A Course in Data Structures and Object-Oriented Design

by Don Sheehy

- Overview
- Basic Python
  - Sequence, Selection, and Iteration
  - Expressions and Evaluation
  - Variables, Types, and State
  - Collections
    - Strings (`str`)
    - Lists (`list`)
    - Tuples (`tuple`)
    - Dictionaries (`dict`)
    - Sets (`set`)
  - Some common things to do with collections
  - Iterating over a collection
  - Other Forms of Control Flow
  - Modules and Imports
- Object-Oriented Programming
  - A simple example
  - Encapsulation and the Public Interface of a Class
  - Inheritance and "is a" relationships
  - Duck Typing
  - Composition and "has a" relationships
- Testing
  - Writing Tests
  - Unit Testing with `unittest`
  - Test-Driven Development
  - What to Test
  - Testing and Object-Oriented Design
- Running Time Analysis
  - Timing Programs
  - Example: Adding the first k numbers.
  - Modeling the Running Time of a Program
    - List Operations
    - Dictionary Operations
    - Set Operations
  - Asymptotic Analysis and the Order of Growth
  - Focus on the Worst Case

    - Big-O
    - The most important features of big-O usage
- Practical Use of the Big-O and Common Functions
    - Bases for Logarithms
- Practice examples
- Stacks and Queues
    - Abstract Data Types
    - The Stack ADT
    - The Queue ADT
    - Dealing with errors
- Deques and Linked Lists
    - The Deque ADT
    - Linked Lists
    - Implementing a Queue with a LinkedList
    - Storing the length
    - Testing Against the ADT
    - The Main Lessons:
    - Design Patterns: The Wrapper Pattern
    - The Main Lessons:
- Doubly-Linked Lists
    - Concatenating Doubly Linked Lists
- Recursion
    - Recursion and Induction
    - Some Basics
    - The Function Call Stack
    - The Fibonacci Sequence
    - Euclid's Algorithm
- Dynamic Programming
    - A Greedy Algorithm
    - A Recursive Algorithm
    - A Memoized Version
    - A Dynamic Programming Algorithm
    - Another example
- Binary Search
    - The Sorted List ADT
- Sorting

    - The Quadratic-Time Sorting Algorithms
    - Sorting in Python
- Sorting with Divide and Conquer
    - Mergesort
        - An Analysis
        - Merging Iterators
    - Quicksort
- Selection
    - The `quickselect` algorithm
    - Analysis
    - One last time without recursion
    - Divide-and-Conquer Recap
    - A Note on Derandomization
- Mappings and Hash Tables
    - The Mapping ADT
    - A minimal implementation
    - The extended Mapping ADT
    - It's Too Slow!
        - Rehashing
    - Factoring Out A Superclass
- Trees
    - Some more definitions
    - A recursive view of trees
    - Tree Traversal
    - If you want to get fancy...
        - There's a catch!
- Binary Search Trees
    - The Sorted Mapping ADT
    - Binary Search Tree Properties and Definitions
    - A Minimal implementation
        - The `floor` function
        - Iteration
    - Removal
- Balanced Binary Search Trees
    - A `BSTMapping` implementation
        - Forward Compatibility of Factories

    - Weight Balanced Trees
    - Height-Balanced Trees (AVL Trees)
    - Splay Trees
- Priority Queues
    - The Priority Queue ADT
    - Using a list
    - Heaps
    - Storing a tree in a list
    - Building a Heap from scratch
    - Changing priorities
- Graphs
    - A Graph ADT
    - The `EdgeSetGraph` Implementation
    - The `AdjacencySetGraph` Implementation
    - Paths and Connectivity
- Graph Search
    - Depth-First Search
    - Removing the Recursion
    - Breadth-First Search
    - Weighted Graphs and Shortest Paths
    - Prim's Algorithm for Minimum Spanning Trees
    - An optimization for Priority-First search

Chapter 0

Overview

This book is designed to cover a lot of ground quickly, without taking shortcuts.

What does that mean? It means that concepts are motivated and start with simple examples. The ideas follow the problems. The abstractions follow the concrete instances.

What does it not mean? The book is not meant to be comprehensive, covering all of data structures, nor is it a complete introduction to all the details of python. Introducing the minimum necessary knowledge to make interesting programs and learn useful concepts is not taking shortcuts, it's just being directed.

There are many books that will teach idiomatic python programming, and many others that will teach problem solving, data structures, or algorithms. There are many books for learning design patterns, testing, and many of the other important practices of software engineering. The aim of this book is cover many of these topics as part of an integrated course.

Towards that aim, the organization is both *simple* and *complex*. The *simple* part is that the overall sequencing of the main topics is motivated by the data structuring problems, as is evident from the chapter titles. The *complex* part is that many other concepts including problem solving strategies, more advanced python, object-oriented design principles, and testing methodologies are introduced bit by bit throughout the text in a logical, incremental way.

As a result, the book is not meant to be a reference. It is meant to be worked through from beginning to end. Many topics dear to my heart were left out to make it possible to work through the whole book in a single semester course.

Chapter 1

Basic Python

This book is not intended as a first course in programming. It will be assumed that the reader has some experience with programming. Therefore, it will be assumed that certain concepts are already familiar to them, the most basic of which is a mental model for programming that is sometimes called *Sequence, Selection, and Iteration*.

Sequence, Selection, and Iteration

A recurring theme in this course is the process of moving from *thinking about* code to *writing* code. We will try to shape the way we think about programs, the way we write programs, and how we go between the two in *both* directions. That is, we want to have facility with both direct manipulation of code as well as high-level description of programs.
A nice model for thinking about (imperative) programming is called Sequence-Selection-Iteration. It refers to:

1. **Sequence**: Performing operations one at a time in a specified order.
2. **Selection**: Using conditional statements such as `if` to select which operations to execute.
3. **Iteration**: Repeating some operations using loops or recursion.

In any given programming language, there are usually several mechanisms for selection and iteration, while sequencing is just the default behavior.
In fact, you usually have to have special constructions in a language to do something other than performing the given operations in the given order.

Expressions and Evaluation

Python can do simple arithmetic. For example, `2+2` is a simple arithmetic **expression**. Expressions get **evaluated** and produce a **value**. Some values are numerical like the `2 + 2` example, but they don't need to be. For example, `5 > 7` is an expression that evaluates to the boolean value `False`. Expressions can become more complex by combining many operations and functions. For example, `5 * (3 + abs(-12) / 3)` contains four different functions. Parentheses and the order of operations determine the order that the functions are evaluated. Recall that in programming the order of operations is also called **operator precedence**. In the above example, the functions are executed in the following order: `abs`, `/`, `+`, `*`.

Variables, Types, and State

Imagine you are trying to work out some elaborate math problem without a computer. It helps to have paper. You write things down, so that you can use them later. It's the same in programming. It often happens that you compute something and want to keep it until later when you will use it. We often refer to stored information as **state**.

We store information in **variables**. In Python, a variable is created by an **assignment** statement. That is a statement of the form:

```
variable_name = some_value
```

The equals sign is *doing* something (assignment) rather than *describing* something (equality). The right side of `=` is an expression that gets evaluated first. Only later does the assignment happen. If the left side of the assignment is a variable name that already exist, it is overwritten. If it doesn't already exist, it is created.

The order of evaluation is very important. Having the right side evaluated first means that assignments like `x = x + 1` make sense, because the value of `x` doesn't change until after `x + 1` is evaluated. Incidentally, there is a shorthand for this kind of update: `x += 1`. There are similar notations for `-=`, `*=`, and `/=`.

An assignment statement is not expression. It doesn't have a value. This turns out to be useful in avoiding a common bug arising from confusing assignment and equality testing (i.e. `x == y`). However, multiple assignment like `x = y = 1` does work as you would expect it to, setting both `x` and `y` to `1`.

Variables are just names. Every name is associated with some piece of data, called an object. The name is a string of characters and it is mapped to an object. The name of a variable, by itself, is treated as an expression that evaluates to whatever object it is mapped to. This mapping of strings to objects is often depicted using boxes to represent the objects and arrows to show the mapping.

Every object has a **type**. The type often determines what you can do with the variable. The so-called **atomic types** in Python are *integers*, *floats*, and *booleans*, but any interesting program will contain variables of many other types as well. You can inspect the type of a variable using the `type()` function. In Python, the word *type* and *class* mean the same thing (most of the time).

The difference between a variable and the **object** it represents can get lost in our common speech because the variable is usually acting as the *name* of the object. There are some times when it's useful to be clear about the difference, in particular when copying objects. You might want to try some examples of copying objects from one variable to another. Does changing one of them affect the other?

```
x = 5
y = 3.2
z = True
print("x has type", type(x))
print("y has type", type(y))
print("z has type", type(z))
```

```
x has type <class 'int'>
y has type <class 'float'>
z has type <class 'bool'>
```

You should think of an object as having three things: an *identity*, a *type*, and a *value*. It's identity cannot change. It can be used to see if two objects are actually the same object with the `is` keyword. For example, consider the following code.

```
x = [1, 2, 3]
y = x
z = [1, 2, 3]

print(x is y)
print(x is z)
print(x == z)
```

```
True
False
True
```

An object cannot change its identity. In Python, you also cannot change the type of an object. You can reassign a variable to point to different object of a different type, but that's not the same thing. There are several functions that may seem to be changing the types of objects, but they are really just creating a new object from the old.

```
x = 2
print("x =", x)
print("float(x) =", float(x))
print("x still has type", type(x))

print("Overwriting x.")
x = float(x)
print("Now, x has type", type(x))
```

```
x = 2
float(x) = 2.0
x still has type <class 'int'>
Overwriting x.
Now, x has type <class 'float'>
```

You can do more elaborate things as well.

```
numstring = "3.1415926"
y = float(numstring)
print("y has type", type(y))

best_number = 73
x = str(best_number)
print("x has type", type(x))

thisworks = float("inf")
print("float(\'inf\') has type", type(thisworks))
infinity_plus_one = float('inf') + 1
```

```
y has type <class 'float'>
x has type <class 'str'>
float('inf') has type <class 'float'>
```

This last example introduced a new type, that of a **string**. A string is a sequence of characters. In Python, there is no special class for a single character (as in C for example). If you want a single character, you use a string of length one.

The value of an object may or may not be changed, depending on the type of object. If the value can be changed, we say that the object is **mutable**. If it cannot be changed, we say that the object is **immutable**. For example, strings are immutable. If you want to change a string, for example, by converting it to lowercase, then you will be creating a new string.

Collections

The next five most important types in Python are strings, lists, tuples, dictionaries, and sets. We call these collections as each can be used for storing a collection of things. We will see many other examples of collections in this course.

Strings (`str`)

Strings are sequences of characters and can be used to store text of all kinds. Note that you can **concatenate** strings to create a new string using the plus sign. You can also access individual characters using square brackets and an **index**. The name of the class for strings is `str`. You can often turn other objects into strings.

```
s = "Hello, "
t = "World."
u = s + t
print(type(u))
print(u)
print(u[9])
n = str(9876)
print(n[2])
```

```
<class 'str'>
Hello, World.
r
7
```

Lists (`list`)

Lists are ordered sequences of objects. The objects do not have to be the same type. They are indicated by square brackets and the **elements** of the list are separated by commas. You can append an item to the end of a list `L` by using the command `L.append(newitem)`. It is possible to index into a list exactly as we did with strings.

```
L = [1,2,3]
print(type(L))
L.append(400)
print(L)
```

```
<class 'list'>
[1, 2, 3, 400]
```

Tuples (`tuple`)

Tuples are also ordered sequences of objects, but unlike lists, they are immutable. You can access the items but you can't change what items are in the tuple after you create it. For example, trying to `append` raises an exception.

```
t = (1, 2, "skip a few", 99, 100)
print(type(t))
print(t)
print(t[4])
t.append(101)
```

```
<class 'tuple'>
(1, 2, 'skip a few', 99, 100)
100
Traceback (most recent call last):
  File "/Users/don/Dropbox/work/research/books/datastructures/docs/8e0ls7h00_code_chunk.py",
    t.append(101)
AttributeError: 'tuple' object has no attribute 'append'
```

Dictionaries (`dict`)

Dictionaries store *key-value* pairs. That is, every element of a dictionary has two parts, a **key** and a **value**. If you have the key, you can get the value. The name comes from the idea that in a real dictionary (book), a word (the key) allows you to find its definition (the value). Notice that the keys can be different types, but they must be immutable types such as atomic types, tuples, or strings. The reason for this requirement is that we will determine where to store something using the key. If the key changes, we will look in the wrong place when it's time to look it up again.

Dictionaries are also known as maps, **mappings**, or hash tables. We will go deep into how these are constructed later in the course. A dictionary doesn't have a fixed order.

```
d = dict()
d[2] = "two"
d[5] = "five"
d["pi"] = 3.1415926

print(d)
print(d["pi"])
```

```
{'pi': 3.1415926, 2: 'two', 5: 'five'}
3.1415926
```

Sets (`set`)

Sets correspond to our notion of sets in math. They are collections of objects without duplicates. We use curly braces to denote them and commas to separate elements. As with dictionaries, a set has no fixed ordering. We say that sets and dictionaries are **nonsequential collections**.

Be careful that empty braces `{}` indicates an empty dictionary and not an empty set. Here is an

example of a newly created set. Some items are added. Notice that the duplicates have no effect on the value as its printed.

```
s = {2,1}
print(type(s))
s.add(3)
s.add(2)
s.add(2)
s.add(2)
print(s)
```

```
<class 'set'>
{1, 2, 3}
```

Some common things to do with collections

There are several operations that can be performed on any of the collections classes (and indeed often on many other types objects).

You can find the number of elements in the collection (the **length**) using `len`.

```
a = "a string"
b = ["my", "second", "favorite", "list"]
c = (1, "tuple")
d = {'a': 'b', 'b': 2, 'c': False}
e = {1,2,3,4,4,4,4,2,2,2,1}

print(len(a), len(b), len(c), len(d), len(e))
```

```
8 4 2 3 4
```

For the sequential types (lists, tuples, and strings), you can **slice** a subsequence of indices using square brackets and a colon as in the following examples. The range of indices is half open in that the slice will start with the first index and proceed up to but not including the last index. Negative indices count backwards from the end. Leaving out the first index is the same as starting at 0. Leaving out the second index will continue the slice until the end of the sequence.

Important: slicing a sequence creates a new object. That means a big slice will do a lot of copying.

It's really easy to write inefficient code this way.

```
a = "a string"
b = ["my", "second", "favorite", "list"]
c = (1, 2,3,"tuple")
print(a[3:7])
print(a[1:-2])
print(b[1:])
print(c[:2])
```

```
trin
 stri
['second', 'favorite', 'list']
(1, 2)
```

Iterating over a collection

It is very common to want to loop over a collection. The pythonic way of doing iteration is with a `for` loop.

The syntax is shown in the following examples.

```
mylist = [1,3,5]
mytuple = (1, 2, 'skip a few', 99, 100)
myset = {'a', 'b', 'z'}
mystring = 'abracadabra'
mydict = {'a': 96, 'b': 97, 'c': 98}

for item in mylist:
    print(item)

for item in mytuple:
    print(item)

for element in myset:
    print(element)

for character in mystring:
    print(character)

for key in mydict:
    print(key)

for key, value in mydict.items():
    print(key, value)

for value in mydict.values():
    print(value)
```

There is class called `range` to represent a sequence of numbers that behaves like a collection. It is often used in for loops as follows.

```
for i in range(10):
    j = 10 * i + 1
    print(j,)
```

Other Forms of Control Flow

Control flow refers to the commands in a language that affect the order or what operations are executed. The `for` loops from the previous section are classic examples if this. The other basic forms of control flow are `if` statements, `while` loops, `try` blocks, and function calls. We'll cover

each briefly and refer the reader to the python documentation for more detailed info.

An `if` statement in its simplest form evaluates an expression and tries to interpret it as a boolean. This expression is referred to as a predicate. If the predicate evaluates to `True`, then a block of code is executed. Otherwise, the code is not executed. This is the *selection* of sequence, selection, and iteration. Here is an example.

```
if 3 + 3 < 7:
    print("This should be printed.")

if 2 ** 8 != 256:
    print("This should not be printed.")
```

```
This should be printed.
```

An `if` statement can also include an `else` clause. This is a second block of code that executes if the predicate evaluates to `False`.

```
if False:
    print("This is bad.")
else:
    print("This will print.")
```

```
This will print.
```

A `while` loop also has a predicate. It is evaluated at the top of a block of code. If it evaluates to `True`, then the block is executed and then it repeats. The repetition continues until the predicate evaluate to `False` or until the code reaches a `break` statement.

```
x = 1
while x < 128:
    print(x)
    x = x * 2
```

```
1
2
4
8
16
32
64
```

A `try` block is the way to catch and recover from errors while a program is running. If you have some code that may cause an error, but you don't want it to crash your program, you can put the code in a `try` block. Then, you can *catch* the error (also known as an **exception**) and deal with it. A simple example might be a case where you want to convert some number to a `float`. Many types of objects can be converted to `float`, but many cannot. If we simply try to do the conversion and it works, everything is fine. Otherwise, if there is a `ValueError`, we can do something else instead.

```
x = "not a number"
try:
    f = float(x)
except ValueError:
    print("You can't do that!")
```

```
You can't do that!
```

A function also changes the control flow. In Python, you define a function with the `def` keyword. This keyword creates an object to store the block of code. The parameters for the function are listed in parentheses after the function name. The `return` statement causes the control flow to revert back to where the function was called and determines the value of the function call.

```
def foo(x, y):
    return 8 * x + y

print(foo(2, 1))
print(foo("Na", " batman"))
```

```
17
NaNaNaNaNaNaNaNa batman
```

Notice that there is no requirement that we specify the types of objects a function expects for its arguments. This is very convenient, because it means that we can use the same function to operate on different types of objects (as in the example above). If we define a function twice, even if we change the parameters, the first will be overwritten by the second. This is exactly the same as assigning to a variable twice. The name of a function is just a name; it refers to an object (the function). Functions can be treated like any other object.

```
def foo(x):
    return x + 2

def bar(somefunction):
    return somefunction(4)

print(bar(foo))
somevariable = foo
print(bar(somevariable))
```

```
6
6
```

Modules and Imports

As we start to write more complex programs, it starts to make sense to break up the code across several files. A single `.py` file is called a module. You can import one module into another using the `import` keyword. The name of a module, by default, is the name of the file (without the `.py` extension). When we import a module, the code in that module is executed. Usually, this should be limited to defining some functions and classes, but can technically include anything. The module also has a namespace in which these functions and classes are defined.

For example, suppose we have the following files.

```
# File: twofunctions.py

def f(x):
    return 2 * x + 3

def g(x):
    return x ** 2 - 1
```

```
# File: theimporter.py
import twofunctions

def f(x):
    return x - 1

print(twofunctions.f(1)) # Will print 5
print(f(1))              # Will print 0
print(twofunctions.g(4)) # Will print 15
```

The `import` brings the module name into the current namespace. I can then use it to identify the functions from the module.

There is very little magic in an import. In some sense, it's just telling the current program about the results of another program. Because the import (usually) results in the module being executed, it's good practice to change the behavior of a script depending on whether it is being run directly, or being run as part of an import. It is possible to check by looking at the `__name__` attribute of the module. If you run the module directly (i.e. as a script), then the `__name__` variable is automatically set to `__main__`. If the module is being imported, the `__name__` defaults to the module name. This is easily seen from the following experiment.

```
# File: mymodule.py
print("The name of this module is", __name__)
```

```
The name of this module is __main__
```

```
# File: theimporter.py

import mymodule
print("Notice that it will print something different when imported?")
```

Here is how we use the `__name__` attribute to check how the program is being run.

```
def somefunction():
    print("Real important stuff here.")

if __name__ == '__main__':
    somefunction()
```

```
Real important stuff here.
```

In the preceding code, the message is printed only when the module is executed as a script. It is not printed (i.e. the `somefunction` function is not called) if the module is being imported. This is a very common python idiom.

One caveat is that modules are only executed the first time they are imported. If, for example, we import the same module twice, it will only be executed once. At that point, the namespace exists and can be accessed for the second one. This also avoids any infinite loops you might try to construct by having two modules, each of which imports the other.

There are a couple other common variations on the standard `import` statement.

1. You can `import` just a particular name or collection of names from a module: `from modulename import thethingIwanted`. This brings the new name `thethingIwanted` into the current namespace. It doesn't need to be preceded by `modulename` and a dot.
2. You can `import` all the names from the module into the current namespace: `from modulename import *`. If you do this, every name defined in the module will be accessible in the current namespace. It doesn't need to be preceded by `modulename` and a dot. Although easy to write and fast for many things, this is generally frowned upon as you often won't know exactly what names you are importing when you do this.
3. You can rename module after importing it: `import numpy as np`. This allows you to use a different name to refer to the objects of the module. In this example, I can write `np.array` instead of `numpy.array`. The most common reason to do this is to have a shorter name. The other, more fundamental use is to avoid naming conflicts.

Chapter 2

Object-Oriented Programming

A primary goal of **object-oriented programming** is to make it possible to write code that is close to the way you think about the things your code represents. This will make it easier to reason about the code and think through its correctness.

A **class** is a data type. In python *type* and *class* are (mostly) synonymous. An **object** is an **instance** of a class. For example, python has a `list` class. If I make a list called `mylist`. Then, `mylist` is an object of type `list`.

```
mylist = []
print(type(mylist))
print(isinstance(mylist, list))
print(isinstance(mylist, str))
```

```
<class 'list'>
True
False
```

There are all kinds of classes built into python. Some you might not expect.

```
def foo():
    return 0

print(type(foo))
```

```
<class 'function'>
```

For the advanced students, here is a more exotic example called a generator. In python you can `yield` instead of `return`. If so, the result will be something called a generator and not a function. This powerful idea shows up a lot in python, but we won't really be able to get our head around it until we understand how classes are able to package up data and code.

```
def mygenerator(n):
    for i in range(n):
        yield i

print(type(mygenerator))
print(type(mygenerator(5)))
```

```
<class 'function'>
<class 'generator'>
```

A simple example

One of the first ways that we learn about to combine multiple pieces of information into a single object is in calculus or linear algebra, with the introduction of vectors. We can think of a 2-dimensional vector as a pair of numbers. If we are trying to write some code that works with 2-dimensional vectors, we could just use tuples. It's not too hard to define some basic functions that work with vectors.

```
u = (3,4)
v = (3,6)

def add(a, b):
    return (a[0] + b[0], a[1] + b[1])

def subtract(a,b):
    return (a[0] - b[0], a[1] - b[1])

def dot(a, b):
    return (a[0] * b[0] + a[1] * b[1])

def norm(a):
    return (a[0] * a[0] + a[1] * a[1]) ** 0.5

def isvertical(a):
    return a[0] == 0

print(norm(u))
print(add(u,v))
print(u + v)
print(isvertical(subtract(v, u)))
```

```
5.0
(6, 10)
(3, 4, 3, 6)
True
```

This could be fine if that's all we wanted to do, but as we fill out the code, things will start to get messier. For example, suppose we want to make sure that the inputs to these functions really are tuples that contain two numbers. We might add some code to every method to check for this error or recover otherwise, but this is not great, because we really just want to operate on vectors. Moreover, we might want to `add` other types of things besides vectors. This would probably require us to make the add function much more complicated, or rename it something more descriptive such as `vectoradd`.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def norm(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5

u = Vector(3,4)

print(u.norm())
print(Vector(5,12).norm())
```

```
5.0
13.0
```

A function defined in a class is called a **method**. It is a standard convention to use `self` as the name of the first parameter to a method. This parameter is object that will generally be operated on by the method. When, calling the method, you don't have to pass a parameter explicitly for `self`. Instead, the dot notation fills in this parameter for you. That is `u.norm()` is translated into `vector.norm(u)`.

The `__init__` method is called a **initializer**. Methods like this one that start and end with two underscores are sometimes called the **magic methods** or also **dunder methods**. You should not make up your own methods starting and ending with two underscores. That's how python sets them apart so you don't actually call your own methods the same thing. Also, dunder methods are usually not called explicitly, but instead provide some other means of calling them. In the case of a initializer, calling the name of the class as a function invokes the initializer. You've seen this before, such as in `float("3.14159")`.

We will use another magic method to implement addition.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def norm(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5

    def __add__(self, other):
        newx = self.x + other.x
        newy = self.y + other.y
        return Vector(newx, newy)

u = Vector(3,4)
v = Vector(3,6)

print(u + v)
```

```
<__main__.Vector object="" at="" 0x103881048="">
```

That output is pretty weird. It's telling me that u + v is a vector object at some memory address, but doesn't tell me what vector it is. We need to implement `__str__` in order to print the vector nicely. This magic method is called by the print function to convert its parameters into a string. It is not obvious how a string ought to be printed for a given class. We have to specify it ourselves.

In the example below, I added a `__str__` method as well as some type checking on the inputs. The result will guarantee that a vector has two floats as coordinates.

```
class Vector:
    def __init__(self, x, y):
        try:
            self.x = float(x)
            self.y = float(y)
        except ValueError:
            self.x = 0.0
            self.y = 0.0

    def norm(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5

    def __add__(self, other):
        newx = self.x + other.x
        newy = self.y + other.y
        return Vector(newx, newy)

    def __str__(self):
        return "(%f, %f)" %(self.x, self.y)

u = Vector(3,4)
v = Vector(3,6)

print(u + v)
```

```
(6.000000, 10.000000)
```

Encapsulation and the Public Interface of a Class

The word **encapsulation** has two different, but related, meanings. The first is the idea of encapsulating or combining into a single thing, data and the methods that operate on that data. In Python, this is accomplished via classes, as we have seen.

The second meaning of encapsulation emphasizes the boundary between the inside and the outside of the class, specifying what is visible to the users of a class. Often this means partitioning the attributes into **public** and **private**. In Python, there is no formal mechanism to keep one from accessing attributes of a class from outside that class. So, in a sense, everything is public. However, there is a convention to make it clear what *ought* to be kept private. Any attribute that starts with an underscore is considered private. Think of it like someone's unlocked diary. You can read it, but you

shouldn’t.

```
class Diary:
    def __init__(self, title):
        self.title = title
        self._entries = []

    def addentry(self, entry):
        self._entries.append(entry)

    def _lastentry(self):
        return self._entries[-1]
```

In the example above, the `addentry` method is public. Anyone can add an entry. However, the `_lastentry` method is private. One should not call this method from outside of the `Diary` class. (Again, you can, but you shouldn’t.) The `title` is also public, but the list `_entries` is private. The collection of all public attributes (in this case, `addentry` and `title`) constitute the **public interface** of the class. This is what a user of the class should interact with. For example, one can use the class above as follows.

```
mydiary = Diary("Don’t read this!!!")
mydiary.addentry("It was a good day.")
print("The diary is called ", mydiary.title)
```

```
The diary is called  Don’t read this!!!
```

Notice that in this example, the encapsulation of the class is not about security. Heck, if it’s my diary, I should be able to read it, right? The reason to respect the private attributes and stick to the public interface is really to help us write working code that continues to work in the future. Code gets changed all the time. If you are modifying a class that is being used elsewhere in the code, you have to be careful not to break that code. If the public interface and its behavior doesn’t change, then one can be confident that the changes don’t affect the other code. One could change the name of a private variable, say changing `_entries` to `_diaryentries` and be confident that this won’t cause some other code somewhere else to break.

Inheritance and "is a" relationships

Whenever we talk about the types of things in our everyday life, it's possible to talk about them at different levels of generality. We can talk about a specific basketball player, say Kylie Irving, or we can talk about professional basketball players, or all basketball players, or people or living creatures. The specific player we started with could be said to belong to any of these classes. The same principle applies to code that we write.

Consider this example from a geometry program.

```
class Triangle:
    def __init__(self, points):
        self._sides = 3
        self._points = list(points)
        if len(self._points) != 3:
            raise ValueError("Wrong number of points.")

    def sides(self):
        return 3

    def __str__(self):
        return "I'm a triangle."

class Square:
    def __init__(self, points):
        self._sides = 4
        self._points = list(points)
        if len(self._points) != 4:
            raise ValueError("Wrong number of points.")

    def sides(self):
        return 4

    def __str__(self):
        return "I'm so square."
```

These are obviously, very closely related classes. One can make another class for which these two classes are **subclasses**. Then, anything common between the two classes can be put into the larger class or **superclass**.

```
class Polygon:
    def __init__(self, sides, points):
        self._sides = sides
        self._points = list(points)
        if len(self._points) != self._sides:
            raise ValueError("Wrong number of points.")

    def sides(self):
        return self._sides

class Triangle(Polygon):
    def __init__(self, points):
        Polygon.__init__(self, 3, points)

    def __str__(self):
        return "I'm a triangle."

class Square(Polygon):
    def __init__(self, points):
        Polygon.__init__(self, 4, points)

    def __str__(self):
        return "I'm so square."
```

Notice that the class definitions of `Triangle` and `Square` now indicate the `Polygon` class in parentheses. This is called **inheritance**. The `Triangle` class **inherits from** (or **extends**) the `Polygon` class. The **superclass** `Polyon` and the **subclasses** are `Triangle` and `Square`. When we call a method on an object, if that method is not defined in the class of that object, Python will look for the method in the superclass. This search for the correct function to call is called the **method resolution order**. If a method from the superclass is redefined in the subclass, then calling the method on an instance of the subclass calls the subclass method instead.

The initializer of the superclass is not called automatically when we create a new instance (unless we didn't define `__init__` in the subclass). In this case, we manually call the `Polygon.__init__` function. This is one of the few times where it's acceptable to call a diner method by name.

When using inheritance, you should always remember the most important rule of inheritance:

> Inheritance means **is a**.

This means that if `ClassB` extends `ClassA`, then a `ClassB` object **is a** `ClassA` object. This

should be true at the conceptual level. So, in our shapes example, we followed this rule, because a triangle is a polygon.

In total, this might look like more code. However, it has less duplication. Duplication is very bad. Even though it's easy to copy and paste on a computer, this is a source of many bugs. The reason is simple. Bugs are everywhere. If you copy and paste a code with a bug, then now you have two bugs. If you find the bug, you have to hope you remember to fix it both places. Anytime you are relying on your memory, you are going to get yourself in trouble.

Software engineers use the acronym **DRY**, to mean **Don't Repeat Yourself**. They will even use it as an adjective, saying "Keep the code DRY". The process of removing duplication by putting common code into a superclass is called **factoring out a superclass**. This is the most common way that inheritance enters a codebase. Sometimes, opportunities for inheritance are identified at the design stage, before coding begins.

Duck Typing

Inheritance is considered a staple of object-oriented programming, and it's important to understand it. That said, it's not nearly as useful in Python as it is in other languages. The reason is that python has built-in (parametric) **polymorphism**. That means we can pass any type of object we want to a function. For example, suppose we have a class to store collections of polygons as follows.

```
class PolygonCollection:
    def __init__(self):
        self._triangles = []
        self._squares = []

    def add(self, polygon):
        if polygon.sides() == 3:
            self._triangles.append(polygon)
        if polygon.sides() == 4:
            self._squares.append(polygon)
```

Notice that the `add` method would work equally well with either version of the `Triangle` and `Squa re` classes defined previously. In fact, we could pass it any object that has a method called `sides`. We didn't *need* inheritance in order to treat Triangles and squares as special cases of the same object. Sometimes, inheritance is the right way to combine classes so they can be treated as a single class, but in Python, it's not as necessary as it is in other languages.

Python's polymorphism is based on the idea of **duck typing**. The name comes from an old expression that if something walks like a duck and quacks like a duck, then it *is* a duck. In the `PolygonCollection` example, if I call the `add` method with an argument that has a `sides` method that returns something that can be compared to an integer, then the code will run without error. Having the right methods is equivalent to "walking and talking like a duck". This means that although inheritance should always mean **is a**, not every *"is a"* relationship in your code needs to be expressed with inheritance.

This idea that *not only inheritance means "is a"* is very important in Python and will be very important throughout this course. We will discuss this more when we look in more depth at abstract data types.

One example that we have already seen is the `str` function. Different types objects can be converted to strings, including objects of classes that we have defined ourselves. As long as we implemented the `__str__` method on our class, then we can call `str` on an instance of that class. That function calls the corresponding method, i.e., `str(t)` for a `Triangle` `t` calls `t.__str__()` which is equivalent to `Triangle.__str__(t)`.

Composition and "has a" relationships

There are many cases where we want objects of different types to share some functionality. Sometimes, inheritance is used to allow for this sharing, but more often we use something called **composition**. This is where one class stores an instance of another class. It allows us to produce more complex objects. The most important rule about composition is the following.

> Composition means "has a".

Consider the case where we want a class to behave like a `list`. For example, we'd like to be able to append to the list and access items by their index, but we don't want any of the other list stuff. In this case, it would be *wrong* to use inheritance. Instead, we would make our class store a list internally (composition). Then, the public interface to our class would contain the methods we want while making calls to the stored `list` instance to avoid duplicating the list implementation. Here is an example.

```
class MyLimitedList:
    def __init__(self):
        self._L = []

    def append(self, item):
        self._L.append(item)

    def __getitem__(self, index):
        return self._L[index]
```

Here, the magic method `__getitem__` will allow us to use the square bracket notation with our class. As with other magic methods, we don't call it directly.

```
L = MyLimitedList()
L.append(1)
L.append(10)
L.append(100)
print(L[2])
```

```
100
```

Chapter 3

Testing

Python is an interpreted language. This gives it a great deal of flexibility, such as duck typing. However, this can also lead to different types of common bugs. For example, if you pass a `float` to a function that really should only receive an `int`, Python won't stop you, but it might lead to unexpected behavior. In general, we have to run the code to get an error, but not all bugs will generate errors. Towards the goal of writing correct code, we use tests to determine two things:

1. **Does it work?** That is, does the code do what it's supposed to do?
2. **Does it still work?** Can you be confident that the changes you made haven't caused other part of the code to break?

Writing Tests

Testing your code means writing more code that checks that the behavior matches your expectations. This is important:

> Test behavior, not implementation.

You have some idea of what code is supposed to do. You run the code. Did it do what you expected? How about some other inputs? In the simplest case, you could simply add some code to the bottom of the module.

```
class Doubler:
    def __init__(self, n):
        self._n = 2 * n

    def n(self):
        return self._n

if __name__ == '__main__':
    x = Doubler(5)
    assert(x.n() == 10)
    y = Doubler(-4)
    assert(y.n() == -8)
```

```
  File "/Users/don/Dropbox/work/research/books/datastructures/docs/1gxmlycnq_code_chunk.py",
    if __name__ == '__main__':
                            ^
SyntaxError: invalid character in identifier
```

The `assert` statement will raise an error if the predicate that follows it is `False`. Otherwise, the program just continues as usual. Assertions are much better than just printing because you don't have to manually check to see that it printed what you expected it to print. Also, people have a tendency to delete old print statements to reduce clutter in their test output. Deleting tests after they pass is a *very bad idea*. Your code is going to change, and you will want to know if a change breaks something that *used* to work.

The line `if __name__ == '__main__':` makes sure that the tests will not run when the module is imported from somewhere else.

For some, learning to test their code runs into a substantial psychological block. They feel that testing the code will reveal its flaws and thus reveal the programmer's flaws. If you feel the slightest hesitation to testing your own code, you should practice the OGAE protocol. It stands for, "Oh Good, An Error!". Every time you get an error, you say this with honest enthusiasm. The computer has just done you a huge favor by identifying something wrong, and it has done so in the safety of your room or office. You can fix it before it becomes any bigger.

Unit Testing with `unittest`

The simple kind of testing in the module described above is fine for tiny programs that will not be

needed again, but for anything remotely serious, you will need proper **unit tests**. The word unit in *unit* testing is meant to imply a single indivisible case. Thus, unit tests are supposed to test a specific behavior of a specific function. This means you will have many tests and you will run them all, every time you change the code.

To make the process go smoothly, there is a standard package called `unittest` for writing unit tests in Python. The package provides a standard way to write the tests, the ability to run the tests all together, and the ability to see the results of the tests in a clear format. In modern software engineering, tests are also run automatically as part of build and deployment systems.

To use the `unittest` package, you will want to import the package in your test file. Then, import the code you want to test. The actual tests will be methods in a class that extends the `unittest.TestCase` class. Every test method must start with the word "test". If it doesn't start with "test", then it will not run. Tests are run by calling the `unittest.main` function.

Here is an example that tests a particular behavior of a hypothetical `DayOfTheWeek` class.

```
import unittest
from dayoftheweek import DayOfTheWeek

class TestDayOfTheWeek(unittest.TestCase):
    def testinitwithabbreviation(self):
        d = DayOfTheWeek('F')
        self.assertEquals(d.name(), 'Friday')

        d = DayOfTheWeek('Th')
        self.assertEquals(d.name(), 'Thursday')

unittest.main()
```

```
  File "/Users/don/Dropbox/work/research/books/datastructures/docs/cjovrq8vl_code_chunk.py",
    d = DayOfTheWeek('F')
                     ^
SyntaxError: invalid character in identifier
```

Notice that even if we have never seen the code for the `DayOfTheWeek` class, we can get a good sense of its expected behavior from reading the tests. In this case, we see that it can be instantiated with the abbreviation `F` and the `name()` function will return the value "Friday". It often happens that unit tests like this give the clearest specification of a data structure's expected behavior.

Moreover, because the tests can be executed, one can be certain that the class really has the expected behavior. With documentation, one sometimes finds that changes in the code are not reflected in the documentation, but passing tests don't have this problem.

Test-Driven Development

Test-Driven Development (TDD) is based on the simple idea that you can write the tests before you write the code. But won't the test fail if the code hasn't been written yet? Yes, if it's a good test. What if it passes? Then, either you're done (unlikely) or there is something wrong with your test.

Writing tests first forces you to do two things:

1. Decide how you want to be able to use some function. What should the parameters be? What should it return?
2. Write only the code that you need. If there is code that doesn't support some desired behavior with tests, then you don't need to write it.

The TDD mantra is **Red-Green-Refactor**. It refers to three phases of the testing process.

- **Red:** The tests fail. They better! You haven't written the code yet!
- **Green:** You get the tests to pass by changing the code.
- **Refactor:** You clean up the code, removing duplication.

The terms "Red" and "Green" refer to many testing fameworks that show failed tests in red and passing tests in green.

Refactoring is the process of cleaning up code, most often referring to the process of removing duplication. Duplication in code, whether it comes from copy-and-paste or just repeating logic can be a source of many errors. If you duplicate code with a bug, now you have two bugs. If you find the bug, you will have to find it twice.

Here is a simple example of refactored code:

Original Code with Minor Duplication:

```
avg1 = sum(L1)/len(L1)
avg2 = sum(L2)/len(L2)
```

Then, it is observed that there should be some default behavior for empty lists so (a test is added

and) the code is updated as follows.

Updated Code Before Refactoring:

```
if len(L1) == 0:
    avg1 = 0
else:
    avg1 = sum(L1) / len(L1)

if len(L2) == 0:
    avg2 = 0
else:
    avg2 = sum(L2) / len(L2)
```

Refactored Code:

```
def avg(L):
    if len(L) == 0:
        return 0
    else:
        return sum(L) / len(L)

avg1 = avg(L1)
avg2 = avg(L2)
```

In the refactored code, the details of the `avg` function are not duplicated. If we want to later modify how the code handles empty lists, we will only have to change it in one place.

The refactored code is also easier to read.

What to Test

Step away from the computer. Think about the problem you are trying to solve. Think about the methods you are writing. Ask yourself, "*What should happen when I run this code?*". Also ask yourself, "*How do I want to use this code?*"

- Write tests that use the code the way it ought to be used.
- Then write tests that use the code incorrectly to test that your code *fails gracefully*. Does it give clear error messages?

- Test the *edge cases*, those tricky cases that may rarely come up. Try to break your own code.
- Turn bugs into tests. A bug or an incorrect behavior can reappear after you fix it. You want to catch it when it does. Sometimes you notice a bug when a different test fails. Write a specific test to reveal the bug, then fix it.
- Test the public interface. Usuually you don’t need to or want to test the private methods of a class. You should treat the test code as a user of the class and it should make no assumptions about private attributes. This way, if a private gets renamed or refactored, you don’t have to change the tests.

Testing and Object-Oriented Design

In object-oriented design, we divide the code into classes. These classes have certain relationships sometimes induced by inheritance or composition. The classes have public methods. We call these public methods the **interface** to the class.

To start a design, we look at the problem and identify nouns (classes) and verbs (methods). In our description, we express what *should* happen. Often these expectations are expressed in "if...then" language, i.e., "if I call this method with these parameters, then this will happen.". A unit test will encode this expectation. It will check that the actual behavior of the code matches the expected behavior.

When writing a class, it helps focus our attention and reduce the number fo things to think about if we assume each class works the way it is supposed to. We try to make this true by testing each class individually. Then, when we compose classes into more complex classes, we can have more confidence that any errors we find are in the new class and not somewhere lurking in the previously written classes.

At first it may seem like a waste of time to thoroughly test your code. However, any small savings in time you might reap early on by skipping tests will very quickly be spent in the headaches of countless hours debugging untested code. When you have lots of untested code, every time there is an unexpected behavior, the error could be anywhere. The debugging process can virtually grind to a halt. If and when this happens to you: Stop. Pick one piece. Test it. Repeat. Being careful and systematic will take you substantially less time overall. **It is faster to go slow**.

Chapter 4

Running Time Analysis

Our major goal in programming is to write code that is *correct*, *efficient*, and *readable*. When writing code to solve a problem, there are many ways to write correct code. When the code is not correct, we debug it. When the code is inefficient, we may not notice when running small tests because we only see the problem when inputs get larger. It can be much harder to track down problems with efficiency.

We want to develop a vocabulary for describing the efficiency of our code. Sentences like "This code is fast" or "This code is slow" tell us very little. How fast or slow is it? Moreover, the same code will take different amounts of time and memory depending on the input. It will also run faster on a faster computer. So, our goals are challenging:

> We want to give a nuanced description of the efficiency of a program that adapts to *different inputs* and to *different computers*.

We will achieve both goals with **asymptotic analysis**. To develop this theory, we will start by measuring the time taken to run some programs. By simple experiments, we can observe how the running times change as the inputs get larger. For example, a program that takes a list as input may run slower if that list has a million items than if it only has ten items, but maybe not. It will depend on the program.

Next, we will give an accounting scheme for counting up the **cost** of a program. This will assign costs to different operations. The cost of the whole program will be the sum of the costs of all the operations executed by the program. Often, the cost will be a function of the input size, rather than a fixed number.

Finally, we will introduce some vocabulary for classifying functions. This is the asymptotic part of asymptotic analysis. The **big-O** notation gives a very convenient way of grouping this (running time) functions into classes that can easily be compared. Thus, we'll be able to talk about and compare the efficiency of different algorithms or programs without having to do extensive experiments.

Thinking about and analyzing the efficiency of programs helps us develop good habits and intuitions that lead to good design decisions. The goal of increasing efficiency in our data structures will be the primary motivation for introducing new structures and new ideas as we proceed.

Timing Programs

To start, let's observe some differences in running time for different functions that do the same thing. We might say these functions have the same behavior or the same **semantics**.

Here is a function that takes a list as input; it returns `True` if there are any duplicates and `False` otherwise.

```
def duplicates1(L):
    n = len(L)
    for i in range(n):
        for j in range(n):
            if i != j and L[i] == L[j]:
                return True
    return False

assert(duplicates1([1,2,6,3,4,5,6,7,8]))
assert(not duplicates1([1,2,3,4]))
```

A basic question that we will ask again and again is the following?

> How fast is this code?

The simplest answer to this question comes from simply running the program and measuring how long it takes. We could do this as follows.

```
import time

for i in range(5):
    n = 1000
    start = time.time()
    duplicates1(list(range(n)))
    timetaken = time.time() - start
    print("Time taken for n = ", n, ": ", timetaken)
```

```
Time taken for n =  1000 :  0.885455846786499
Time taken for n =  1000 :  0.29903507232666016
Time taken for n =  1000 :  0.3108839988708496
Time taken for n =  1000 :  0.26366209983825684
Time taken for n =  1000 :  0.26892900466918945
```

Notice that we see some variation in the time required. This is caused by many factors, but the main one is that the computer is performing many other tasks at the same time. It is running an operating system and several other programs at the same time. Also, if run on different computers, one could expect to see wildly different results based on the speed of the processor and other differences between the computers. For now, let's run the code several times and take the average to smooth them out for a given computer. We'll wrap this idea in a general function that takes another function as input, and a length. It runs the given function on lists of the given length and averages the time take per run.

```
import time

def timetrials(func, n, trials = 10):
    totaltime = 0
    start = time.time()
    for i in range(trials):
        func(list(range(n)))
        totaltime += time.time() - start
    print("average =%10.7f for n = %d" % (totaltime/trials, n))
```

We can now look at the average running time as the length of the list gets longer. It's not surprising to see that the average time goes up as the length n increases.

```
for n in [50, 100, 200, 400, 800, 1600, 3200]:
    timetrials(duplicates1, n)
```

```
average = 0.0020274 for n = 50
average = 0.1165188 for n = 100
average = 0.3106624 for n = 200
average = 0.3800069 for n = 400
average = 0.9648227 for n = 800
average = 3.5948985 for n = 1600
average =10.8566278 for n = 3200
```

Let's try to make our code faster. Simple improvements can be made by eliminating situations where we are doing unnecessary or redundant work. In the `duplicates1` function, we are comparing each pair of elements twice because both `i` and `j` range over all n indices. We can eliminate this using a standard trick of only letting `j` range up to `i`. Here is what the code would look like.

```
def duplicates2(L):
    n = len(L)
    for i in range(1,n):
        for j in range(i):
            if L[i] == L[j]:
                return True
    return False
```

```
for n in [50, 100, 200, 400, 800, 1600, 3200]:
    timetrials(duplicates2, n)
```

```
average = 0.0009027 for n = 50
average = 0.0029751 for n = 100
average = 0.1852228 for n = 200
average = 0.5709280 for n = 400
average = 0.3970987 for n = 800
average = 1.3742860 for n = 1600
average = 5.5493487 for n = 3200
```

There is a python shortcut the kind of loop used in `duplicates2`. The `any` function takes an iterable collection of booleans and returns `True` if any of the booleans are true. You can make an iterable collection in an expression in the same way one does for comprehensions. This can be very

handy.

```
def duplicates3(L):
    n = len(L)
    return any(L[i] == L[j] for i in range(1,n) for j in range(i))
```

```
for n in [50, 100, 200, 400, 800, 1600, 3200]:
    timetrials(duplicates3, n)
```

```
average = 0.0010976 for n = 50
average = 0.0047515 for n = 100
average = 0.2850778 for n = 200
average = 0.4441026 for n = 400
average = 0.5427464 for n = 800
average = 1.8794848 for n = 1600
average = 7.2552740 for n = 3200
```

This last optimization reduces the number of lines of code and may be desirable for code readability, but it doesn't improve the speed.

If we want to make a real improvement, we'll need a substantial new idea. One possibility is to sort the list and look at adjacent elements in the list. If there are duplicates, they will be adjacent after sorting.

```
def duplicates4(L):
    n = len(L)
    L.sort()
    for i in range(n-1):
        if L[i] == L[i+1]:
            return True
    return False
```

```
def duplicates5(L):
    n = len(L)
    L.sort()
    return any(L[i] == L[i+1] for i in range(n-1))

def duplicates6(L):
    s = set()
    for e in L:
        if e in s:
            return True
        s.add(e)
    return False

def duplicates7(L):
    return len(L) == len(set(L))

def duplicates8(L):
    s = set()
    return any(e in s or s.add(e) for e in L)
```

```
for n in [50, 100, 200, 400, 800, 1600, 3200]:
    print("Quadratic: ", end="")
    timetrials(duplicates3, n)
    print("Sorting:   ", end="")
    timetrials(duplicates5, n)
    print("Sets:      ", end="")
    timetrials(duplicates7, n)
    print('--------------------------')
```

```
Quadratic: average = 0.0010944 for n = 50
Sorting:   average = 0.0000747 for n = 50
Sets:      average = 0.0000256 for n = 50
---------------------------
Quadratic: average = 0.0052706 for n = 100
Sorting:   average = 0.0347797 for n = 100
Sets:      average = 0.0017203 for n = 100
---------------------------
Quadratic: average = 0.1026614 for n = 200
Sorting:   average = 0.0413665 for n = 200
Sets:      average = 0.0000539 for n = 200
---------------------------
Quadratic: average = 0.2487745 for n = 400
Sorting:   average = 0.0004653 for n = 400
Sets:      average = 0.0001281 for n = 400
---------------------------
Quadratic: average = 0.5434304 for n = 800
Sorting:   average = 0.0009512 for n = 800
Sets:      average = 0.0002215 for n = 800
---------------------------
Quadratic: average = 1.8862489 for n = 1600
Sorting:   average = 0.0030383 for n = 1600
Sets:      average = 0.0005270 for n = 1600
---------------------------
Quadratic: average = 7.2847819 for n = 3200
Sorting:   average = 0.0048042 for n = 3200
Sets:      average = 0.0008232 for n = 3200
---------------------------
```

Some key ideas: The Data Structures really matter. The speed of the set membership testing or set construction gives a big improvement. We made assumptions about the elements of the list, that they were comparable or hashable. These assumptions can be thought of as assumptions about the *type* of the elements. This is not the same as the class. It's duck typing. The assumption is that certain methods can be called. The example also gives some practice using generator expressions.

The main important idea is that the running time depends on the size of the input. The time goes up as the length goes up. Sometimes, the gap between two pieces of code will increase as the input size grows.

Example: Adding the first k numbers.

Below is a program that adds up the first k positive integers and returns both the sum and time required to do the computation.

```
import time

def sumk(k):
    start = time.time()

    total = 0
    for i in range(k+1):
        total = total + i
    end = time.time()

    return total, end-start
```

```
for i in range(5):
    print("Sum: %d, time taken: %f" % sumk(10000))
```

```
Sum: 50005000, time taken: 0.000870
Sum: 50005000, time taken: 0.000871
Sum: 50005000, time taken: 0.000868
Sum: 50005000, time taken: 0.000868
Sum: 50005000, time taken: 0.000869
```

Notice that we see some variation in the time required. This is caused by many factors, but the main one is that the computer is performing many other tasks at the same time. It is running an operating system and several other programs at the same time. Also, if run on different computers, one could expect to see wildly different results based on the speed of the processor and other differences between the computers. For now, let's run the code several times and take the average to smooth them out for a given computer.

```
def timetrials(func, k, trials = 10):
    totaltime = 0
    for i in range(trials):
        totaltime += func(k)[1]
    print("average =%10.7f for k = %d" % (totaltime/trials, k))
```

```
timetrials(sumk, 10000)
timetrials(sumk, 100000)
timetrials(sumk, 1000000)
timetrials(sumk, 10000000)
```

```
average = 0.0009853 for k = 10000
average = 0.0745956 for k = 100000
average = 0.1778027 for k = 1000000
average = 1.4838944 for k = 10000000
```

Seeing the times for different values of `k` reveals a rather unsuprising pattern. As `k` goes up by a factor of 10, the time required for `sumk` also goes up by a factor of `10`. This makes sense, because it has to do about `k` additions and assignments. To say it another way, the time is proportional to `k`. We will often be more concerned with finding what the running time is proportional to than finding the exact time itself. We expect to see this relationship between the running time and the input `k` regardless of what computer we run the code on.

The code we wrote seems to be correct, however, there is another, much simpler way to compute the sum of the numbers from `1` to `k` using a formula that is very important for this class. It will show up again and again. To prove that

$$\sum_{i=1}^{k} i = 1 + 2 + 3 + \cdots + k = k(k+1)/2,$$

it suffices to observe that you can add the numbers in pairs, matching i with $k - i + 1$ starting with 1 and k. There are $k/2$ such pairs and each adds up to $k + 1$. Let's use this formula to rewrite our `sumk` function and time it.

```
import time

def sumk2(k):
    start = time.time()
    total = (k*(k+1)//2)
    end = time.time()
    return total, end-start

timetrials(sumk2, 10000)
timetrials(sumk2, 100000)
timetrials(sumk2, 1000000)
timetrials(sumk2, 10000000)
timetrials(sumk2, 100000000)
```

```
average = 0.0000003 for k = 10000
average = 0.0000005 for k = 100000
average = 0.0000003 for k = 1000000
average = 0.0000005 for k = 10000000
average = 0.0000002 for k = 100000000
```

This is much much faster. Even as `k` becomes very large, it doesn't seem to slow down.

Modeling the Running Time of a Program

We will introduce a general technique for describing and summarizing the number of operations required to run a piece of code, be it a single line, a function, or an entire program. Along the way, we will develop a vocabulary for comparing the efficiency of algorithms that doesn't require us to run them and time them.

It's not enough to count lines of code. A single line of code can do a lot of stuff. Here's a one line function that does all kinds of stuff. It creates a list of 200 items and sums all the entries for each of value of i from 0 to k-1 and returns a list of the results.

```
def f001(k):
    return [sum([i, i + 1] * 100) for i in range(k)]

print(f001(9))
```

```
[100, 300, 500, 700, 900, 1100, 1300, 1500, 1700]
```

Instead, we will want to count operations a little more carefully. The unit we will use to describe the **running time** of an algorithm is the number of atomic operations. This is not exactly a unit of time, but at some level, the atomic operations that we will describe can all be performed in a small number of clock cycles on your CPU and so correspond to a real amount of time.

(Please don't say "runtime" as a replacement for "running time". These are not the same thing!)

Atomic operations include

- arithmetic and boolean operations
- variable assignment
- accessing the value of a variable from its name
- branching (jumping to another part of the code for if/for/while statements)
- calling a function
- returning from a function

Below, there is listings of the asymptotic running time of the most common operations on the standard python collections classes. You should familiarize yourself with these listings. In particular, you should be aware of which operations on collections produce a new copy of the collection. For example, concatenation and slicing both produce a new collection and thus the running times are proportional to the length of the newly created collection. In almost all cases, one can see the reason for the running times by understanding what work the algorithms must do and also how the data structure is laid out in memory.

List Operations

Operation Name	Code	Cost
index access	`L[i]`	1
index assignment	`L[i] = newvalue`	1
Append	`L.append(newitem)`	1
Pop (from end of list)	`L.pop()`	1
Pop (from index `i`)	`L.pop(i)`	$n - i$

Insert at index `i`	`insert(i, newitem)`	$n - i$
Delete an item (at index `i`)	`del(item)`	$n - i$
Membership testing	`item in L`	n
Slice	`L[a:b]`	$b - a$
Concatenate two lists	`L1 + L2`	$n_1 + n_2$
Sort	`L.sort()`	$n \log_2 n$

Note that these running times are the same for the other sequential collections, `list` and `str` assuming the operation exists for those immutable types. For example, index access, membership testing, slicing, and concatenation all work. Remember that slicing and concatenation produce new objects and don't change the originals.

Dictionary Operations

Unlike the `list` operations, the costs of `dict` operations are a bit mysterious. Some may seem downright impossible. Should it really cost just one atomic operation to test if a given item is in a set of a billion elements? There are three things to remember:

1. We will study how dictionaries are implemented and how they exploit one of the wonderful, clever ideas of computer science.
2. This is just a model, albeit a useful and accurate one.
3. The actual cost is a kind of average. It could take longer sometimes.

Operation Name	Code	Cost
Get item	`D[key]`	1
Set item	`D[key] = value`	1
(key) membership testing	`key in D`	1
Delete an item by its key	`del D[key]`	1

Set Operations

A `set` is very much like a `dict` where the entries have keys but no values. They are implemented

the same way and so, the running times for their common operations are the same. Some set operations that correspond to our mathematical idea of a set do not correspond to operations on dictionaries. Operations that produce a new set will leave the input sets unchanged. Below, let n_A be the size of set `A` and let n_B be the size of set `B`.

Operation Name	Code	Cost
Add a new item	`A.add(newitem)`	1
Delete an item	`A.delete(item)`	1
Union	`A | B`	$n_A + n_B$
Intersection	`A & B`	$\min\{n_A, n_B\}$
Set differences	`A - B`	n_A
Symmetric Difference	`A ^ B`	$n_A + n_B$

Asymptotic Analysis and the Order of Growth

The goal is not to predict exactly how much time an algorithm will take, but rather to predict the **order of growth** of the time as the input size grows. That is, if we have algorithm that operates on a list of length n, the running time could be proportional to n. In that case, the algorithm will take 100 times longer on a list that is 100 times longer. A second algorithm might have a running time proportional to n^2 for inputs of length n. Then, the algorithm will take 10000 times longer on a list that is 100 times longer. Notice, that *the exact constant of proportionality is not important for these facts.*

The *size of the input* refers to the number of bits needed to encode it. As we will be ignoring constant factors, we could just as easily refer to the number of words (a word is 64 bits) needed to encode it. An integer or a float is generally stored in one word. Technically, we can store some really big numbers in a python integer which would require many more words, but as a convention, we will assume that ints and floats fit in a constant number of bits. This is necessary to assume that arithmetic takes constant time.

Focus on the Worst Case

Usually, different inputs of the same length may have different running times. The standard convention we will use most of the time is to consider the worst case. We are looking for **upper bounds** on the running time. If the algorithm has a running time that is better than the analysis predicts, that's okay.

Big-O

We will almost always describe running times as a function of the input size. That is, the running time on an input of size n might be $5n^2 + 3n + 2$. If we were to write this as a (mathematical) function of n, we could call it f and write $f(n) = 5n^2 + 3n + 2$. In this example, the $5n^2$ is by far the most important

The formal mathematical definition that allows us to ignore constant factors is called the **big-O notation**. As a warmup example, we say that

$$f(n) = O(n^2)$$

if there exists a constant c such that

$$f(n) \leq cn^2 \text{for all sufficiently large } n.$$

This is correct for $f(n) = 5n^2 + 3n + 2$ because if we take $c = 6$, we see that as long as $n > 4$, we have

$$f(n) = 5n^2 + 3n + 2 < 5n^2 + 4n < 5n^2 + n^2 \leq 6n^2.$$

The above inequalities came from repeated using the fact that $n > 4$.

We can now state the formal definition of the big-O notation: Given (nondecreasing) functions f and g, we say $f(n) = O(g(n))$ if there exist constants c and n_0 such that for all $n > n_0$ we have $f(n) \leq cg(n)$.

The most important features of big-O usage

1. The big-O *hides constant factors*. Any term that does not depend on the size of the input is considered a constant and will be suppressed in the big-O.
2. The big-O tells us about what will eventually be true *when the input is sufficiently large*.

These two features are present in the formal definition. The constant c is the constant that stands in for all other constant factors. This constant also allows us to suppress lower order terms. The constant n_0 is the threshold after which the inequality is true.

Practical Use of the Big-O and Common Functions

Even though the definition of the big-O notation allows us to compare all kinds of functions, we will usually use it to simplify functions, eliminating extraneous constants and low order terms. So, for example, you should write $O(n)$ instead of $O(3n)$ and $O(n^2)$ instead of $O(5n^2 + 3n + 2)$. There are several functions that will come up so often that we will want to have them in our vocabulary.

- **Constant Functions**, $O(1)$
- **Logarithmic Functions**, $O(\log n)$
- **Linear Functions**, $O(n)$
- **"n Log n"**, $O(n \log n)$
- **Quadratic Functions**, $O(n^2)$
- **Polynomial Functions**, $O(n^k)$ for some constant k.
- **Exponential Functions**, $O(2^n)$ (this is different from $2^{O(n)}$)
- **Factorial Functions**, $O(n!)$

Bases for Logarithms

You may have noticed that we didn't give a base for the logarithm above. The reason is that inside the big-O, logarithms of any constant base are the same. That is, $\log_a(n) = O(\log_b(n))$, where a and b are any two constants. Here's the proof. Let $c = \frac{1}{\log_b(a)}$ and $n_0 = 0$.

$$\log_a(n) = \frac{\log_b(n)}{\log_b(a)} \leq c \log_b(n) \text{ for all } n > n_0.$$

Practice examples

In each of the following examples, assume the input is a list of length n.

```
def f002(L):
    newlist = []  # 2 creating a new list and variable assignment
    for i in L:   # loops n times
        if i % 2 == 0:  # 1
            newlist.append(i) # 1 (append is constant time on lists)
    return newlist  # 1 return
```

Let's count up the cost of each line of code. The costs are in the comments. So, the total cost is something like $2n + 3$ in the worst case (i.e. when all the items are even). We would report this as $O(n)$ and we would call this a **linear-time** algorithm, or sometimes simply a linear algorithm.

```
def f003(L):
    x = 0   # 1
    for i in L:   # loop n times
        for j in L:   # loop n times
            x += i*j  # 3 two arithmetic operations and assignment
    return x # 1
```

Again, let's count up the cost of each line of code. The costs are in the comments. The inner loop costs $3n$ and it runs n times, so the total for the whole method is $3n^2 + 2$. We would report this as $O(n^2)$ and call this a **quadratic-time** algorithm, or sometimes simply a quadratic algorithm.

Here's an example we've seen several times.

```
def f004(L):
    x = 0 # 1
    for j in range(1,len(L))  # loops n-1 times
        for i in range(j) # loops j times
            x += L[i] * L[j] # 5 two list accesses, two arithmetic operations and assignment
    return x # 1
```

It's a little trickier to figure this one out because the number of times the inner loop runs changes with each iteration of the outer loop. In this case, it's not hard to add up the cost of each of the outer loops one at a time. The first costs 5, the second costs 10, and so on to that the jth costs $5j$. The total costs (including initializing `x` and returning is

$$2 + \sum_{i=1}^{n-1} 5i = 2 + 5\sum_{i=1}^{n-1} i = 2 + \frac{5n(n-1)}{2} = O(n^2).$$

We will see this kind of sum often so it's worth recognizing it (both in the code and as a mathematical expression).

Chapter 5

Stacks and Queues

Abstract Data Types

Throughout the book, we will use **abstract data types** or **ADTs** as a starting point for any discussion of a particular data structure. An ADT is not a data structure, but it does tell us about data structures. The way we use the term ADT, it will be very similar to the term **interface**. An ADT answers two main questions:

1. What is the *data* to be stored or represented?
2. What can we do with the data?

These together describe the **behavior** or **semantics** of the data structure. When we give an ADT, we will list the names of the methods that will be present, what kind of input they take, and what is their expected output. The ADT may also describe error situations and what should happen if they occur. A **data structure** is an implementation of an ADT. To make, this distinction, it is sometimes useful to call them **concrete data structures**, though we will usually omit the word "concrete".

The ADT tells us what methods the data structure will implement. However, the ADT *does not* give an hints or prescriptions for how the data structure is implemented. This is important both as a definition, but also as a guiding design idea in object-oriented programming, that I will write it again:

> **The ADT should be independent of all concerns about its implementation.**

You may have noticed that the two questions ADTs answer are related to the definition of **encapsulation** we gave in our earlier discussion of object-oriented programming. As such, when we

implement data structures in python, we will package them as classes.

The Stack ADT

- **push** - add a new item to the stack.
- **pop** - remove and return the next item in Last In First Out (LIFO) ordering.
- **peek** - return the next item in LIFO ordering.
- **size** - returns the number of items in the stack (we'll use the pythonic `__len__` method)
- **isempty** - return `True` if the stack has no items and return `False` otherwise.

This ADT can be implemented quite easily using a `list`. We will implement it with a class called `ListStack`. Here, we are giving hints about the implementation in the name. This is more common in Java programming, but we adopt the convention in the book to help us distinguish between different implementations of the same ADT.

```
class ListStack:
    def __init__(self):
        self._L = []

    def push(self, item):
        self._L.append(item)

    def pop(self):
        return self._L.pop()

    def peek(self):
        return self._L[-1]

    def __len__(self):
        return len(self._L)

    def isempty(self):
        return len(self) == 0
```

The `Stack` class above illustrates the object-oriented strategy of *composition* (the `Stack` has a `list`). It is also an example of the **Wrapper Pattern**. The Python builtin `list` is doing all the heavy lifting, but from the user's perpective, they don't know or care how the methods are implemented. This is not exactly true. The user would start to care if the performance is bad. It wouldn't be too

hard to make this inefficient. For example, we could have implemented the Stack by pushing new items into the front of the list. Here we use inheritance to avoid rewriting the methods that will not be changing.

```
class BadStack(Stack):
    def push(self, item):
        self._L.insert(0, item)

    def pop(self):
        return self._L.pop(0)

    def peek(self):
        return self._L[0]
```

A simple asymptotic analysis shows why this implementation is far less efficient. Inserting a new item into a list requires that all the other items in the list have to move over to make room for the new item. If we insert at the beginning of the list, then every item has to be copied to a new position. Thus, the `insert` call in `push` takes $O(n)$ time. Similarly, if we pop an item at the beginning of the list, then every other item in the list gets moved over one space to fill in the gap. Thus, the `list.pop` call in our `pop` method will take $O(n)$ time as well. So, `push` and `pop` both take linear time in this implementation. It really earns its name.

The Queue ADT

- **enqueue** - add a new item to the queue.
- **dequeue** - remove and return the next item in First In First Out (FIFO) ordering.
- **len** - returns the number of items in the queue.
- **isempty** - return `True` if the queue has no items and return `False` otherwise.

```
class Queue:
    def __init__(self):
        self._L = []

    def enqueue(self, item):
        self._L.append(item)

    def dequeue(self):
        return self._L.pop(0)

    def __len__(self):
        return len(self._L)

    def isempty(self):
        return len(self) == 0
```

"But wait," you say. "I thought calling `pop(0)` was a bad thing to do."

Yes, it takes time proportional to the length of the list, but what can we do? If we dequeue off the end of the list, we would have to enqueue by inserting into the front of the list. That's bad too.

Here's a different idea. Let's not really delete things from the front of the list. Instead, we'll ignore them by keeping the index of the head of the queue.

```
class Queue:
    def __init__(self):
        self._head = 0
        self._L = []

    def enqueue(self, item):
        self._L.append(item)

    def dequeue(self):
        item = self._L[self._head]
        self._head += 1
        return item

    def __len__(self):
        return len(self._L) - self._head

    def isempty(self):
        return len(self) == 0
```

There is something a little odd about this code: it never gets rid of old items after they have been dequeued. Even if it deleted them, it still keeps a place in the list for them. This is a kind of **lazy** update. Shouldn't we clean up after ourselves? Yes, but let's wait. Here's the idea. If the list ever gets half empty, that is, if `_head` is more than half the length of `_L`, then we will bite the bullet and replace `_L` with a slice of it. "Biting the bullet" is an especially good turn of phrase here if you view this process as a kind of amputation of the old gangrenous stump of the list.

```
class ShrinkingQueue(Queue):
    def dequeue(self):
        item = self._L[self._head]
        self._head += 1
        if self._head > len(self._L)//2:
            self._L = self._L[self._head:]
            self._head = 0
        return item
```

So, it looks like we lost all the benefits of our lazy update, because now we have a `dequeue` method that sometimes takes linear time. However, we don't do that *every* time. How expensive is it really? If we do all our `enqueue` operations first, and then dequeue all our items afterwards, then some items at the end of the list get moved (i.e. copied to a new memory location during the slicing operation) many times. The first half of the items don't get moved at all. The next quarter of the list

(from $1/2$ to $3/4$) gets moved exactly one time. The next eighth of the list moves exactly twice. Of n total items, there are $n/2^i$ items that get moved $i-1$ times. Thus the total number of moves for n items is at most $n\sum_{i=1}^{\log n}\frac{i-1}{2^i} < n$. So, "on average", the cost per item is constant.

This kind of lazy update is very important. In fact, it's how python is able to do `list.pop` quickly. Technically, `pop()` can also take linear time for some calls but on average, the cost is constant per operation. The same idea makes `append` fast. In that case, python allocates extra space for the list and every time it fills up, the list is copied to a bigger area, that roughly doubles in size.

Dealing with errors

Often, we make assumptions about how a class can or ought to be used. This is also considered part of the semantics of the class and it affects how we program it. We would like to write error-free code. We'd like to make it so that it always works no matter how it gets misused, but sometimes, an error message is the correct behavior.

In python we **raise an error** the way one might throw an egg. Either someone gently catches it or it crashes. Depending on the situation, either might be the right thing to do.

In the case of a stack, it is never correct usage to `pop` from an empty stack. Thus, it makes sense that someone using our `Stack` class should have their program crash and see an error message if they attempt to call `pop` when there are no items left on the stack. In the list implementation above, this does happen:

```
s = ListStack()
s.push(5)
s.pop()
s.pop()
```

```
Traceback (most recent call last):
  File "/Users/don/Dropbox/work/research/books/datastructures/docs/vmzyt5d2w_code_chunk.py",
    s.pop()
  File "/Users/don/Dropbox/work/research/books/datastructures/docs/vmzyt5d2w_code_chunk.py",
    return self._L.pop()
IndexError: pop from empty list
```

If we look at the error message, it even seems pretty good. It says we tried to `pop from empty lis t`. But if you look at the code, you might ask, "What list?" We know how the `ListStack` class is

implemented, and the name even gives a hint at its implementation, so we might guess what's going on, but the user does have to search up the stack trace a little to see the line of their own code that caused the problem. We could catch the exception in our `pop` method and raise a different error so the source of the problem is more obviously the user's code. Otherwise, the stack trace reports the error in our code. Then, a user, might have to try to understand our class in order to backtrack to understand what they did wrong in their code. Instead, give them an error that explains exactly what happened.

```
class AnotherStack(ListStack):
    def pop(self):
        try:
            return self._L.pop()
        except IndexError:
            raise RuntimeError("pop from empty stack")

s = AnotherStack()
s.push(5)
s.pop()
s.pop()
```

```
Traceback (most recent call last):
  File "/Users/don/Dropbox/work/research/books/datastructures/docs/8w3cer6w1_code_chunk.py",
    return self._L.pop()
IndexError: pop from empty list

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/Users/don/Dropbox/work/research/books/datastructures/docs/8w3cer6w1_code_chunk.py",
    s.pop()
  File "/Users/don/Dropbox/work/research/books/datastructures/docs/8w3cer6w1_code_chunk.py",
    raise RuntimeError("pop from empty stack")
RuntimeError: pop from empty stack
```

Chapter 6

Deques and Linked Lists

A **deque** (pronounced "deck") is a doubly-ended queue. It acts like both a stack and a queue, storing an ordered collection of items with the ability to add or remove from both the beginning and the end. Here is the ADT.

The Deque ADT

- **addfirst(item)** - add `item` to the front of the deque.
- **addlast(item)** - add `item` to the end of the deque.
- **removefirst(item)** - remove and return the first item in the deque.
- **removelast(item)** - remove and return the last item in the deque.
- **len** - return the number of items in the deque.

As a start, let's see what a deque implementation would look like using a list.

```
class ListDeque:
    def __init__(self):
        self._L = []

    def addfirst(self, item):
        self._L.insert(0, item)

    def addlast(self, item):
        self._L.append(item)

    def removefirst(self):
        return self._L.pop(0)

    def removelast(self):
      return self._L.pop()
```

The code is simple enough, but a couple of operations will start to get slow as the deque gets large. Inserting and popping at index 0 takes $O(n)$ time. This follows from the way lists are stored sequentially in memory. There is no way to change the beginning of the list without shifting all the other elements to make room or fill gaps. To make this work, we are going to have to give up on the idea of having the items laid out sequentially in memory. Instead, we'll store some more information with each item that we can use to extract the order.

Linked Lists

Linked lists are a simple data structure for storing a sequential collection. Unlike a standard python list, it will allow us to insert at the beginning quickly. The idea is to store the items in individual objects called **nodes**. A special node is the head of the list. Each node stores a reference to the next node in the list.

We will use the usual trick of looking back into our intuitive description above to find the nouns that will be the types we will need to define. In this case, there is the list itself and the nodes. Let's write a class for a `ListNode`.

```
class ListNode:
    def __init__(self, data, link = None):
        self.data = data
        self.link = link
```

Now, to start the `LinkedList`, we will store the head of the list. We will provide two methods, `addfirst` and `removefirst` both of which modify the beginning of the list. These will behove roughly like the push and pop operations of the stack. This first implementation will hide the nodes from the user. That is, from a users perspective, they can create a linked list, and they can add and remove nodes, but they don't have to touch (or even know about) the nodes. This is abstraction (hiding details)!

```
class LinkedList:
    def __init__(self):
        self._head = None

    def addfirst(self, item):
        self._head = ListNode(item, self._head)

    def removefirst(self):
        item = self._head.data
        self._head = self._head.link
        return item
```

Implementing a Queue with a LinkedList

Recall that even our best list implementation of a Queue required linear time in the worst case for `dequeue` operations (though constant on average). We could hope to do better with a linked list. However, right now, we have no way to add to or remove from the end of the linked list. Here is an inefficient, though simple and correct way to do it.

```
class LinkedList:
    def __init__(self):
        self._head = None

    def addfirst(self, item):
        self._head = ListNode(item, self._head)

    def addlast(self, item):
        if self._head is None:
            self.addfirst(item)
        else:
            currentnode = self._head
            while currentnode.link is not None:
                currentnode = currentnode.link
            currentnode.link = ListNode(item)

    def removefirst(self):
        item = self._head.data
        self._head = self._head.link
        return item

    def removelast(self):
        if self._head.link is None:
            return self.removefirst()
        else:
            currentnode = self._head
            while currentnode.link.link is not None:
                currentnode = currentnode.link
            currentnode.link = None
            return currentnode.data
```

The new `addlast` method implements a very common pattern in linked lists. It starts at the head of the linked list and **traverses** to the end by following the `link`s. It uses the convention that the `link` of the last node is `None`.

Similarly, `removelast` traverses the list until it reaches the second to last element.

This traversal approach is not very efficient. For a list of length n, we would need $O(n)$ time to find the end.

A different approach might be to store the last node (or **tail**) of the list so we don't have to search for

it when we need it. This requires a bit of overhead to make sure it always stores the correct node. Most of the special cases happen when there is only one item in the list. We will be able to use this to get some improvement for `addlast`, because we will be able to jump right to the end without traversing. We will also be able to clean up the code for `removelast` a little by eliminating the `link .link` stuff and instead just check if we reached the tail.

```
class LinkedList:
    def __init__(self):
        self._head = None
        self._tail = None

    def addfirst(self, item):
        self._head = ListNode(item, self._head)
        if self._tail is None: self._tail = self._head

    def addlast(self, item):
        if self._head is None:
            self.add(item)
        else:
            self._tail.link = ListNode(item)
            self._tail = self._tail.link

    def removefirst(self):
        item = self._head.data
        self._head = self._head.link
        if self._head is None: self._tail = None
        return item

    def removelast(self):
        if self._head is self._tail:
            return self.removefirst()
        else:
            currentnode = self._head
            while currentnode.link is not self._tail:
                currentnode = currentnode.link
            item = self._tail.data
            self._tail = currentnode
            self._tail.link = None
            return item
```

Now we can implement the Queue ADT with a linked list. It will be surprisingly easy.

```
# linkedqueue.py
class LinkedQueue:
    def __init__(self):
        self._L = LinkedList()

    def enqueue(self, item):
        self._L.addlast(item)

    def dequeue(self):
        return self._L.removefirst()
```

Storing the length

With our `ListQueue`, we implemented `__len__` to give the number of items in the queue. To implement the same method on the `LinkedQueue`, we will want **delegate** the length computation to the `LinkedList`.

Let's add the ability to get the length of the linked list. We'll do it by storing the length and updating it with each operation.

```
class LinkedList:
    def __init__(self):
        self._head = None
        self._tail = None
        self._length = 0

    def addfirst(self, item):
        self._head = ListNode(item, self._head)
        if self._tail is None: self._tail = self._head
        self._length += 1

    def addlast(self, item):
        if self._head is None:
            self.addfirst(item)
        else:
            self._tail.link = ListNode(item)
            self._tail = self._tail.link
            self._length += 1

    def removefirst(self):
        item = self._head.data
        self._head = self._head.link
        if self._head is None: self._tail = None
        self._length -= 1
        return item

    def removelast(self):
        if self._head is self._tail:
            return self.removefirst()
        else:
            currentnode = self._head
            while currentnode.link is not self._tail:
                currentnode = currentnode.link
            item = self._tail.data
            self._tail = currentnode
            self._tail.link = None
            self._length -= 1
            return item

    def __len__(self):
        return self._length
```

We still have to iterate through the whole list in order to remove from the end. It seems very hard to avoid this. As a result, our `removelast` method still takes linear time. We won't fix this until the next chapter and it will require a new idea. So, instead of testing the deque ADT methods now, we'll instead see how to implement an efficient queue using a linked list.

Testing Against the ADT

Recall that the Queue ADT specifies the expected behavior of a Queue data structure. It describes the **public interface** to the class. Now that we can use our `LinkedList` class to implement a Queue with worst-case constant time operations, we have multiple distinct implementations of the ADT. Good tests for these data structures would only assume that they implement what is provided in the ADT. In fact, we probably want to test both implementations with the same tests.

If we rename our first Queue implementations `ListQueue`, we might have had the following tests for it.

```
import unittest
from listqueue import ListQueue

class TestListQueue(unittest.TestCase):
    def testinit(self):
        q = ListQueue()

    def testaddandremoveoneitem(self):
        q = ListQueue()
        q.enqueue(3)
        self.assertEqual(q.dequeue(), 3)

    def testalternatingaddremove(self):
        q = ListQueue()
        for i in range(1000):
            q.enqueue(i)
            self.assertEqual(q.dequeue(), i)

    def testmanyoperations(self):
        q = ListQueue()
        for i in range(1000):
            q.enqueue(2 * i + 3)
        for i in range(1000):
            self.assertEqual(q.dequeue(), 2 * i + 3)

    def testlength(self):
        q = ListQueue()
        self.assertEqual(len(q), 0)
        for i in range(10):
            q.enqueue(i)
        self.assertEqual(len(q), 10)
        for i in range(10):
            q.enqueue(i)
        self.assertEqual(len(q), 20)
        for i in range(15):
            q.dequeue()
        self.assertEqual(len(q), 5)

if __name__ == '__main__':
    unittest.main()
```

Now that we have another implementation, we might be tempted to just copy and paste the old

tests, changing the references from `ListQueue` to `LinkedQueue`. That is a lot of code duplication and code duplication leads to problems. For example, suppose we realize that our code has issues if we try to `dequeue` from an empty queue. If we decide on the right behavior, we will enforce it with a test. Do we also copy the test to the other (copied) test file? What if one implementation is fixed and the other is not?

This is a standard situation where inheritance is called for. We wanted to copy a bunch of methods to be included in two different classes (`TestListQueue` and `TestLinkedQueue`). Instead we want them to *share* the methods. So, we **refactor** the code, by **extracting a superclass**. Our new class will be called `TestQueue`. Both `TestListQueue` and `TestLinkedQueue` will extend `TestQueue. Remember extending means inheriting from.

```
# testqueue.py
class TestQueue:
    def newQueue():
        raise NotImplementedError

    def testinit(self):
        q = self.newQueue()

    def testaddandremoveoneitem(self):
        q = self.newQueue()
        q.enqueue(3)
        self.assertEqual(q.dequeue(), 3)

    def testalternatingaddremove(self):
        q = self.newQueue()
        for i in range(1000):
            q.enqueue(i)
            self.assertEqual(q.dequeue(), i)

    def testmanyoperations(self):
        q = self.newQueue()
        for i in range(1000):
            q.enqueue(2 * i + 3)
        for i in range(1000):
            self.assertEqual(q.dequeue(), 2 * i + 3)

    def testlength(self):
        q = self.newQueue()
        self.assertEqual(len(q), 0)
        for i in range(10):
            q.enqueue(i)
        self.assertEqual(len(q), 10)
        for i in range(10):
            q.enqueue(i)
        self.assertEqual(len(q), 20)
        for i in range(15):
            q.dequeue()
        self.assertEqual(len(q), 5)
```

The new class looks almost exactly the same as our old `TestListQueue` class exactly for a couple small changes. Instead of creating a new `ListQueue` object in each test, we call a method called `newQueue`. In this class, that method is *not implemented* and you can see it will raise an error telling

you so if you were to try to call it. This method will be overwritten in the extending class to provide the right kind of Queue object. There is another *important difference* between the `TestQueue` class and our old `TestListQueue` class---the `TestQueue` class does not extend `unittest.TestCase`. It just defines some methods that will be **mixed into** `TestListQueue` and `TestLinkedQueue`. Here are our new test files.

```
# testlistqueue.py
import unittest
from testqueue import TestQueue
from listqueue import ListQueue

class TestListQueue(unittest.TestCase, TestQueue):
    def newQueue(self):
        return ListQueue()

if __name__ == '__main__':
    unittest.main()
```

```
# testlinkedqueue.py
import unittest
from testqueue import TestQueue
from linkedqueue import LinkedQueue

class TestListQueue(unittest.TestCase, TestQueue):
    def newQueue(self):
        return LinkedQueue()

if __name__ == '__main__':
    unittest.main()
```

The classes, `TestListQueue` and `TestLinkedQueue`, extend both `unittest.TestCase` *and* `TestQueue`. This is called **multiple inheritance**. In other languages like C++ that support multiple inheritance, it is considered a bad design decision. However, in python, it is appropriate to use it for this kind of **mix in**. The only thing to remember is that the golden rule of inheritance should still be observed: **inheritance mean 'is a'**.

The Main Lessons:

- Use the public interface as described in an ADT to **test** your class.

- You can use **inheritance** to share functionality between classes.
- Inheritance means **'is a'**.

Design Patterns: The Wrapper Pattern

In the last two chapters, we saw several different implementations of the Queue ADT. The main ones, `LinkedQueue` and `ListQueue` were very simple. In both cases, we used **composition**, the class stored an object of another class, and then **delegates** most of the operations to the other class. These are examples of something called the **Wrapper Pattern**. The Queue in both cases is a wrapper around another data structure.

Design patterns also known as **object-oriented design patterns** or simply **patterns** are ways of organizing classes in to solve common programming problems. In the case of the Wrapper Pattern, we have a class that already "sort of" does what we want, but it has different names for the operations and it possibly has many other operations that we don't want to support. So, we create a new class that **has an** instance of another class (a `list` or `LinkedList` in our example) and then provide methods that operate on that object. From outside the class, we don't have to know anything about the wrapped class. Sometimes, this separation is called a **layer of abstraction**. The user of our class does not have to know anything about our implementation in order to use the class.

The Main Lessons:

- Use design patterns where appropriate to organize your code and improve readability.
- The Wrapper Pattern gives a way to provide an alternative interface to (a subset of) the methods in another class.
- Composition means **'has a'**.

Chapter 7

Doubly-Linked Lists

Previously, we introduced the Deque ADT and we gave two implementations, one using a list and one using a linked list. In our linked list implementation, all of the basic operations ran in constant time except `removelast`. In this chapter, we're going to introduce a new data structure that allows us to do all the Deque operations in constant time. The key idea will be to store two links in each node, one forwards and one backwards so that we can traverse the list in either direction. In this **doubly-linked list**, removing from the end will be symmetric to removing from the beginning, with the roles of head and tail reversed.

When we create a new `ListNode`, we can specify the nodes before and after so that the appropriate links can be established. We want it to always be true that `b == a.link` if and only if `a = b.prev` for any two nodes `a` and `b`. To help ensure this invariant, we set `self.prev.link = self` and `self.link.prev = self` unless `prev` or `link` respectively are `None`.

```
class ListNode:
    def __init__(self, data, prev = None, link = None):
        self.data = data
        self.prev = prev
        self.link = link
        if prev is not None:
            self.prev.link = self
        if link is not None:
            self.link.prev = self
```

First, we'll look at adding items a `DoublyLinkedList`. These operations are very similar to the `addfirst` operation on a `LinkedList`. One has to do a little more work to update the `prev` node that

was not present in our `LinkedList`.

```
class DoublyLinkedList:
    def __init__(self):
        self._head = None
        self._tail = None
        self._length = 0

    def addfirst(self, item):
        if len(self) == 0:
            self._head = self._tail = ListNode(item, None, None)
        else:
            newnode = ListNode(item, None, self._head)
            self._head.prev = newnode
            self._head = newnode
        self._length += 1

    def addlast(self, item):
        if len(self) == 0:
            self._head = self._tail = ListNode(item, None, None)
        else:
            newnode = ListNode(item, self._tail, None)
            self._tail.link = newnode
            self._tail = newnode
        self._length += 1

    def __len__(self):
        return self._length
```

The code above is begging for refactoring. It's clear that there is a lot of shared logic between the two methods. We should use this as an opportunity to simplify the code. In this case, we might consider the more general problem of adding a node between two other nodes. We will just need to consider those cases where the nodes `before` or `after` or both are `None`.

```
    def _addbetween(self, item, before, after):
        node = ListNode(item, before, after)
        if after is self._head:
            self._head = node
        if before is self._tail:
            self._tail = node
        self._length += 1

    def addfirst(self, item):
        self._addbetween(item, None, self._head)

    def addlast(self, item):
        self._addbetween(item, self._tail, None)
```

Symmetry is also apparent in the code to remove an item from either end. As with the `add` methods, we factor out a (private) method that both use to remove a node and return its data. It includes a little logic to detect if the head or tail or both change with the removal.

```
    def _remove(self, node):
        before, after = node.prev, node.link
        if node is self._head:
            self._head = after
        else:
            before.link = after
        if node is self._tail:
            self._tail = before
        else:
            after.prev = before
        self._length -= 1
        return node.data

    def removefirst(self):
        return self._remove(self._head)

    def removelast(self):
        return self._remove(self._tail)
```

Concatenating Doubly Linked Lists

There are several operations that are very fast on doubly linked lists compared to other lists. One of the most useful is the ability to concatenate two lists. Recall that the plus sign can be used to concatenate two `list` objects.

```
A = [1,2,3]
B = [4,5,6]
C = A + B
print(A)
print(B)
print(C)
```

```
[1, 2, 3]
[4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

For lists, concatenation creates a new list. It takes time proportional to the length of the newly created list `C` and it doesn't modify the lists `A` or `B`. For doubly linked lists, we could achieve the same asymptotic running time by incrementally building up a new list. However, if we are allowed to modify the lists, the concatenation can be accomplished by pointing the tail of the first list at the head of the second.

```
    def __iadd__(self, other):
        if other._head is None: return
        if self._head is None:
            self._head = other._head
        else:
            self._tail.link = other._head
            other._head.prev = self._tail
        self._tail = other._tail
        self._length = self._length + other._length

        # Clean up the other list.
        other.__init__()
        return self

L = DoublyLinkedList()
[L.addlast(i) for i in range(11)]
B = DoublyLinkedList()
[B.addlast(i+11) for i in range(10)]

L += B

n = L._head
while n is not None:
    print(n.data)
    n = n.link
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

We have to be a bit careful about the differences between this concatenation operation and concatenation of regular lists. With the doubly-linked list, the concatenation operation empties the other list. It does this so that we don't have multiple doubly-linked lists with the same `ListNode`s. That would be a problem if we tried to edit just one of the lists, because the changes would be reflected in the other list as well, possibly with disastrous consequences.

Chapter 8

Recursion

Recursion is an important idea across computer science. For us, in learning data structures and basic algorithms, there are several ways to both think about and use recursion. At its simplest, one can think of **recursion** as: *when a function calls itself*.

Although true, this definition doesn't really tell us how or why to use it, and what to do with it when we find it in the wild. Our objectives for this chapter are three-fold:

1. Understand how recursion is implemented in a computer and translate this into a model for how to think about recursive functions.
2. Use recursion as a problem solving technique, recognizing the role of "subproblems" as a primary motivation for recursion.
3. Be able to analyze the running time of recursive functions.

Here is an example of a recursive function.

```
def f(k):
    if k > 0:
        return f(k-1) + k
    return 0

print(f(5))
```

```
15
```

This is actually just our familiar `sumk` function in disguise. How does it work? To find the sum of

numbers from 1 to k, it just finds the sum of the numbers from 1 to $k-1$ and then adds k. Yes, we know there is a better way to compute this sum, but there is something satisfying about this approach. We were asked to solve a problem given some input k and we pretended we already had a solution for smaller numbers ($k-1$ in this case). Then, we used that solution to construct a solution for k. The magic comes from the fact that the "pretended" solution was actually the same function. I call this the *pretend someone else wrote it* approach to recursion. It's very handy for both writing and analyzing recursive functions.

Recursion and Induction

The function `f` above does the same thing as our `sumk` function that we saw earlier. You can check it using something called induction. In fact, this checking process is exactly what it means to "do a proof by induction". There, you prove a fact, by checking that it is true in the base case. Then, you prove that it's true for `k` assuming it's true for `k-1`. In the special case of checking that our function `f` is identical to the `sumk` function that simply returns $\frac{k(k+1)}{2}$, it looks as follows. First, check that indeed

$$f(0) = 0 = \frac{0 * (0+1)}{2}.$$

Then observe that

$$f(k) = f(k-1) + k = \frac{(k-1)(k-1+1)}{2} + k = \frac{(k-1)k + 2k}{2} = \frac{k(k+1)}{2}.$$

We won't get many opportunities to do such proofs in this class, but it's worth remembering that there is a strong connection between recursion and induction and working through this connection can enrich your understanding of both concepts.

Some Basics

Here are some basic rules that you should try to follow to make sure your recursive algorithms terminate.

1. Have a base case. If every function call makes another recursive call, you will never terminate.
2. Recursive calls should move towards the base case. This usually means that the recursive calls are made on "smaller" subproblems. Here, "smaller" can mean different things.

It's not hard to write a recursive function that never stops *in theory*. However, in practice, so-called infinite recursion is pretty quickly discovered with a `RecursionError`. Python has a limit on the recursion depth. This limit is usually around 1000.

The Function Call Stack

To think clearly about recursion, it helps to have an idea of how recursion works on a real computer. It is easiest to first understand how all function calls work and then it should be clear that there is no technical difference in making a recursive function call compared to any other function call.

```
def f(k):
    var = k ** 2
    return g(k+1) + var

def g(k):
    var = k + 1
    return var + 1

print(f(3))
```

```
15
```

The following code gives a `RecursionError` even though none of the functions call themselves directly. The error is really just signaling that the call stack reached its limit.

```
def a(k):
    if k == 0: return 0
    return b(k)

def b(k):
    return c(k)

def c(k):
    return a(k-1)

a(340)
```

An interesting recursive example con be constructed by creating two lists, each one containing the

other.

```
A = [2]
B = [2]
A.append(A)
B.append(B)
A == B
```

```
Traceback (most recent call last):
  File "/Users/don/Dropbox/work/research/books/datastructures/docs/a25memlwa_code_chunk.py",
    A == B
RecursionError: maximum recursion depth exceeded in comparison
```

In this case, the recursive function is `list.__eq__`, the method that compares two lists when you use `==`. It compares the list by checking if the elements are equal. The lists `A` and `B` each have length 2. The first elements match. The second element of each list is another list. To compare them, there is another call to `list.__eq__`. This is a repeat of the first call and the process repeats until the recursion limit is reached.

The Fibonacci Sequence

Here is a classic example of a recursively defined function. It was originally named for Leonardo Fibonacci who studied it in the context of predicting the growth of rabbit populations.

```
def fib(k):
    if k in [0,1]: return k
    return fib(k-1) + fib(k-2)

print([fib(i) for i in range(15)])
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

This works, but it starts to get really slow, even for small values of `k`. For example, I tried to run it for `k = 40` and gave up on it.

This is a case that we will encounter many times in this course. Once we embrace recursion as a way to think about breaking problems into smaller pieces, we may find very short recursive

algorithms, but for one reason or another, we may want to rewrite them without recursion. Here's a version that uses a loop instead.

```
def fib(k):
    a, b = 0,1
    for i in range(k):
        a, b = b, a + b
    return a

print(fib(400))
```

```
176023680645013966468226945392411250770384383304492191886725992896575345044216019675
```

This is much better! At first look, it might seem like the only difference in the total work is just in the extra overhead of making function calls rather than updating local variables. This is wrong. Let's work out in more detail exactly what function calls are made in the recursive implementation. Let's say we called `fib(6)`. This makes calls to `fib(5)` and `fib(4)`. This, in turn makes calls to `fib(4)` and `fib(3)` as well as `fib(3)` and `fib(2)`. Each of these four function calls will make two more function calls each. We can draw them out in a tree. Notice, that already, we have multiple calls to `fib(4)` as well as to `fib(3)`. We might as well ask, how many times will we call the same function with the same value? If we compute `fib(k)`, the answer, interestingly enough is related to the Fibonacci numbers themselves. Let $T(k)$ denote the number of calls to `fib` when computing `fib(k)`. It's easy to see that $T(k) = T(k-1) + T(k-2) + 1$. In this case, the result is nearly exactly the Fibonacci numbers ($T(1) = 1, T(2) = 2, T(3) = 4, T(4) = 7$, ...). In each case, the value of T is one less than a Fibonacci number. Thus, the running time of `fib` will grow like the Fibonacci numbers, or equivalently, it will grow like ideal rabbit families, exponentially. The kth Fibonacci number is about ϕ^k, where $\phi = \frac{1+\sqrt{5}}{2} \sim 1.618$ is known as the Golden Ratio.

Euclid's Algorithm

Euclid's algorithm is a classic in every sense of the word. The input is a pair of integers `a, b` and the output is the greatest common divisor, i.e., the largest integer that divides both `a` and `b` evenly. It is a very simple recursive algorithm. We'll look at the code first and then try to figure out what its doing, why it works, and how we can improve it.

```
def gcd(a, b):
    if a == b:
        return a
    if a > b:
        a, b = b, a
    return gcd(a, b - a)
```

Like all recursive algorithms, there is a base case. Here, if `a == b`, then `a` (or equivalently, `b`) is the answer and we return it. Otherwise, we arrange it so `a < b` and make a recursive call: `gcd(a, b - a)`.

When we walk through some examples, by hand, it seems like the algorithm makes many recursive calls when `b` is much bigger than `a`. In fact, it's not hard to see that we'll need at least `a // b` recursive calls before we swap `a` and `b`. These calls are just repeatedly subtracting the same number until it gets sufficiently small. This is known to elementary school students as division. This is actually a deep idea that's worth pondering. Division is iterated subtraction the same way that multiplication is iterated addition, exponentiation is iterated multiplication, and logarithms are iterated division. Theoretical computer scientists even have a use for iterated logarithms (we call it $\log^*$, pronounced "log star").

We can just do the division directly rather than repeatedly subtracting the smaller from the bigger. The result is a slight change to the base case and replacing the subtraction with the modulus operation. Here is the revised code.

```
def gcd(a, b):
    if a > b:
        a, b = b, a
    if a == 0:
        return b
    return gcd(a, b % a)

print("GCD of 12 and 513 is", gcd(12, 513))
print("GCD of 19 and 513 is", gcd(19, 513))
print("GCD of 19 and 515 is", gcd(515 ,19))
```

```
GCD of 12 and 513 is 3
GCD of 19 and 513 is 19
GCD of 19 and 515 is 1
```

Incidentally, if `a` and `b` are allowed to be arbitrary numbers, you might try find examples where the gcd algorithm gets caught in an infinite recursion. In this way, you might discover the irrational numbers. If `a` and `b` are rational, then the algorithm is also guaranteed to terminate.

If you wanted to find an example that was as bad as possible, you might try to find a pair `(a,b)` such that after one iteration, you get the pair `(b-a, a)` where the ratio of the numbers is the same. Then, you can check that this will continue and therefore, you'll never get closer to that base case. But is that possible? Is there a pair of numbers with this property? The answer is yes. One could use $a = 1$ and $b = \phi$, the Golden Ratio.

Chapter 9

Dynamic Programming

The term **dynamic programming** refers to an approach to writing algorithms in which a problem is solved using solutions to the same problem on smaller instances. Recall that this was the same intuition behind recursive algorithms. Indeed, there are many instances in which you might arrive at a dynamic programming algorithm by starting with a recursive algorithm that is inefficient because it repeatedly makes recursive calls to a function with the exact same parameters, wasting time.

We start with the problem of giving change using the fewest coins. Imagine that you have a list of coin amounts [1, 5, 10, 25] and an amount of change that you want to produce. A first attempt at such a solution might be to assign the change in a **greedy** way. In that approach, you would simply try to add the largest coin you can until you are done. Here is an implementation. *(Notice the relationship to Euclid's GCD algorithm)*

A Greedy Algorithm

```
def greedyMC(coinvalueList, change):
    coinvalueList.sort()
    coinvalueList.reverse()
    numcoins = 0
    for c in coinvalueList:
        # Add in as many coins as possible of the next largest value.
        numcoins += change // c
        # Update the amount of change left to return.
        change = change % c
    return numcoins

print(greedyMC([1,5,10,25], 63))
print(greedyMC([1, 21, 25], 63))
print(greedyMC([1, 5, 21, 25], 63))
```

```
6
15
7
```

Clearly, this does not work. The second example returns 15 and the third returns 7. In both cases, the right answer should have been 3 as can be achieved by returning three 21 cent coins. The problem is not a bug in the code, but an incorrect algorithm. The greedy solution does not work here. Let's try recursion instead.

A Recursive Algorithm

```
def recMC(coinValueList, change):
   minCoins = change
   if change in coinValueList:
     return 1
   else:
      for i in [c for c in coinValueList if c <= change]:
         numCoins = 1 + recMC(coinValueList,change-i)
         if numCoins < minCoins:
            minCoins = numCoins
   return minCoins

# print(recMC([1,5,10,25],63)) # Seriously don't even try to run this
print(recMC([1, 21, 25],63))
print(recMC([1, 5, 21, 25],63))
```

```
3
3
```

This works, but it's very slow. A natural thing to try here is to **memoize** solutions from previous recursive calls. This way, we don't repeat work by computing the number of coins for a specific amount of change more than one time. Don't forget to pass along the dictionary of known results when making recursive calls.

A Memoized Version

```
def memoMC(coinValueList, change, knownResults):
    minCoins = change
    if change in coinValueList:
        knownResults[change] = 1
        return 1
    elif change in knownResults:
        return knownResults[change]
    else:
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + memoMC(coinValueList, change-i, knownResults)
            if numCoins < minCoins:
                minCoins = numCoins
                knownResults[change] = minCoins
    return minCoins

print(memoMC([1,5,10,25],63, {}))

knownresults = {}
print(memoMC([1, 5, 10, 21, 25], 63, knownresults))
print(knownresults)
```

```
6
3
{1: 1, 5: 1, 6: 2, 7: 3, 8: 4, 9: 5, 10: 1, 11: 2, 12: 3, 13: 4, 14: 5, 15: 2, 16: 3, 17: 4,
```

This is much faster and now can work with much larger instances. It was slightly awkward that we had to pass in an empty dictionary to get things started, but that's not a big deal. Let's look at the dictionary `knownresults` that gets populated by the algorithm. It contains all but a few of the values from 1 to 63. Maybe instead of recursive calls, starting with the full amount of change and working our way down, we could build up the dictionary of values starting with the small values and working our way up. This is the essence of dynamic programming.

It happens that a list makes sense here because we want to access the items by their integer indices starting from 1. In each step, the next best answer can be found by trying each coin, subtracting its value from the current value, and checking the list to find how many coins are needed to make up the rest of the change if that coin is used.

A Dynamic Programming Algorithm

Finally, we can repackage this into a dynamic programming algorithm. The idea is to explicitly fill in the table of all values, using previously computed values to compute new values.

```
def dpMakeChange(coinValueList, change):
    # Create a list to store the answers to the subproblems
    minCoins = [None]*(change + 1)

    # For each value from 0 to change, compute the min number of coins needed.
    for cents in range(change+1):
        # Assume at first that all 1's are used.
        minCoins[cents] = cents
        # Check if any coin leads to a better solution to our current best.
        for c in coinValueList:
            if cents >= c:
                minCoins[cents] = min(minCoins[cents], minCoins[cents - c] + 1)

    # Return just the element in the table corresponding to the desired value.
    return minCoins[change]

print(dpMakeChange([1,5,10,21,25], 63))
print(dpMakeChange([1,5,10,21,25], 64))
```

```
3
4
```

A major difference between the dynamic programming approach and the memoized recursion approach is that the dynamic program builds the results *from the bottom up*, while the recursive version works top down, starting work on the largest problem first and working down to the smaller problems as needed. *Do either of these approaches correspond to the way that you solve programming problems?*

Another example

In order to see dynamic programming as a general approach to solving problems, we need to see another example. As with recursion, the key is to look for smaller problems to solve. This is really a theme throughout this course. All problems must be broken into smaller problems. If those smaller problems are instances of the same problem we started with then maybe recursion or dynamic

programming are appropriate.

The next problem we'll solve with dynamic programming is called the longest common subsequence (LCS) problem. A **subsequence** of a string `s` is string `t` such that the characters of `t` all appear in `s` in the same order. For example `'abc'` is a subsequence of `'xxxxxaxxxbxxxcxxxx'`

The input to the LCS problem is a pair of strings, we'll call them `X` and `Y`. The output is the longest string that is a subsequence of both `X` and `Y`.

One view of this problem is that we want to find the smallest set of characters to remove from `X` and `Y` so that the results will be equal. However, we don't want to try all possible subsequences. There are 2^n subsequences of a string of length n. That's too many! Our code would never finish for even $n = 100$.

Here is the trick that cracks open this problem. If `X` and `Y` end in the same character, then that character is the last character in the longest common substring. That is, if `X[-1] == Y[-1]` then the `LCS(X,Y)` is `LCS(X[:-1], Y[:-1]) + X[-1]`. On the other hand, if `X[-1] != Y[-1]`, then at least one of `X[-1]` or `Y[-1]` is *not* in the LCS. In that case, `LCS(X,Y)` is the longer of `LCS(X[:-1],Y)` and `LCS(X,Y[:-1])`. We could turn this into a very tidy recursive algorithm.

```
def reclcs(X,Y):
    if X == "" or Y == "":
        return ""
    if X[-1] == Y[-1]:
        return reclcs(X[:-1], Y[:-1]) + X[-1]
    else:
        return max([reclcs(X[:-1], Y), reclcs(X, Y[:-1])], key = len)
```

However, when we run this on moderately size inputs, it seems to run forever. It's not hard to see that if we had two long strings that didn't match any characters, then the tree of recursive calls would be a complete binary tree of depth n. That's 2^n recursive calls. However, there aren't that many distinct recursive calls. Each such call is of the form `reclcs(X[:i], Y[:j])`, i.e. it takes the first `i` characters of `X` and the first `j` characters of `Y` for some `i` and `j`. This means that there should only be $O(n^2)$ *distinct* recursive calls. As n^2 is much smaller than 2^n, many recursive calls are being repeated. That's wasted work that we can avoid. We could add memoization as before, but really, what we want is dynamic programming. We'll store solutions to subproblems in a dictionary `t` so that `t[(i,j)] == LCS(X[:i], Y[:j])`. We initialize the table with `""` for subproblems with `i = 0` or `j = 0` (the base case). Here is the code.

```
def lcs(X, Y):
    t = {}
    for i in range(len(X)+1): t[(i,0)] = ""
    for j in range(len(Y)+1): t[(0,j)] = ""

    for i, x in enumerate(X):
        for j, y in enumerate(Y):
            if x == y:
                t[(i+1,j+1)] = t[(i, j)] + x
            else:
                t[(i+1,j+1)] = max([t[(i, j+1)], t[(i+1, j)]], key = len)
    return t[(len(X), len(Y))]
```

The initialization takes linear time and the main pair of loops iterate $O(n^2)$ times. The inner loop takes time proportional to the LCS in the worst case, because we will concatenate strings of that length. The total running time is $O(kn^2)$ where k is the length of the output.

Can you think of a way to get the running time down to $O(n^2)$? This would require a different way to store solutions to subproblems.

Chapter 10

Binary Search

Binary search is a classic algorithm. It is particularly nice to think about as a recursive algorithm. If you are looking for an item in a sorted list, you break the list in half and repeat the search on whichever side could contain the missing element, which can be found by comparing with the median element. Then, repeating on the smaller list is just a single recursive call.

```
def bs(L, item):
    if len(L) == 0: return False
    median = len(L) // 2
    if item == L[median]:
        return True
    elif item < L[median]:
        return bs(L[:median], item)
    else:
        return bs(L[median + 1:], item)
```

This code, although correct, is not nearly as efficient as it could be. To analyze it, we use the same technique as we will use for all recursive algorithms in this course. We will count up *all* the operations that will be performed by the function except the recursive calls. Then we will draw the tree of all recursive calls and add up the cost of each call.

In this case, the worst-case running time of `bs` on a list of length n is $n/2$ plus a constant. So, the first call costs $n/2$. The second costs $n/4$. The third costs $n/8$. Adding them up gives a number close to n. So, we are taking linear time to test membership. We would be faster to just iterate one time through the list by using the `in` function (i.e. `list.__contains__`).

If we're going to do faster, we have to avoid all that slicing. Let's only pretend to slice the list and

instead pass the indices defining the range of the list we want to search. Here is a first attempt.

```
def bs(L, item, left = 0, right = None):
    if right is None: right = len(L)
    if right - left == 0: return False
    if right - left == 1: return L[left] == item
    median = (right + left) // 2
    if item < L[median]:
        return bs(L, item, left, median)
    else:
        return bs(L, item, median, right)
```

Note that we had to do a little work with default parameters so that we can still call this function as `bs(mylist, myitem)`. This involves a check to see if `right is None` and sets it to be the length of the list if necessary.

When we analyze this recursive algorithm just as before, we see that all the operations take constant time, so the total running time will be proportional to the total number of recursive calls. The tree of function calls is a single chain of length at most $O(\log n)$. *(Why $\log(n)$? This is the number of times you can cut n in half before it gets down to 1.)* So, we see that the asymptotic running time is $O(\log n)$.

In the analysis, we observed that the tree of function calls is a single chain. This is called **linear recursion**. Here, we have a special case in which the function directly returns the result of the recursive function call. That is called **tail recursion**.

In general, tail recursion can always be replaced by a loop. The idea is to simply update the parameter variables and loop rather than making a recursive call. Here is the idea in action.

```
def bs(L, item):
    left, right = 0, len(L)
    while right - left > 1:
        median = (right + left) // 2
        if item < L[median]:
            right = median
        else:
            left = median
    return right > left and L[left] == item
```

Note that this solution is simpler to write than our original, and is also probably slightly faster. It does

require a little more thought to see why it's correct and why it runs in $O(\log n)$ time.

The Sorted List ADT

- `add(item)` - adds `item` to the sorted list.
- `remove(item)` - removes the first occurrence of `item` from the sorted list. Raise a `ValueError` if the `item` is not present.
- `__getitem__(index)` - returns the item with the given `index` in the sorted list. This is also known as **selection**.
- `__contains__(item)` - returns true if there is an item of the sorted list equal to `item`.
- `__iter__` - returns an iterator over the sorted list that yields the items in sorted order.
- `__len__` - returns the length of the sorted list.

Here is a very simple implementation of the sorted list ADT.

```
class SimpleSortedList:
    def __init__(self):
        self._L = []

    def add(self, item):
        self._L.append(item)
        self._L.sort()

    def remove(self, item):
        self._L.remove(item)

    def __getitem__(self, index):
        return self._L[index]

    def __contains__(self, item):
        return item in self._L

    def __len__(self):
        return len(self._L)

    def __iter__(self):
        return iter(self._L)
```

This is a classic example of the Wrapper pattern. The `list` storing the items is kept private so we can enforce that it stays ordered. It looks a bit like a cheat to implement `add` in this way, but we'll

see later as we cover sorting algorithms that this might be an efficient approach after all.

The one algorithm in this mix that seems most relevant to improve is the `__contains__` method. Even though, we are just calling out to python's built-in method for checking membership in a list, we have some hoping of improving the efficiency because we know that the list is sorted.

Let's replace it with binary search as we implemented it above.

```
class BSSortedList(SimpleSortedList):
    def __contains__(self, item):
        left, right = 0, len(self._L)
        while right - left > 1:
            median = (right + left) // 2
            if item < self._L[median]:
                right = median
            else:
                left = median
        return right > left and self._L[left] == item
```

We might also try to use binary search to find the index at which we want to insert a new node in order to speed up `add`. Unfortunately, after finding the index, we still need to spend linear time in the worst-case to insert an item into a list at a particular index.

Chapter 11

Sorting

The Quadratic-Time Sorting Algorithms

Before we dive into particular sorting algorithms, let's first ask an easier question:

> **Given a list `L`, determine if `L` is sorted or not.**

After a little, thought, you will probably come up with some code that looks like the following.

```
def is_sorted(L):
    for i in range(len(L)-1):
        if L[i]>L[i+1]:
            return False
    return True

print(is_sorted([1,2,3,4,5]))
print(is_sorted([1,4,5,7,2]))
```

```
True
False
```

This is much better than the following, more explicit code.

```
def is_sorted_slow(L):
    for i in range(len(L)-1):
        for j in range(i+1, len(L))
            if L[j] < L[i]:
              return False
    return True
```

The latter code checks that every pair of elements is in the right order. The former code is more clever in that it only checks adjacent elements. If the list is not sorted, then there will be two adjacent elements that will be out of order. This is true because ordering relations are **transitive**, that is, if `a < b` and `b < c`, then `a < c`. It's important to realize that we are using this assumption, because later, we will define our own ways of ordering items, and we will need to make sure this is true so that sorting even makes sense (if `a < b < c < a`, then what is the correct sorted order?).

Let's use the `is_sorted` method to write a sorting algorithm. Instead of returning False when we find two elements out of order, we'll just fix them and move on. Let's call this (for reasons to become clear) `dumberSort`.

```
def dumberSort(L):
    for i in range(len(L)-1):
        if L[i]>L[i+1]:
            L[i], L[i+1] = L[i+1], L[i]
```

The main problem with this code is that *it doesn't sort the list*.

```
L = [5,4,3,2,1]
dumberSort(L)
print(L)
```

```
[4, 3, 2, 1, 5]
```

We should probably do it twice *or more*. We could even repeat the algorithm until it works.

```
def dumbSort(L):
    while (not isSorted(L)):
        dumberSort(L)
```

```
L = [5,4,3,2,1]
dumbSort(L)
print(L)
```

```
[1, 2, 3, 4, 5]
```

Will it ever get the list sorted?

How many times will we `dumberSort` the list?

After some thinking, you will notice that if `n = len(L)`, then looping `n-1` times is enough. There are several different ways to see that this is correct. You may see that every element in the list moves at least one place closer to its final location with each iteration.

Another way to see that this works is to check that after calling `dumberSort(L)` one time, the largest element will move all the way to the end of the list and stay there through all subsequent calls. Calling `dumberSort(L)` a second time will cause the second largest element to *bubble* up to the second to last to last place in the list. We are uncovering what is called an **invariant**, something that is true every time we reach a certain point in the algorithm. For this algorithm, the invariant is that after `i` calls to `dumberSort(L)`, the last `i` items are in their final (sorted) locations in the list. The nice thing about this invariant is that if it holds for `i = n`, then the whole list is sorted, and we can conclude the algorithm is correct.

At this point, we would test the code and think about refactoring. Remember the DRY (**D** on't **R** epeat **Y** ourself) principle. Because we know how many times to loop, we could just use a `for` loop.

```
def bubbleSort(L):
    for iteration in range(len(L)-1):
        for i in range(len(L)-1):
            if L[i]>L[i+1]:
                L[i], L[i+1] = L[i+1], L[i]

alist = [0, 100000,54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

```
[0, 17, 20, 26, 31, 44, 54, 55, 77, 93, 100000]
```

At this point, we have a correct algorithm and it's quite easy to bound its running time. It's $O(n^2)$, a quadratic time algorithm.

We lost something compared to `dumbSort`, namely, we no longer stop early if the list is already sorted. Let's bring that back. We'll use a flag to check if any swaps were made.

```
def bubbleSort(L):
    keepgoing = True
    while keepgoing:
        keepgoing = False
        for i in range(len(L)-1):
            if L[i]>L[i+1]:
                L[i], L[i+1] = L[i+1], L[i]
                keepgoing = True
```

Now, that we know an invariant that leads to a correct sorting algorithm, maybe we could work backwards from the invariant to an algorithm. Recall, the invariant was that after `i` iterations of a loop, the `i` largest elements are in their final positions. We can write an algorithm that just makes sure to achieve this, by selecting the largest among the first n-i elements and moving that element into place.

```
def selectionSort(L):
    n = len(L)
    for i in range(n-1):
        max_index=0
        for index in range(n - i):
            if L[index] > L[max_index]:
                max_index = index
        L[n-i-1], L[max_index] = L[max_index], L[n-i-1]
```

There is another invariant that we have implicitly considered previously when working with sorted lists. Recall there, we assumed that we had a sorted list and one new element was added that had to moved into the correct position. We saw several different variations of this. We can turn this invariant into yet another sorting algorithm. To keep it as close to possible as the previous attempts, we will say that the invariant is that after `i` iterations, the last `i` elements in the list are in sorted order.

Do you see the difference between this and our previous invariant?

There are many ways we could enforce this invariant. We'll do it by "bubbling" element `n-i` into

position in the `i` th step. Note this is not the final position, but rather the position that satisfies the invariant.

```
def insertionSort(L):
    n = len(L)
    for i in range(n):
        for j in range(n-i-1, n-1):
            if L[j]>L[j+1]:
                L[j], L[j+1] = L[j+1], L[j]
```

As before, we can make this algorithm go faster if the list is already sorted (or almost already sorted). We stop the inner loop as soon as the element is in the right place

```
def insertionSort(L):
    n = len(L)
    for i in range(n):
        j = n - i - 1
        while j < n - 1 and L[j]>L[j+1]:
            L[j], L[j+1] = L[j+1], L[j]
            j+=1
```

There is an important point to remember in the code above. If `j == n-1` , then evaluating `L[j] < L[j+1]` would cause an error. However, this code does not cause an error, because the expression `j < n - 1 and L[j]>L[j+1]` evaluates the first part first. As soon as `j < n-1` evaluates to False, it doesn't need to evaluate the other clause of the `and` . It skips it.

You often see `insertionSort` written in a way that keeps the sorted part of the list in the front rather than the end of the list.

Could you write such an insertion sort? Try it.

Sorting in Python

Python has two main functions to sort a list. They are called `sort()` and `sorted()` . The difference is that the former sorts the list and the latter returns a new list that is sorted. Consider the following simple demonstration.

```
X = [3,1,5]
Y = sorted(X)
print(X, Y)

X.sort()
print(X)
```

```
[3, 1, 5] [1, 3, 5]
[1, 3, 5]
```

When working with your own classes, you may want to sort elements. To do so, you only need to be able to compare elements.

In the following example, the elements are sorted by decreasing value of `b`. Then, the list is sorted again, but the key function is supplied give a different comparator. Notice that the parameter is the function itself and not the evaluation of the function.

```python
from random import randrange

class Foo:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def __lt__(self, other):
        return other.b < self.b

    def __str__(self):
        return "(%d, %d, %d)" % (self.a, self. b, self. c)

    def geta(self):
        return self.a

L = [Foo(randrange(100),randrange(100), randrange(100)) for i in range(6)]
L.sort()
for foo in L:
    print(foo)

print("--------")

for foo in sorted(L, key = Foo.geta):
    print(foo)
```

```
(28, 89, 84)
(65, 51, 96)
(55, 44, 60)
(89, 36, 25)
(57, 35, 84)
(2, 5, 93)
--------
(2, 5, 93)
(28, 89, 84)
(55, 44, 60)
(57, 35, 84)
(65, 51, 96)
(89, 36, 25)
```

If the key function returns a tuple, it will sort by the first element and break ties with subsequent elements. This kind of sorting is called lexicographic because it is how you would sort words in alphabetical order.

Here is an example of sorting strings by their length (longest to shortest) using the alphabetical order (ignoring case) to break ties.

```
strings = "here are Some sample strings to be sorted".split()

def mykey(x):
    return -len(x), x.upper()

print(sorted(strings, key=mykey))
```

```
['strings', 'sample', 'sorted', 'here', 'Some', 'are', 'be', 'to']
```

Chapter 12

Sorting with Divide and Conquer

Previously, we saw some simple quadratic algorithms for sorting. We saw `bubbleSort`, `selectionSort`, and `insertionSort`. We considered all these algorithms from the perspective of correctness. We wanted to have ways to argue that the result was indeed a sorted list, and only afterwards applied some optimizations (such as stopping early). We used the idea of an **invariant**. Specifically, it was a **loop invariant**, something that is true on each iteration of the loop. We started with the invariant that after `i` iterations, the last `i` elements are in sorted order.

Invariants help us to reason about our code the same way we do with recursive algorithms. Once the code is written, we assume the algorithm works on small examples in the same way we assume a loop invariant holds from a previous iteration of the loop.

Now, our goal is to write a faster sorting algorithm.

Divide and Conquer is a paradigm for algorithm design. It usually consists of 3 (plus one) parts. The first part is to **divide** the problem into 2 or more pieces. The second part is the **conquer** step, where one solves the problem on the pieces. The third part is the **combine** step where one combines the solutions on the parts into a solution on the whole.

The description of these parts leads pretty directly to recursive algorithms, i.e. using recursion for the conquer part. The other part that appears in many such algorithms is a **base case**, as one might expect in a recursive algorithm. This is where you deal with inputs so small that they cannot be divided.

Mergesort

The most direct application of the Divide and Conquer paradigm to the sorting problem is the **mergesort** algorithm. In this algorithm, all the difficult work is in the merge step.

```
def mergeSort(L):
    # Base Case!
    if len(L) < 2:
        return

    # Divide!
    mid = len(L) // 2
    A = L[:mid]
    B = L[mid=]

    # Conquer!
    mergeSort(A)
    mergeSort(B)

    # Combine!
    merge(A, B, L)

def merge(A, B, L):
    i = 0 # index into A
    j = 0 # index into B
    while i < len(A) and j < len(B):
        if A[i] < B[j]:
            L[i+j] = A[i]
            i = i + 1
        else:
            L[i+j] = B[j]
            j = j + 1
      # Add any remaining elements once one list is empty
      L[i+j:] = A[i:] + B[j:]
```

That last line might look a little strange. The right side of the assignment is concatenating the remaining elements from the two lists (of which one should be empty). Then, this list is assigned into a slice. In its wonderfulness, python allows you to assign into a slice the same way you would assign into an index.

We could also use some more logic in the loop to avoid this last step, though I have found students disagree as to which approach is simpler.

```
def merge(A, B, L):
    i, j = 0, 0
    while i < len(A) or j < len(B):
        if j == len(B) or (i < len(A) and A[i] < B[j]):
            L[i+j] = A[i]
            i = i + 1
        else:
            L[i+j] = B[j]
            j = j + 1
```

The complex `if` statement above relies heavily on something called **short-circuited** evaluation of boolean expressions. If we have a boolean operation like `or`, and the first operand is `True`, then we don't have to evaluate the second operand to find out that the overall result will be `True`. Using this fact, python will not even evaluate the second operand. Similarly, if we have an `and` expression and the first operand is `False`, then the second operand is never evaluated. The key to remember is that the order does matter here. The expression `(i < len(A) and A[i] < B[j]) or j == len(B)` is logically equivalent, but if we use this expression instead, it will raise an IndexError when `j == len(B)`.

Short-circuit evaluation is *not* part of python magic, it's a standard feature in most programming languages.

An Analysis

The analysis of `mergesort` will proceed in the usual way for recursive algorithms. We will count the basic operations in the methods, not counting recursive calls. Then, we will draw the tree of recursive calls and write down the cost of each function call in the tree. Because the input size will change from call to call, the cost will also change. Then, we add them all up to get the total running time.

The `merge` function for two lists whole length add up to n takes $O(n)$ time. This is because we only need to do a comparison and some assignment for each item that gets added to the final list (of which there are n).

In the tree of recursive calls, the top (or root) costs $O(n)$. The next level has two calls on lists of length $n/2$. The second level down has four calls of lists of length $n/4$. On down the tree, each level i has 2^i calls, each on lists of length $n/2^i$. A nice trick to add these costs up, is to observe that the costs of all nodes on any given level sum to $O(n)$. There are about $\log_2 n$ levels. (How many times

can you divide n by 2 until you get down to 1?)

So, we have $\log n$ levels, each costing $O(n)$, and thus, the total cost is $O(n \log n)$.

Merging Iterators

The merge operation is another example of using some structure on our data (that two lists are themselves sorted) to perform some operation quickly (find the minimum overall element in constant time). However, after you've written enough python, you might start to feel like you are doing something wrong if you are messing around with a lot of indices. When possible, it is much nicer to use iterators. We'll use this problem as an example to motivate some deeper study into iterators.

Recall that an object is an **Iterable** in python if it has a method called `__iter__` that returns an iterator, and an **Iterator** is a object that has an `__iter__` method and a `__next__` method. These magic methods are called *from the outside* as `iter(my_iterable)` and `next(my_iterator)`. They are most commonly used by the `for` keyword either in for loops or in **generator expressions** as in comprehensions.

```
class SimpleIterator:
    def __init__(self):
        self._count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._count < 10:
            self._count += 1
            return self._count
        else:
            raise StopIteration

iterator1 = SimpleIterator()
for x in iterator1:
    print(x)

iterator2 = SimpleIterator()
L = [2 * x for x in iterator2]
```

```
1
2
3
4
5
6
7
8
9
10
```

Iterators are a fundamental pattern in object-oriented programming and they appear in other programming languages with slightly different methods. For example, some programming languages have iterators that support a `hasnext` method that can tell in advance whether there is another item to iterate over. In a standard python iterator, you simply try to get the next item and it will raise `StopIteration` if there isn't one. This is an example of python's *easier to ask forgiveness than permission (EAFP)* approach. Checking in advance is like asking for permission. However, such a method can be handy. With it, one might also want a method called `peek` that will return the next item without advancing past it.

Here is how one might implement these methods in python. The `BufferedIterator` class below is built from any iterator. It stays one step of the iteration ahead of the user and stores the next item in a buffer.

```
class BufferedIterator:
    def __init__(self, i):
        self._i = iter(i)
        self._hasnext = True
        self._buffer = None
        self._advance()

    def peek(self):
        return self._buffer

    def hasnext(self):
        return self._hasnext

    def _advance(self):
        try:
            self._buffer = next(self._i)
        except StopIteration:
            self._buffer = None
            self._hasnext = False

    def __iter__(self):
        return self

    def __next__(self):
        if self.hasnext():
            output = self.peek()
            self._advance()
            return output
        else:
            raise StopIteration
```

We can use this `BufferedIterator` to implement another iterator that takes two iterators and merges them as in the `merge` operation of `mergesort`.

```
def merge(A, B):
    a = BufferedIterator(A)
    b = BufferedIterator(B)
    while a.hasnext() or b.hasnext():
        if not a.hasnext() or (b.hasnext() and b.peek() < a.peek()):
            yield next(b)
        else:
            yield next(a)
```

This iterator looks very different from our previous one. First of all, it's not a class, but appears to just be a method. This is a slick python way of defining a simple iterator. The "method" here has `yield` statements instead of `return` statements. Python recognizes this and creates a **generator** object. Calling `merge` will return an object of type `generator`. A `generator` is an iterator. The easiest way to think of it is as a method that pauses every time it reaches `yield` and resumes when asked for the next item. The magic here is explained by understanding that this really is packaged into an iterator object that is returned.

We can use this new `merge` iterator to write a new version of `mergesort`.

```
def mergesort(L):
    if len(L) > 1:
        m = len(L) // 2
        A, B = L[:m], L[m:]
        mergesort(A)
        mergesort(B)
        L[:] = merge(A, B)
```

Quicksort

The `mergesort` code does a lot of slicing. Recall that this creates a copy. Eventually, we will try to get to a version that doesn't do this. A sorting algorithm, that just rearranges the elements in a single list is called an **in-place** sorting algorithm. Before we get there, let's see the easiest version of `quicksort`. One way to think about the motivation for `quicksort` is that we want to do divide and conquer, but we want the combine step to be as easy as possible. Recall that in `mergesort`, most of our cleverness was devoted to doing the combining. In `quicksort`, the harder step is the dividing.

```
def quickSorted(L):
    #base case
    if len(L) < 2:
        return L[:]

    # Divide!
    pivot = L[-1]
    LT = [e for e in L if e < pivot]
    ET = [e for e in L if e == pivot]
    GT = [e for e in L if e > pivot]

    # Conquer
    A = quickSorted(LT)
    B = quickSorted(GT)

    # Combine
    return A + ET + B
```

Let's do an in-place version. For this, we want to avoid creating new lists and concatenating them at the end.

```
def quicksort(L, left = 0, right = None):
    if right is None:
        right = len(L)

    if right - left > 1:
        # Divide!
        mid = partition(L, left, right)

        # Conquer!
        quicksort(L, left, mid)
        quicksort(L, mid+1, right)

        # Combine!
        # Nothing to do!

def partition(L, left, right):
    pivot = right - 1
    i = left        # index in left half
    j = pivot - 1   # index in right half

    while i < j:
```

```
        # Move i to point to an element >= L[pivot]
        while L[i] < L[pivot]:
            i = i + 1

        # Move j to point to an element < L[pivot]
        while i < j and L[j] >= L[pivot]:
            j = j - 1

        # Swap elements i and j if i < j
        if i < j:
            L[i], L[j] = L[j], L[i]

    # Put the pivot in place.
    if L[pivot] <= L[i]:
        L[pivot], L[i] = L[i], L[pivot]
        pivot = i

    # Return the index of the pivot.
    return pivot

# Simple test to see if it works.
L = [5,2,3,1,4]
quicksort(L)
print(L)
```

```
[1, 2, 3, 4, 5]
```

Here is a version without all the comments. It uses a helper function rather than using default parameters to handle the initial call. It's helpful to look at at two different implementations of the same function and compare the different choices that were made between the two.

```
def quicksort(L):
    _quicksort(L, 0, len(L))

def _quicksort(L, left, right):
    if right - left > 1:
        mid = partition(L, left, right)
        _quicksort(L, left, mid)
        _quicksort(L, mid+1, right)

def partition(L, left, right):
    i, j, pivot = left, right - 2, right - 1
    while i < j:
        while L[i] < L[pivot]:
            i += 1
        while i < j and L[j] >= L[pivot]:
            j -= 1
        if i < j:
            L[i], L[j] = L[j], L[i]
    L[pivot], L[i] = L[i], L[pivot]
    return i
```

Chapter 13

Selection

Consider the following problem.

> Given a list of numbers, find the median element.

Recall that the median element is the one that would be in the middle if we sorted the list. So, the following could would be correct.

```
def median(L):
    L.sort()
    return L[len(L) // 2]
```

The code does rearrange the list and we'd have to decide if that was okay for our given list. If we use a fast sorting algorithm, we can expect this algorithm to run in $O(n \log n)$ time.

Can we do better? How about a linear time algorithm?

Let's consider a related problem, first. If our goal is to find the second largest item in a list, we could do this in linear time as follows.

```
def secondsmallest(L):
    a, b = None, None
    for item in L:
        if a is None or item <= b:
            a, b = item, a
        elif b is None or item <= a:
            b = item
    return b
```

We just stored a variable for each of the two largest we've seen so far and update them as we look at new items. You could probably make this work for the third largest or more, but it would start to get cumbersome and slow, especially if you wanted to extract the median.

Some other approach will be necessary to compute the median in linear time. Let's try divide and conquer. To make this work, we're going to take an approach that often appears when dealing with recursion:

> **To solve a problem with recursion, sometimes it's easier to solve a harder problem.**

So, rather than solving **the median problem** directly, we will solve **the selection problem** defined as follows.

> **The Selection Problem:** Given a list of numbers and a number k, find the kth smallest number in the list.

The `quickselect` algorithm

We can use a divide and conquer approach derived from `quicksort` to do selection. The idea is to partition the list and then recursive do the selection on one part or the other. Unlike `quicksort`, we only have to do a recursive search on one half of the list (as in binary search). (Note that the smallest element will be returned for $k = 1$, *not* $k = 0$, so there are several cases where we add one to the pivot index.)

```
def quickselect(L, k):
    return _quickselect(L, k, 0, len(L))

def _quickselect(L, k, left, right):
    pivot = partition(L, left, right)
    if k <= pivot:
        return _quickselect(L, k, left, pivot)
    elif k == pivot + 1:
        return L[pivot]
    else:
        return _quickselect(L, k, pivot + 1, right)

def partition(L, left, right):
    pivot = randrange(left, right)
    L[pivot], L[right -1] = L[right -1], L[pivot]
    i, j, pivot = left, right - 2, right - 1
    while i < j:
        while L[i] < L[pivot]:
            i += 1
        while i < j and L[j] >= L[pivot]:
            j -= 1
        if i < j:
            L[i], L[j] = L[j], L[i]
    L[pivot], L[i] = L[i], L[pivot]
    return i
```

Just as with `quicksort`, we use a randomized pivot so that we can expect to eliminate a constant fraction of the list with each new step. Unlike `quicksort`, we will not make a recursive call on both sides. As a result, we'll see that the average running time is only $O(n)$. That means we can do selection faster than we can sort, which makes sense, but it is *not obvious* how to do it. We'll make the analysis formal below.

Analysis

The `quickselect` algorithm is a **randomized algorithm**.
The worst-case running time is bad. If we are trying to select the largest element and the pivots all land on the smallest elements, then the running time will be quadratic.

Instead, we'll analyze the **expected running time**. The word **expected** comes from probability and refers to the *average* over all random choices the algorithm makes.

Given a list of n numbers, the partition function will pick a random element to serve as the *pivot*. We say a pivot is *good* if it lands in the range of indices from $n/4$ to $3n/4$. So, choosing randomly, there is a 50% chance of picking a good pivot.

With each recursive call, we get a smaller list. Let's say that n_i is the size of the list on the ith recursive call, so $n = n_0 > n_1 > \cdots > n_k$, where k is the (unknown) number of recursive calls. There is a $1/2$ probability of a good pivot at any step, so the expected value of n_i can be bounded as follows.

$$E[n_i] \leq (1/2)\left(\frac{3E[n_{i-1}]}{4}\right) + (1/2)(E[n_{i-1}]) = \left(\frac{7}{8}\right)E[n_{i-1}].$$

This bound combines two bounds, each that holds at least half the time, first the good pivot case where $n_i \leq 3n_{i-1}/4$ and second, the bad pivot case, where at least $n_i \leq n_{i-1}$. The actual expectation will be smaller, but this upper bound suffices.

Repeating this fact by plugging it into itself (recursively) for smaller values gives the next bound in terms of n.

$$E[n_i] \leq \left(\frac{7}{8}\right)n_{i-1} \leq \left(\frac{7}{8}\right)^2 n_{i-2} \leq \left(\frac{7}{8}\right)^3 n_{i-3} \leq \cdots \leq \left(\frac{7}{8}\right)^i n.$$

Each recursive call takes linear time, so the total running time will be

$$T(n) = \sum_{i=0}^{k} O(n_i) = O(\sum_{i=0}^{k} n_i).$$

So, we need to bound the sum of the n_is, but we only need to bound the expected (average) sum. The most important fact in all of probability is the **linearity of expectations**. It says that the sum of two expected values is the expected value of the sum. We use it here to finish our analysis.

$$E[T(n)] = O\left(\sum_{i=0}^{k} E[n_i]\right) \leq O\left(\sum_{i=0}^{k}\left(\frac{7}{8}\right)^i n\right) = O(n).$$

The last step follows by the well-known equation for geometric series. You should have seen it in a math class by now, but don't hesitate to look it up if you forgot it.

One last time without recursion

The `quickselect` algorithm is an example of linear recursion. Each function call makes just one more function call. It's also **tail recursive** because that single recursive call is the last operation prior to returning. So, as we've seen several times before, it's possible to eliminate the recursion and replace it with a single loop. The code is below, but you might want to try to write this yourself to see if you understand both the algorithm and the process of tail recursion elimination.

```
def quickselect(L, k):
    left, right = 0, len(L)
    while left < right:
        pivot = partition(L, left, right)
        if k <= pivot:
            right = pivot
        elif k == pivot + 1:
            return L[pivot]
        else:
            left = pivot + 1
      return L[left]
```

As long as the difference between left and right shrinks by a constant factor every couple of pivots, we can be sure to *prune* enough of the search to get a linear time algorithm.

Divide-and-Conquer Recap

There are three main classes of divide-and-conquer algorithms and analyses that we've seen for lists.

The first,and perhaps simplest, were those like binary search, where each recursive step takes constant time and makes a single recursive call on a list that is a constant times smaller. The total running time in those cases is proportional to the depth of the recursion, $O(\log n)$.

The second class of recursive, divide-and-conquer algorithms we saw were binary recursion associated with sorting. In those, the running time is linear plus the time to make recursive calls on shorter lists whose total length is $O(n)$. In those cases, as long as the depth of the recursion is $O(\log n)$, the total running time is $O(n \log n)$.

Now, we have a third class of divide-and-conquer algorithms, those like `quickselect`. Here, like in the sorting algorithms, the running time is linear plus the cost of recursive calls. However, like in binary search, there is only one recursive call. In these cases, as long as the subproblems shrink by a constant factor at each step, the running time will be $O(n)$.

A Note on Derandomization

An algorithm that does not use randomness is called **deterministic**. So far, all the algorithms we have covered are deterministic so we didn't need to make the distinction. In the case of the selection problem, it's possible to solve it in linear time in the worst-case, without randomness.

If you recall the `quickselect` algorithm depends on getting an approximate median in order to reduce the length of the list by a constant fraction. So, if you had an algorithm for finding the median in linear time, you could use it to choose the pivots in `quickselect` in order to get a linear-time algorithm.

That should sound a little wrong. We started this whole section with the idea that we can solve the median problem using an algorithm for selection. Now, we want to solve the selection problem using an algorithm for median. Will we be stuck with a chicken and egg situation?

The algorithm is sometimes called the *median of medians* algorithm. You can find more info about it online.

Chapter 14

Mappings and Hash Tables

A **mapping** is an association between two sets of things. The standard built-in data type in python for mappings is the dictionary (`dict`). This kind of mapping is used by python itself to associate names of variables (strings) with objects. We usually refer to these associated pairs as **key-value pairs**.

Not all programming languages come with a built-in data type for mappings. We're going to pretend for a short time that we don't have a python dictionary available to us and go through the process of implementing one ourselves. This will allow us to resolve one of the major unsolved mysteries from earlier in the course:

> Why does accessing or assigning a value in a dictionary take only constant time?

The Mapping ADT

A **mapping** is a collection of key-value pairs such that the keys are unique (i.e. no two pairs share a key). It supports the following methods.

- **`get(k)`** - return the value associate to the key `k` . Usually an error (`KeyError`) is raised if the given key is not present.
- **`put(k, v)`** - Add the key-value pair `(k,v)` to the mapping.

These are the two main operations. They are what make a mapping, and are generally implemented as `__getitem__` and `__setitem__` in python in order to support the familiar square bracket notation. We will put off anything more elaborate for now.

A minimal implementation

Here is a very lightweight method for using a `list` as a mapping. We start with a little class to store key-value pairs, then give two methods to implement get and put.

```
class Entry:
    def __init__(self, key, value):
        self.key = key
        self.value = value

    def __str__(self):
        return str(self.key) + " : " + str(self.value)

def mapput(L, key, value):
    for e in L:
        if e.key == key:
            e.value = value
            return
    L.append(Entry(key, value))

def mapget(L, key):
    for e in L:
        if e.key == key:
            return e.value
    raise KeyError
```

```
m = []
mapput(m, 4, 'five')
mapput(m, 1, 'one')
mapput(m, 13, 'thirteen')
mapput(m, 4, 'four')
assert(mapget(m, 1) == 'one')
assert(mapget(m, 4) == 'four')
```

At this point, it seems that the only advantage of the `dict` structure is that it provides some useful syntax for adding and getting entries. There are some other advantages, but we'll only reveal them by trying to build a map ourselves.

But first, let's put our new data structure in a class. This will allow us to encapsulate the underlying list so that users don't accidentally mess it up, for example, by appending to it rather than using `put`

. We'd like to protect users from themselves, especially when there are properties of the structure we want to maintain. In this case, we want to make sure keys stay unique.

```
class ListMappingSimple:
    def __init__(self):
        self._entries = []

    def put(self, key, value):
        for e in self._entries:
            if e.key == key:
                e.value = value
                return
        self._entries.append(Entry(key, value))

    def get(self, key, default = "NotARealDefaultValue"):
        for e in self._entries:
            if e.key == key:
                return e.value
        if default == "NotARealDefaultValue":
            raise KeyError
        else:
            return default
```

This is an okay start. It support `get` and `put` and that's maybe enough to be a mapping, but we'd like several more interesting methods to really put such a structure to use. Let's extend the ADT with more features.

The extended Mapping ADT

As with any collection, we might want some other methods such as `__len__`, `__contains__`, or various iterators.

The standard behavior for iterators in dictionaries is to iterate over the keys. Alternative iterators are provided to iterate over the values or to iterate over the key-value pairs as tuples. For a `dict` object this is done as follows.

```
d = {'key1': 'value1', 'key2': 'value2'}

for k in d:
    print(k)

for v in d.values():
    print(v)

for k, v in d.items():
    print(k, v)
```

```
key2
key1
value2
value1
key2 value2
key1 value1
```

We'll add the same kind of functionality to our Mapping ADT. So, the **extended Mapping ADT** includes the following methods (with get and put renamed for python magic).

- `__getitem__(k)` ** - return the value associate to the key `k`. Usually an error (`KeyError`) is raised if the given key is not present.
- `__setitem__(k, v)` - Add the key-value pair `(k,v)` to the mapping.
- `__len__` - return the number of keys in the dictionary.
- `__contains__(k)` - return true if the mapping contains a pair with key `k`.
- `__iter__` - return an iterator over the keys in the dictionary.
- `values` - return an iterator over the values in the dictionary.
- `items` - return an iterator over the key-value pairs (as tuples).

It is very important to recall from the very beginning of the course that the `dict` class is a **non-sequential collection**. That is, there is no significance to the ordering of the items and furthermore, you should never assume to know anything about the ordering of the pairs. You should not even assume that the ordering will be consistent between two iterations of the same `dict`. This same warning goes for the mappings we will implement and we'll see that the ability to rearrange the order of how they are stored is the secret behind the mysteriously fast running times. However, this first implementation will have the items in a fixed order because we are using a `list` to store them.

```
class ListMapping:
    def __init__(self):
        self._entries = []

    def put(self, key, value):
        e = self._entry(key)
        if e is not None:
            e.value = value
        else:
            self._entries.append(Entry(key, value))

    def get(self, key):
        e = self._entry(key)
        if e is not None:
            return e.value
        else:
            raise KeyError

    def _entry(self, key):
        for e in self._entries:
            if e.key == key:
                return e
        return None

    def __str__(self):
        return str([str(e) for e in self._entries])

    def __getitem__(self, key):
        return self.get(key)

    def __setitem__(self, key, value):
        self.put(key, value)

    def __len__(self):
        return len(self._entries)

    def __contains__(self, key):
        if self._entry(key) is None:
            return False
        else:
            return True

    def __iter__(self):
```

```
        return (e.key for e in self._entries)

    def values(self):
        return (e.value for e in self._entries)

    def items(self):
        return ((e.key, e.value) for e in self._entries)
```

Note that I took the opportunity to factor out some duplication in the `get` and `put` methods.

It's Too Slow!

Our goal is to to get the same kind of constant-time operations as in the `dict` class. Right now, we are very far from that. Currently, we need linear time to get put, and check membership. To do better, we're going to need a new idea. The `ListMapping` takes linear time because it has to iterate through the list. We could make this faster if we had many short lists instead of one large list. Then, we just need to have a quick way of knowing which short list to search or update.

We're going to store a list of `ListMappings`. For any key `k`, we want to compute the index of the *right* `ListMapping` for `k`. We often call these `ListMapping`s *buckets*. This term goes back to the idea that you can quickly group items into buckets.

This means, we want an integer. A **hash function** takes a key and computes an integer. Most classes in python implement a method called `__hash__` that does just this. We can use it to implement a simple mapping scheme that improves on the `ListMapping`.

```
class HashMappingSimple:
    def __init__(self):
        self._size = 100
        self._buckets = [ListMapping() for i in range(self._size)]

    def __setitem__(self, key, value):
        m = self._bucket(key)
        m[key] = value

    def __getitem__(self, key):
        m = self._bucket(key)
        return m[key]

    def _bucket(self, key):
        return self._buckets[hash(key) % self._size]
```

Let's look more closely at this code. It seems quite simple, but it hides some mysteries.

First, the initializer creates a list of 100 ListMaps. These are called the buckets. If the keys get spread evenly between the buckets then this will be about 100 times faster! If two keys are placed in the same bucket, this is called a **collision**.

The `__getitem__` and `__setitem__` methods call the `_bucket` method to get one of these buckets for the given key and then just use that ListMap's get and set methods. So, the idea is just to have several list maps instead of one and then you just need a quick way to decide which to use. The `hash` function, returns an integer based on the value of the given key. The collisions will depend on the hash function.

The number 100 is pretty arbitrary. If there are many many entries, then one might get 100-fold speedup over ListMap, but not much more. It makes sense to use more buckets as the size increases. To do this, we will keep track of the number of entries in the map. This will allow us to implement `__len__` and also grow the number of buckets as needed. Here is the code.

```
class HashMap:
    def __init__(self, size = 100):
        self._size = size
        self._buckets = [ListMapping() for i in range(self._size)]
        self._length = 0

    def __setitem__(self, key, value):
        m = self._bucket(key)
        if key not in m:
            self._length += 1
        m[key] = value

        # Check if we need more buckets.
        if self._length > self._size:
            self._double()

    def __getitem__(self, key):
        m = self._bucket(key)
        return m[key]

    def _bucket(self, key):
        return self._buckets[hash(key) % self._size]

    def _double(self):
        # Save the old buckets.
        oldbuckets = self._buckets
        # Double the size.
        self._size *= 2
        # Create new buckets
        self._buckets = [ListMapping() for i in range(self._size)]
        # Add in all the old entries.
        for bucket in oldbuckets:
            for key, value in bucket.items():
                # Identify the new bucket.
                m = self._bucket(key)
                m[key] = value

    def __len__(self):
        return self._length
```

Rehashing

The most interesting part of the code above is the `_double` method. This is a method that increases the number of buckets. It's not enough to just append more buckets to the list, because the `_bucket` method that we use to find the right bucket depends on the number of buckets. When that number changes, we have to reinsert all the items in the mapping so that they can be found when we next `get` them.

Factoring Out A Superclass

We have given two different implementations of the same ADT. There are several methods that we implemented in the `ListMapping` that we will also want in the `HashMapping`. It makes sense to avoid duplicating common parts of these two (concrete) data structures. Inheritance provides a nice way to do this.

This is the most common way that inheritance appears in code. Two classes want to share some code, so we **factor out a superclass** that both can inherit from and share the underlying code.

There are some methods that we expect to be implemented by the subclass. We can enforce this by putting the methods in the subclass, but raising an error if they are called. This way, the error will only be raised if the subclass does not override those methods.

Here is the code for the superclass.

```
class Mapping:
    class Entry:
        def __init__(self, key, value):
            self.key = key
            self.value = value

        def __str__(self):
            return "%d: %s" % (self.key, self.value)

    # Child class needs to implement this!
    def get(self, key):
        raise NotImplemented

    # Child class needs to implement this!
    def put(self, key, value):
        raise NotImplemented

    # Child class needs to implement this!
```

```
    def __len__(self):
        raise NotImplemented

    # Child class needs to implement this!
    def _entryiter(self):
        raise NotImplemented

    def __iter__(self):
      return (e.key for e in self._entryiter())

    def values(self):
        return (e.value for e in self._entryiter())

    def items(self):
        return ((e.key, e.value) for e in self._entryiter())

    def __contains__(self, key):
        try:
            self.get(key)
        except KeyError:
            return False
        return True

    def __getitem__(self, key):
        return self.get(key)

    def __setitem__(self, key, value):
        self.put(key, value)

    def __str__(self):
        return "{%s}" % (", ".join([str(e) for e in self]))
```

There is a lot here, but notice that there are really only four methods that a subclass has to implement: `get`, `put`, `__len__`, and a method called `_entryiter` that iterates through the entries. This last method is private because the user of this class does not need to access `Entry` objects. They have the Mapping ADT methods to provide access to the data. This is why the `Entry` class is an inner class (defined inside the `Mapping` class).

Now, the `ListMapping` can be rewritten as follows.

```
class ListMapping(Mapping):
    def __init__(self):
        self._entries = []

    def put(self, key, value):
        e = self._entry(key)
        if e is not None:
            e.value = value
        else:
            self._entries.append(Entry(key, value))

    def get(self, key):
        e = self._entry(key)
        if e is not None:
            return e.value
        else:
            raise KeyError

    def _entry(self, key):
        for e in self._entries:
            if e.key == key:
                return e
        return None

    def _entryiter(self):
        return iter(self._entries)

    def __len__(self):
        return len(self._entries)
```

All the magic methods as well as the public iterators and string conversion are handled by the superclass. The subclass only has the parts that are specific to this implementation.

The `HashMapping` class can also be rewritten this was as follows.

```
class HashMapping(Mapping):
    def __init__(self, size = 100):
        self._size = size
        self._buckets = [ListMapping() for i in range(self._size)]
        self._length = 0

    def _entryiter(self):
        return (e for b in self._buckets for e in b._entryiter())

    def get(self, key):
        b = self._bucket(key)
        return b[key]

    def put(self, key, value):
        b = self._bucket(key)
        if key not in b:
            self._length += 1
        b[key] = value

        # Check if we need more buckets.
        if self._length > self._size:
            self._double()

    def __len__(self):
        return self._length

    def _bucket(self, key):
        return self._buckets[hash(key) % self._size]

    def _double(self):
        # Save the old buckets
        oldbuckets = self._buckets
        # Reinitialize with more buckets.
        self.__init__(self._size * 2)
        for bucket in oldbuckets:
            for key, value in bucket.items():
                self[key] = value
```

Chapter 15

Trees

Trees data types are ideal for representing hierarchical structure. Trees are composed of **nodes** and nodes have 0 or more **children** or **child nodes**. A node is called the **parent** of its children. Each node has (at most) one parent. If the children are ordered in some way, then we have an **ordered tree**. We are concerned primarily with **rooted trees**, i.e. there is a single special node called the **root** of the tree. The root is the only node that does not have a parent. The nodes that do not have any children are called **leaves** or **leaf nodes**.

There are many many examples of hierarchical (tree-like) structures:

- Class Hierarchies (assuming single inheritance). The subclass is the child and the superclasses is the parent. The python class `object` is the root.
- File folders or directories on a computer can be nested inside other directories. The parent-child relationship encode containment.
- The tree of recursive function calls in the execution of a divide-and-conquer algorithm. The leaves are those calls where the base case executes.

Even though we use the family metaphor for many of the naming conventions, a family tree is not actually a tree. The reason is that these violate the one parent rule.

When we draw trees, we take an Australian convention of putting the root on the top and the children below. Although this is backwards with respect to the trees we see in nature, it does correspond to how we generally think of most other hierarchies.

Some more definitions

A **path** in a tree is a sequence of nodes in which each node is a child of the previous node. We say it is a path from x to y if the first node is x and the last node is y. There exists a path from the root to every node in the tree. The **length of the path** is the number of hops or edges which is one less than the number of nodes in the path. The **descendants** of a node x are all those nodes y for which there is a path from x to y. If y is a descendant of x, then we say x is an **ancestor** of y. Given a tree T and a node n in that tree, the **subtree rooted at** n is the tree whose root is n that contains all descendants of n.

The **depth** of a node is the length of the path to the node from the root. The **height** of a tree is the maximum depth of any node in the tree.

A recursive view of trees

A tree can be defined recursively as a root with 0 or more children, each of which is tree. This will be convenient at first and will be important later on as we think about doing more elaborate algorithms on trees, many of which are easiest to describe recursively. Be warned however, that our object-oriented instincts will later kick in as we separate the notions of trees and nodes. Having two different kinds of things will lead us to use two different classes to represent them.

We can use lists to represent a hierarchical structure by making lists of lists. To make this a little more useful, we will also store some data in each node of the tree. We'll use the convention that a list is a tree in which the first item is the data and the remaining items are the children (each should be a list).

Here is a simple example.

```
['a', ['p'], ['n'], ['t']]
```

This is a tree with `'a'` stored in the root. The root has 3 children storing respectively, `'p'`, `'n'`, and `'t'`. Here is a slightly bigger example that contains the previous example as a subtree.

```
T = ['c', ['a', ['p'], ['n'], ['t']], ['o', ['n']]]
```

We can print all the nodes in such a tree with the following code.

```
def printtree(T):
    print(T[0])
    for child in range(1, len(T)):
        printtree(T[child])

printtree(T)
```

Whenever I see code like that above, I want to replace it with something that can work with any iterable collection. This would involve getting rid of all the indices and using an iterator. Here's the equivalent code that uses an iterator.

```
def printtree(T):
    iterator = iter(T)
    print(next(iterator))
    for child in iterator:
        printtree(child)
```

Here we use the iterator to extract the first list item, then loop through the rest of the children.

The information in the tree is all present in this *list of lists* structure. However, it can be cumbersome to read, write, and work with. Let's package this into a class that allows us to write code that is as close as possible to how we think about and talk about trees.

```
class Tree:
    def __init__(self, L):
        iterator = iter(L)
        self.data = next(iterator)
        self.children = [Tree(c) for c in iterator]
```

The initializer takes a *list of lists* representation of a tree as input. A `Tree` object has two attributes, `data` stores data associated with a node and `children` stores a lists a child `Tree` objects. The recursive aspect of this tree is clear from the way the children are generated as `Tree`'s.

Let's add our print function to the class.

```
def printtree(self):
    print(self.data)
    for child in self.children:
        child.printtree()
```

This is the most common pattern for algorithms that operated on trees. It has two parts; one part operates on the data and the other part applies the function recursively on the children. Here is another example of the same pattern. Let's check if two trees are equal in the sense of having the same shape and data. We use the `__eq__` method so this method will be used when we use `==` to check equality between `Tree`'s.

```
def __eq__(self, other):
    return self.data == other.data and self.children == other.children
```

Here, it is less obvious that we are doing recursion, but we are because `self.children` and `other.children` are lists and list equality is determined by testing the equality of the items. In our case, the items in the `.children` lists are `Tree`'s, so our `__eq__` method will be invoked for each one.

Here's another example. Let's write a function that computes the height of the tree. We can do this by computing the height of the subtrees and return one more than the max of those. If there are no children, the height is 0.

```
def height(self):
    if len(self.children) == 0:
        return 0
    else:
        return 1 + max(child.height() for child in self.children)
```

Tree Traversal

Previously, all the collections we stored were either sequential (i.e., `list`, `tuple`, and `str`) or non-sequential (i.e., `dict` and `set`). The tree structure seems to lie somewhere between the two. There is some structure, but it's not linear. We can give it a linear (sequential) structure by iterating through all the nodes in the tree, but there is not a unique way to do this. For trees, the process of visiting all the nodes is called **tree traversal**. For ordered trees, there are two standard traversals, called **preorder** and **postorder**, both are naturally defined recursively.

In a preorder traversal, we *visit* the node first followed by the traversal of its children. In a postorder traversal, we traverse all the children and then visit the node itself. The *visit* refers to whatever computation we want to do with the nodes. The `printtree` method given previously is a classic example of a preorder traversal. We could also print the nodes in a postorder traversal as follows.

```
def printpostorder(self):
    for child in self.children:
        child.printpostoder()
    print(self.data)
```

It is only a slight change in the code, but it results in a different output.

If you want to get fancy...

It was considered a great achievement in python to be able to do this kind of traversal with a generator. Recall that a generator is an iterator defined using a method that yields instead of returning. Recursive generators seem a little mysterious, especially at first. However, if you break down this code and walk through it by hand, it will help you have a better understanding of how generators work.

```
def preorder(self):
    yield self.data
    for child in self.children:
        for data in child.preorder():
            yield data
```

You can also do this to iterate over the trees (i.e. nodes). I have made this one private because the user of the tree likely does not want or need access to the nodes.

```
def _preorder(self):
    yield self
    for child in self.children:
        for descendant in child._preorder():
            yield descendant
```

There's a catch!

This recursive generator does not run in linear time! Recall that each call to the generator produces an object. The object does not go away after yielding because it may need to yield more values. If one calls this method on a tree, each value yielded is passed all the way from the node to the root. Thus, the total running time is proportional to the sum of the depths of all the nodes in the tree. For a degenerate tree (i.e. a single path), this is $O(n^2)$ time. For a perfectly balanced binary tree, this is

$O(n \log n)$ time.

Chapter 16

Binary Search Trees

Let's start with an ADT. A **SortedMapping** is a mapping data type for which the keys are comparable. It should support all the same operations as any other mapping as well as some other operations similar to those in a `SortedList` such as predecessor search.

The Sorted Mapping ADT

A **sorted mapping** stores a collection of key-value pairs (*with comparable keys*) supporting the following operations.

- `get(k)` - Return the value associate to the key `k`. An error (`KeyError`) is raised if the given key is not present.
- `put(k, v)` - Add the key-value pair `(k,v)` to the mapping.
- `floor(k)` - Return a tuple `(k,v)` corresponding to the key-value pair in the mapping with the largest key that is less than or equal to `k`.
- `remove(k)` - Remove the key-value pair with key `k` from the sorted mapping. An error (`KeyError`) is raised if the given key is not present.

Binary Search Tree Properties and Definitions

A tree is called a **binary tree** if every node has at most two children. We will continue to assume that we are working with ordered trees and so we call the children `left` and `right`. We say that a binary tree is a **binary search tree** if for every node `x`, all the keys in the subtree `x.left` are less than the key at `x` and all the keys in the subtree `x.right` are greater than the key of `x`. This

ordering property, also known as **the BST property** is what makes a binary search tree different from any other kind of binary tree.

The BST property is related to a new kind of tree traversal, that was not possible with other trees. Recall that previously we saw preorder and postorder traversal of trees. These traversals visit all the nodes of the tree. The preorder traversal visits the root of each subtree prior to visiting any nodes in the children. The postorder traversal visits all the nodes in the children prior to visiting the root. The new traversal we introduce here is called **inorder traversal** and it visits all the nodes in the left child prior to visiting the root and then visits all the nodes in the right child after visiting the root. This order results in a traversal of the nodes *in sorted order according to the ordering of the keys*.

A Minimal implementation

We're going to implement a sorted mapping using a binary search tree. As we have already written an abstract class for packaging up a lot of the magic methods we expect from a mapping, we will inherit from that class to get started.

Also, unlike in the previous section, we're going to distinguish between a class for the tree (`BSTMapping`) and a class for the nodes (`BSTNode`). We will maintain a convention that the operations on the `BSTNode` class will operate on and return other `BSTNode` objects when appropriate, whereas the `BSTMapping` class will abstract away the existence of the nodes for the user, only returning keys and values.

Here is just the minimum requirements to be a `Mapping` . It's a top down implementation, so it delegates all the real work to the `BSTNode` class.

```
class BSTMapping(Mapping):
    def __init__(self):
        self._root = None

    def get(self, key):
        if self._root:
            return self._root.get(key).value
        raise KeyError

    def put(self, key, value):
        if self._root:
            self.root = self._root.put(key, value)
        else:
            self._root = BSTNode(key, value)

    def __len__(self):
        return len(self._root) if self._root else 0

    def _entryiter(self):
        if self._root:
            yield from self._root

    def floor(self, key):
        if self._root:
            floornode = self._root.floor(key)
            return floornode.key, floornode.value
        else:
            return None, None

    def remove(self, key):
        if self._root:
            self._root = self._root.remove(key)
        else:
            raise KeyError

    def __delitem__(self, key):
        self.remove(key)
```

The code above gives us almost everything we need. There are a couple of mysterious lines to pay attention to. One is the line in the `put` method that updates the root. We will use this convention extensively. As methods may rearrange the tree structure, such methods return the node that ought to be the new root of the subtree. The same pattern appears in the `remove` function.

One other construct we haven't seen before is the `yield from` operation in the iterator. This takes an iterable and iterates over it, yielding each item. So, `yield from self._root` is the same as `for item in iter(self._root): yield item`. It implies that our `BSTNode` class will have to be iterable.

Let's see how these methods are implemented. We start with the initializer and some handy other methods.

```
class BSTNode:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.left = None
        self.right = None
        self._length = 1

    def __len__(self):
        return self._length
```

The `get` method uses binary search to find the desired key.

```
    def get(self, key):
        if key == self.key:
            return self
        elif key < self.key and self.left:
            return self.left.get(key)
        elif key > self.key and self.right:
            return self.right.get(key)
        else:
            raise KeyError
```

Notice that we are using `self.left` and `self.right` as booleans. This works because `None` evaluates to `False` and `BSTNode`'s always evaluate to `True`. We could have implemented `__bool__` to make this work, but it suffices to implement `__len__`. Objects that have a `__len__` method are `True` if and only if the length is greater than `0`. This is the default way to check if a container is empty. So, for example, it's fine to write `while L: L.pop()` and it will never try to pop from an empty list. In our case, it will allow us to write `if self.left` to check if there is a left child rather than writing `if self.left is not None`.

Next, we implement `put`. It will work by first doing a binary search in the tree. If it finds the key

already in the tree, it overwrites the value (keys in a mapping are unique). Otherwise, when it gets to the bottom of the tree, it adds a new node.

```
    def put(self, key, value):
        if key == self.key:
            self.value = value
        elif key < self.key:
            if self.left:
                self.left.put(key, value)
            else:
                self.left = BSTNode(key, value)
        elif key > self.key:
            if self.right:
                self.right.put(key, value)
            else:
                self.right = BSTNode(key, value)
        self._updatelength()


    def _updatelength(self):
        len_left = len(self.left) if self.left else 0
        len_right = len(self.right) if self.right else 0
        self._length 1 + len_left + len_right
```

The `put` method also keeps track of the length, i.e. the number of entries in each subtree.

The `floor` function

The `floor` function is just a slightly fancier version of `get`. It also does a binary search, but it has different behavior when the key is not found, depending on whether the last search was to the left or to the right. Starting from any node, if we search to the right and the result is `None`, then we return the the node itself. If we search to the left and the result is `None`, we also return `None`.

```
    def floor(self, key):
        if key == self.key:
            return self
        elif key < self.key:
            return self.left.floor(key) if self.left else None
        elif key > self.key:
            return (self.right.floor(key) or self) if self.right else self
```

To parse the expressions above, it's helpful to remember that boolean operations on objects can evaluate to objects. In particular `False or myobject` evaluates to `myobject`. Use the python interactive shell to try out some other examples and to see how `and` behaves.

Iteration

As mentioned above, binary search trees support inorder traversal. The result of an inorder traversal is that the nodes are yielded *in the order of their keys*.

Here is an inorder iterator for a binary search tree implemented using recursive generators. This will be fine in most cases.

```
    def __iter__(self):
        if self.left:
            yield from self.left
        yield self
        if self.right:
            yield from self.right
```

Removal

To implement removal in a binary search tree, we'll use a standard algorithmic problem solving approach. We'll start by thinking about how to handle the easiest possible cases. Then, we'll see how to turn every case into an easy case.

The start of a removal is to find the node to be removed using binary search in the tree. Then, if the node is a leaf, we can remove it without any difficulty. It's also easy to remove a node with only one child because we can remove the node and bring its child up without violating the BST property.

The harder case come when the node to be removed has both a left and a right child. In that case, we find the node with the largest key in its left subtree (i.e. the rightmost node). We swap that node with our node to be removed and call `remove` again on the left subtree. The next time we reach that node, we know it will have at most one child, because the node we swapped it with had no right child (otherwise it wasn't the rightmost before the swap).

A note of caution. Swapping two node in a BST will cause the BST property to be violated. However, the only violation is the node to be removed will have a key greater than the node that we swapped it with. So, the removal will restore the BST property.

Here is the code to do the swapping and a simple recursive method to find the rightmost node in a subtree.

```
def _swapwith(self, other):
    self.key, other.key = other.key, self.key
    self.value, other.value = other.value, self.value

def maxnode(self):
    return self.right.maxnode() if self.right else self
```

Now, we are ready to implement `remove`. As mentioned above, it does a recursive binary search to find the node. When it finds the desired key, it swaps it into place and makes another recursive call. This swapping step will happen only once and the total running time is linear in the height of the tree.

```
def remove(self, key):
    if key == self.key:
        if self.left is None: return self.right
        if self.right is None: return self.left
        self._swapwith(self.left.maxnode())
        self.left = self.left.remove(key)
    elif key < self.key and self.left:
        self.left = self.left.remove(key)
    elif key > self.key and self.right:
        self.right = self.right.remove(key)
    else: raise KeyError
    self.updatelength()
    return self
```

Chapter 17

Balanced Binary Search Trees

In the previous chapter, we saw how to implement a Mapping using a BST. However, in the analysis of the main operations, it was clear that the running time of all the basic operations depended on the height of the tree.

A BST with n nodes can have height $n-1$. It's easy to get such a tree, just insert the keys in sorted order. On the other hand, there always exists a BST with height $O(\log n)$ for any set of n keys. This is achieved by inserting the median first, followed by the medians of each half and so on recursively. So, we have a big gap between the best case and the worst case. Our goal will be to get as close as possible to the best case while keeping the running times of all operations proportional to the height of the tree.

We will say that a BST with n nodes is **balanced** if the height of the tree is at most some constant times $\log n$. To balance our BSTs, we want to avoid rebuilding the whole tree. Instead, we'd like to make a minimal change to the tree that will preserve the BST property.

The most basic such operation is called a **tree rotation**. It comes in two forms, `rotateright` and `rotateleft`. Rotating a node to the right will move it to be the right child of its left child and will update the children of these nodes appropriately. Here it is in code.

```
def rotateright(self):
    newroot = self.left
    self.left = newroot.right
    newroot.right = self
    return newroot
```

Notice that `rotateright` returns the new root (of the subtree). This is a very useful convention when working with BSTs and rotations. Every method that can change the structure of the tree will return the new root of the resulting subtree. Thus, calling such methods will always be combined with an assignment. For example, if we want to rotate the left child of a node `parent` to the right, we would write `parent.left = parent.left.rotateright()`.

A `BSTMapping` implementation

Here is the code for the `BSTMapping` that implements rotations. We make sure to also update the lengths after each rotation. The main difference with our previous code is that now, all methods that can change the tree structure are combined with assignments. It is assumed that only `put` and `remove` will rearrange the tree, and so `get` and `floor` will keep the tree structure as is.

```
from mapping import Mapping, Entry


class BSTNode:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.left = None
        self.right = None
        self._length = 1

    def newnode(self, key, value):
        return BSTNode(key, value)

    def get(self, key):
        if key == self.key:
            return self
        elif key < self.key and self.left:
            return self.left.get(key)
        elif key > self.key and self.right:
            return self.right.get(key)
        else:
            raise KeyError

    def put(self, key, value):
        if key == self.key:
            self.value = value
```

```
        elif key < self.key:
            if self.left:
                self.left = self.left.put(key, value)
            else:
                self.left = self.newnode(key, value)
        elif key > self.key:
            if self.right:
                self.right = self.right.put(key, value)
            else:
                self.right = self.newnode(key, value)
        self.updatelength()
        return self

    def updatelength(self):
        len_left = len(self.left) if self.left else 0
        len_right = len(self.right) if self.right else 0
        self._length = 1 + len_left + len_right

    def floor(self, key):
        if key == self.key:
            return self
        elif key < self.key:
            return self.left.floor(key) if self.left else None
        elif key > self.key:
            return (self.right.floor(key) or self) if self.right else self

    def rotateright(self):
        newroot = self.left
        self.left = newroot.right
        newroot.right = self
        self.updatelength()
        newroot.updatelength()
        return newroot

    def rotateleft(self):
        newroot = self.right
        self.right = newroot.left
        newroot.left = self
        self.updatelength()
        newroot.updatelength()
        return newroot

    def maxnode(self):
```

```
        return self.right.maxnode() if self.right else self

    def _swapwith(self, other):
        """
        Swaps the key and value of a node.
        This operation has the potential to break the BST property.
        Use with caution!
        """
        self.key, other.key = other.key, self.key
        self.value, other.value = other.value, self.value

    def remove(self, key):
        if key == self.key:
            if self.left is None: return self.right
            if self.right is None: return self.left
            self._swapwith(self.left.maxnode())
            self.left = self.left.remove(key)
        elif key < self.key and self.left:
            self.left = self.left.remove(key)
        elif key > self.key and self.right:
            self.right = self.right.remove(key)
        else:
            raise KeyError
        self.updatelength()
        return self

    def __iter__(self):
        if self.left: yield from self.left
        yield Entry(self.key, self.value)
        if self.right: yield from self.right

    def preorder(self):
        yield self.key
        if self.left: yield from self.left.preorder()
        if self.right: yield from self.right.preorder()

    def __len__(self):
        return self._length

class BSTMapping(Mapping):
    Node = BSTNode

    def __init__(self):
```

```
        self._root = None

    def get(self, key):
        if self._root is None: raise KeyError
        return self._root.get(key).value

    def put(self, key, value):
        if self._root:
            self._root = self._root.put(key, value)
        else:
            self._root = self.Node(key, value)

    def floor(self, key):
        if self._root:
            floornode = self._root.floor(key)
            if floornode is not None:
                return floornode.key, floornode.value
        return None, None

    def remove(self, key):
        if self._root is None: raise KeyError
        self._root = self._root.remove(key)

    def _entryiter(self):
        if self._root:
            yield from self._root

    def preorder(self):
        if self._root:
            yield from self._root.preorder()

    def __len__(self):
        return len(self._root) if self._root else 0

    def __str__(self):
        return str(list(self.preorder()))
```

Forward Compatibility of Factories

We are looking into the future a little with this code. In particular, we want this class to be easily extendable. To do this, we don't create new instances of the nodes directly. In the `BSTMapping` class, we have the assignment `Node = BSTNode` and create new nodes by writing `self.Node(key,`

`value)` rather than `BSTNode(key, value)`. This way, when we extend this class (as we will several times), we can use a different value for `Node` in order to produce different types of nodes without having to rewrite all the methods from `BSTMapping`. Similarly, we use a `newnode` method in the `BSTNode` class. A method that generates new instances of a class is sometimes called a **factory**.

With regards to *forward compatibility*, it's a challenge to walk the fine line between, good design and premature optimization. In our case, the use of a factory to generate nodes only came about after the need was clear from writing subclasses. This should be a general warning to students when reading code in textbooks. One often only sees how the code ended up, but not how it started. It's helpful to ask, "Why is the code written this way or that way?", but the answer might be that it will make later code easier to write.

Weight Balanced Trees

Now, we're ready to balance our trees. One way to be sure the tree is balanced is to have the key at each node `x` be the median of all the keys in the subtree at `x`. Then, the situation would be analogous to binary search in a sorted `list`. Moving down the tree, every node would have have as many nodes in its subtree than its parent. Thus, after $\log n$ steps, we reach a leaf and so the BST would be balanced.

Having medians in every node a nice ideal to have, but it is both too difficult to achieve and too strong an invariant for balancing BSTs. Recall that in our analysis of `quicksort` and `quickselect`, we considered *good pivots* to be those that lay in the middle half of the list (i.e. those with rank between $n/4$ and $3n/4$). If the key in each node is a good pivot among the keys in its subtree, then we can also guarantee an overall height of $\log_{4/3} n$.

We'll say a node `x` is **weight-balanced** if

`len(x) + 1 < 4 * (min(len(x.left),len(x.right)) + 1)`.

It's easy enough to check this condition with our `BSTMapping` implementation because it keeps track of the length at each node. If some change causes a node to no longer be weight balanced, we will recover the weight balance by rotations. In the easiest case, a single rotation suffices. If one is rotation is not enough, then two rotations will be enough. We'll have to see some examples and do some simple algebra to see why.

The `rebalance` method will check for the balance condition and do the appropriate rotations. We'll

call this method after every change to the tree, so we can assume that the unbalanced node `x` is only just barely imbalanced. Without loss of generality, let's assume `x` has too few nodes in its left subtree. Then, when we call `x.rotateleft()`, we have to check that both `x` becomes weight balanced and also that its new parent `y` (the former right child of `x`) is weight balanced. Clearly, `len(x) == len(x.rotateleft())` as both contain the same nodes. So, there is an imbalance at `y` after the rotation only if its right child is too light. In that case, we rotate `y` right before rotating `x` left. This will guarantee that all the nodes in the subtree are weight balanced.

This description is not enough to be convincing that the algorithm is correct, but it is enough to write code. Let's look at the code first and see the rebalance method in action. Then, we'll go back and do the algebra to prove it is correct.

```
from bstmapping import BSTMapping, BSTNode


class WBTreeNode(BSTNode):
    def newnode(self, key, value):
        return WBTreeNode(key, value)

    def toolight(self, other):
        otherlength = len(other) if other else 0
        return len(self) + 1 >= 4 * (otherlength + 1)

    def rebalance(self):
        if self.toolight(self.left):
            if self.toolight(self.right.right):
                self.right = self.right.rotateright()
            newroot = self.rotateleft()
        elif self.toolight(self.right):
            if self.toolight(self.left.left):
                self.left = self.left.rotateleft()
            newroot = self.rotateright()
        else:
            return self
        return newroot

    def put(self, key, value):
        newroot = BSTNode.put(self, key, value)
        return newroot.rebalance()

    def remove(self, key):
        newroot = BSTNode.remove(self, key)
        return newroot.rebalance() if newroot else None

class WBTreeMapping(BSTMapping):
    Node = WBTreeNode
```

The `toolight` method is for checking if a subtree has enough nodes to be a child of a weight balanced node. We use it both to check if the current node is weight balanced and also to check if one rotation or two will be required.

The `put` and `remove` methods call the corresponding methods from the superclass `BSTNode` and then rebalance before returning. We tried to reuse as much as possible our existing implementation.

Height-Balanced Trees (AVL Trees)

The motivation for balancing our BSTs was to keep the height small. Rather than balancing by weight, we could also try to keep the heights of the left and right subtrees close. In fact, we can require that these heights differ by at most one. Similar to weight balanced trees, we'll see if the height balanced property is violated and if so, fix it with one or two rotations.

Such height balanced trees are often called AVL trees. In our implementation, we'll maintain the height of each subtree and use these to check for balance. Often, AVL trees only keep the balance at each node rather than the exact height, but computing heights is relatively painless.

```
from bstmapping import BSTMapping, BSTNode

def height(node):
    return node.height if node else -1

def update(node):
    if node:
        node.updatelength()
        node.updateheight()

class AVLTreeNode(BSTNode):
    def __init__(self, key, value):
        BSTNode.__init__(self, key, value)
        self.updateheight()

    def newnode(self, key, value):
        return AVLTreeNode(key, value)

    def updateheight(self):
        self.height = 1 + max(height(self.left), height(self.right))

    def balance(self):
        return height(self.right) - height(self.left)

    def rebalance(self):
        bal = self.balance()
        if bal == -2:
            if self.left.balance() > 0:
                self.left = self.left.rotateleft()
            newroot = self.rotateright()
```

```
        elif bal == 2:
            if self.right.balance() < 0:
                self.right = self.right.rotateright()
            newroot = self.rotateleft()
        else:
            return self
        update(newroot.left)
        update(newroot.right)
        update(newroot)
        return newroot

    def put(self, key, value):
        newroot = BSTNode.put(self, key, value)
        update(newroot)
        return newroot.rebalance()

    def remove(self, key):
        newroot = BSTNode.remove(self, key)
        update(newroot)
        return newroot.rebalance() if newroot else None

class AVLTreeMapping(BSTMapping):
    Node = AVLTreeNode
```

Splay Trees

Let's add in one more balanced BST for good measure. In this case, we won't actually get a guarantee that the resulting tree is balanced, but we will get some other nice properties.

In a **splay tree**, every time we get or put an entry, its node will get rotated all the way to the root. However, instead of rotating it directly, we consider two steps at a time. The `splayup` method looks two levels down the tree for the desired key. If its not exactly two levels down, it does nothing. Otherwise, it rotates it up twice. If the rotations are in the same direction, the bottom one is done first. If the rotations are in opposite directions, it does the top one first.

At the very top level, we may do only a single rotation to get the node all the way to the root. This is handled by the `splayup` method in the `SplayTreeMapping` class.

A major difference from our previous implementations is that now, we will modify the tree on calls to

`get`. As a result, we will have to rewrite `get` rather than inheriting it. Previously, get would return the desired value. However, we want to return the new root on every operation that might change the tree. So, which should we return? Clearly, we need that value to return, and we also need to not break the tree. Thankfully, there is a simple solution. The splaying operation conveniently rotates the found node all the way to the root. So, the `SplayTreeNode.get` method will return the new root of the subtree, and the `SplayTreeMapping.get` returns the value at the root.

```
from bstmapping import BSTMapping, BSTNode

class SplayTreeNode(BSTNode):
    def newnode(self, key, value):
        return SplayTreeNode(key, value)

    def splayup(self, key):
        newroot = self
        if key < self.key:
            if key < self.left.key:
                newroot = self.rotateright().rotateright()
            elif key > self.left.key:
                self.left = self.left.rotateleft()
                newroot = self.rotateright()
        elif key > self.key:
            if key > self.right.key:
                newroot = self.rotateleft().rotateleft()
            elif key < self.right.key:
                self.right = self.right.rotateright()
                newroot = self.rotateleft()
        return newroot

    def put(self, key, value):
        newroot = BSTNode.put(self, key, value)
        return newroot.splayup(key)

    def get(self, key):
        if key == self.key:
            return self
        elif key < self.key and self.left:
            self.left = self.left.get(key)
        elif key > self.key and self.right:
            self.right = self.right.get(key)
        else:
            raise KeyError
```

```
        return self.splayup(key)

class SplayTreeMapping(BSTMapping):
    Node = SplayTreeNode

    def splayup(self, key):
        if key < self._root.key:
            self._root = self._root.rotateright()
        if key > self._root.key:
            self._root = self._root.rotateleft()

    def get(self, key):
        if self._root is None: raise KeyError
        self._root = self._root.get(key)
        self.splayup(key)
        return self._root.value

    def put(self, key, value):
        BSTMapping.put(self, key, value)
        self.splayup(key)
```

Chapter 18

Priority Queues

We're going to discuss a versatile and fundamental data structure called a **priority queue**. It will resemble a queue except that items all have a priority and will come out ordered by their priority rather than their order of insertion.

By this point, we know enough about different data structures that we will be able to implement this data structure in several different ways. Eventually, we will introduce a data structure called a **heap** that will give good performance in many respects, but it won't be

The Priority Queue ADT

The **Priority Queue ADT** is a data type that stores a collection of items with priorities (not necessarily unique) that supports the following operations.

- `insert(item, priority)` - Add `item` with the given `priority`.
- `findmin()` - Return the item with the minimum priority. If there are multiple items with the minimum priority, ties may be broken arbitrarily.
- `removemin()` - Remove and return the item with the minimum priority. Ties are broken arbitrarily.

Using a list

As with every ADT presented in this book, there is a simple, though not necessarily efficient, implementation using a list. Let's start with a list and put all the entries in the list. To find the min, we

can iterate through the list.

```
class SimpleListPQ:
    def __init__(self):
        self._L = []

    def insert(self, item, priority):
        self._L.append((item, priority))

    def findmin(self):
        return min(self._L, key = lambda x : x[1])[0]

    def removemin(self):
        item, priority = min(self._L, key = lambda x : x[1])
        self._L.remove((item, priority))
        return item
```

Although this code will work, there are a couple design decisions that are not great. First, there is something we might call *magical indices*. As the data are stored as tuples, we have to use an index to pull out the priority or the item as needed. This is an undocumented convention. Someone reading the code would have to read the insert code to understand the `findmin` code. Also, there is some duplication in the use of `min` with the same `key` lambda expression for both `removemin` and `findmin`. These shortcomings can be our motivation to go back and clean it up, i.e. **refactor** it.

We'll make a class that stores the entries, each will have an `item` and a `priority` as attributes. We'll make the entries comparable by implementing `__lt__` and thus the comparison of entries will always just compare their priorities.

```
class Entry:
    def __init__(self, item, priority):
        self.priority = priority
        self.item = item

    def __lt__(self, other):
        return self.priority < other.priority
```

Now, we can rewrite the list version of the priority queue. It's almost the same as before, except, that by using the `Entry` class, the code is more explicit about the types of objects, and hopefully, easier to read and understand.

```
class UnsortedListPQ:
    def __init__(self):
        self._entries = []

    def insert(self, item, priority):
        self._entries.append(Entry(item, priority))

    def findmin(self):
        return min(self._entries).item

    def removemin(self):
        entry = min(self._entries)
        self._entries.remove(entry)
        return entry.item
```

Let's analyze the running time of the methods in this class. The `insert` method will clearly only require constant time because it is a single list append. Both `findmin` and `removemin` take $O(n)$ time for a priority queue with n items, because the `min` function will iterate through the while list to find the smallest priority entry.

The `findmin` would be much faster if the list were sorted. Then we could just return the first item in the list. An even better approach would sort the list backwards. Then `removemin` could also be constant time, using the constant-time `pop` function on a list.

```
class SortedListPQ:
    def __init__(self):
        self._entries = []

    def insert(self, item, priority):
        self._entries.append(Entry(item, priority))
        self._entries.sort(reverse = True)

    def findmin(self):
        return self._entries[-1].item

    def removemin(self):
        return self._entries.pop().item
```

I used a nice feature of the python `sort` method, a named parameter called `reversed` which, as one might guess, reverses the sorting order.

The asymptotic running time of both `findmin` and `removemin` is reduced to constant-time. The `insert` method no longer runs in constant time, because it sorts. This means that the running time will depend on the time it takes to sort. In this case, assuming no other methods rearrange the (private!) `_entries` attribute, there will be only one entry out of sorted order, the one we just appended. In this case, a good implementation of insertion sort would run in linear time (it only has to insert one item before everything is sorted). Although, for code brevity, we called out to python's `sort` method, (which could take $O(n \log n)$-time) this is really a linear-time operation).

What we have with our two list implementations of the priority queue ADT is **tradeoff** between two operations. In one case, we have fast `insert` and slow `removemin` and in the other we have slow `insert` and fast `removemin`. Our goal will be to get all of the operations pretty fast. We may not achieve constant time for all, but logarithmic-time is achievable.

Heaps

The data structure we'll use for an efficient priority queue is called a **heap**. Heaps are almost always used to implement a priority queue so that as you look at other sources, you might not see a distinction between the two ideas. As we are using it, the priority queue is the ADT, the heap is the data structure.

Matters are complicated by two other vocabulary issues. First, there are many different kinds of heaps. We'll study just one example, the so-called binary heap. Second, the usual word used for the *priority* in a heap is "key". This can be confusing, because unlike *keys* in mapping data structures, there is no requirement that priorities be unique. We will stick with the word "priority" despite the usual conventions in order to keep these ideas separate.

We can think of a binary heap as a binary tree that is arranged so that smaller priorities are above larger priorities. The name is apt as any good heap of stuff should have the big things on the bottom and small things on the top. For any tree with nodes that have priorities, we say that the **tree is heap-ordered** if for every node, the priority of its children are at least as large as the the priority of the node itself. This naturally implies that the minimum priority is at the root.

Storing a tree in a list

Previously, we stored trees as a **linked data structure**. This means having a node class that stores references to other nodes, i.e. the children. We'll use the heap as an opportunity to see a different

way to represent a tree. We'll put the heap entries in a list and let the list indices encode the parent-child relationships of the tree.

For a node at index `i`, its left child will be at index `2 * i + 1` and the right child will be at index `2 * i + 2`. This means that the parent of the node at index `i` is at index `(i-1) // 2`. This mapping of nodes to indices puts the root at index `0` and the rest of the nodes appear level by level from left-to-right, top-to-bottom. Using this mapping, we say a **list is heap-ordered** if the corresponding binary tree is heap-ordered. This is equivalent to saying that for all $i > 0$, the priority of the entry at index i is greater than or equal to the priority at index $(i-1)//2$.

We will maintain the invariant that after each operation, the list of entries is heap-ordered. To insert a new node, we append it to the list and then repeated swap it with its parent until it is heap-ordered. This updating step will be called `_upheap` and is similar to the inner loop of an insertion sort. However, unlike insertion sort, it only does at most $O(\log n)$ swap operations (each one reduces the index by half).

Similarly, there is a `_downheap` operation that will repeatedly swap an entry with its child until it's heap-ordered. This operation is useful in the next section for building a heap from scratch.

```
class HeapPQ:
    def __init__(self):
        self._entries = []

    def insert(self, item, priority):
        self._entries.append(Entry(item, priority))
        self._upheap(len(self._entries) - 1)

    def _parent(self, i):
        return (i - 1) // 2

    def _children(self, i):
        left = 2 * i + 1
        right = 2 * i + 2
        return range(left, min(len(self._entries), right + 1))

    def _upheap(self, i):
        L = self._entries
        parent = self._parent(i)
        if i > 0 and L[i] < L[parent]:
            L[i], L[parent] = L[parent], L[i]
            self._upheap(parent)

    def findmin(self):
        return self._entries[0].item

    def removemin(self):
        L = self._entries
        item = L[0].item
        L[0] = L[-1]
        L.pop()
        self._downheap(0)
        return item

    def _downheap(self, i):
        L = self._entries
        children = self._children(i)
        if children:
            child = min(children, key = lambda x: L[x])
            if L[child] < L[i]:
                L[i], L[child] = L[child], L[i]
                self._downheap(child)
```

Building a Heap from scratch

Just using the public interface, one could easily construct a `HeapPQ` from a list of item-priority pairs. For example, the following code would work just fine.

```
pq = HeapPQ()
pairs = [(10, 10), (2, 2), (30, 30), (4,4)]
for item, priority in pairs:
    pq.insert(item, priority)
```

The `insert` method takes $O(\log n)$ time, so the total running time for this approach is $O(n \log n)$ time. *You should double check that you believe this claim despite the fact that for each insertion, there are fewer than n entries.*

Perhaps surprisingly, we can construct the `HeapPQ` in linear time. We call this **heapifying a list**. We will exploit the `_downheap` method that we have already written. The code is deceptively simple.

```
def _heapify(self):
    n = len(self._entries)
    for i in range(n):
        self._downheap(n - i - 1)
```

Look at the difference between the `heapify` code above and the `_heapify_slower` code below. The former works from the end and "downheaps" every entry and the latter starts at the beginning and "upheaps" every entry. Both are correct, but one is faster.

```
def _heapify_slower(self):
    n = len(self._entries)
    for i in range(n):
        self._upheap(i)
```

They may seem to be the same, but they are not. To see why, we have to look a little closer at the running time of `_upheap` and `_downheap`. Consider the tree perspective of the list. For `_upheap`, the running time depends on the depth of the starting node. For `_downheap`, the running time depends on the height of the subtree rooted at the starting node. Looking at a complete binary tree, half of the nodes are leaves and so `_downheap` will take constant time and `_upheap` will take $O(\log n)$ time. Thus, `_heapify_slower` will take at least $\frac{n}{2} \log_2 n = O(n \log n)$ time.

On the other hand, to analyze `_heapify`, we have to add up the heights of all the nodes in a complete binary tree. Formally, this will be $n \sum_{i=1}^{\log_2 n} i/2^i$. There is a cute trick to bound this sum. Simply observe that if from every node in the tree, we take a path that goes left on the first step and right for every step thereafter, no two paths will overlap. This means that the the sum of the lengths of these paths (which is also to the sum of the heights) is at most the total number of edges, $n - 1$. Thus, `_heapify` runs in $O(n)$ time.

Changing priorities

There is another "standard" operation on priority queues. It's called `reducepriority`. It does what it says, reducing the priority of an entry. This is very simple to implement in our current code if the index of the entry to change is known. It would suffice to change the priority of the entry and call `_upheap` on that index.

However, the usual way to reduce the priority is to specify the item and its new priority. This requires that we can find the index of an item. We'll do this by storing a dictionary that maps items to indices. We'll have to update this dictionary every time we rearrange the items. To make sure this happens correctly, we will add a method for swapping entries and use this whenever we make a swap.

The full code including the `_heapify` method is given below. This full version of the priority queue will be very useful for some graph algorithms that we will see soon.

```
class PriorityQueue:
    def __init__(self, entries = None):
        entries = entries or []
        self._entries = [Entry(i, p) for i, p in entries]
        self._itemmap = {i: index for index, (i,p) in enumerate(entries)}
        self._heapify()

    def insert(self, item, priority):
        index = len(self._entries)
        self._entries.append(Entry(item, priority))
        self._itemmap[item] = index
        self._upheap(index)

    def _parent(self, i):
        return (i - 1) // 2

    def _children(self, i):
```

```
        left, right = 2 * i + 1, 2 * i + 2
        return range(left, min(len(self._entries), right + 1))

    def _swap(self, a, b):
        L = self._entries
        va = L[a].item
        vb = L[b].item
        self._itemmap[va] = b
        self._itemmap[vb] = a
        L[a], L[b] = L[b], L[a]

    def _upheap(self, i):
        L = self._entries
        parent = self._parent(i)
        if i > 0 and L[i] < L[parent]:
            self._swap(i,parent)
            self._upheap(parent)

    def reducepriority(self, item, priority):
        i = self._itemmap[item]
        entry = self._entries[i]
        entry.priority = min(entry.priority, priority)
        self._upheap(i)

    def findmin(self):
        return self._entries[0].item

    def removemin(self):
        L = self._entries
        item = L[0].item
        self._swap(0, len(L) - 1)
        del self._itemmap[item]
        L.pop()
        self._downheap(0)
        return item

    def _downheap(self, i):
        L = self._entries
        children = self._children(i)
        if children:
            child = min(children, key = lambda x: L[x])
            if L[child] < L[i]:
                self._swap(i, child)
```

```
                self._downheap(child)

    def __len__(self):
        return len(self._entries)

    def _heapify(self):
        n = len(self._entries)
        for i in range(n):
            self._downheap(n - i - 1)
```

Chapter 19

Graphs

The **graph** is a fundamental mathematical object in computer science. Graphs are used to abstract networks, maps, program control flow, probabilistic models, and even data structures among many other computer science concepts.

A good mental model to start with is a road map of a country. There are cities and roads connecting those cities. Graphs are ideal for describing such situations where there are items and connections between them.

Formally, a graph is a pair (V, E) where V is any set and E is a set of pairs of elements of V. We call V the **vertex set** and E the **edge set**. Just using the definition of a graph and the standard python collections, we could store a graph as follows.

```
G = ({1,2,3,4}, {(1,2), (1,3), (1,4)})
```

The graph `G` has 4 vertices and 3 edges. Such a graph is often depicted 4 labeled circles for vertices and lines between them depicting the edges. Because tuples are ordered, we would add arrowheads. This indicates a **directed graph** or **digraph**. If the ordering of the vertices in an edge does not matter, we have an **undirected graph**. There will be no major difference between an undirected graph and a digraph that has a matched pair (v, u) for every (u, v) in E.

Usually, we will disallow **self-loops**, edges that start and end at the same vertex, and **multiple edges** between the same pair of vertices. Graphs without self-loops and multiple edges are called **simple graphs**.

If two vertices are connected by an edge, we say they are **adjacent**. We all call such vertices

neighbors. For an edge $e = (u, v)$, we say that the vertices u and v are **incident** to the edge e. The **degree of a vertex** is the number of neighbors it has. For digraphs, we distinguish between **in-degree** and **out-degree**, the number of in-neighbors and out-neighbors respectively.

A Graph ADT

First, let's establish the minimal ADT for a simple, directed graph. We should be able to use any (hashable) type for the vertices. We want to be able to create the graph from a collection of vertices and a collection of ordered pairs of vertices.

- `__init__(V, E)` : Initialize a new graph with vertex set `V` and edge set `E`.
- `vertices()` : Return an iterable collection of the vertices.
- `edges()` : Return an iterable collection of the edges.
- `addvertex(v)` : Add a new vertex to the graph. The new vertex is identified with the `v` object.
- `addedge(u, v)` : Add a new edge to the graph between the vertices with keys `u` and `v`.
- `nbrs(v)` : Return an iterable collection of the (out)neighbors of `v`, i.e. those vertices `w` such that `(v, w)` is an edge. (For directed graphs, this is the collection of out-neighbors.)

There are several error conditions to consider. For example, what happens if a vertex is added more than once? What happens if an edge `(u,v)` is added, but `u` is not a vertex in the graph? What if one calls `nbrs(v)` and `v` is not a vertex in the graph? We will deal with these later. For now, let's get something working.

The `EdgeSetGraph` Implementation

Cleverness comes second. First, let's write a Graph class that is as close as possible to the mathematical definition of a graph. It will store a set for the vertices and a set of pairs (2-tuples) for the edges. To get the neighbors of a vertex, it will iterate over all edges in the graph.

```
class EdgeSetGraph:
    def __init__(self, V, E):
        self._V = set()
        self._E = set()
        for v in V: self.addvertex(v)
        for u,v in E: self.addedge(u,v)

    def vertices(self):
        return iter(self._V)

    def edges(self):
        return iter(self._E)

    def addvertex(self, v):
        self._V.add(v)

    def addedge(self, u, v):
        self._E.add((u,v))

    def nbrs(self, v):
        return (w for u,w in self._E if u == v)
```

To make an undirected version of this class, we can simply modify the `addedge` method to add an edge in each direction. We also change the `nbrs` method to work with unordered pairs (sets).

```
class UndirectedEdgeSetGraph(EdgeSetGraph):
    def addedge(self, u, v):
        self._E.add((u,v))
        self._E.add((v,u))

    def edges(self):
        edgeset = set()
        for u,v in self._E:
            if (v,u) not in edgeset:
                edgeset.add((u,v))
        return iter(edgeset)
```

The `AdjacencySetGraph` Implementation

The major problem with previous implementation is that it's very slow to enumerate the neighbors of

a vertex. It should not be necessary to go through all the edges, just to find the ones incident to a given vertex.

An alternative approach is to store a set with each vertex and have this set contain all the neighbors of that vertex. This allows for fast access to the neighbors. In fact, it means that we don't have to store the edges explicitly at all.

```
class AdjacencySetGraph:
    def __init__(self, V, E):
        self._V = set()
        self._nbrs = {}
        for v in V: self.addvertex(v)
        for u,v in E: self.addedge(u,v)

    def vertices(self):
        return iter(self._V)

    def edges(self):
        for u in self._V:
            for v in self.nbrs(u):
                yield (u,v)

    def addvertex(self, v):
        self._V.add(v)
        self._nbrs[v] = set()

    def addedge(self, u, v):
        self._nbrs[u].add(v)

    def nbrs(self, v):
        return iter(self._nbrs[v])
```

```
G = AdjacencySetGraph({1,2,3}, {(1,2),(2,1),(1,3)})
print("neighbors of 1:", list(G.nbrs(1)))
print("neighbors of 2:", list(G.nbrs(2)))
print("neighbors of 3:", list(G.nbrs(3)))
```

```
neighbors of 1: [2, 3]
neighbors of 2: [1]
neighbors of 3: []
```

Let's do a very simple example to make sure it works as we expect it to.

One design decision that may seem rather strange is that we do not have a class for vertices or edges. As of now, we are exploiting the **polymorphism** of python, which allows us to use any (hashable) type for the vertex set. Recall that polymorphism (in this context) is the ability to call the same function on different types of objects. This can be very convenient as it allows one to put a graph structure on top of any set.

As written, there is no clear reason to use a set for the neighbors rather than a list. One case where the set is better is if we want to test if the graph contains a particular edge. With the `AdjacencySetGraph` above, this method could be implemented as follows.

```
    def hasedge(self, u, v):
        return v in self._nbrs[u]
```

If `self._nbrs[u]` were a list, the method could take time linear in the degree of `u`.

Paths and Connectivity

A **path** in a graph $G = (V, E)$ is a sequence of vertices connected by edges. That is, a nonempty sequence of vertices $(v_0, v_1, \ldots, v_k)$ is a path from v_0 to v_k as long as $(v_{i-1}, v_i) \in E$ for all $i \in \{1, \ldots, k\}$. We say a **path is simple** if it does not repeat any vertices. The **length of a path** is the number of edges. A single vertex can be seen as a path of length zero.

A **cycle** is a path of length at least one that starts and ends at the same vertex. The **length of a cycle** is the number of edges. A **cycle is simple** if is is a cycle and removing the last edge results in a simple path, i.e., there are no repeated vertices until the last one.

To solidify these definitions, we could write a couple methods to check them.

```
def ispath(G, V):
  """Return True if and only if the vertices V form a path in G."""
  return V and all(G.hasedge(V[i-1], V[i]) for i in range(1, len(V)))

def issimplepath(G, V):
  """Return True if and only if the vertices V form a simple path in G."""
  return ispath(G, V) and len(V) == len(set(V))

def iscycle(G, V):
    """Return True if and only if the vertices V form a cycle in G."""
    return ispath(G, V) and V[0] == V[-1]

def issimplecycle(G, V):
    """Return True if and only if the vertices V form a simple cycle in G."""
    return iscycle(G,V) and issimplepath(G, V[:-1])
```

We say that u is **connected** to v if there exists a path that starts at u and ends at v. For an undirected graph, if u is connected to v, then v is connected to u. In such graphs, we can partition the vertices into subsets called **connected components** that are all pairwise connected.

For a directed graph, two vertices u and v are **strongly connected** if u is connected to v *and also* v is connected to u.

Let's consider a simple method to test of two vertices are connected. We will add it to our `AdjacencySetGraph` class. As a first exercise, we could try to work recursively. The idea is simple: in the base case, see if you are trying to get from a vertex to itself. Otherwise, it suffices to check if any of the neighbors of the first vertex are connected to the last vertex.

```
    def connected(self, a, b):
        if a == b: return True
        return any(self.connected(nbr, b) for nbr in self.nbrs(a))
```

Can you guess what will go wrong? Here's an example.

```
G = AdjacencySetGraph({1,2,3}, {(1,2), (2,3)})
assert(G.connected(1,2))
assert(G.connected(1,3))
assert(not G.connected(3,1))
print("First graph is okay.")

H = AdjacencySetGraph({1,2,3}, {(1,2), (2,1), (2,3)})
try:
    H.connected(1,3)
except RecursionError:
    print("There was too much recursion!")
```

```
First graph is okay.
There was too much recursion!
```

It's clear that if the graph has any cycles, we can't check connectivity this way. To deal with cycles, we can keep a set of visited vertices. Recall that this is called memoization.

```
    def connected(self, a, b):
        return self._connected(a, b, set())

    def _connected(self, a, b, visited):
        if a in visited: return False
        if a == b: return True
        visited.add(a)
        return any(self._connected(nbr, b, visited) for nbr in self.nbrs(a))
```

Now, we can try again and see what happens.

```
H = AdjacencySetGraph({1,2,3}, {(1,2), (2,1), (2,3)})
try:
    assert(H.connected(1,2))
    assert(H.connected(1,3))
except RecursionError:
    print('There was too much recursion!')
print('It works now!')
```

```
It works now!
```

Chapter 20

Graph Search

Now that we have a vocabulary for dealing with graphs, we can revisit some previous topics. For example, a tree can now be viewed as a graph with edges directed from parents to children. Given a graph that represents a rooted tree, we could rewrite our preorder traversal using the Graph ADT. Let's say we just want to print all the vertices.

```
def printall(G, v):
    print(v)
    for n in G.nbrs(v):
        printall(G, n)
```

This is fine for a tree, but it quickly gets very bad as soon as there is a cycle. In that case, there is nothing in the code to keep us from going around and around the cycle. We can take the same approach as Hansel and Gretel: we're going to leave bread crumbs to see where we've been. The easiest way to do this is to just keep around a set that stores the vertices that have been already visited in the search. Thus, our `printall` method becomes the following.

```
def printall(G, v, visited):
    print(v)
    for n in G.nbrs(v):
        if n not in visited:
            visited.add(n)
            printall(G, n)
```

This is the most direct generalization of a recursive tree traversal into something that also traverses the vertices of a graph. Unlike with (rooted) trees, we have to specify the starting vertex, because

there is no specially marked "root" vertex in a graph. Also, this code will not necessarily print all the vertices in a graph. In this case, it will only print those vertices that are connected to the starting vertex. This pattern can be made to work more generally.

Depth-First Search

A **depth-first search** (or **DFS**) of a graph G starting from a vertex v will visit all the vertices connected to v. It will always prioritize moving "outward" in the direction of new vertices, backtracking as little as possible. The `printall` method above prints the vertices in a **depth-first order**. Below is the general form of this algorithm.

```
def dfs(self, v):
    visited = {v}
    self._dfs(v, visited)
    return visited

def _dfs(self, v, visited):
    for n in self.nbrs(v):
        if n not in visited:
            visited.add(n)
            self._dfs(n, visited)
```

With this code, it will be easy to check if two vertices are connected. For example, one could write the following.

```
def connected(G, u, v):
    return v in G.dfs(u)
```

It's possible to modify our `dfs` code to provide not only the set of connected vertices, but also the paths used in the search. The idea is to store a dictionary that maps vertices to the previous vertex in the path from the starting vertex. This is really encoding a tree as a dictionary that maps nodes to their parent. The root is the starting vertex.

It differs from the trees we've seen previously in that it naturally goes up the tree from the leaves to the root rather than the reverse. The resulting tree is called the **depth-first search tree**. It requires only a small change to generate it. We use the convention that the starting vertex maps to `None`.

```
def dfs(self, v):
    tree = {v: None}
    self._dfs(v, tree)
    return tree

def _dfs(self, v, tree):
    for n in self.nbrs(v):
        if n not in tree:
            tree[n] = v
            self._dfs(n, tree)
```

Removing the Recursion

The `dfs` code above uses recursion to keep track of previous vertices, so that we can backtrack (by `return`ing) when we reach a vertex from which we can't move forward. To remove the recursion, we replace the function call stack with our own stack.

This is not just an academic exercise. By removing the recursion, we will reveal the structure of the algorithm in a way that will allow us to generalize it to other algorithms. Here's the code.

```
def dfs(self, v):
    tree = {}
    tovisit = [(None, v)]
    while tovisit:
        a,b = tovisit.pop()
        if b not in tree:
            tree[b] = a
            for n in self.nbrs(b):
                tovisit.append((b,n))
    return tree
```

Breadth-First Search

We get another important traversal by replacing the stack with a queue. In this case, the search prioritizes breadth over depth, resulting in a **breadth-first search** of **BFS**.

```
def bfs(self, v):
    tree = {}
    tovisit = Queue([(None, v)])
    while tovisit:
        a,b = tovisit.dequeue()
        if b not in tree:
            tree[b] = a
            for n in self.nbrs(b):
                tovisit.enqueue((b,n))
    return tree
```

A wonderful property of breadth-first search is that the paths encoded in the resulting **breadth-first search tree** are the shortest paths from each vertex to the starting vertex. Thus, BFS can be used to find the shortest path connecting pairs of vertices in a graph.

```
def distance(G, u, v):
    tree = G.bfs(u)
    if v not in tree:
        return float('inf')
    edgecount = 0
    while v is not u:
        edgecount += 1
        v = tree[v]
    return edgecount
```

Weighted Graphs and Shortest Paths

In the **single source, all shortest paths problem**, the goal is to find the shortest path from every vertex to a given source vertex. If the edges are assumed to have the same length, then BFS solves this problem. However, it is common to consider **weighted graphs** in which a (positive) real number called the **weight** is assigned to each edge. We will augment our graph ADT to support a function `wt(u,v)` that returns the weight of an edge. Then, the weight of a path is the sum of the weights of the edges on that path. Now, simple examples make it clear that the shortest path may not be what we get from the BFS tree.

One nice algorithm for the single source, all shortest paths problem on weighted graphs is called Dijkstra's algorithm. It looks a lot like DFS and BFS except now, the stack or queue is replaced by a priority queue. The vertices will be visited in order of their distance to the source. These distances

will be used as the priorities in the priority queue.

We'll see two different implementations. The first, although less efficient is very close to DFS and BFS. Recall that in those algorithms, we visit the vertices, recording the edges used in a dictionary and adding all the neighboring vertices to a stack or a queue to be traversed later. We'll do the same here except that we'll use a priority queue to store the edges to be searched. We'll also keep a dictionary of the distances from the start vertex that will be updated when we visit a vertex. The priority for an edge `(u,v)` will be the distance to `u` plus the weight of `(u,v)`. So, if we use this edge, the shortest path to `v` will go through `u`. In this way, the tree will encode all the shortest paths from the start vertex. Thus, the result will be not only the lengths of all the paths, but also an efficient encoding of the paths themselves.

```
def dijkstra(self, v):
    tree = {}
    D = {v: 0}
    tovisit = PriorityQueue()
    tovisit.insert((None,v), 0)
    while tovisit:
        a,b = tovisit.removemin()
        if b not in tree:
            tree[b] = a
            if a is not None:
                D[b] = D[a] + self.wt(a,b)
            for n in self.nbrs(b):
                tovisit.add((b,n), D[b] + self.wt(b,n))
    return tree, D
```

Prim's Algorithm for Minimum Spanning Trees

Recall that a subgraph of an undirected graph $G = (V, E)$ is a **spanning tree** if it is a tree with vertex set V. For a weighted graph, the weight of a spanning tree is the sum of the weights of its edges. The **Minimum Spanning Tree** (**MST**) Problem is to find *a* spanning tree of an input graph with minimum weight. The problem comes up naturally in many contexts.

To find an algorithm for this problem, we start by trying to describe which edges should appear in the minimum spanning tree. That is, we should think about the object we want to construct first, and only then can we think about *how* to construct it. We employed a similar strategy when discussing sorting algorithms. In that case, we first tried to write a function that would test for correct output.

We won't go that far now, but we will ask, "How would we know if we had the minimum spanning tree?"

One thing that would certainly be true about the minimum spanning tree is that if we removed an edge (resulting in two trees), we couldn't find a lighter weight edge connecting these two trees. Otehrwise, that would be a spanning tree of lower weight.

Something even a little more general is true. If we split the vertices into any two sets A and B, the lowest weight edge with one end in A and the other end in B must be in the MST. Suppose for contradiction that this edge is not in the MST. Then, we could add that edge and form a cycle, which would have another edge spanning A and B. That edge could then be removed, leaving us with a lighter spanning tree. This is a contradiction, because we assumed that we started with the MST.

So, in light of our previous graph algorithms, we can try to always add the lightest edge from the visited vertices to an unvisited vertex. This can easily be encoded in a priority queue.

```
def prim(self, v):
    tree = {}
    tovisit = PriorityQueue()
    tovisit.insert((None, v), 0)
    while tovisit:
        a,b = tovisit.removemin()
        if b not in tree:
            tree[b] = a
            for n in self.nbrs(b):
                tovisit.add((b,n), self.wt(b,n))
    return tree
```

An optimization for Priority-First search

The implementation above, although correct, is not technically Dijkstra's Algorithm, but it's close. By trying to improve its asymptotic running time, we'll get to the real Dijkstra's Algorithm. When thinking about how to improve an algorithm, an easy first place to look is for wasted work. In this case, we can see that many edges added to the priority queue are later removed without being used, because they lead to a vertex that has already been visited (by a shorter path).

It would be better to not have to these edges in the priority queue, but we don't know when we first see an edge whether or not a later edge will provide a short cut. We can avoid adding a new entry to

the priority and instead modify the existing entry that is no longer valid.

The idea is to store vertices rather than edges in the priority queue. Then, we'll use the `reducepriority` method to update an entry when we find a new shorter path to a given vertex. Although we won't know the distances at first, we'll store the shortest distance we've seen so far. If we find a shortcut to a given vertex, we will reduce it's priority and update the priority queue. Updating after finding a shortcut is called **edge relaxation**. It works as follows. The distances to the source are stored in a dictionary `D` that maps vertices to the distance, based on what we've searched so far. If we find that `D[n] > D[u] + G.wt(u,n)`, then it would be a shorter path to `n` if we just took the shortest path from the source to `u` and appended the edge `(u,n)`. In that case, we set `D[n] = D[u] + G.wt(u,n)` and update the priority queue. *Note that we had this algorithm in mind when we added `reducepriority` to our Priority Queue ADT.*

Here's the code.

```
def dijkstra2(self, v):
    tree = {v: None}
    D = {u: float('inf') for u in self.vertices()}
    D[v] = 0
    tovisit = PriorityQueue([(u, D[u]) for u in self.vertices()])
    while tovisit:
        u = tovisit.removemin()
        for n in self.nbrs(u):
            if D[u] + self.wt(u,n) < D[n]:
                D[n] = D[u] + self.wt(u,n)
                tree[n] = u
                tovisit.reducepriority(n, D[n])
    return tree, D
```

An important difference with our DFS/BFS code is that the main data structure now stores vertices, not edges. Also, we don't have to check if we've already visited a vertex, because if we have already visited it, we won't find an edge to relax (vertices are visited in order of their distance to the source).