

We already know a bunch of Python objects. For example, strings, lists, and dictionaries are all objects. A string is an instantiation of the string class, and has two primary attributes: the sequence of characters, and the length of the sequence. The string class has methods such as `split()` and `rstrip()` among many others. The list class has methods `append()` and `extend()`. So we have been making use of objects for quite a long time. What we will learn in this chapter is to define our own classes, and instantiate objects from our own classes. Fortunately, our work with ADTs has already laid the foundations for our study.

19.2 Defining classes and instantiating objects

The Python keyword `class` begins a class definition. The name of the class follows, and the name is usually capitalized. In brackets following the class name, we write (`object`) followed by a friendly colon `:` (the box is only used here to draw attention to to colon). Every remaining part of the class

definition is indented relative to the keyword `class`, and the class definition looks somewhat like a function definition. Particularly, the (`object`) looks like a function's parameter, but it is not. We'll say more about this later. For now, accept it as a standard part of a class definition.

A class definition should start with a function whose name is `__init__()`. This function is actually a class method that is called implicitly by Python when an object of the class is being instantiated. This method, along with any other method defined in the class, must have as its first argument the parameter `self`. This practice is familiar to us as part of our convention when we implemented various ADTs: the first argument for an ADT operation was the data structure. The case is exactly the same when defining methods. The parameter `self` refers to the object itself. It is the object's way of saying *me* or *I*. This first parameter is not optional. Methods may require parameters that come after the parameter `self`, as needed by the design. Again, we've seen ADT operations like this.

The `__init__()` method is responsible for setting up and initializing the object. The use of the underscore characters in its name implies that this is not a method a programmer calls directly; Python makes sure this method is used at the time an object is instantiated. The initialization is much the same as our ADTs' `create()` functions. However, defining a class makes the initialization a little more programmer-friendly.

19.3 Simple examples

Consider the following very simple class definition:

```python
class Hero(object):
    def __init__(self, nn, pp):
        self.name = nn
        self.power = pp

    def say_hello(self):
        print('Hello, evil-doers! My name is', self.name)
        print('My super power is', self.power)


if __name__ == '__main__':
    diana = Hero('Wonder Woman', 'super strength')
    diana.say_hello()

    bruce = Hero('Batman', 'martial arts')
    bruce.say_hello()
```

In this example, we define a class with two methods only. Near the end of the example, two objects are instantiated. They have the same class, but they have different attributes.

The class name is Hero. The `__init__()` function (lines 2-4) has three parameters, one of which is `self`, which is required for any method. On lines 3-4, the object's attributes `name` and `power` are being created and initialized using the dot-notation and the object `self`. The `__init__()` function must not return anything. Apart from that, it is a normal function that gets called by Python when an object is being instantiated.

Lines 12 and 15 show how to instantiate an object from a class definition. Notice that the class name is used, as if it were a function. Behind the scenes, Python figures out that the Hero class is being instantiated, and Python creates a blank object (this is the `object` on line 1). Then Python calls Hero's `__init__()` function, giving three arguments: the new object, and the two arguments we see on line 12. Lines 3-4 create two attributes, and assign the string arguments. Again, behind the scenes, Python arranges for a reference to the new object to be returned, so that the variable on line 12 can refer to the newly instantiated Hero object.

Lines 6-9 define a method for the Hero class. This method has one parameter only, `self`, and doesn't do much except display some text to the console. Notice on lines 13 and 16 that the method is called using the dot-notation, and also that no arguments are given. Using the dot-notation is convenient for programmers, and behind the scenes, Python arranges for the parameter `self` to refer to the appropriate object.

This is where our procedural ADT examples differ from object oriented programming. In a procedural ADT, we use dot-notation as well, but we use the module name before the dot, and the operation name after the dot, e.g., `Queue.pop(aqueue)`. We also use the data structure explicitly as an argument. In object oriented programming, the object comes before the dot, and the object is never given explicitly as an argument.

Just to make the comparison between procedural programming and object oriented programming perfectly clear, we present below the procedural version of Hero:

```
1  def create(nn, pp):
2      hero = {}
3      hero['name'] = nn
4      hero['power'] = pp
5      return hero
6
7  def say_hello(ahero):
8      print('Hello, evil-doers! My name is', ahero['name'])
9      print('My super power is', ahero['power'])
10
11
12 if __name__ == '__main__':
13     diana = create('Wonder Woman', 'super strength')
14     say_hello(diana)
15
16     bruce = create('Batman', 'martial arts')
17     say_hello(bruce)
```

The object oriented version is a little different in detail, but very similar in principle.

19.4   Encapsulation and access control

A class contains attributes and methods; we call this *encapsulation*. One of the motivations for our procedural ADT design was to hide the data behind an interface, allowing access only through the operations defined as part of the interface. By restricting access to the data, we prevent the typical kinds of bugs that might occur when programmers have unrestricted access to data. But as you no

doubt noticed during our unit on trees, calling a function to access all the attributes of a tree node leads to programs that are at best verbose, and at worst, inconvenient.

Among the concepts that object oriented programming brings to software design is called *access control*. The class can indicate that some attributes should have restricted access, and others can have less restricted access. This flexibility allows programmers to protect what needs protecting, and to loosen the restrictions when doing so makes certain behaviours more convenient.

Python has a convention that governs access control for attributes. An attribute can be *public*, *protected*, or *private*.

- A *public* attribute can be accessed by any script, without restrictions.
  We use public attributes when allowing access makes other scripts and programs more convenient, and when having unrestricted access to the data will not put the data at too much risk. In Python, a public attribute starts with a normal lower-case letter; unlike private or protected attributes which start with at least one underscore character (_).
- A *private* attribute cannot be accessed by any script outside of the class definition. We use private attributes when limiting access to the data is essential to the correct behaviour of the object. Any attempt to access a private attribute outside the class definition will raise a run-time error. In Python, a private attribute starts with 2 underscore characters (__).
- A *protected* attribute is neither public nor private. It can be accessed outside the class definition, but only in related scripts designed specifically for the class, where insider knowledge will not put the data at risk. In Python, a protected attribute starts with exactly 1 underscore character (_).

For example, we've revised the Hero class above to make use of different levels of access control.

```python
class Hero(object):
    def __init__(self, name, power, sid):
        self.name = name
        self._power = power
        self.__secret = sid

if __name__ == '__main__':
    bruce = Hero('Batman', 'martial arts', 'Bruce Wayne')
    print(bruce.name)
    print(bruce._power)
    print(bruce.__secret)
```

In this example, the attribute name is public, the attribute _power is protected, and the new attribute, __secret is private. The main script below the class is able to display the Hero's name and super power, but a runtime error is raised as a result of trying to access the private attribute.

In some languages, for example, Java and C++, access to the attributes is very strictly enforced. In Python, the access control is more or less a suggestion. If you try hard enough, you can always access any attribute of any object of any class. That's part of the flexibility of Python programming. Still, use access control to indicate what level of access you think is essential, and trust that other Python programmers will have the wisdom to respect your design. Certainly, no one will have any sympathy for you if you ignore access control for Python objects and end up with a lot of buggy code because you were making changes to data you should not have been changing.

You can always provide methods as operations for your attributes, just as we saw in procedural

ADTs, such as treenode's `set_left()` or `get_left()`. In object oriented programming, these kinds of operations are called *getters and setters*, for obvious reasons. By providing an operation, you can do things like check the validity of the operation, or hide details that might change with future designs. But you are also increasing the effort required to use your objects. Choosing whether to make attributes public, private, or protected is a matter of experience.

19.5 Perspective

Earlier we said that the object oriented programming paradigm is by far the most common approach to the design of very large software systems. But that is not to say it is the only modern programming paradigm. Functional programming languages, like Scala and Haskell, are showing very great potential, and modern languages like Python and Java are increasingly incorporating functional programming ideas. The point of this comment is that one should not expect software development to remain static. There is still much that computer science has to learn about software design, and many great ideas about software have not even been thought yet. Computer Science is still too young to become too rigid about programming!

Mutable vs Immutable
Inheritance: Giving several classes some
common properties
Over-riding a method
Summary

20 — Object Oriented Programming

Learning Objectives

After studying this chapter, a student should be able to:

- Describe the difference between mutable and immutable objects.
- Explain the concept of inheritance, using a simple example.
- Explain the concept of function over-riding.
- Implement a method that over-rides `__str__()` for a simple class.

20.1 Mutable vs Immutable

In the previous chapter, we presented some of the basics of object oriented programming. In this chapter we will be exploring some of the implications of the paradigm. It is not the purpose of this course to teach everything there is to know about objects and classes. There are other courses that cover these matters very thoroughly. But there are a few concepts which will be helpful for your work in this course.

Consider the following class definition:

```
1  class Rectangle(object):
2      def __init__(self, width, length, x, y):
3          self.length = length
4          self.width = width
5          self.x = x
6          self.y = y
7
8  if __name__ == '__main__':
9
10     small = Rectangle(5, 5, 0, 0)      #   5x5 at (0,0)
```

```
11      big = Rectangle (10, 10, 0, 0)        # 10x10 at (0,0)
12
13      # move them
14      small.x += 10
15      small.y += -5
16
17      big.x = 15
18      big.y = 0
```

The initialization method creates and initializes four public attributes, The main script creates two instances of the Rectangle class, and then, by accessing the public attributes directly, moves the rectangles to different locations. The fact that the attributes are public means that they can be changed by any script. This is convenient, but leaves the object open to certain kinds of errors. For example, lines 14-15 move the small rectangle by a certain amount, whereas lines 17-18 move the bigger rectangle to a specific place. The difference between += and = is pretty small, and could be prone to error. One way to improve the situation is to protect the attributes, and provide setters and getters, as follows:

```
class Rectangle (object):
    def __init__(self, width, length, x, y):
        self._length = length
        self._width = width
        self._x = x
        self._y = y

    def move_by(self, by_x, by_y):
        self._x += by_x
        self._y += by_y

    def move_to(self, to_x, to_y):
        self._x = to_x
        self._y = to_y

if __name__ == '__main__':
    small = Rectangle (5, 5, 0, 0)      #    5x5 at (0,0)
    big = Rectangle (10, 10, 0, 0)      # 10x10 at (0,0)

    # move them
    small.move_by (10, -5)

    big.move_to (15, 0)
```

This has the advantage of making the intention very clear, as well as being less prone to errors.

The two methods change the object's attributes. Thus we call the object mutable. A *mutable* object has at least some methods that change the object's attributes. An *immutable* object has no methods at all that change the object's attributes. Typically, an immutable object has methods that create new objects rather than change an existing object. For example, consider the following class

definition:

```python
class Currency(object):
    def __init__(self, dollars, cents):
        self.__dollars = dollars
        self.__cents = cents

if __name__ == '__main__':

    movie_price = Currency(10, 50)
    pop_corn_price = Currency(7, 95)
    drink_price = Currency(5, 95)
```

Objects of the Currency class represent some amount of money, stored in two private attributes.
Because they are private, they cannot be accessed outside the class definition. We will design a few
methods to make the Currency class somewhat more useful, but we will insist that no one is able to
make changes to the value stored in an existing Currency object. In other words, we want Currency
objects to be immutable.

Suppose we want to take two Currency objects and add them together to get a combined value;
e.g., maybe we want to add the cost of popcorn and a soft drink. One way to do this is to write a
method called add() in the Currency class as follows:

```python
    def add(self, other):
        """
        Add the other Currency value to this one.
        :param other: A Currency value
        :return: (none)
        """
        cents = (100 * (self.__dollars + other.__dollars)
                + (self.__cents + other.__cents))
        self.__dollars = cents//100
        self.__cents = cents % 100
```

This example is instructive, because it shows that objects can be passed to other methods. In this
case, both objects are Currency objects, and the add() method is part of the Currency class, so all
the private attributes are accessible here. However, this method violates our design decision that a
currency object will not change its value. The method also makes a fairly arbitrary choice in deciding
that self should be the one to change, and not other. Why not the other way around?

Alternatively, we could design add() to return a new Currency object, whose value is the sum
of the two currency objects, as follows:

```python
    def add(self, other):
        """
        Add two Currency values together, producing a new
        Currency value
        :param other: A Currency value
        :return: A Currency value
        """
```

```
        cents = 100 * (self.__dollars + other.__dollars) \
                + (self.__cents + other.__cents)
        result = Currency(cents//100, cents % 100)
        return result
```

A new Currency object is being created to contain the sum, and it is being returned as the answer. It's not doing anything more complicated than the previous version. We could use it in a script as follows:

```
if __name__ == '__main__':

    movie_price = Currency(10, 50)
    pop_corn_price = Currency(7, 95)
    drink_price = Currency(5, 95)

    sub_total = Currency(0, 0)
    sub_total = sub_total.add(movie_price)
    sub_total = sub_total.add(pop_corn_price)
    sub_total = sub_total.add(drink_price)
```

While it looks like the variable `sub_total` refers to an object that is changing, a careful understanding of the `add()` method reveals that every time `add()` is called, a new Currency object is created, and a reference to a new Currency object replaces the previous reference to an older Currency object. You could imagine a number of other similar methods, one for subtraction, and one for multiplication, that also return new Currency objects. As long as no method actually changes any attributes, the object can be considered immutable, and the way we interact with immutable objects reflects our prior discussions in Chapter 13. There is no magic for immutable objects; just a different pattern for how they can be used.

All of the immutable objects we know about are like our Currency class: strings, numbers, and tuples have no methods to change the data stored inside; they can only be used in expressions that give rise to new instantiations. For example, we can't instruct Python to change the number 3. When we write 3 + 1 we are not changing 3 or 1; we are creating a new number with the value 4. We can use the value 4 in an expression, or we can save it by using an assignment statement.

20.2 Inheritance: Giving several classes some common properties

We started the chapter with a simple example of the Rectangle class. A rectangle instance has a position (x, y), and a height h and a width w. But if you know anything about geometry, you know that there are many other shapes we could want to define: squares, and circles, for example. If we want to use squares and circles in some application, they will also need position and size information. In fact, any shape we can think of will need a position; they have this in common. All shapes have a size, too, but we usually describe circles and squares differently. A circle has a radius, and a square has a side-length.

It would be trivial to create two new classes, one called Circle, and the other called Square, which is more or less copy/paste from the Rectangle class that we saw earlier.

```
class Rectangle(object):
    def __init__(self, width, length, x, y):
```

```python
        self._length = length
        self._width = width
        self._x = x
        self._y = y

    def move_by(self, by_x, by_y):
        self._x += by_x
        self._y += by_y

    def move_to(self, to_x, to_y):
        self._x = to_x
        self._y = to_y

class Square(object):
    def __init__(self, side, x, y):
        self._side = side
        self._x = x
        self._y = y

    def move_by(self, by_x, by_y):
        self._x += by_x
        self._y += by_y

    def move_to(self, to_x, to_y):
        self._x = to_x
        self._y = to_y

class Circle(object):
    def __init__(self, radius, x, y):
        self._radius = radius
        self._x = x
        self._y = y

    def move_by(self, by_x, by_y):
        self._x += by_x
        self._y += by_y

    def move_to(self, to_x, to_y):
        self._x = to_x
        self._y = to_y
```

This is not a good design. By now, you should be wary of copy/paste as a design strategy.

Python (and every other object-oriented language) provides tools that allow programmers to express the idea that some things are common to a group of classes, while other things will be different. The key is to give a name to the common properties, and to give them their own class. For example, we stated above that all shapes have a position. The Shape class below captures the idea of

position:

```
class Shape(object):
    def __init__(self, x, y):
        self._x = x
        self._y = y

    def move_by(self, by_x, by_y):
        self._x += by_x
        self._y += by_y

    def move_to(self, to_x, to_y):
        self._x = to_x
        self._y = to_y
```

A Shape knows about position, but not about size. Its methods `move_to()` and `move_by()` give us the tools to move shapes around.

Python (and every other object-oriented language) also lets us indicate that a Rectangle is a kind of a Shape:

```
1  class Rectangle(Shape):
2      def __init__(self, width, length, x, y):
3          Shape.__init__(self, x, y)
4          self.length = length
5          self.width = width
```

On the first line of the class definition for Rectangle, we see the name `Shape` being used instead of `object`. This is how we tell Python that a Rectangle is a kind of Shape. Specifically, it means that a Rectangle object has all the properties of a Shape as defined above: it has the two Shape attributes, `self.x` and `self.y`, and the two Shape methods `move_to()` and `move_by()`. The `__init__()` method for Rectangle has 5 parameters, and on line 3, it calls the `__init__()` method for Shape, allowing Shape to initialize the Shape attributes for the object. This is unlike most of the method calls we have seen because it mentions the class name, Shape explicitly, to avoid ambiguity.

The same technique can be used to say that a Square is a kind of Rectangle, specifically, a rectangle where the length and width are equal:

```
1  class Square(Rectangle):
2      def __init__(self, side, x, y):
3          Rectangle.__init__(self, side, side, x, y)
```

Square's `__init__()` method does nothing else but instruct Rectangle to initialize its attributes. Rectangle initializes `self.width` and `self.length` to the same value. and Rectangle instructs Shape to initialize `self.x` and `self.y`, as before.

We say that Square is *a kind of* Rectangle, or that Square is a *sub-class* of Rectangle. We say that Rectangle is the *super-class* for Square, or that Rectangle is the *parent class* for Square. Likewise, Rectangle is a sub-class of Shape, and Shape is the parent class of Rectangle. We should point out that this extends to the class `object`: Shape is a sub-class of `object` and `object` is the parent class of Shape. This language is not accidentally similar to the terminology we use for trees.

Our example has only shown a sequence of parent-child relationships from `object` to `Square`, but Currency is sub-class of `object` as well, making it a sibling, so to speak, of Shape. All classes that we have seen, including lists, tuples, dictionaries, strings, numbers and Booleans, are part of this tree of classes, and the root of that tree is the `object` class. It's not a binary tree; the object class has many sub-classes. The tree of classes is called a class hierarchy.

Each class inherits all the methods of the classes in its ancestry (up the hierarchy to the root). When each class calls the `__init__()` method of its parent class, it inherits all the attributes of the classes in its ancestry. As a result, Square and Rectangle have `move_to()` and `move_by()` methods, and attributes `self.x` and `self.y`.

We can define the Circle class to be a sub-class of Shape as well.

```
1  class Circle(Shape):
2      def __init__(self, radius, x, y):
3          Shape.__init__(self, x, y)
4          self.radius = radius
```

It inherits attributes and methods from Shape, and methods from `object`.

We can give each of our classes a method to calculate area. For Square and Rectangle, the method is the same; for Circle, we need to use a different calculation. In the definition for class Rectangle, we'd add the method:

```
1      # in Rectangle
2      def area(self):
3          return self.length * self.width
```

and in Circle, we'd add a method of the same name, but with a different calculation:

```
1      # in Circle
2      def area(self):
3          return Math.pi * self.radius ** 2
```

This is how we can give different properties to classes that have the same parent class. It's very interesting to note that we don't have to give Square an area method at all. Square will inherit Rectangle's area method, and that's exactly correct for Square, because length and width are the same.

When we call an object method, for example `sq.area()`, Python looks for the `area()` method in the object's class. If the method is defined there, Python uses it. But if the method is not defined by the class, Python checks the object's ancestor classes, one by one, until it finds one (good), or until it reaches the root of the class hierarchy, and cannot find a method with that name (a runtime error). The first one Python finds is the one that gets called.

20.3   Over-riding a method

As it stands, our Currency objects may be correctly summing up, but we have no way to see what their value is. We can't simply use `print()` to print out an object, and we can't access the private attributes outside of the class definition. But we could write a method to construct a string representing the value of a currency object:

```
    def to_string(self):
        return '$'+str(self.__dollars)+'.'+str(self.__cents)
```

The function doesn't display anything. It simply creates a string that could be displayed, as follows:

```
    print('Before taxes:', sub_total.to_string())
```

This is a very simple solution to a common problem.

You might wonder why we couldn't simply use Python's str() function to display a Currency object. It turns out that the answer is that we can, but the result is not what we want. By default, the str() function will create a string reflecting a Currency object's identity (i.e., its address on the heap), rather than its value (a dollar amount). For example, str(sub_total) would result in a string that looks like '<Currency object at 0x102d2c4e0>'. The str() function currently does not know enough about the Currency object to do any better.

We can do better by changing the name of our to_string() method, giving it the name __str__() instead. Everything else looks the same:

```
    def __str__(self):
        return '$'+str(self.__dollars)+'.'+str(self.__cents)
```

Now, when we call Python's str() function, Python knows to use Currency's __str__() method:

```
    print('Before taxes:', str(sub_total))
```

The object class has a definition for __str__(); this method returns a bland and unhelpful return string like '<object at 0x102d2c4e0>'. It is the best object can do without help from the programmer. But any class in the hierarchy can define its own version of __str__(), which can create a useful string based on the attributes of the class instances. That's what we did above with Currency. It has the same name as the method in object, but it has a different behaviour. We say that the __str__() method in the Currency class *over-rides* the __str__() method in the object class.

Python's str() function looks something like this:

```
def str(obj):
    return obj.__str__()
```

In other words, when you call str(total), Python translates that into call to the __str__() method in total's class. If total's class has a __str__() method, that's the one that gets used. If not, Python checks the parent class for a __str__() method, and up the hierarchy as described in the previous section. Since object does have a __str__() method, we always get something, even if it is not too informative.

The object class has dozens of methods like __str__() that make programming with objects and classes easier. For example, we could over-ride the __add__() method, by changing the name of Currency's add() to __add__(). With this new name in place, we can use normal arithmetic syntax on Currency objects:

```
if __name__ == '__main__':

    movie_price = Currency(10, 50)
```

```
    pop_corn_price = Currency(7, 95)
    drink_price = Currency(5, 95)

    sub_total = Currency(0, 0)
    sub_total = sub_total + movie_price
    sub_total = sub_total + pop_corn_price
    sub_total = sub_total + drink_price
```

This works because Python translates expressions using + into method calls using `__add__()`, much the way it translates the function `str()` into a method call for `__str__()`.

20.4 Summary

Objects and classes provide another technique for designing ADTs. We saw how to design ADTs using procedural programming in Chapters 6-17. We can also implement ADTs using classes and objects. We have learned that data can be encapsulated in objects along with the methods that operate on the data. We have seen that objects can be mutable, if some method allows the data to be modified. We can create immutable objects by ensuring that none of the object's methods is allowed to change the data; operations on immutable objects create new class instances containing the result of the operations. We discussed class hierarchy in terms of sub-classes and super-classes, and how a class inherits methods and attributes from its parent class. We have seen that some common Python functions like `str()` can be applied to instances of classes we define by over-riding `object` methods.

21 — Binary Search Trees

Learning Objectives

After studying this chapter, a student should be able to:

- Describe what a binary search tree is, in terms of its basic organizational principle.
- Describe basic operations of binary search tree, and demonstrate the behaviour in example binary search trees.
- Explain how tree balance affects the efficiency of various binary search tree operations.

21.1 Introduction

Many applications and algorithms depend on being able to store data in collections, like lists and dictionaries, so that the data can be looked up efficiently. It is so common that modern languages like Python provide tools to assist programmers. We've already mentioned lists and dictionaries, but it is also important to point out the Python operator `in` which can be used to check if a data value appears in a collection. For example, given a Python list `alist`, the expression `x in alist` performs a linear search, returning `True` if the value appears in the list. The time complexity of linear search of a list with N elements is $O(N)$.

As we have seen in the previous course, keeping the data organized can make a big difference. For example, if the data in the list is sorted, we can apply binary search to find a given value. The worst case time complexity of binary search of a sorted list with N elements is $O(\log N)$. It is very important to understand how much of an improvement binary search is over linear search. It's not just a little better. It's outrageously better.

To get the benefit of binary search in lists, we have to start with a sorted list, and it's important to understand the costs of sorting. The best sorting methods , for example, the Quick Sort algorithm, have average case time complexity of $O(N \log N)$. It turns out that no general purpose sorting algorithm can be expected to do better than $O(N \log N)$ on average.

Figure 21.1: A binary search tree. Values on the left are smaller. Values on the right are larger.
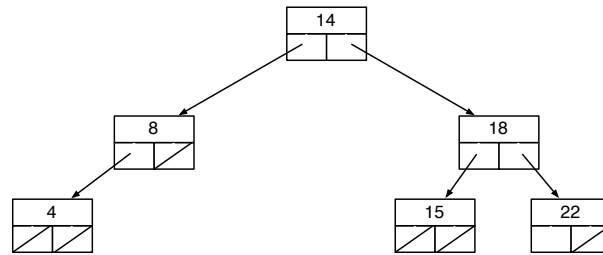
So if we have a list of data that never changes, sorting it once at the beginning, and using binary search for repeated look-up is a very efficient strategy. On the other hand, if we have an application where values are added and deleted from the list frequently, keeping it in sorted order can be very expensive. It is possible to keep sorting the data after every addition or deletion, paying the cost of sorting, i.e., repeatedly doing $O(N \log N)$ steps. It is also possible to ensure that the data is always sorted by inserting new data in the appropriate place, to maintain sorted order; this amounts to linear search, repeated for every new data value, at a cost of $O(N)$ every time. The question is whether we can do better than either of these two options, and the answer is yes.

21.2 Binary Search Trees

A *binary search tree* (BST) is a special kind of binary tree that allows us to keep data organized. The binary search tree is a collection, like a list. We'll be able to add data, delete data, and search for data in a binary search tree far more efficiently than using a Python list. The tree structure of the data allows us to avoid having to use a sorting algorithm to keep the data organized.

A binary search tree is just a binary tree, with up to two children per node, and data stored as usual. What makes a binary search tree special is the way the data in the tree is organized.

> **Definition 21.1**   A binary tree satisfies the *binary search tree property* if at every node v in the tree:
>
>   1. All the data stored in the left subtree of v is smaller than the data at v.
>   2. All the data stored in the right subtree of v is larger than the data at v.
>
> If a binary tree satisfies this property, it is called a binary search tree.

An example of a binary search tree is shown in Figure 21.1. The definition above implicitly assumes that a binary search tree does not store multiple copies of the same data value, though the definition could be adjusted to allow multiple copies. Also, the terms *smaller* and *larger* could mean before and after according to a given rule, e.g. alphabetical order. A binary search tree can be used to store numerical data, text data, or any other kind of data that has a natural ordering to the values. It is interesting to note that an in-order traversal of a binary search tree processes the data in sorted order.

Just a word of caution: It is common for students to assume that every binary tree is a binary search tree. This is not true! There are lots of different ways to organize data in a binary tree, and the BST property is only one possible way. Another organizational principle is called the *heap property*.

21.3   Basic operations

There are three important operations on binary search trees that we have to discuss:

1. `member_prim(tnode, value)`: checks if the `value` is in the binary search tree rooted at `tnode`.
2. `insert_prim(tnode, value)`: adds the `value` to the binary search tree rooted at `tnode`.
3. `delete_prim(tnode, value)`: removes the `value` from the binary search tree rooted at `tnode`.

We will discuss these in order.

Before we start, we will review the tree-node ADT, this time expressed using an object-oriented version.

```python
class TreeNode(object):
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right
```

This version does not define any operations or methods; there are three attributes, and by design, all the attributes are public. This is very convenient as our Python functions will be far less verbose. I the code examples that follow, we will assume that the TreeNode class is defined in a Python module whose name is the same as the class, and we'll `import` `TreeNode as TN`. When we want to create a new TreeNode instance, we will write `TN.TreeNode(5)`, mentioning the module name first, then the class.

21.3.1   Checking if the value is in the BST

If we were searching for a value in an **arbitrary** binary tree (one that does not have the BST property), we would have to check three different places to find a particular value.

1. First, check the root. If the value you want is stored as data in the node, you have successfully found what you were looking for.
2. Second, check if the value is on the left side of the tree.
3. If the item was not in the left side of the tree, check the right side of the tree.

This analysis lends itself to a simple recursive function.

When we are searching for a value in a binary search tree, we can be a little more intelligent about our search. Given a tree *t* with the BST property and a data value:

1. If *t* is *empty*, return `False`
2. If *t* is *stores the same data value*, return `True`
3. If the data value is *smaller than the root*, look *left* recursively
4. If the data value is *larger than the root*, look *right* recursively

In Python, the function is as follows:

```python
def member_prim(tnode, value):
    """
    Purpose:
        Check if value is stored in the binary search tree.
```

```
    Preconditions:
        :param tnode: a binary search tree
        :param value: a value
    Postconditions:
        none
    :return: True if value is in the tree
    """
    if tnode is None:
        return False
    else:
        cval = tnode.data
        if cval == value:
            # found the value
            return True
        elif value < cval:
            # use the BST property
            return member_prim(tnode.left, value)
        else:
            return member_prim(tnode.right, value)
```

It's important to draw attention to how the function handles a search that comes up empty. Essentially, the search keeps stepping left or right until it steps right off the tree, i.e., the subtree we stepped to is empty. While a human might be able to keep a context in mind, especially if a tree diagram is available, the function does not make any attempt to keep track of which nodes were looked at.

Search in a binary search tree only follows one of the possible two branches at any node. If the search is lucky, the value we're looking for is near the top of the tree. But if we're not lucky, the value is stored in a leaf node, or, the value is not in the tree at all. As a result, the worst case time complexity for `member_prim()` happens when the search ends up exploring a path to the deepest part of the tree.

As with all recursive functions, we analyze the computational complexity of the function by counting the number of recursive calls. The number of recursive calls needed to get to the bottom of the tree depends on the height of the tree. In a complete binary tree with height h, the number of nodes is $N = 2^h - 1$. Solving for h, we can say that the height of a complete binary tree with N nodes is $h = \log(N + 1)$. Thus, in the worst case, the computational complexity for search in a balanced binary search tree is $O(\log N)$.

Binary trees don't have to be balanced, though. Informally, in a balanced tree, most nodes have 2 children. It is very common for trees to have at least a few nodes with only one child. It is possible to build a tree in which each node has no more than 1 child. We call such trees *degenerate* because it is not very tree-like. In a degenerate tree, every node except the leaf node has one child. Therefore the number of nodes in a degenerate tree of height N is exactly N.

The worst case time complexity for search in general binary search trees is $O(N)$, simply because in the worst case, a binary search tree might be degenerate.

21.3.2 Adding a value to the BST

When we talk about adding a value to a binary search tree, we will assume that the value is not already in the tree. If the value happens to be in the tree already, we will not add a second copy. We could modify the rules to allow second copies, but that adds detail without enhancing clarity, so we'll leave it alone.

The key algorithmic insight that we use to put a value into a binary search tree is to put the value exactly where `member_prim()` would find it if it were in the tree already. In other words, our algorithm would start by searching the tree for the value. Since the value is (presumably) not in the tree, eventually, the search will wind up looking at the value `None`, only to realize that the value is not in the tree. The new value that we're adding to the tree will always be a leaf node, as either a left or right child of a node that is currently in the tree. For example, to add the value 10 to the tree in Figure 21.1, we'd search for the value 10 by stepping left from 14, and right from 8. The new node containing the value 10 should be the right subtree of the node containing 8.

Remember that we made note of this situation when we discussed the `member_prim()` function. We observed that search does not make any attempt to keep track of how it arrived at a `None`, i.e., an empty tree. This `None` that we're looking at is exactly the `None` we want to replace, by adding a new leaf. But this new leaf cannot connect itself to its parent node; the parent node has to connect itself to the leaf, by storing the reference to the new leaf as a left or right subtree.

It is possible to write some tricky Python code to handle the special situation that arises when a node has to attach a new leaf node, but there is a simpler solution. Instead of having special code that only attaches a leaf, we'll always make sure that every node re-attaches its child node to the appropriate place. In most cases, this is simply an assignment that was unnecessary, but in the case of adding a new leaf, the new connection will be made with a very small amount of fuss.

Given a tree *t* with the BST property and a data value:

1. If the tree is *empty*, return *a new tree with the value stored*
2. If the root *stores the same data value*, return the root
3. If the data value is *smaller than the root*, *update* the *left* subtree, recursively
4. If the data value is *larger than the root*, *update* the *right* subtree recursively

In Python, the function is as follows:

```python
def insert_prim(tnode, value):
    """
    Insert a new value into the binary tree.
    Preconditions:
        :param tnode: a binary search tree
        :param value: a value
    Postconditions:
        If the value is not already in the tree, it is added
    Return
        :return: flag, tree
        Flag is True if insertion succeeded;
                tree contains the new value
        Flag is False if the value is already in the tree,
                tree unchanged
    """
```

```
16
17      if tnode is None:
18          return True, TN.TreeNode(value)
19      else:
20          cval = tnode.data
21          if cval == value:
22              return False, tnode
23          elif value < cval:
24              flag, subt = insert_prim(tnode.left, value)
25              if flag:
26                  tnode.left =  subt
27              return flag, tnode
28          else:
29              flag, subt = insert_prim(tnode.right, value)
30              if flag:
31                  tnode.right = subt
32              return flag, tnode
```

There are two base cases. The first base case, lines 17-18, handles the creation of a new treenode, which happens as soon as the search reaches an empty tree. Pay special attention to the fact that a tuple is being returned: the value `True` to indicate a successful insertion, and the newly created leaf node. The other base case, lines 21-22, handles the situation that the value is already in the tree. Again a tuple is returned: this time the value `False` to indicate that no value was inserted, and the tree unchanged.

The two recursive cases similar. If the new value should be inserted on the left (lines 24-27), the recursive call starts on the left subtree (line 24). But notice how the resulting tuple is used. If the insertion was successful (line 25), the node sets the left subtree (line 26). Here `subt` could be the new leaf node, or it might be just the old subtree with a new leaf attached somewhere far below `tnode`. In either case, the left subtree is attached. The case for the right subtree is essentially the same (lines 29-32).

Since the `insert_prim()` function essentially does a search, the computational complexity again depends on the height of the tree. If the tree is balanced, we have $O(\log N)$, as before. If the tree is degenerate, we have $O(N)$, again, as before, for exactly the same reasons.

21.3.3 Removing a value from the BST

Like the previous two examples, the function to remove a value from the tree first has to find the value in the tree. Unlike insertion, where we always insert a new treenode as a leaf in the tree, there are four different situations for deletion:

1. The node to be deleted has no children. For example, node 4 in Figure 21.1. In this case we can just remove the leaf, and the resulting tree will remain a binary search tree. The way to remove the leaf is to replace the reference to the leaf with a `None`.
2. The node to be deleted has only a left subtree. For example, node 8 in Figure 21.1. In this case we can just connect the parent of node 8 to node 8's left child, and the resulting tree will remain a binary search tree. This connection can be implemented by replacing the reference to node 8 with the reference to node 8's left child.

3. The node to be deleted has only a right subtree. This is similar to case 2.
4. The node to be deleted has two subtrees. For example, node 14 in Figure 21.1. This is the only case where the right thing to do is perhaps not obvious. We discuss it below in detail.

These 4 cases have to be handled separately.

Case 4: the node to be deleted has a left and right child

Suppose we want to delete the node containing 14 in Figure 21.1. We'll need to make one tree by connecting the two subtrees, somehow. There are a bunch of different ways we could do this, but we have to make sure that, whatever we do, the resulting tree will satisfy the binary search tree property.

If we look carefully at the tree, and think about the binary search tree property, we can make an important observation: every value in 14's left subtree is smaller than any value in 14's right subtree. This observation suggests a strategy: We can connect 14's `right` subtree, the tree rooted at 18, to the right of the biggest node in 14's `left` left subtree. In this example tree, the value 8 is the biggest value in 14's left subtree, since it has no right child. But in general, 14's left subtree might contain values to the right of 8, which are bigger than 8, but less than 14. This strategy will connect the whole tree starting at 18 as 8's right subtree.

In general, we have to find the largest value in a subtree to manage case 4. Fortunately, the binary search tree property tells us exactly where this value is: keep stepping right until you find a node that has no right subtree.

To summarize, given a tree with the BST property and a data value to delete, we can find the node to be deleted as follows:

- If the tree is *empty*, return False
- If the root *stores the same data value*, *reconnect* the root's children (see Algorithm reconnect)
- If the data value is *smaller than the root*, *delete* the value from the *left* subtree, recursively
- If the data value is *larger than the root*, *delete* the value from the *right* subtree recursively

Once found, the node to be deleted is at the root of a subtree using the reconnect algorithm, as follows. Given a tree with the BST property, whose root is to be deleted:

- If the root has *no children*, return None (the empty tree)
- If the root has *a left child, but no right child*, return the left child.
- If the root has *a right child, but no left child*, return the right child.
- If the root *has 2 children*:

  – Start at the left child, and step down, always going right, to find left's largest node.
  – Attach the root's right child as the right subtree of left's largest node
  – Return left.

The deletion operation has two phases: first, finding the node to be deleted; second, reconnecting the tree to maintain the binary search tree property. To find the node to be deleted, we have to descend into the tree, and in the worst case, we have to go to the deepest node in the tree. The first phase is essentially the same as the search algorithm. In the second phase, cases 1-3 are very simple, requiring a couple of assignments only, and therefore, $O(1)$. Case 4 warrants special consideration, since the reconnection requires stepping down the node's left subtree. But observe that the total amount of stepping, in the worst case, reaches the deepest part of the tree. In other words, the worst case depends on the height of the tree, as with search, and with insert. So, when the tree is balanced, the worst-case time complexity is $O(\log N)$, or if the tree is degenerate, the worst case time complexity is $O(N)$.

21.4 The importance of Binary Search Trees

If we can guarantee that the binary search trees are always balanced, every operation we've considered in this chapter has worst case time complexity of $O(\log N)$, where N is the number of data values stored in the tree. In comparison, the cost of keeping a sorted list sorted while adding and deleting values is at best $O(N)$. The improvement of $O(\log N)$ of $O(N)$ is astonishingly good. It may seem trivial if you misunderstand the nature of the logarithm function. When your costs can be described by $O(\log N)$, it's the practical equivalent of saying *no normal, practically realistic increase in the problem size will have any noticeable effect on the runtime*. This is an astounding statement. Keep reading it until you are astounded. We'll wait.

Keep in mind that we have said nothing about keeping the trees balanced. None of our algorithms made any attempt to do that. One can easily imagine a sequence of insertions and deletions that could cause binary search trees to become degenerate, and thereby lose the logarithmic advantage. In fact, there are more complicated algorithms that rearrange the tree at every insertion or deletion to guarantee balanced search trees at all times, but those algorithms are not studied in first year computer science. They are a second year topic!

22 — The Table ADT

Learning Objectives

After studying this chapter, a student should be able to:

- Describe what a Table is, in terms of its possible applications.
- Explain how a variation of the treenode data structure can be used to implement the Table ADT.
- Explain how the binary search tree can be used to make the Table ADT implementation efficient.

22.1 Introduction

In the previous chapter we saw how to use binary tree data structure, and the binary search tree property to organize data so that tasks such as data look-up, insertion, and deletion are very efficient. In this chapter we will complete the picture by defining an ADT called a Table, which will build on binary search trees, but will allow arbitrary data to be stored.

The Table ADT will store `keyed` data. A key is a value that is unique to the data being stored, like a Student number. The key tells us which data we're looking for, but the data may be much more than the key, for example, a complete Student record. The keys have to be comparable, like numbers or strings.

The Table ADT will provide yet another example of an ADT, but more importantly, will be a direct application of binary search trees. Furthermore, by studying the implementation of the Table ADT, we will get an insight into one of Python's key data structures. A dictionary is just Python's own implementation of our Table ADT.

The primary operations for the Table ADT are as follows:

- `is_empty()`: Query if table is empty
- `size()`: Query size of table

- `insert()`: Insert key, value into table
- `delete()`: Delete key, value from table
- `retrieve()`: Retrieve the data for a given key

Our Table ADT will be implemented using the object-oriented variation of the TreeNode class we saw in Chapter 21. Our variation, called a KVTreeNode, will store two different pieces of information at a node: a key value, which we will use to organize the tree structure, and the data value, which will be the payload. For example, in a student information system, the key would be the Student number, and the data value would be the complete student record.

The KVTreeNode is defined as follows:

```python
class KVTreeNode(object):
    def __init__(self, key, value, left=None, right=None):
        """
        Create a new KVTreeNode for the given data.
        Pre-conditions:
            key:    A key used to identify the node
            value:  Any data value to be stored in the KVTreeNode
            left:   Another KVTreeNode (or None, by default)
            right:  Another KVTreeNode (or None, by default)
        """
        self.value = value
        self.left = left
        self.right = right
        self.key = key
```

We've made all the attributes public, for convenience.

22.2 Table ADT implementation

The implementation of the Table ADT will use the KVTreeNode discussed above, as well as slight variations of the binary search tree algorithms discussed in Chapter 21.

We'll make it a class. The `__init__()` method, given below, demonstrates that a Table is an object with two attributes, `root` and `size`:

```python
class Table(object):
    def __init__(self):
        self.__root = None
        self.__size = 0
```

The `size` field allows us to keep track of the number of nodes in the Table, so that we don't have to count them when someone asks for the size of the Table.

The retrieve method takes a key, and returns the data value associated with the key in the Table, if it exists. The operation uses an internal function, which is very similar to the `member_prim()` function from Chapter 21. The only significant difference is that the key value is used to locate the appropriate treenode (e.g., line 17), but the data value is returned (line 19).

```python
1    def retrieve(self, key):
2        """
```

```
3            Return the value associated with the given key.
4            Preconditions:
5                :param key: a key
6            Postconditions:
7                none
8            Return
9                :return: True, value if the key appears in the table
10                        False, None otherwise
11           """
12
13           def retrieve_prim(tnode):
14               if tnode is None:
15                   return False, None
16               else:
17                   ckey = tnode.key
18                   if ckey == key:
19                       return True, tnode.value
20                   elif key < ckey:
21                       return retrieve_prim(tnode.left)
22                   else:
23                       return retrieve_prim(tnode.right)
24
25           return retrieve_prim(self.__root)
```

The other two operations, namely `insert()` and `delete()` are left as exercises.

22.3 The efficiency of the Table ADT implementation

The Table ADT implementation is simply a binary search tree, where the keys are used to keep the data values organized. As a result, the operations retrieve, insert, and delete operations have the same computational complexity as the simpler variations studied in Chapter 21. If the trees are balanced, the worst case time complexity for all three operations is $O(\log N)$. As a result, Tables can get extremely large, and we wouldn't expect to notice any significant runtime costs when accessing data there. Degenerate trees are still a problem, but not a problem we have to solve in a first year course!

Introduction
Using frequency information to design a
code
Encoding and decoding
Efficiency considerations

23 — Case Study: Huffman codes

Learning Objectives

After studying this chapter, a student should be able to:

- Describe the difference between fixed-length codes and variable-length codes.
- Describe the Huffman tree algorithm for designing variable length codes for a given message.
- Outline and explain the time complexity for the various stages of encoding and decoding using Huffman codes.
- Explain why a pair of queues makes the most efficient implementation of the Huffman tree algorithm.

23.1 Introduction

In this section, we'll study an interesting application of trees, namely the construction of a code to support file compression for text files. This is an interesting case study for a number of reasons. File compression is a real application. The use of trees to support file compression is far from obvious, but easy enough to present at an introductory level. The algorithms we will study in this chapter reveal a depth of analysis and design, capturing the spirit of the field of computer science, going beyond the notion of *learning to program.*

Computers use binary digits ("bits") to represent all data, but particularly for the purposes of this chapter, text data. Numbers, letters, and symbols like + are represented inside the computer using sequences of bits. The representation amounts to being a code.

Fixed-length codes, like ASCII, use bit-sequences that are all the same length. ASCII codes were designed to use 7 bits to represent the typical characters and symbols you can type on a keyboard. A modern, standardized set of codes, known as Unicode, allows for 8 bit codes (e.g., UTF-8) or 16 bit codes (e.g., UTF-16). Standardized codes like these allow computer applications to exchange

Figure 23.1: Two binary trees.
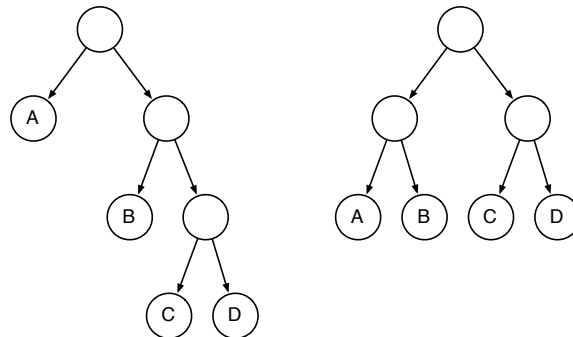
information, and having a fixed-length means that computers, and computer programs can manipulate the data relatively easily.

The practical motivation for studying Huffman codes is the fact that in human languages, some characters are used more frequently that others. For example, the letter 'e' appears far more frequently in English text than the letter 'w'. Using this fact motivates the concept of a variable-length code, in which more frequently used characters have shorter codes. Variable length codes can result in more efficient transmission of text data from one computer to another. We can get the most out of this idea by designing a code that is specific to the message being transmitted: characters or symbols that do not appear in the text need not be coded.

It turns out that there is a relationship between codes and binary trees. A given code (fixed-length or variable-length), can be represented by a binary tree in which the characters or symbols to be encoded are leaf nodes (importantly, leaf nodes have no children). Since there is a unique path from the root to any leaf node, we can describe the path from root to leaf using '0' if the path takes a left branch, or '1' if the path takes a right branch. A fixed-length code has all the symbols (leaf nodes) at the same level in the binary tree. A variable-length code may have symbols (leaf nodes) at different levels of the tree.

For example, consider Figure 23.1, which has two trees. On the left, is a tree where the leaf nodes are at different levels, and and on the right is a tree where the leaf nodes are at the same level. The tree on the right corresponds to a fixed-length code. Using the rule that taking a left branch corresponds to a '0', and a right branch corresponds to a '1', we can see the following codes:

00	'A'
01	'B'
10	'C'
11	'D'

Using the fixed-length code, the message 'ABACABAD' is encoded by the sequence '0001001000010011'.

Using the tree on the left corresponds to a variable-length code. Using the rule that taking a left branch corresponds to a '0', and a right branch corresponds to a '1', we can see the following codes:

0	'A'
10	'B'
110	'C'
111	'D'

Using the fixed-length code, the message `'ABACABAD'` is encoded by the sequence `'01001100100111'`.

The variable-length code would be more efficient at representing messages with a high number of `'A'` in them, and relatively few `'D'`. But if the message had roughly equal numbers of letters `'A'` through `'D'`, the fixed length code would be more efficient. A simple exercise in counting reveals that in the example above, the variable-length code needs fewer binary digits than the fixed-length code. The difference is small here, because the message was small; a larger message will demonstrate more impressive savings. As we will see, the algorithm we will study in this section will find the most efficient code given a particular message. For some messages, it will produce a fixed-length code, precisely because all the symbols with roughly the same frequency in the message.

To make the discussion very concrete, we will assume that we are given a message that we want to encode. The messages we use in our discussion will be relatively short, but in general, they could be as long as you like: essays, Wikipedia articles, DNA sequences, 100 years of temperature data...

Huffman codes are designed by constructing a binary tree based on the symbol frequencies in the message, so that common symbols are nearer the root than uncommon symbols. The process to build a Huffman code has two steps:

1. Determine the frequency of the symbols in the message.
2. Build a binary tree using frequency of the symbols.

We'll take a closer look at the second step a bit later. Once the tree is built, the message can be encoded, either from the tree directly, or from a table created for symbol-code pairs. What ever form is chosen, the data structure that is used to encode the message is called a *codec*.

In order to decode a message that was encoded using a Huffman code, the transmission needs to include in the message the information needed to decode the message. In other words, the encoded message has to include the codec in the transmission. When an encoded message is received, the decoding program has to read the codec, and then apply the codec to the encoded message to recover the original message.

23.2   Using frequency information to design a code

The first step in the encoding process is to analyze the message. The analysis has to record the symbols in the message, and count how often they are used. In Python, we could use a dictionary whose keys are symbols, and whose values are counts.

```python
# analyze the frequency of characters in the message
frequencies = {}
for char in message:
    if char in frequencies:
        # seen char before
        frequencies[char] += 1
    else:
        # first time seeing char
```

```
frequencies[char] = 1
```

For example, if the message to be encoded is 'ABACABAD', we can count 4 'A', 2 'B', and 1 each of 'C' and 'D'.

The second step is a little more interesting. We want to build a binary tree that has symbols stored in the leaf nodes, and we want to organize the tree so that uncommon symbols are deeper in the tree than more common symbols.

The Huffman algorithm for building a tree with the desired organization is simple and clever. It builds the binary tree up in stages. Initially, every symbol in the message is stored, along with its frequency, in a leaf node, i.e., a binary tree with no children, and all these leaf nodes are stored in a list.

The tree-building process continues as follows: As long as there are at least 2 trees in the list, the two trees with the lowest frequencies are selected and removed from the list, and merged. The merge process creates a new tree whose children are the two selected trees, and whose frequency is the sum of the frequencies of its children. The new tree is placed in the list with the other trees. Note that if two trees are removed form the list, and an new tree is placed in the list, the list gets smaller by one element.

The process of selection, removal, merging, and replacing continues until there is only one tree left in the list, which is the tree we'll use to derive the code for the message. This tree is called a Huffman tree.

Using our example message 'ABACABAD', the four symbols in our message would be stored as follows:
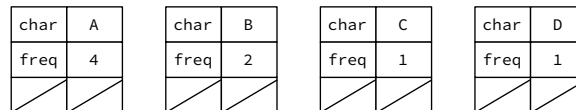
It's important to note that the frequencies are stored in the treenodes as well. Also, these Huffman trees would be stored in a list, which is not explicitly shown in the diagram.

In our example, the two lowest frequency trees are the trees for 'C' and 'D'. These are removed from the list, and merged, giving them a common parent node with no character, but a combined frequency of 2:
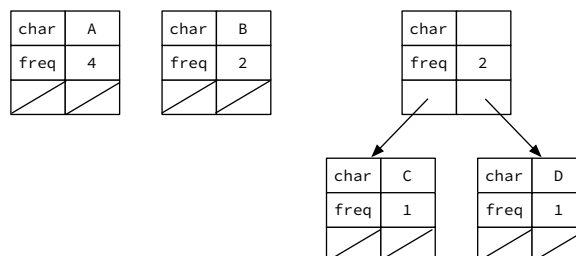
The tree-building process continues by choosing the two lowest frequency trees in the list, namely the tree for 'B', and the tree combining 'C' and 'D'. These trees are merged, giving them a common parent node with no character but a combined frequency of 4:

```
char  A        char
freq  4        freq  4
                 /      \
         char  B          char
         freq  2          freq  2
                            /      \
                    char  C          char  D
                    freq  1          freq  1
```
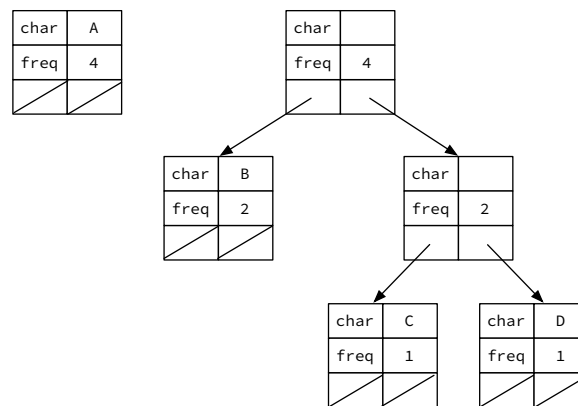
Since there are only two trees left, these are combined in the same way, giving them a common parent node with no character but a combined frequency of 8:

```
                char
                freq  8
                 /        \
        char  A             char
        freq  4             freq  4
                             /      \
                     char  B          char
                     freq  2          freq  2
                                       /      \
                               char  C          char  D
                               freq  1          freq  1
```
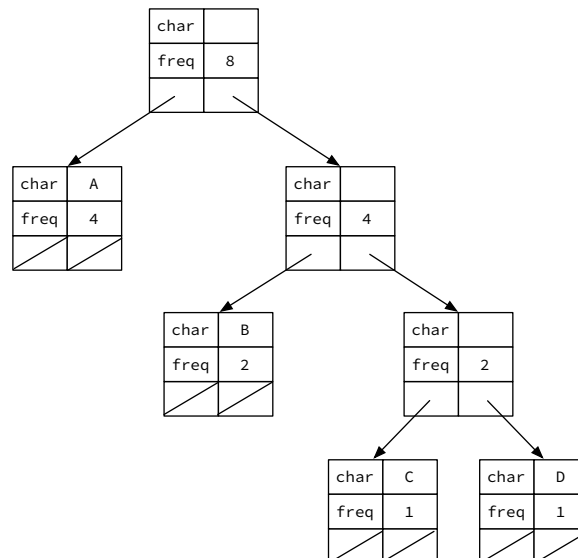
This tree is similar in structure to one of the trees in Figure 23.1, which should not be a big surprise.

There are two places in the Huffman tree building process that involve arbitrary choices. If two or more trees have the same lowest frequency, it does not matter which one (or two) are chosen; the choice is arbitrary. As long as the new tree node has the selected nodes as children, it does not matter which is the left child, and which is the right child; the choice is arbitrary. As a result of arbitrary choices, the process will result in different Huffman trees depending on the choices made arbitrarily. In this situation, we say that Huffman trees are not unique. By this we mean that any of the trees that might have resulted are as good for the purpose of producing codes as any of the others.

23.3 Encoding and decoding

Once we have a Huffman tree we have several options for how to use it. Every symbol in the message appears in the tree as a leaf. A symbol's Huffman code is simply the description of the path from the

root of the Huffman tree to the leaf, using a '0' when a left branch is taken, and a '1' when a right branch is taken. Since every leaf has a unique path from root to leaf, every symbol has a unique code.

The simplest way to use a Huffman tree is to traverse the tree building up a code, i.e., a string of '0' and '1' sequences. The sequence is extended by a '0' every time the traversal descends into a left subtree, and by a '1' every time the traversal descends into a right subtree. When the traversal reaches a leaf node, the code, and the symbol can be entered into a dictionary, with the symbol as the key, and the code as the value.

Once this dictionary is constructed, the encoding process is a matter of stepping through the message, and using the dictionary to look up each symbol's code.

```
encoded = ''
for char in message:
    encoded += code[char]
```

Decoding an encoded message is a little bit trickier. We will assume that a code-symbol dictionary has been constructed; this is simply the inverse of the symbol-code dictionary we built from the Huffman tree. In a real application, the decoding dictionary will need to be contructed from the information transmitted along with the encoded message.

The reason decoding is a bit trickier is that Huffman codes are variable length. We don't know in advance which bit sequences belong to which symbols.

The process of decoding an encoded message consists of stepping through the encoded message one character at a time, building up a short sequence of binary digits until the sequence is recognized by the decoding dictionary as a code. Once a code is recognized, the corresponding symbol can be added to the decoded message.

```
message = ''
first = 0
last = first + 1
while first < len(encoded):
    tryit = encoded[first:last]
    if tryit in codes:
        message += codes[tryit]
        first = last
        last = first + 1
    else:
        last += 1
```

23.4 Efficiency considerations

The Huffman tree algorithms were described at a fairly high level, temporarily ignoring practical considerations. Careful implementation of these ideas will result in a fairly efficient program. Let us suppose that the message has k unique symbols, and has length N; further, we'll assume the encoded message has length M. Counting the frequency of the symbols should require $O(N)$ steps; we have to look at each symbol once to count it. Given a codec for the message, encoding the message requires $O(N)$ codes to be joined together. Given a codec, decoding the message requires looking at each binary digit in the encoded message, i.e., $O(M)$.

The one place where we have to work a bit harder is in the merge process. The process has to build k leaf nodes, and performs $k-1$ merges (because the list of trees gets smaller by one every time). The process requires us to find the two smallest trees in the list. If k is small enough, it doesn't matter how we find the two smallest trees.

But considering the costs here, even if k is not too big, leads to an interesting study. There are two obvious ways to find the smallest tree in a list of trees. First, we could perform a linear search. Repeating the linear search k times will result in a time complexity of $O(k^2)$. The second option is to keep the list sorted, in order of increasing frequency. With the list organized, finding the two smallest trees is easy: we can remove the first two trees in the list in $O(1)$ time. However, we have to keep the list sorted. That could involve a linear search, putting the newly merged tree into the sorted list at the correct position to keep the list organized by increasing frequency, which requires $O(k)$ steps. Or we could simply re-sort the list, requiring $O(k \log k)$ steps.

There is an interesting property of the tree-merging process that we can use to our advantage. Consider what happens when we merge two trees: the result is a single tree whose frequency is the sum of the two selected trees. It turns out that at any stage of the merging process, any tree we construct by merging two other trees will have a greater frequency than any previously merged tree. Consider the sequence of merges in the example above: our merged trees had frequencies 2, 4, and 8, in that order. Because this is guaranteed to be true, we do not have to sort the trees we build, we can store them in a FIFO queue.

We can implement an efficient merge process by keeping two queues: the sorted leaf nodes, and the queue of merged trees. The smallest tree will appear at the front of one of these two queues, and the appropriate tree can be dequeued in $O(1)$ time. The result is always a merged tree, so it should be enqueued (on the back, FIFO, remember) to the queue of merged trees, again in $O(1)$ time. The process of merging trees will still require $O(k)$ merges, but each merge is $O(1)$ time! The dominating cost of this algorithm is the cost to sort the initial list of treenodes, i.e. $O(k \log k)$ time.

A good implementation would hide the two queues behind an ADT where the dequeue operation returned the smallest tree, and the enqueue operation puts trees only on the merged list.

24 — Three Kinds of Tasks

Learning Objectives

After studying this chapter, a student should be able to:

- To distinguish between search tasks, decision tasks, and optimization tasks.
- To give examples of search tasks, decision tasks, and optimization tasks.

24.1 Introduction

In computer science, and software development, we face many kinds of problems. The fundamental problem, from which all the others seem to flow, is that we want our computers to perform a specific task. In other words, the fundamental problem in computer science is to develop algorithms. An algorithm is a sequence of instructions that accomplish a stated task. We've seen a fair number of algorithms so far. Some are fairly mundane, like calculating an average value for some collection of data, or finding the nth Fibonacci number. Others are a bit more interesting, like simulating an MM1 queue, or building a Huffman tree.

Before we start our study of algorithms, it is useful to provide a loose classification of the kinds of tasks that we typically need algorithms to do. These are broad categorizations with somewhat vague descriptions, because a lot of different examples fit into them.

The first is the class of *search tasks*. A search task is any task in which the objective is the discovery or construction of a thing (object or value) that satisfies a given set of requirements. Some example search tasks:

- Calculating the square root of a given number. The value we need has the requirement that if it is squared (i.e., multiplied by itself), the result is equal to the given number.
- Determining the solution of a set of equations involving a set of mathematical variables. The thing we need as a result is a set of numerical values, one for each variable, which makes the given equations true.

- Planning an itinerary for a trip from Saskatoon to Hong Kong. The thing we want to have at the end is a plan of travel. This plan must be feasible, in the sense that each step of the itinerary is a real travel service (e.g., a real flight on a real airline), and each step in the sequence completes before the next step is supposed to start.

The thing about search tasks is that the answer can be checked, or *certified*, just by looking at the answer and the requirements.

The second kind of task is a *decision task*. A decision task is any task in which the objective is to determine if a set of requirements can be satisfied. A decision task is usually phrased as a yes/no or true/false question. Some examples of decision tasks:

- Does a given number have a real-valued square root?
- Does a set of equations involving a set of mathematical variables have a solution?
- Is it possible to travel from Saskatoon to Hong Kong?
- Is there a time this week that everyone on the project can meet for an hour?

Decision tasks are closely related to search tasks. Sometimes, the answer to a decision task is to perform the related search task, and answer affirmatively if the search task has a certified answer. Sometimes, the decision task is a lot easier; for example, by checking the sign of a number, we can determine if it has real roots without having to find the roots first.

The third kind of task is an *optimization* task. An optimization task is a task in which the objective is to find the best, according to a given set of criteria, out of all the possible things that satisfy a set of requirements. Some examples of optimization tasks:

- Find the largest (or smallest) number in a collection (list or tree).
- Find an itinerary for travel from Saskatoon to Hong Kong that has the lowest dollar cost.
- Find a code for text data that results in the highest compression rate of all possible codes.
- Find a one-hour meeting time this week at which the largest number of project members can attend.

An optimization task is a special kind of search task, in that we're trying to find a thing (object or value) that satisfies some requirements. An optimization task is distinct from a search task in that the answer to an optimization task cannot be checked by looking at the answer and the requirements; it also requires checking that no other possible answer is better. For example, the square root task (a search task), we can determine if a value is correct by checking if squaring it gives us the right answer. In the task of finding the maximum value from a list (an optimization task), we can't tell if it's the largest value just by looking at it; we have to know something about the other values in the list as well.

24.2   Example Problems

In this section we'll describe several example problems, describing their requirements, and giving variations according to the classification scheme discussed in the previous section. Algorithms to solve these problems will be discussed in the next chapter.

As you read through the examples that follow, keep in mind that these are tasks that can be stated fairly simply, and only make use of concepts that we have already studied. As you learn more about computer science, you will encounter a much larger variety!

Also keep in mind the idea of abstraction: the tasks below involve lists and trees and quantities that may not seem to have any hook into a real, practical problem. It would be a mistake for you to

conclude that the tasks have no practical value, simply because they have been stated abstractly. It is because a task can be stated abstractly that a solution to the abstract problem can be applied to any number of real, practical problems. This is a skill that all computer scientists develop: reducing a real problem to an abstract task that represents the essential aspects. Once the trappings of a practical application have been removed, and the essential aspects are revealed, it is very common to discover that the task is well-understood, with well-known algorithms for solving it. Alternatively, it might be similar to a known task, so reading and studying the similar task may suggest a solution. Finally, if the task is not already well-studied, the effort to understand it, and solve it is considered a contribution to science.

24.2.1 Subset Sum

Given a list of numbers, L, and a numeric target value T, find a list of numbers M, taken from L, whose sum is exactly T. For example, let $L = [1, 3, 5, 7]$, and let $T = 8$. One solution is $M = [1, 7]$.

The subset sum problem is a search problem, because we can check any postulated answer M by adding the numbers up and comparing it to T.

24.2.2 Maximum Slice

A slice of a list is any contiguous sequence of numbers in the list, starting at index a and including all numbers at index a up to but not including index b. This is analogous to Python's notion of a slice.

Given a list of numbers, L, find the slice from a to b that has the largest sum of all possible slices of L. For example, let $L = [1, -2, 3, 4, -5]$. The slice from index 2 up to but not including index 4 has a sum equal to 7, which is the largest of all possible slices. In the maximum slice problem, we want the maximum sum, and typically, the range a to b is not part of the answer.

The maximum slice problem is an optimization problem. We can verify that a given slice has a given sum, we cannot know it is the maximum sum just by looking at the slice.

24.2.3 Making Change

Given D, an integer in the range 0 through 99, and a list L of coin values, find a list of integers C, indicating how many of each coin value are needed to have the value of D exactly.

For example, let $D = 37$, and let $L = [1, 5, 10, 25]$, i.e., the coins are pennies (no longer in circulation in Canada), nickel, dimes, and quarters. A solution is $C = [2, 2, 0, 1]$, meaning: 2 pennies, 2 nickels, 0 dimes, and 1 quarter. A quick calculation proves that the solution has monetary value equal to 37.

As stated, the problem is a search task. Given a list of the numbers of coins, and their values, we can check if it the value of those coins is D. We can turn it into an optimization task by asking for the smallest number of coins equal to the value D. In our example above, the solution with the smallest number of coins replaces the 2 nickels with 1 dime: $C = [2, 0, 1, 1]$.

24.2.4 Maximum Tree Path

Given a binary tree T with integer data values, and a target value V, find a path from the root of T to a leaf node on which the sum of the data values is at least V. For example, given the tree in Figure 24.1, and $V = 10$, the path that takes the left branch every time has data values that add up to 11.

As stated, the problem is a search problem: it's easy to verify that the path sum satisfies the requirement. A variant of the task can be stated as a decision problem: is there a path whose sum is

Figure 24.1: A binary tree with integer values.
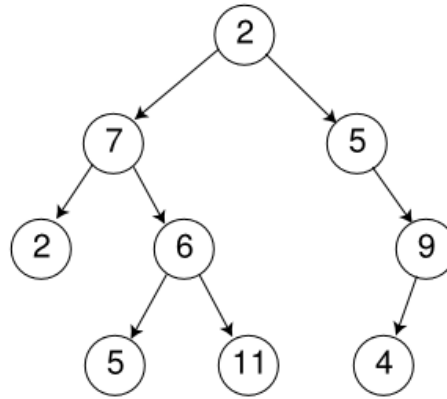
greater than V? A related optimization task would be to ask for the path in the tree whose sum is the largest over all paths from root to leaf. In this variant, we are given the tree T, but not a target value.

Another search variant of the Maximum Tree Path problem is to find a path from root to leaf whose sum is not greater than V.

24.2.5   Leap Line

This task is inspired by side-scroller video games, but is highly simplified. Consider a game avatar controlled by a player, who can make the avatar to run and jump in one direction along a track. The track has rewards (e.g., coins), and penalties (e.g., mushrooms) placed uniformly, and the goal is to collect the highest total score by running through the coins to collect them, and jumping over mushrooms to avoid penalties. The avatar cannot jump past the end of the track.

We can create an abstract representation of this game by using integer values in a list L: positive values for coins, and negative values for mushrooms. The avatar's path through the game is decided by having a choice: either moving from index i to index $i+1$ (a *step*), and adding the value $L[i+1]$ to the player's score, or moving from index i to index $i+2$ (a jump of size 2), collecting only the value $L[i+2]$, avoiding the item at index $i+1$. The Leap Line task is to determine the maximum possible score for the given list of integers L.

The task can be generalized a little more, if we change the distance that the avatar can jump. In general, a step takes the avatar from index i to index $i+1$, and a jump takes the avatar from index i to index $i+k$, where k is a positive integer.

25 — Algorithms

Learning Objectives

After studying this chapter, a student should be able to:

- For each of the strategies (brute-force, greedy, divide-and-conquer, recursive back-tracking, and dynamic programming), identify a practical example to which it would apply.
- Use a greedy approach to solve an appropriate problem and determine if the greedy rule chosen leads to an optimal solution.
- Use a divide-and-conquer algorithm to solve an appropriate problem.
- Use recursive backtracking to solve a problem such as navigating a maze.

25.1 Introduction

In this chapter we'll describe some basic algorithm styles. By studying these styles, you will have a terminology to help you describe algorithms, and also you will have a foot-hold on some general approaches that will help you develop algorithms. These algorithm styles apply to all the task categories presented in the previous chapter.

Designing algorithms is not the same as designing software. We do not insist on robustness, adaptability, or reusability when designing algorithms, even though these are essential for good software. Algorithms may be interesting for their own sake; we might design an algorithm to prove that something can be done, or to provide the basis of some crucial component of software. An algorithm may eventually become part of an application, but the work on designing the algorithm would focus mostly on how to accomplish the task, and not at all on how to turn it into quality software. Once the algorithm has been designed, we can turn it into quality software using the concepts we've studied in previous chapters.

The study of algorithms does not require the use of computers at all, though undoubtedly they

can be quite useful. Some of the most fundamental work on algorithms was done by computer scientists before computers were practical or common. The study of algorithms is the pure science aspect of computer science.

25.2   Algorithm styles

25.2.1   Brute Force Algorithms

A *brute force* algorithm solves a search task or an optimization task by trying every possibility. To design a brute force algorithm, you have to figure out what the possibilities are, and how to work through them systematically. The easiest brute force algorithm that we've seen is linear search through a list: stepping through the possibilities is as simple as writing a for-loop.

Other tasks from the previous chapter are not as easy. For the Making Change problem, the possibilities are the set of all possible combinations of any number of coins. For the Subset Sum task, the possibilities are every possible sub-list of the list L. For the Maximum Tree Path, the possibilities are every path in the tree from root to leaf.

A brute force algorithm does not try to explore the possibilities in a clever way (hence *brute*, implying a deficit of cleverness). However, a brute force algorithm will stop as soon as one of the possibilities satisfies the task's requirements.

The only *force* that a brute force algorithm can apply is the speed of the computer. If the number of possibilities is very large, the time complexity of a brute force algorithm might be so high as to render it useless for anything but small examples. A brute force algorithm is usually the simplest to design for any given problem, and writing one can be useful for deepening your understanding of the task you are trying to accomplish.

25.2.2   Backtracking Algorithms

A *backtracking* algorithm solves a search task or optimization task by organizing the exploration of possibilities using recursion. Typically, a backtracking algorithm explores the space of possibilities by trying all the choices recursively. As a result, a backtracking algorithm may be a brute force algorithm (but not always). For example, a brute force algorithm for the Maximum Tree Path search task (find a path from root to leaf in T whose sum is at least V), would explore the tree using recursion, because trees are inherently recursive.

Backtracking algorithms are not always brute force. Depending on the task, a backtracking algorithm might be able to avoid a very large number of possibilities by checking whether deeper recursion can possibly lead to a solution or not. If it can be determined that a particular choice can not lead to a solution, that choice can be eliminated before it is tried, and more time can be spent exploring other more feasible choices.

25.2.3   Divide and Conquer Algorithms

A *divide and conquer* algorithm solves a task by dividing the task into independent sub-tasks. The independent tasks are usually solved recursively, and the solution to the original task is constructed from the solutions to the sub-tasks. For example, the Quick Sort algorithm is a divide and conquer algorithm: it separates a list into sub-lists, sorts them, and puts the pieces together. A divide and conquer approach is valuable when the computational complexity of solving the sub-tasks is much smaller than the cost of solving the whole problem.

Divide and conquer algorithms are frequently designed as recursive functions. Normal recursive functions that break a task of size N into one task of size $N-1$ and another of size 1 are also divide and conquer algorithms. Typically, thought, the subtasks for a divide and conquer algorithm are substantially smaller than the given task, say two problems of size $N/2$.

25.2.4 Greedy Algorithms

A *greedy* algorithm solves a search task by avoiding the exploration of all the possibilities. To avoid looking at all the possibilities, a greedy algorithm tries to make good choices about the task based on the information that the algorithm has available at the time. The word "tries" in the previous sentence is important, because sometimes, the information that the algorithm has is too limited to allow the algorithm to the right or perfect choice. But sometimes, the algorithm has all the information it needs to make the right choice.

The binary search algorithm is a greedy algorithm. It decides where to look for a given element of a list based on the value at the midpoint of the list. If the list is sorted, the limited information guarantees that binary search will return the correct answer. But if the list is not sorted, the decision to search half of the list only will almost certainly not return the correct answer.

While binary search is familiar, it is not typical of greedy algorithms. More typically, greedy algorithms construct a solution to a search task or an optimization task piece by piece. For example, we can solve the Make Change task by deciding sequentially how many quarters to use, followed by dimes, nickels and pennies. It's greedy because a decision about the answer is made without exploring the possibilities. This particular greedy algorithm makes every choice correctly, and so it can solve the optimization version of this task without trying all possibilities.

A greedy algorithm is also used when exploring all possibilities is completely infeasible, and an answer, even if not the right or best answer, is better than no answer at all. Many algorithms in Artificial Intelligence and Machine Learning are greedy algorithms of this sort.

25.2.5 Dynamic Programming Algorithms

A *dynamic programming* algorithm is a divide and conquer algorithm combined with the technique of memoization. This kind of algorithm is highly valuable if the divide and conquer algorithm would require the re-solving of the same sub-tasks repeatedly, and the memo allows the algorithm to save time by looking up the answer instead of re-computing it.

Consider for example, the Leap Line problem. It is fairly easy to implement a backtracking algorithm that tries every possible combination of stepping and jumping. If we simply implement a memo (i.e., a dictionary that remembers what calculations have already been made), we can avoid a lot of repeated calculations.

The simplest dynamic programming algorithms can be described as being divide and conquer algorithms supplemented by a memo. Many dynamic programming algorithms are not written using recursion, but build up a data structure, similar to our memos, using loops instead of recursion. This is usually done when recursion would incur too high a cost in terms of its use of the system stack, but frequently results in algorithms that are much more difficult to design and understand afterward.

The memoization incurs a memory cost, in exchange for saving runtime costs. This trade-off usually leads to a very effective solution, but one must always analyze these costs to be sure.

25.3 How to solve it

If you are faced with a task that you don't have an algorithm to solve, you might wonder where you should start. This section tries to give some advice, in the form of a sequence of things to try.

1. Try to identify the essential aspects of the problem, independent of the application. You're no doubt trying to solve a problem with a practical application, but it usually pays to strip away the details, and look at the essential aspects. At this point, you may recognize your task as being related to a task you already know how to solve.
2. Determine if your task is a search task, decision task, or optimization task. This will help you focus on what your algorithm needs to do.
3. Design a brute force algorithm for your task. This will help you understand what a solution is, and can be useful for small examples.
4. Try a divide and conquer approach.
5. Design a dynamic programming algorithm, by simply adding a memo to your divide and conquer algorithm, if it does a lot of repeated calculations.
6. Design a greedy algorithm that makes simple choices without trying all possibilities. This might be good enough.
7. Design a backtracking algorithm. It might be brute force, or you might be able to save some work by cutting off fruitless possibilities before they are fully explored.

26 — Algorithm Examples

Learning Objectives

After studying this chapter, a student should be able to:

- To identify examples of different algorithm styles.
- To practice abstraction and generalization for specific tasks.
- To become aware of the kinds of mathematical tools that computer scientists use to express abstractions.

26.1 Introduction

The best way to improve your algorithm design skills is to study other algorithms for tasks you haven't seen before. Learning to extract the essential aspects of a task will help you recognize similarities. It's also important to learn about the kinds of mathematical tools and notation that computer scientists use to express abstractions: mathematical logic, set theory, graphs, etc. This chapter will present some algorithms for the tasks first described in Chapter 24.

26.2 Subset Sum

Given a list of numbers, L, and a numeric target value T, find a list of numbers M, taken from L, whose sum is exactly T. For example, let $L = [1,3,5,7]$, and let $T = 8$. One solution is $M = [1,7]$.

    The subset sum problem is a search problem, because we can check any postulated answer M by adding the numbers up and comparing it to T.

26.2.1 Brute Force

For the subset sum problem, the space of possibilities is the space of all possible sublists of the given list. A brute force algorithm has to create every possible sublist of the of the given list, and to

try them all, until one of the sublists has the target sum. This is most easily done using recursion. The internal function, `trysum()`, shown below, has a recursive case (lines 17-24) that makes two recursive calls. First (line 19), it tries to use the first element in the given list. If an answer can be found using that element, the answer is returned (lines 20-21). If not, the function is called again (line 24), this time not using the first element.

```python
 1  def subsetsum_v1(alist, target):
 2      """
 3      Given a list of numbers, find a subset whose
 4      sum is equal to the target.
 5      """
 6
 7      def trysum(alist, subset):
 8          """
 9          Recursively construct a subset from alist, checking
10          if the sum is equal to the target.
11          """
12          if sum(subset) == target:
13              return True, subset
14          elif len(alist) == 0:
15              return False, None
16          else:
17              tryit = alist[0]
18              # try to find a subset sum using the first value
19              flag, answer = trysum(alist[1:], subset + [tryit])
20              if flag:
21                  return flag, answer
22              else:
23                  # try without the first value
24                  return trysum(alist[1:], subset)
25
26      flag, result = trysum(alist, [])
27      if flag: return result
28      else: return None
```

This technique of trying all possible subsets by including or excluding an element is fundamental to problems involving subsets or sublists. You should imagine the collection of all possible subsets, organized like a binary tree. The root of this imaginary tree is empty, since no elements have been added. The left branch is created by putting the first element into the sublist; the right branch is created by excluding the first element. All possible subsets are constructed by either including or excluding elements from the original list. Figure 26.1 shows a small example of a tree of all the subsets of the list $[1,2,3]$.

The worst case time complexity of the brute-force algorithm is $O(2^N)$, because there are 2^N sublists of a list of size N.

Figure 26.1: A tree of all the subsets of the list $[1, 2, 3]$, that would be created by the function `trysum()` for the Subset Sum task.
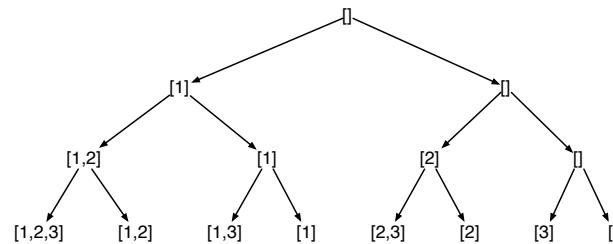
26.2.2 Divide-and-conquer

A divide and conquer algorithm can be designed for the subset sum task. The idea is pretty simple: A recursive function named `binary()` (lines 8-24) tries to find a subset with the appropriate sum using only elements from the first half of the list (line 16). If an answer cannot be found in the first half of the list, the algorithm will try using only the second half of the list (line 20). If neither of those work, the brute-force algorithm as described above will be used on the whole list (line 24). Notice the base case of the `binary()` function. For lists of size 5 or smaller, a brute force algorithm, `trysum()` is performed.

The code is shown below. To keep the listing as short as possible, the listing does not include the `trysum()` internal function, which is called on line 24. It's identical to the function of the same name in the brute force backtracking algorithm from the previous section.

```
1  def subsetsum_v2(alist, target):
2      """
3      Given a list of numbers, find a subset whose
4      sum is equal to the target.
5      """
6      # this does divide and conquer, in the form of
7      # dividing the list in half
8      def binary(alist):
9          if len(alist) < 5:
10             # for small lists, just use brute force!
11             return trysum(alist, [])
12         else:
13             mid = len(alist)//2
14
15             # try to find a solution in a list half the size
16             flag, ans = binary(alist[:mid])
17             if flag: return True, ans
18
19             # if not, try the other half
20             flag, ans = binary(alist[mid + 1:])
21             if flag: return True, ans
22
```

```
23              # if not, try the whole list
24              return trysum(alist, [])
25
26     flag, result = binary(alist)
27     if flag: return result
28     else: return None
```

In the worst case, trying to find a solution in lists of half the size may result in no solution, and so the divide-and-conquer algorithm resorts to using brute force on the whole list. As a result, the worst case time complexity is the same as for the brute force algorithm above. But the algorithm tends to be very fast in a limited range of experiments. This suggests that an average case time complexity analysis may reveal that the algorithm belongs in a lower order complexity class. The average case analysis is beyond the scope of the course!

26.3   Maximum Slice

Given a list of numbers, *L,* find the slice from *a* to *b* that has the largest sum of all possible slices of *L.* For example, let $L = [1, -2, 3, 4, -5]$. The slice from index 2 up to but not including index 4 has a sum equal to 7, which is the largest of all possible slices. In the maximum slice problem, we want the maximum sum, and typically, the range *a* to *b* is not part of the answer.

The maximum slice problem is an optimization problem.

26.3.1   Brute Force

For the maximum slice problem, the set of all possibilities is the set of all slices. A slice can start at any element in the list. For each starting element, the slice can end at any element after the starting element. In other words, there are *N* slices starting at index 0, $N-1$ slices starting at index 1, etc. In total, there are $O(N^2)$ possible slices of a list of size *N.*

In the brute force algorithm shown below, a couple of for-loops are used to control the start and end of the slices (lines 9-13). Once the start and end of a slice is determined, the sum can be calculated using Python's `sum()` function (line 11).

```
1  def maxslice_brute_force_v0(alist):
2      """
3      Find the maximum sum of all slices of alist.
4      :param alist: a list of numbers
5      :return: a number, the maximum slice sum
6      """
7      # using brute force: look at all possible slices
8      maxsum = alist[0]
9      for i in range(len(alist)):
10         for j in range(len(alist)):
11             slice = sum(alist[i:j + 1])
12             if slice > maxsum:
13                 maxsum = slice
14     return maxsum
```

Figure 26.2: An example of the use of a dictionary to store slice sums in the Maximum Slice task. Calculating the sum of the slice $[3, 4, -5]$ can be done by storing sums of a smaller slice $[4, -5]$ in a dictionary, and using them when needed.

$$s[4, -5] = sum([4, -5])$$

$$[1, -2, 3, 4, -5]$$

$$s[3, 5] = sum([3, 4, -5])$$
$$= 3 + s[4, -5]$$
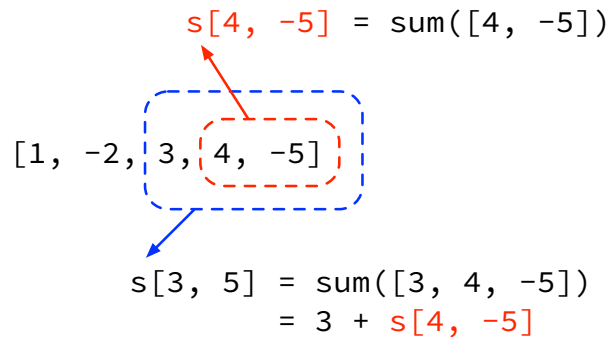
Because this is an optimization problem, the brute force algorithm has to step through every possible slice, keeping track of the largest sum it has seen so far (lines 12-13).

The worst case time complexity of this implementation is $O(N^3)$, because the sum of each slice (line 11) requires $O(N)$ steps, and there are $O(N^2)$ slices.

A substantial improvement to this algorithm could be made by employing an idea from dynamic programming. Consider what happens after computing the sum of the slice from index 2 to index 4 (just to be concrete). Some time later, when the starting index is increased, the brute force algorithm has to calculate the sum of the slice from index 3 to index 4. Notice that the elements in this slice were already added up when calculating the sum from 2 to 4. In other words, the brute force algorithm given above recomputes a lot of sums on line 11. We can avoid recomputing any sums by storing the partial sums in a dictionary. See Figure 26.2. In the code below, a dictionary named s is used to store slice sums from index i to index j.

```
 1  def maxslice_brute_force_v1(alist):
 2      """
 3      Find the maximum sum of all slices of alist.
 4      :param alist: a list of numbers
 5      :return: a number, the maximum slice sum
 6      """
 7      # using brute force: look at all possible slices
 8      # but store all the partial sums in a dictionary
 9      # where s[i,j] stores the value sum(alist[i,j+1])
10      s = {}
11      maxsum = alist[0]
12      for i in range(len(alist)):
13          s[i,i] = alist[i]
14          if s[i,i] > maxsum: maxsum = s[i,i]
15          for j in range(i+1, len(alist)):
16              s[i, j] = s[i,j-1] + alist[j]
17              if s[i,j] > maxsum: maxsum = s[i,j]
```

```
18        return maxsum
```

Because the algorithm computes slice sums by using the diction s, the cost of computing the sum of any slice is $O(1)$, make the time complexity of the improved brute force algorithm $O(N^2)$. It uses dynamic programming to save time, but it is still a brute force algorithm because it has to check all the possibilities.

26.3.2 Divide and conquer

To apply divide and conquer to a task, it's a good idea to consider what you could do if you split your task into two roughly equal subtasks. For the Maximum Slice task, we have to imagine what could happen to the maximum slice (even though we don't know where it is). It's possible that the maximum slice is contained entirely in the first half of the list, in which case, a recursive function call with the first half of the list will find it. Similarly, it might be in the second half of the list, so a recursive function call with the second half of the list will find it. It's also possible that the maximum slice starts in the first half of the list, and extends into the second half.

The problem is that we don't know where the maximum slice is! So we will try all three possibilities. Our divide and conquer will find the maximum slice for the first half of the list, the last half of the list, and the maximum slice that extends through the middle, and it will return whichever one of these three is better.

In the code below, the function `maxslice_rec()` (lines 36-51) recursively tries to find the maximum slice in both halves (lines 46, 47), and through the middle (lines 48-49).

```
1   def maxslice_DC(alist):
2       """
3       Find the maximum sum of all slices of alist.
4       :param alist: a list of numbers
5       :return: a number, the maximum slice sum
6       """
7       # using divide and conquer
8
9       def max_tail(left, right):
10          """
11          Calculate the maximum slice that ends at right
12          (from any point starting at left or later)
13          """
14          s = {}
15          s[right] = alist[right]
16          maxsum = s[right]
17          # calculate the sums from right to left (backwards)
18          for i in range(right - 1, left - 1, -1):
19              s[i] = s[i + 1] + alist[i]
20              if s[i] > maxsum: maxsum = s[i]
21          return maxsum
22
23      def max_head(left, right):
24          """
```

```
25          Calculate the maximum slice that starts at left
26          (to any point up to and including right)
27          """
28          s = {}
29          s[left] = alist[left]
30          maxsum = s[left]
31          for i in range(left + 1, right + 1):
32              s[i] = s[i - 1] + alist[i]
33              if s[i] > maxsum: maxsum = s[i]
34          return maxsum
35
36      def maxslice_rec(left, right):
37          """
38          Recursively find the maximum slice between left and right.
39          """
40          # using divide and conquer
41          if left == right:
42              return alist[left]
43          else:
44              # divide, and solve
45              mid = (right + left) // 2
46              max_left  = maxslice_rec(left, mid)
47              max_right = maxslice_rec(mid + 1, right)
48              max_cross = (max_tail(left, mid)
49                              + max_head(mid + 1, right))
50              # conquer
51              return max(max_left, max_right, max_cross)
52
53      # body of maxslice_DC
54      return maxslice_rec(0, len(alist) - 1)
```

The interesting part of the algorithm is how the maximum slice is computed when it passes through the middle of the list. We've defined two internal functions to manage this. The function `max_tail()` (lines 9-21) uses a dictionary and starts from the middle and steps towards the left side of the list, storing slice sums in the dictionary. The function `max_head()` (lines 23-34) uses a dictionary and starts from the middle and steps towards the right side of the list, storing slice sums in the dictionary. A careful reading of these two functions tells us that both functions have worst case time complexity of $O(N)$. In other words, focussing our search on a slice that passes through the middle of the list means we're ignore very many slices; the combination of `max_tail()` and `max_head()` only looks at a very small subset of all the possible slices.

The worst case time complexity of the divide and conquer algorithm can be shown to be $O(N \log N)$. The course has not developed the mathematics needed to reach this conclusion, but we can wave our hands a bit. Consider an imaginary binary tree (again), starting with the whole list as the root, and each branch taking the left half or the right half of the list. We can split the list in half at most $O(\log N)$ times, so the tree has at $O(\log N)$ levels. One important thing to notice is that every element in the original list appears exactly once in every level of the tree. The other thing to notice

Figure 26.3: An example showing how `max_tail()` and `max_head()` work together to calculate the maximum slice that passes through the middle of a list. This technique is part of the divide and conquer algorithm for Maximum Slice.

[1, -2, 3, 4, -5, 1, 2,｜3, 4, 5, -6, -7, -8]
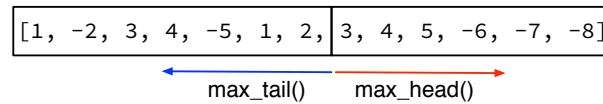
max_tail()   max_head()

is that `max_tail()` and `max_head()` are called on every level in the tree. Since these functions have time complexity $O(N)$, we can see that the total amount of work on each level is also $O(N)$. Considering there are $O(\log N)$ levels in the imaginary tree, we get the result $O(N \log N)$. This kind of analysis is a bit beyond what we're hoping for in this course, but the tools you need for this kind of thing will be obtained from further computer science courses.

26.4 Make Change

Given D, an integer in the range 0 through 99, and a list L of coin values, find a list of integers C, indicating how many of each coin value are needed to have the value of D exactly.

For example, let $D = 37$, and let $L = [1, 5, 10, 25]$, i.e., the coins are pennies (no longer in circulation in Canada), nickel, dimes, and quarters. A solution is $C = [2, 2, 0, 1]$, meaning: 2 pennies, 2 nickels, 0 dimes, and 1 quarter. A quick calculation proves that the solution has monetary value equal to 37.

As stated, the problem is a search task. Given a list of the numbers of coins, and their values, we can check if it the value of those coins is D. We can turn it into an optimization task by asking for the smallest number of coins equal to the value D. In our example above, the solution with the smallest number of coins replaces the 2 nickels with 1 dime: $C = [2, 0, 1, 1]$.

26.4.1 Greedy

A greedy algorithm is used to avoid systematically considering every possibility. It does so by making choices about which possibility to consider using the information it has available. A greedy algorithm for the Make Change task simply calculates how many coins of each value that it needs. In our example, we assume the use of quarters (25), dimes (10), nickels (5), and pennies (1).

The greedy algorithm considers these coins in order of decreasing value, i.e., quarters, dimes, nickels, pennies. We can calculate how many quarters we need by dividing the amount by the value of the coin, using integer division (line 13). Taking the remainder of division by the value of a coin (line 15) tells us how much remains to be accounted for by smaller coins.

```
1  def change_v2(cents):
2      """
3      Make change for the given cents value.
4      Assumes coin values 25c, 10c, 5c, 1c
5      :param cents: an integer
6      :return: a list of counts for the coins used.
7      """
8      coins = [25, 10, 5, 1]
```

```
 9       coin_index = 0
10       counts = [0] * len(coins)
11       remaining = cents
12       while remaining > 0:
13           n = remaining // coins[coin_index]
14           counts[coin_index] = n
15           remaining = remaining % coins[coin_index]
16           coin_index += 1
17       return counts
```

It's interesting to consider the time complexity of this algorithm. The loop on lines 12-16 repeats the calculation once for each coin, and there are 4 coins. So the time complexity is $O(1)$. This explains why even children can master this algorithm very easily. It's fast, and doesn't require a lot of memory!

You might consider how many combinations of coins there are that have a total value of D. A brute force algorithm to explore all of these possible combinations would be quite a bit more expensive in terms of time!

26.5 Maximum Tree Path

Given a binary tree T with integer data values, and a target value V, find a path from the root of T to a leaf node on which the sum of the data values is at least V. For example, given the tree in Figure 24.1, and $V = 10$, the path that takes the left branch every time has data values that add up to 11. As stated, the problem is a search problem: it's easy to verify that the path sum satisfies the requirement.

26.5.1 Brute force

To find the maximum sum of data values from root to leaf in a given tree, we have to reach each leaf, and check the sum on the path from the root. This is most easily done recursively, by determining the maximum sum from the current node through one of its children. This idea is implemented in the `max_path()` function below.

```python
def max_path(atree):
    """
    Find the maximum sum of all paths from root to leaf
    in atree.
    :param atree: A TreeNode
    :return: the maximum sum
    """
    if atree is None:
        return 0
    else:
        left = max_path(atree.left)
        right = max_path(atree.right)
        return max(left, right) + atree.data
```

The computational complexity of this algorithm depends on the size of the tree. Each node in the tree is added to a sum exactly once, so if there are N nodes in the tree, the algorithm has $O(N)$ time complexity. The shape of the tree doesn't matter, nor does the arrangement of the values.

26.5.2 Greedy

A greedy algorithm for this task serves mainly as an example of what kinds of decisions could be made to avoid systematically working through every possibility. As we noted above, the brute force algorithm for this task is $O(N)$, which is completely feasible. We would not need to avoid the work in this task.

The greedy algorithm chooses a single path from root to leaf by checking the values stored at its children. The greedy choice, which illustrates the reason why it's called "greedy", is to choose the path that leads to the child with the largest value.

```python
def greedy_path(atree):
    """
    Find the maximum sum of all paths from root to leaf
    in atree.
    :param atree: A TreeNode
    :return: the maximum sum
    """
    if atree.is_leaf():
        return atree.data
    elif atree.left.data > atree.right.data:
        # the left branch seems promising
        return atree.data + greedy_path(atree.left)
    else:
        # the right branch seems promising
        return atree.data + greedy_path(atree.right)
```

The greedy algorithm can easily make a choice that is not correct. For example, the binary tree in Figure 26.4 has high value leaf nodes below the node with value 2, and low value leaf nodes below the node with value 3. The greedy algorithm will choose the right branch from the root, because 3 is bigger than 2.

This example shows that a greedy choice does not always lead to the correct answer. The information available at the time of the decision was not sufficient to guarantee the right answer.

The computational complexity of the greedy search is the length of a path from root to leaf, i.e., the height of the tree. Depending on the way the tree branches, the height of the binary tree could be as low as $O(\log N)$, or as high as $O(N)$, where N is the number of nodes in the tree. So in the worst case, the computational complexity is $O(N)$.

It seems important to emphasize that a greedy algorithm like this can be useful when it is infeasible to explore every possibility. The real value of this example is demonstrating how the information available to an algorithm could be used, and that it might not be enough to obtain a correct or best solution.

Figure 26.4: A binary tree with high values hidden below a node with a relatively low value. A greedy algorithm for Maximum Tree Path will not return the correct result.
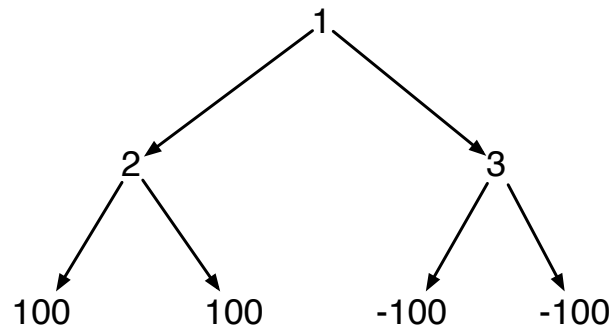
26.6 Leap Line

We are given a list *L* of positive and negative integers. A path through *L* is decided by having a choice at every point in *L*: either moving from index i to index $i+1$ (a *step*), and adding the value $L[i+1]$ to the score, or moving from index i to index $i+2$ (a jump of size 2), collecting only the value at $L[i+2]$, avoiding the item at index $i+1$. The Leap Line task is to determine the maximum possible score for the given list of integers *L*.

26.6.1 Brute Force

A brute force algorithm for this task requires systematic examination of every possible path through the list. The algorithm below accomplishes this using recursion. At every index in the list, there are 2 possibilities: take a step from index i to index $i+1$, or take a jump from index i to index $i+2$. The best path from index i is the better of the two choices. The base case for the recursion considers the options available when the avatar is near the end of the list.

The internal function `leap_or_step()` systematically checks every path using recursion.

```
 1  def maximumScoreFrom_v0(track):
 2      """
 3      Calculate the maximum score that can be obtained from
 4      stepping and leaping along the given track
 5      :param track: a list of integers
 6      :return: the maximum score
 7      """
 8
 9      def leap_or_step(loc):
10          """
11          Calculate the maximum score that can be obtained from
12          stepping and leaping along the given track, starting
13          at the given location
14          """
15          step_loc = loc + 1
16          leap_loc = loc + 2
```

```
17
18              if step_loc == len(track):
19                  return track[loc]
20              elif leap_loc >= len(track):
21                  return track[loc] + leap_or_step(step_loc)
22              else:
23                  return track[loc] + max(leap_or_step(step_loc),
24                                          leap_or_step(leap_loc))
25
26          return leap_or_step(0)
```

As with all brute force algorithms, the worst case time complexity depends on the number of possibilities. As with any recursive function that has 2 recursive calls in the recursive case, the computations can be modelled by an imaginary binary tree. Most of the nodes in the imaginary tree have 2 children, so the number of nodes in the tree is $O(2^h)$ where h is the height of the tree. When the choice is between stepping $(+1)$ or jumping $(+2)$ through a list of size N, there will be leaf nodes from height $N/2$ down to N. So $h = O(N)$, and thus the time complexity is $O(2^N)$.

26.6.2 Dynamic Programming

It's really instructive to see how adding a memo can help an algorithm like this. In this case, the memo keeps track of the value of the best path starting from index i. Because there is only one list, about half of all paths will step on index i, and the other half will jump over i. In any case, the brute force algorithmm is expensive because it recomputes the best path starting from index i every time a possible path reaches index i. The memo avoid the recomputation by storing the value of the best path starting from i the first time index i is encountered, and using a lookup for the value every other time.

The following code is essentially the brute force algorithm with an added memo. See line 10 for the creation of the dictionary, lines 19-20 to check to see if the value from i has already been computed once before, and 28-29 for storing the value in the memo the first time.

```
1   def maximumScoreFrom_v1(track):
2       '''
3       Calculate the maximum score that can be obtained from
4       stepping and leaping along the given track
5       :param track: a list of integers
6       :return: the maximum score
7       '''
8       # using memoization
9       # memo[loc] stores the best score starting from loc.
10      memo = {}
11
12      def leap_or_step(loc):
13          """
14          Calculate the maximum score that can be obtained from
15          stepping and leaping along the given track, starting
16          at the given location
```

```
17              """
18              # check if the best score is already known
19              if loc in memo:
20                  return memo[loc]
21
22              step_loc = loc + 1
23              leap_loc = loc + 2
24
25              if step_loc == len(track):
26                  return track[loc]
27              elif leap_loc >= len(track):
28                  result = track[loc] + leap_or_step(step_loc)
29                  memo[loc] = result
30                  print('memo', loc, result)
31                  return result
32              else:
33                  result = track[loc] + max(leap_or_step(step_loc),
34                                            leap_or_step(leap_loc))
35                  memo[loc] = result
36                  print('memo', loc, result)
37                  return result
38
39          return leap_or_step(0)
```

With a memo, it can be shown that the dynamic programming algorithm requires $O(N)$ time.

26.6.3 Greedy

It turns out that under certain conditions, a greedy solution can obtain the correct result for the Leap Line problem. The algorithm shown below chooses to step to $i+1$ or leap to $i+2$ depending on whether the value at $i+1$ is positive (step) or negative (leap). See lines 22-27.

```
1  def maximumScoreFrom_v2(track):
2      """
3      Calculate the maximum score that can be obtained from
4      stepping and leaping along the given track
5      :param track: a list of integers
6      :return: the maximum score
7      """
8
9      def leap_or_step(loc):
10          """
11          Calculate the maximum score that can be obtained from
12          stepping and leaping along the given track, starting
13          at the given location
14          """
15          step_loc = loc + 1
```

```python
16            leap_loc = loc + 2
17
18            if step_loc == len(track):
19                return track[loc]
20            elif leap_loc >= len(track):
21                return track[loc] + leap_or_step(step_loc)
22            elif track[step_loc] < 0:
23                # greedy choice: avoid negatives
24                return track[loc] + leap_or_step(leap_loc)
25            else:
26                # greedy choice: collect 1
27                return track[loc] + leap_or_step(step_loc)
28
29        return leap_or_step(0)
```

The greedy algorithm looks only at one possibility, and since the list has N elements, the greedy algorithm has worst case time complexity of $O(N)$.

The greedy algorithm will obtain the correct maximum score if the list contains only 2 values, e.g., -1 and 1, and the leap is limited to $i + 2$. If the values in the list have a more variety, or if the leaps are longer, a greedy solution cannot find the correct maximum score. This claim will be stated without any justification, but the tools you need to do the analysis are part of future computer science courses.

Part I

Appendices

A — Abstract Data Types

A.1 The full ADT Registry implementation

The node-based queue is discussed in Chapter 6. Here we present the implementation without remark.

```python
# CMPT 145: Abstract Data Types
# Defines the Registry ADT

#  A registry records a Boolean value for a range of integers.
#  It can be used to record observations or events.

# Implementation:
#  - a simple list of Boolean values
#  - set() and reset() are standard names for Boolean operations.


def create(size, value):
    """
    Purpose:
        Create a registry of a give size filled with the given value
    Pre-conditions:
        size: number of elements in the registry
        value: the default initial value for all elements
    Post-conditions:
        (none)
    Return:
        the newly created registry
    """
    return [value for i in range(size)]
```

```python
def set(reg, i):
    """
    Purpose:
        Set the registry element at i to True
    Pre-conditions:
        reg: the registry
        i: an index, in the correct range for reg
    Post-conditions:
        the registry value at i is set to True
    Return:
        (none)
    """
    reg[i] = True

def reset(reg,i):
    """
    Purpose:
        Set the registry element at i to False
    Pre-conditions:
        reg: the registry
        i: an index, in the correct range for reg
    Post-conditions:
        the registry value at i is set to False
    Return:
        (none)
    """
    reg[i] = False

def is_registered(reg, i):
    """
    Purpose:
        Returns the value stored at registry element i
    Pre-conditions:
        reg: the registry
        i: an index, in the correct range for reg
    Post-conditions:
        (none)
    Return:
        the value stored at element i
    """
    return reg[i]
```

A.2   The full ADT Statistics implementation

The node-based queue is discussed in Chapter 6. Here we present the implementation without remark.

```python
# CMPT 145: Abstract Data Types
# Defines the Statistics ADT
# Calculate mean and variance.

# Implementation
# Do the calculations without storing all the data!
# Uses a dictionary as a record to store three quantities:
#    'count':     the number of data values added
#    'avg':       the running average of the values added
#    'sumsqdiff': the sum of the square differences between the
#                 values added and the mean so far
# These values can be modified every time a new data value is
# added, so that the mean and variance can be calculated quickly
# as needed.  This approach means that we do not need to store
# the data values themselves, which could save a lot of space.


def create():
    """
    Purpose:
        Create a Statistics record.
    Pre-conditions:
        (none)
    Post-conditions:
        a new record is allocated
    Return:
        A reference to a Statistics record.
    """
    b = {}
    b['count'] = 0       # how many data values have been seen
    b['avg'] = 0         # the running average so far
    b['sumsqdiff'] = 0   # the sum of the square differences
    return b


def add(stat, value):
    """
    Purpose:
        Use the given value in the calculation of statistics.
    Pre-Conditions:
        stat: a Statistics record
        value: the value to be added
```

```
    Post-Conditions:
        none
    Return:
        none
    """
    stat['count'] += 1
    k = stat['count']              # convenience
    diff = value - stat['avg']   # convenience
    stat['avg'] += diff/k
    stat['sumsqdiff'] += ((k-1)/k)*(diff**2)


def mean(stat):
    """
    Purpose:
        Return the mean of all the values seen so far.
    Pre-conditions:
        stat: the Statistics record
    Post-conditions:
        (none)
    Return:
        The mean of the data seen so far.
        Note: if no data has been seen, 0 is returned.
             This is clearly false.
    """
    return stat['avg']


def var(stat):
    """
    Purpose:
        Return the variance of all the values seen so far.
        (variance is the average of the squared difference
        between each value and the average of all values)
    Pre-conditions:
        stat: the Statistics record
    Post-conditions:
        (none)
    Return:
        The variance of the data seen so far.
        Note: if 0 or 1 data values have been seen, 0 is returned.
             This is clearly false.
    """
    return stat['sumsqdiff']/stat['count']
```

```python
def sampvar(stat):
    """
    Purpose:
        Return the sample variance of all the values seen so far.
    Pre-conditions:
        stat: the Statistics record
    Post-conditions:
        (none)
    Return:
        The sample variance of the data seen so far.
        Note: if 0 or 1 data values have been seen, 0 is returned.
            This is clearly false.
    """
    return stat['sumsqdiff']/(stat['count'] - 1)
```

B — Applications of Stacks and Queues

B.1 Queueing simulation

The queueing simulation, based on the concept of a M/M/1 queue, is discussed in Chapter 9. Here we present the implementation without remark.

```python
# CMPT 145: Linear ADTs
# Application: M/M/1 Queuing simulation
# (based on example by Sedgewick, Wayne, Dondero)
#
# Simulate the arrival and service of customers using a queue.
# Calculate the average wait time for customers, given
# assumptions about the rate of arrival and service.

import TQueue as Queue
import Statistics as Statistics
import random as rand

# the assumptions
arrival_rate = 20/60  # the number of customer arrivals per minute
service_rate = 20/60  # the number of customers served per minute
sim_length = 8*60  # the end of the simulation in minutes


def sample_time(x):
    '''
    Return a random sample time until an event.
    In the long run, the events will have a rate of x
    events per unit time.
```

```python
    Preconditions:
        x: the desired arrival rate, per unit time
    Post-conditions:
        none
    Return:
        a random sample time that obeys the rate x
    '''
    return rand.expovariate(x)


# customers wait in a service queue
service_queue = Queue.create()

# we're interested in the average time in the queue
waiting = Statistics.create()

# keep track of when a new customer will arrive
nextArrival = sample_time(arrival_rate)

# keep track of when a service event is complete
nextService = nextArrival + sample_time(service_rate)


# simulate the arrival-service process
while nextService < sim_length:

    # handle all arrivals that occur before the service is complete
    while nextArrival < nextService:
        Queue.enqueue(service_queue, nextArrival)
        nextArrival = nextArrival + sample_time(arrival_rate)

    # service time is over; serve the first customer in the queue
    this_arrival = Queue.dequeue(service_queue)

    # how long was the customer waiting?
    waited = nextService - this_arrival
    Statistics.add(waiting, waited)

    # determine when service to the next customer will end
    if Queue.is_empty(service_queue):
        nextService = nextArrival + sample_time(service_rate)
    else:
        nextService = nextService + sample_time(service_rate)

print('Average wait time:', Statistics.mean(waiting))
```

B.2 Bracket checking

The bracket-checking program is discussed in Chapter 9. Here we present the implementation without remark.

```python
# CMPT 145: Linear ADTs
# Application: Bracket Matching Algorithm
#
# Check a string for matching brackets
# Uses the Queue and Stack ADTs

import MyQueue as MyQueue
import Stack as Stack


example = '( ( ( ) ( ) ) )'

# create the initial empty containers
chars    = MyQueue.create()
brackets = Stack.create()
unmatched_close = False


# put all the characters in the Queue
for c in example:
    MyQueue.enqueue(chars, c)


# brackets match if and only if every '(' has
# a corresponding ')'
while not MyQueue.is_empty(chars) and not unmatched_close:
    c = MyQueue.dequeue(chars)
    if c == '(':
        Stack.push(brackets,c)
    elif c == ')' and not Stack.is_empty(brackets):
        Stack.pop(brackets)
    elif c == ')' and Stack.is_empty(brackets):
        unmatched_close = True
    else:
        pass


# check how the analysis turned out
if unmatched_close:
    print("Found a ')' with no matching '('")
elif not Stack.is_empty(brackets):
    print("At least one '(' without a matching ')'")
else:
```

```
        print('Brackets matched')
```

B.3 Post-fix Arithmetic evaluator

The post-fix arithmetic evaluator is discussed in Chapter 9. Here we present the implementation without remark.

```python
# CMPT 145: Linear ADTs
# Post-fix arithmetic evaluation

# An application of the Stack ADT
# and the Queue ADT

import MyQueue as Queue
import Stack as Stack


def evaluate(expr):
    """
    Evaluate a postfix expression.
    Pre-conditions:
        expr: a string containing a postfix expression
    Post-Conditions:
        none
    Return:
        the value of the expression
    """

    # create the initial empty data structures
    expression = Queue.create()
    evaluation = Stack.create()

    expr_list = expr.split()
    # put all the items in the Queue
    for c in expr_list:
        Queue.enqueue(expression, c)

    # evaluate the expression
    while not Queue.is_empty(expression):
        c = Queue.dequeue(expression)

        if c == '*':
            v1 = Stack.pop(evaluation)
            v2 = Stack.pop(evaluation)
            Stack.push(evaluation, v1*v2)
        elif c == '/':
```

```python
            v1 = Stack.pop(evaluation)
            v2 = Stack.pop(evaluation)
            Stack.push(evaluation, v2/v1)
        elif c == '+':
            v1 = Stack.pop(evaluation)
            v2 = Stack.pop(evaluation)
            Stack.push(evaluation, v1+v2)
        elif c == '-':
            v1 = Stack.pop(evaluation)
            v2 = Stack.pop(evaluation)
            Stack.push(evaluation, v2-v1)
        else:
            Stack.push(evaluation, float(c))

    return Stack.pop(evaluation)


example = "f 56 + "
print('('+example+')', '=',evaluate(example))
```

C — List-based Stacks and Queues

C.1 The full list-based Queue implementation

The list-based queue is discussed in Chapter 10. Here we present the implementation without remark.

```
# CMPT 145: Linear ADTs
# Implements the Queue ADT
#
# A queue (also called a FIFO queue) is a compound data
# structure in which the data values are ordered according to
# the FIFO (first-in first-out) protocol.
#
# Implementation notes:
# This implementation uses Python lists directly.  Essentially,
# it adapts the list and imposes the queue protocol upon it.



def create():
    """
    Purpose
        creates an empty queue
    Return
        an empty queue
    """
    return list()


def is_empty(queue):
```

```
    """
    Purpose
        checks if the given queue has no data in it
    Pre-conditions:
        queue is a queue created by create()
    Return:
        True if the queue has no data, or false otherwise
    """
    return len(queue) == 0


def size(queue):
    """
    Purpose
        returns the number of data values in the given queue
    Pre-conditions:
        queue: a queue created by create()
    Return:
        The number of data values in the queue
    """
    return len(queue)


def enqueue(queue, value):
    """
    Purpose
        adds the given data value to the given queue
    Pre-conditions:
        queue: a queue created by create()
        value: data to be added
    Post-condition:
        the value is added to the queue
    Return:
        (none)
    """
    queue.append(value)


def dequeue(queue):
    """
    Purpose
        removes and returns a data value from the given queue
    Pre-conditions:
        queue: a queue created by create()
    Post-condition:
        the first value is removed from the queue
```

```
    Return:
        the first value in the queue
    """
    return queue.pop(0)

def peek(queue):
    """
    Purpose
        returns the value from the front of given queue
        without removing it
    Pre-conditions:
        queue: a queue created by create()
    Post-condition:
        None
    Return:
        the value at the front of the queue
    """
    return queue[0]
```

C.2 The full list-based Stack implementation

The list-based stack is discussed in Chapter 10. Here we present the implementation without remark.

```
# CMPT 145: Linear ADTs
# Defines the Stack ADT
#
# A stack (also called a pushdown or LIFO stack) is a compound
# data structure in which the data values are ordered according
# to the LIFO (last-in first-out) protocol.
#
# Implementation:
# This implementation uses Python lists directly.  Essentially,
# it adapts the list and imposes the stack protocol upon it.
# Here, stack grows from 0, and push/pop happen at the "end" of
# the list


def create():
    """
    Purpose
        creates an empty stack
    Return
        an empty stack
    """
    return list()
```

```python
def is_empty(stack):
    """
    Purpose
        checks if the given stack has no data in it
    Pre-conditions:
        stack is a stack created by create()
    Return:
        True if the stack has no data, or false otherwise
    """
    return len(stack) == 0


def size(stack):
    """
    Purpose
        returns the number of data values in the given stack
    Pre-conditions:
        stack: a stack created by create()
    Return:
        The number of data values in the queue
    """
    return len(stack)


def push(stack, value):
    """
    Purpose
        adds the given data value to the given stack
    Pre-conditions:
        queue: a stack created by create()
        value: data to be added
    Post-condition:
        the value is added to the stack
    Return:
        (none)
    """
    stack.append(value)


def pop(stack):
    """
    Purpose
        removes and returns a data value from the given stack
    Pre-conditions:
        stack: a stack created by create()
```

```
    Post-condition:
        the top value is removed from the stack
    Return:
        returns the value at the top of the stack
    """
    return stack.pop()

def peek(stack):
    """
    Purpose
        returns the value from the front of given stack
        without removing it
    Pre-conditions:
        stack: a stack created by create()
    Post-condition:
        None
    Return:
        the value at the front of the stack
    """
    return stack[-1]
```

D — Node-based Stacks and Queues

D.1 The full list-based Queue implementation

The node-based queue is discussed in Chapter 13. Here we present the implementation without remark.

```python
# CMPT 145: Node-based Queues
# Implements the Queue ADT
#
# A queue (also called a FIFO queue) is a compound data
# structure in which the data values are ordered according to
# the FIFO (first-in first-out) protocol.
#
# Implementation notes:
# This implementation uses the linked node structure.


import node as node

def create():
    """
    Purpose
        creates an empty queue
    Return
        an empty queue
    """
    queue = {}
    queue['size'] = 0          # how many elements in the queue
    queue['front'] = None      # the node chain starts here
    queue['back'] = None       # the node chain ends here
```

```python
    return queue


def size(queue):
    """
    Purpose
        returns the number of data values in the given queue
    Pre-conditions:
        queue: a queue created by create()
    Return:
        The number of data values in the queue
    """
    return queue['size']


def is_empty(queue):
    """
    Purpose
        checks if the given queue has no data in it
    Pre-conditions:
        queue is a queue created by create()
    Return:
        True if the queue has no data, or false otherwise
    """
    return queue['size'] == 0


def enqueue(queue, value):
    """
    Purpose
        adds the given data value to the given queue
    Pre-conditions:
        queue: a queue created by create()
        value: data to be added
    Post-condition:
        the value is added to the queue
    Return:
        (none)
    """
    new_node = node.create(value, None)

    if is_empty(queue):
        queue['front'] = new_node
        queue['back'] = new_node
    else:
        prev_last_node = queue['back']
```

```python
        node.set_next(prev_last_node, new_node)
        queue['back'] = new_node

    queue['size'] += 1



def dequeue(queue):
    """
    Purpose
        removes and returns a data value from the given queue
    Pre-conditions:
        queue: a queue created by create()
    Post-condition:
        the first value is removed from the queue
    Return:
        the first value in the queue, or None
    """
    assert not is_empty(queue), 'dequeued an empty queue'

    prev_first_node = queue['front']
    result = node.get_data(prev_first_node)
    queue['front'] = node.get_next(prev_first_node)
    queue['size'] -= 1
    if queue['size'] == 0:
        queue['back'] = None
    return result



def peek(queue):
    """
    Purpose
        returns the value from the front of given stack
        without removing it
    Pre-conditions:
        queue: a non-empty queue created by create()
    Post-condition:
        None
    Return:
        the value at the front of the queue
    """
    assert not is_empty(queue), 'peeked into an empty queue'

    first_node = queue['front']
    result = node.get_data(first_node)
    return result
```

D.2 The full list-based Stack implementation

The node-based stack is discussed in Chapter 13. Here we present the implementation without remark.

```python
# CMPT 145: Node-Based Data Structures
# Implements the Stack ADT
#
# A stack (also called a pushdown or LIFO stack) is a compound
# data structure in which the data values are ordered according
# to the LIFO (last-in first-out) protocol.
#
# Implementation:
# This implementation uses the linked node structure.


import node as node

def create():
    """
    Purpose
        creates an empty stack
    Return
        an empty stack
    """
    stack = {}
    stack['size'] = 0        # how many elements in the stack
    stack['top'] = None      # the node chain starts here
    return stack


def size(stack):
    """
    Purpose
        returns the number of data values in the given queue
    Pre-conditions:
        queue: a queue created by create()
    Return:
        The number of data values in the queue
    """
    return stack['size']


def is_empty(stack):
```

```python
    """
    Purpose
        checks if the given queue has no data in it
    Pre-conditions:
        queue is a queue created by create()
    Return:
        True if the queue has no data, or false otherwise
    """
    return stack['size'] == 0


def push(stack, value):
    """
    Purpose
        adds the given data value to the given stack
    Pre-conditions:
        stack: a stack created by create()
        value: data to be added
    Post-condition:
        the value is added to the stack
    Return:
        (none)
    """
    new_node = node.create(value, stack['top'])
    stack['top'] = new_node
    stack['size'] += 1




def pop(stack):
    """
    Purpose
        removes and returns a data value from the given stack
    Pre-conditions:
        stack: a stack created by create()
    Post-condition:
        the first value is removed from the stack
    Return:
        the first value in the stack, or None
    """
    assert not is_empty(stack), 'popped an empty stack'

    prev_first_node = stack['top']
    result = node.get_data(prev_first_node)
    stack['top'] = node.get_next(prev_first_node)
```

```
    stack['size'] -= 1
    return result


def peek(stack):
    """
    Purpose
        returns the value from the front of given stack
        without removing it
    Pre-conditions:
        stack: a stack created by create()
    Post-condition:
        None
    Return:
        the value at the front of the stack
    """
    assert not is_empty(stack), 'peeked into an empty stack'

    first_node = stack['top']
    result = node.get_data(first_node)
    return result
```

E — The treenode ADT

E.1   The full Treenode ADT implementation

The node-based queue is discussed in Chapter 17. Here we present the implementation without remark.

```
# CMPT 145: Binary trees
# Defines the tree node ADT
#
# A treenode is a simple container with three pieces of information
#    data: the contained information
#    left:  a reference to another treenode or None
#    right: a reference to another treenode or None


# Implementation notes:
#    This implementation uses a Python dictionary as a record.


def create(data, left=None, right=None):
    """
    Create a new treenode for the given data.
    Pre-conditions:
        data:  Any data value to be stored in the treenode
        left:  Another treenode (or None, by default)
    Post-condition:
        none
    Return:
        the treenode created
    """
```

```python
    return {'data':data, 'left':left, 'right':right}


def get_data(treenode):
    """
    Retrieve the contents of the data field.
    Pre-conditions:
        node: a node created by create()
    Post-conditions:
        none
    Return
        the data value stored previously in the node
    """
    return treenode['data']


def get_left(tnode):
    """
    Retrieve the contents of the left field.
    Pre-conditions:
        tnode: a treenode created by create()
    Post-conditions:
        none
    Return
        the value stored in left field
    """
    return tnode['left']


def get_right(tnode):
    """
    Retrieve the contents of the right field.
    Pre-conditions:
        tnode: a treenode created by create()
    Post-conditions:
        none
    Return
        the value stored in right field
    """
    return tnode['right']


def set_data(tnode, val):
    """
    Set the contents of the data field to val.
    Pre-conditions:
```

```
        tnode: a node created by create()
        val:  a data value to be stored
    Post-conditions:
        stores the new data value, replacing the existing value
    Return
        none
    """
    tnode['data'] = val


def set_left(tnode, val):
    """
    Set the contents of left field to val.
    Pre-conditions:
        tnode: a treenode created by create()
        val:   a treenode, or the value None
    Post-conditions:
        stores the val in left field, replacing the existing value
    Return
        none
    """
    tnode['left'] = val

def set_right(tnode, val):
    """
    Set the contents of right field to val.
    Pre-conditions:
        tnode: a treenode created by create()
        val:   a treenode, or the value None
    Post-conditions:
        stores the val in right field, replacing the existing value
    Return
        none
    """
    tnode['right'] = val
```

F — Algorithms

F.1 Examples

A variety of algorithms were discussed in Chapter 26. Here we present the full implementations without remark.

F.2 Maximum Slice

```python
# CMPT 145 - Algorithms
# The Maximum Slice Problem
# Given a list A containing (positive and negative) numbers,
# Find the slice A[a:b] that has the maximum sum

import random as rand
import time as time


# version 0: naively sums each slice
def maxslice_brute_force_v0(alist):
    """
    Find the maximum sum of all slices of alist.
    :param alist: a list of numbers
    :return: a number, the maximum slice sum
    """
    # using brute force: look at all possible slices
    maxsum = alist[0]
    for i in range(len(alist)):
        for j in range(len(alist)):
            slice = sum(alist[i:j + 1])
```

```python
            if slice > maxsum:
                maxsum = slice
    return maxsum


# version 1: sums the slices more cleverly
def maxslice_brute_force_v1(alist):
    """
    Find the maximum sum of all slices of alist.
    :param alist: a list of numbers
    :return: a number, the maximum slice sum
    """
    # using brute force: look at all possible slices
    # but store all the partial sums in a dictionary
    # where s[i,j] stores the value sum(alist[i,j+1])
    s = {}
    maxsum = alist[0]
    for i in range(len(alist)):
        s[i,i] = alist[i]
        if s[i,i] > maxsum: maxsum = s[i,i]
        for j in range(i+1, len(alist)):
            s[i, j] = s[i,j-1] + alist[j]
            if s[i,j] > maxsum: maxsum = s[i,j]
    return maxsum


# version 2 uses divide and conquer
# Divide the list into 2 halves
# The maximum slice can occur in one of 3 ways:
# 1. Starts and finishes on the left half
# 2. Starts and finishes on the right half
# 3. Starts somewhere in the left half, and
#    continues somewhere to the right half
def maxslice_DC(alist):
    """
    Find the maximum sum of all slices of alist.
    :param alist: a list of numbers
    :return: a number, the maximum slice sum
    """
    # using divide and conquer

    def max_tail(left, right):
        """
        Calculate the maximum slice that ends at right
        (from any point starting at left or later)
        """
```

```python
        s = {}
        s[right] = alist[right]
        maxsum = s[right]
        # calculate the sums from right to left (backwards)
        for i in range(right - 1, left - 1, -1):
            s[i] = s[i + 1] + alist[i]
            if s[i] > maxsum: maxsum = s[i]
        return maxsum

    def max_head(left, right):
        """
        Calculate the maximum slice that starts at left
        (to any point up to and including right)
        """
        s = {}
        s[left] = alist[left]
        maxsum = s[left]
        for i in range(left + 1, right + 1):
            s[i] = s[i - 1] + alist[i]
            if s[i] > maxsum: maxsum = s[i]
        return maxsum

    def maxslice_rec(left, right):
        """
        Recursively find the maximum slice between left and right.
        """
        # using divide and conquer
        if left == right:
            return alist[left]
        else:
            # divide, and solve
            mid = (right + left) // 2
            max_left  = maxslice_rec(left, mid)
            max_right = maxslice_rec(mid + 1, right)
            max_cross = (max_tail(left, mid)
                          + max_head(mid + 1, right))
            # conquer
            return max(max_left, max_right, max_cross)

    # body of maxslice_DC
    return maxslice_rec(0, len(alist) - 1)


# put the versions through their paces
if __name__ == '__main__':
    examples = [[1, 2, 3, 4, 5],
```

```
                [5, 4, -3, 2, 1],
                [1, 2, -3, 4, 5],
                [1, -2, 3, 4, -5],
                [1, -2, 3, 4, -5, 1, 2, 3, 4, 5, -6, -7, -8],
                [rand.randint(-100, 100) for i in range(1000)],
                [rand.randint(-100, 100) for i in range(4000)]
               ]

    for ex in examples:
        print('Example: list of length:', len(ex))

        print('Brute Force version 0:')
        start = time.process_time()
        result = maxslice_brute_force_v0(ex)
        end = time.process_time()
        print('Result:', result, 'Time:', (end - start))

        print('Brute Force version 1:')
        start = time.process_time()
        result = maxslice_brute_force_v1(ex)
        end = time.process_time()
        print('Result:', result, 'Time:', (end - start))

        print('Divide and conquer:')
        start = time.process_time()
        result = maxslice_DC(ex)
        end = time.process_time()
        print('Result:', result, 'Time:', (end - start))

        print()
```

F.3 Subset Sum

```
# CMPT 145 - Algorithms
# The Subset Sum problem
#
# Given a list of integers L, and a target T
# Find a subset of numbers from L whose sum is T


import random as rand
import time as time



# Brute force solution
```

```python
def subsetsum_v1(alist, target):
    """
    Given a list of numbers, find a subset whose
    sum is equal to the target.
    """

    def trysum(alist, subset):
        """
        Recursively construct a subset from alist, checking
        if the sum is equal to the target.
        """
        if sum(subset) == target:
            return True, subset
        elif len(alist) == 0:
            return False, None
        else:
            tryit = alist[0]
            # try to find a subset sum using the first value
            flag, answer = trysum(alist[1:], subset + [tryit])
            if flag:
                return flag, answer
            else:
                # try without the first value
                return trysum(alist[1:], subset)

    flag, result = trysum(alist, [])
    if flag: return result
    else: return None


# Divide and Conquer solution
def subsetsum_v2(alist, target):
    """
    Given a list of numbers, find a subset whose
    sum is equal to the target.
    """

    def trysum(alist, subset):
        """
        Recursively construct a subset from alist, checking
        if the sum is equal to the target.
        """
        if sum(subset) == target:
            return True, subset
        elif len(alist) == 0:
            return False, None
```

```python
        else:
            tryit = alist[0]
            # try to find a subset sum using the first value
            flag, answer = trysum(alist[1:], subset + [tryit])
            if flag:
                return flag, answer
            else:
                # try without the first value
                return trysum(alist[1:], subset)

    # this does divide and conquer, in the form of
    # dividing the list in half
    def binary(alist):
        if len(alist) < 5:
            # for small lists, just use brute force!
            return trysum(alist, [])
        else:
            mid = len(alist)//2

            # try to find a solution in a list half the size
            flag, ans = binary(alist[:mid])
            if flag: return True, ans

            # if not, try the other half
            flag, ans = binary(alist[mid + 1:])
            if flag: return True, ans

            # if not, try the whole list
            return trysum(alist, [])

    flag, result = binary(alist)
    if flag: return result
    else: return None


if __name__ == '__main__':

    example1 = [1, 2, 3, 4, 5]
    example2 = [1, 3, 5, 7]
    target = 8

    for ex in [example1, example2]:
        print('Example: list of length:', len(ex))

        print('Brute Force (Version 1):')
        start = time.process_time()
```

```python
        subset = subsetsum_v1(ex, target)
        end = time.process_time()
        print('Target:', target,
              'Time:', end - start,
              'Result:', subset)

        print('Divide and Conquer (Version 2):')
        start = time.process_time()
        subset = subsetsum_v2(ex, target)
        end = time.process_time()
        print('Target:', target,
              'Time:', end - start,
              'Result:', subset)

        print()

example_size = 80
example3 = [rand.randint(-1000, 1000)
            for i in range(example_size)]
target3 = rand.randint(-2000, 2000)

number_of_trials = 5
for i in range(number_of_trials):
    target = rand.randint(-2000, 2000)

    print('Brute Force (Version 1) on list of size',
          example_size, ':')
    start = time.process_time()
    subset = subsetsum_v1(example3, target3)
    end = time.process_time()
    print('Target:', target,
          'Time:', end - start,
          'Result:', subset)

    print('Divide and Conquer (Version 2) on list of size',
          example_size, ':')
    start = time.process_time()
    subset = subsetsum_v2(example3, target3)
    end = time.process_time()
    print('Target:', target,
          'Time:', end - start,
          'Result:', subset)

    print()
```

F.4 Maximum Tree Path

```python
# CMPT 145 - Algorithms
# The Maximum Path Problem
# Given a Binary Tree (not necessarily a Binary Search Tree)
# containing (positive and negative) numbers,
# Find the path from root to leaf that has the maximum sum


import random as rand
import time


# a quick and dirty implementation of TreeNodes
class Treenode(object):
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right

    def is_leaf(self):
        return self.left is None and self.right is None


def random_tree(h, lor=-200, hir=200):
    """
    Construct a complete binary tree of height h with
    randomly chosen data values
    :param h: the height of the tree
    :param lor: an integer
    :param hir: an integer, lor <= hir
    :return: a complete binary TreeNode of height h
    """
    if h == 1:
        return Treenode(rand.randint(lor, hir))
    else:
        return Treenode(rand.randint(lor, hir),
                        random_tree(h - 1, lor, hir),
                        random_tree(h - 1, lor, hir))


# Brute force solution: check all possible paths
def max_path(atree):
    """
    Find the maximum sum of all paths from root to leaf
    in atree.
```

```python
    :param atree: A TreeNode
    :return: the maximum sum
    """
    if atree is None:
        return 0
    else:
        left = max_path(atree.left)
        right = max_path(atree.right)
        return max(left, right) + atree.data


# Greedy solution
# Note: this algorithm may not return the true maximum!
def greedy_path(atree):
    """
    Find the maximum sum of all paths from root to leaf
    in atree.
    :param atree: A TreeNode
    :return: the maximum sum
    """
    if atree.is_leaf():
        return atree.data
    elif atree.left.data > atree.right.data:
        # the left branch seems promising
        return atree.data + greedy_path(atree.left)
    else:
        # the right branch seems promising
        return atree.data + greedy_path(atree.right)


if __name__ == '__main__':
    # build the tree; O(2^N) time!
    height = 21
    maxval = 100
    print('Building tree...')
    start = time.process_time()
    some_tree = random_tree(height)
    end = time.process_time()
    print('...took:', end - start, 'seconds')

    print('Brute Force Implementation:')
    start = time.process_time()
    result = max_path(some_tree)
    end = time.process_time()
    print("Result:", result, 'Time:', end - start)
```

```python
    print('Greedy Implementation:')
    start = time.process_time()
    result = greedy_path(some_tree)
    end = time.process_time()
    print("Result:", result, 'Time:', end - start)
```

F.5 Make Change

```python
# CMPT 145 - Algorithms
# The Make Change Problem
#
# Given a number D in the range 0-99
# Find the smallest collection of coins whose value is D



# a greedy solution, adding coins one at a time
def change_v1(cents):
    """
    Make change for the given cents value.
    Assumes coin values 25c, 10c, 5c, 1c
    :param cents: an integer
    :return: a list of counts for the coins used.
    """
    coins = [25, 10, 5, 1]
    coin_index = 0
    counts = [0, 0, 0, 0]
    remaining = cents
    while remaining > 0:
        if coins[coin_index] <= remaining:
            counts[coin_index] += 1
            remaining -= coins[coin_index]
        else:
            coin_index += 1
    return counts


# a greedy solution, calculating each quantity
# of coins exactly
def change_v2(cents):
    """
    Make change for the given cents value.
    Assumes coin values 25c, 10c, 5c, 1c
    :param cents: an integer
    :return: a list of counts for the coins used.
```

```python
    """
    coins = [25, 10, 5, 1]
    coin_index = 0
    counts = [0] * len(coins)
    remaining = cents
    while remaining > 0:
        n = remaining // coins[coin_index]
        counts[coin_index] = n
        remaining = remaining % coins[coin_index]
        coin_index += 1
    return counts


# A brute force solution
# Find the smallest of all combinations of coins
def change_v3(cents):
    """
    Make change for the given cents value.
    Assumes coin values 25c, 10c, 5c, 1c
    :param cents: an integer
    :return: a list of counts for the coins used.
    """

    coins = [25, 10, 5, 1]

    def combinations(counts):
        """
        Cycle through every possible combination, looking
        for a combo that adds up to the right value, and
        has the smallest number of coins
        :param counts: a 4-tuple
        :return: a pair (True, list) if list is the best
                 combination
        """
        value = sum([counts[i] * v for i, v in enumerate(coins)])
        if value == cents:
            return True, counts
        elif value > cents:
            # don't add more coins to this combination
            # because it's already too big
            return False, None
        else:
            (c0, c1, c2, c3) = counts
            # add 1 to each number of coins separately
            trying = [(c0 + 1, c1, c2, c3), (c0, c1, c2 + 1, c3),
                      (c0, c1 + 1, c2, c3), (c0, c1, c2, c3 + 1)]
```

```python
            # try to find the best combination, by using
            # the lowest number of coins
            best_size = 100
            best_counts = None
            for combo in trying:
                flag, res = combinations(combo)
                if flag and sum(res) < best_size:
                    best_size = sum(res)
                    best_counts = res
            if best_size == 100:
                # nothing worked!
                return False, None
            else:
                return True, best_counts

    flag, result = combinations((0, 0, 0, 0))
    if flag:
        return result
    else:
        return None


if __name__ == '__main__':
    examples = [0, 25, 37, 49, 51, 87, 99]

    for e in examples:
        print('Version 1:', e, change_v1(e))
        print('Version 2:', e, change_v2(e))
        print('Version 3:', e, change_v3(e))

        print()
```

F.6 Leap Line

```python
# CMPT 145 - Algorithms
# The Leap Line Problem
# Given a list A containing (positive and negative) numbers,
# Assuming a collection policy
# Find the maximum score that can be collected
#
# Collection policy:
#   Mario must leap over 1 item or step into it
#   Leaping over item prevents collection
#   Stepping into an item ensures collection
```

```python
import time as time


# The collection policy.
# Change the return value for leaps!
def collection(action):
    """
    Isolate the collection policy for easy modification!
    :param action: a string, either 'step' or 'leap'
    :param loc: the current position
    :return: a new position
    """
    if action == 'step':
        return 1
    elif action == 'leap':
        return 3
    else:
        raise Exception('Invalid action for collection policy')


# version 0: Brute force
def maximumScoreFrom_v0(track):
    """
    Calculate the maximum score that can be obtained from
    stepping and leaping along the given track
    :param track: a list of integers
    :return: the maximum score
    """

    def leap_or_step(loc):
        """
        Calculate the maximum score that can be obtained from
        stepping and leaping along the given track, starting
        at the given location
        """
        step_loc = loc + 1
        leap_loc = loc + 2

        if step_loc == len(track):
            return track[loc]
        elif leap_loc >= len(track):
            return track[loc] + leap_or_step(step_loc)
        else:
            return track[loc] + max(leap_or_step(step_loc),
                                    leap_or_step(leap_loc))
```

```python
    return leap_or_step(0)


# version 1: Using Memoization
def maximumScoreFrom_v1(track):
    '''
    Calculate the maximum score that can be obtained from
    stepping and leaping along the given track
    :param track: a list of integers
    :return: the maximum score
    '''
    # using memoization
    # memo[loc] stores the best score starting from loc.
    memo = {}

    def leap_or_step(loc):
        """
        Calculate the maximum score that can be obtained from
        stepping and leaping along the given track, starting
        at the given location
        """
        # check if the best score is already known
        if loc in memo:
            return memo[loc]

        step_loc = loc + 1
        leap_loc = loc + 2

        if step_loc == len(track):
            return track[loc]
        elif leap_loc >= len(track):
            result = track[loc] + leap_or_step(step_loc)
            memo[loc] = result
            print('memo', loc, result)
            return result
        else:
            result = track[loc] + max(leap_or_step(step_loc),
                                      leap_or_step(leap_loc))
            memo[loc] = result
            print('memo', loc, result)
            return result

    return leap_or_step(0)


# version 2: Greedy
```

```python
# looks one location ahead, and leaps over -1
def maximumScoreFrom_v2(track):
    """
    Calculate the maximum score that can be obtained from
    stepping and leaping along the given track
    :param track: a list of integers
    :return: the maximum score
    """

    def leap_or_step(loc):
        """
        Calculate the maximum score that can be obtained from
        stepping and leaping along the given track, starting
        at the given location
        """
        step_loc = loc + 1
        leap_loc = loc + 2

        if step_loc == len(track):
            return track[loc]
        elif leap_loc >= len(track):
            return track[loc] + leap_or_step(step_loc)
        elif track[step_loc] < 0:
            # greedy choice: avoid negatives
            return track[loc] + leap_or_step(leap_loc)
        else:
            # greedy choice: collect 1
            return track[loc] + leap_or_step(step_loc)

    return leap_or_step(0)


examples = [
    ['easy',
        [0, -1, -1, 1, -1, -1, 0]],
    ['medium',
        [0, -1, 1, -1, -1, 1, -1, -1, 1, -1, 1, 0]],
    ['challenging',
        [0, -1, 1, -1, -1, 1, -1, -1, 1, -1, 1, -1, -1, 0]],
    ['hardmode',
        [0, -1, 1, -1, -1, 1, -1, -1, 1, -1, 1, -1, -1, 1,
         1, 1, -1, -1, -1, 1, -1, 1, -1, -1, 1, -1, -1, 1,
         -1, 1, -1, -1, 1, 1, 1, -1, 0]],
    ['very_hard',
        [0, -1, 1, -1, -1, 1, -1, -1, 1, -1, 1, -1, -1, 1,
         1, 1, -1, 1, -1, -1, 1, 1, 1, 1, 1, -1, 1, -1, -1,
```

```
          1, -1, -1, 1, -1, 1, 1, -1, -1, 1, -1, -1, 1, -1,
          1, -1, -1, 0]]
]

for e in examples:
    print('Example: list of length:', len(e[1]))

    print('Brute Force version (v0):', end=" ")
    start = time.process_time()
    result = maximumScoreFrom_v0(e[1])
    end = time.process_time()
    print('Result', '(' + e[0] + '):', result,
          'Time:', (end - start))

    print('Memoized version (v1):', end=" ")
    start = time.process_time()
    result = maximumScoreFrom_v1(e[1])
    end = time.process_time()
    print('Result', '(' + e[0] + '):', result,
          'Time:', (end - start))

    print('Greedy version (v2):', end=" ")
    start = time.process_time()
    result = maximumScoreFrom_v2(e[1])
    end = time.process_time()
    print('Result', '(' + e[0] + '):', result,
          'Time:', (end - start))

    print()
```