Part I

Python Essentials

Lab 1

The Standard Library

Lab Objective: *Python is designed to make it easy to implement complex tasks with little code. To that end, any Python distribution includes several built-in functions for accomplishing common tasks. In addition, Python is designed to import and reuse code written by others. A Python file that can be imported is called a module. All Python distributions include a collection of modules for accomplishing a variety of common tasks, collectively called the* Python Standard Library. *In this lab we become familiar with the Standard Library and learn how to create, import, and use modules.*

Built-in Functions

Every Python installation comes with several built-in functions. These functions may be used at any time and Python will recognize them. IPython's object introspection feature makes it easy to learn about built-in functions. Start IPython from the terminal and use ? to bring up technical details on each function.

```
In [1]: min?
Docstring:
min(iterable[, key=func]) -> value
min(a, b, c, ...[, key=func]) -> value

With a single iterable argument, return its smallest item.
With two or more arguments, return the smallest argument.
Type:      builtin_function_or_method

In [2]: len?
Docstring:
len(object) -> integer

Return the number of items of a sequence or collection.
Type:      builtin_function_or_method
```

The following code demonstrates each of the common built-in functions listed in Table 1.1. They are quite intuitive and easy to use.

Function	Returns
`abs()`	The absolute value of a real number, or the magnitude of a complex number.
`min()`	The smallest element of a single iterable, or the smallest of several arguments. Strings are compared based on lexicographical order: numerical characters first, then upper-case letters, then lower-case letters.
`max()`	The largest element of a single iterable, or the largest of several arguments.
`len()`	The number of items of a sequence or collection.
`sum()`	The sum of a sequence of numbers.

Table 1.1: Some common built-in functions. Documentation on all built-in functions can be found at `https://docs.python.org/2/library/functions.html`.

```
# abs() can be used with real or complex numbers.
>>> abs(-7)
7
>>> abs(3 + 4j)
5.0

# min() can be used on a list, string, or several arguments.
>>> min([4, 2, 6])
2
>>> min('aXbYcZ')              # Characters are ordered lexicographically.
'X'
>>> min(1, 'a', 'A')
1

# max() works the same way min() does.
>>> print max([4, 2, 6]), max('aXbYcZ'), max(1, 'a', 'A')
6, c, a

# len() can be used on a string, list, set, dict, tuple, or other iterable.
>>> len([2, 7, 1])
3
>>> len('abcdef')
6
>>> len({1, 'a', 'a'})         # Duplicates are not added to sets.
2

# sum() can be used on iterables containing numbers, but not strings.
>>> my_list = [1, 2, 3]
>>> my_tuple = (4, 5, 6)
>>> my_set = {7, 8, 9}
>>> sum(my_list) + sum(my_tuple) + sum(my_set)
45
>>> sum([min(my_list), max(my_tuple), len(my_set)])
10
```

Problem 1. Write a function that accepts a list of numbers as input and returns a new list with the minimum, maximum, and average of the original list (in that order). Remember to use floating point division to calculate the average. Can you implement this function in a single line?

Namespaces and Mutability

Names

All objects created in Python resides in memory. These objects may be primitive data, data structures, functions, or any other sort of Python object. A *name* is a reference to a Python object. A *namespace* is a dictionary that maps names to Python objects.

```
# The number 4 is the object, 'number_of_students' is the name.
>>> number_of_sudents = 4

# The list is the object, and 'students' is the name.
>>> students = ["John", "Paul", "George", "Ringo"]

# Python statements defining a function form an object.
# The name for this function is 'add_numbers'.
>>> def add_numbers(a, b):
...     return a + b
...
```

A single equals sign assigns an object to a name. If a name is assigned to another name, that new name refers to the same object that the orignal name refers to (or a copy of it).

```
>>> students = ["John", "Paul", "George", "Ringo"]
>>> band_members = students
>>> print(band_members)
['John', 'Paul', 'George', 'Ringo']
```

Note

Many programming languages distinguish between *variables* and *pointers*. A pointer typically holds a memory location where the value of some other variable is stored. Pointer arithmetic and manipulation is delicate, occasionaly very useful, but often cumbersome.

Python names are essentially pointers, but typical pointer operations are done automatically, and objects in memory that have nothing pointing to them are automatically deleted. Understanding how Python handles memory access via names is important for implementing reference-based data structures, such as linked lists and trees.

Mutability

Python object types are either mutable or immutable. An *immutable* object cannot be altered once created, so assigning a new name to it creates a copy in memory. A *mutable* object's value may be changed, so assigning a new name to it does *not* create a copy. Therefore, if two names refer to the same mutable object, any changes to the object will be reflected in both names.

Warning

Mutability can be very useful in some settings, but it can also cause hard-to-find problems when a copy was intended to be made. For example, suppose we had a dictionary mapping items to their base prices, and we wanted make a new dictionary that accounts for a small sales tax.

```
>>> original = {"pencil": 1, "pen": 2, "paper": 3, "computer": 2000}
>>> tax_prices = original                    # Make a "copy" for processing.
>>> for item in tax_prices:
...     price = tax_prices[item]             # Get the original price.
...     tax_prices[item] += .07 * price      # Add on a 7% tax.
...
# Now the base prices have been updated to the total price.
>>> print(tax_prices)
{'pencil': 1.07, 'pen': 2.14, 'paper': 3.21, 'computer': 2140.0}

# However, dictionaries are mutable, so 'prices' and 'original' actually
# refer to the same object. The original base prices have now been lost.
>>> print(original)
{'pencil': 1.07, 'pen': 2.14, 'paper': 3.21, 'computer': 2140.0}
```

To avoid this, we create a copy of an object. Changes made to the copy will not change the original object. In this case, we replace the 2^{nd} line of the above code with the following:

```
>>> tax_prices = dict(original)
```

Then, after running the same procedure, the two dictionaries will be different.

Problem 2. Python has several methods that seem to change immutable objects. These methods actually work by making copies of objects. We can therefore determine which object types are mutable and which are immutable by using the equal sign and "changing" the objects.

```
>>> dict_1 = {1: 'x', 2: 'b'}          # Create a dictionary.
>>> dict_2 = dict_1                    # Assign it a new name.
>>> dict_2[1] = 'a'                    # Change the 'new' dictionary.
>>> dict_1 == dict_2                   # Compare the two names.
True
```

Since altering one name altered the other, we conclude that no copy has been made and that therefore Python dictionaries are mutable. If we repeat this process with a different type and the two names are different in the end, we will know that a copy had been made and the type is immutable.

Write a function in which you programmatically determine which object types are mutable and which are immutable, as in the example code above. Use the following operations to modify each of the given types.

numbers	num_1 += 1
strings	str_1 += 'a'
lists	list_1.append(1)
tuples	tuple_1 += (1,)
dictionaries	dict_1[1] = 'a'

Print a statement of your conclusions to the terminal.

Modules

In general, a Python *module* is a file containing Python code that is meant to be used in some other setting, and not necessarily run directly. The `import` statement loads the code from a specified Python file. That is, when a module containing some functions, classes, or other objects is imported, those functions, classes, and objects are made available for use.

In Python files, all `import` statements should occur at the top, below the header but before any other code. Thus we expand our example of typical Python file from the previous lab to the following:

```
# filename.py
"""This is the file header.
The header contains basic information about the file.
"""

import math
import numpy as np
from scipy import linalg as la
import matplotlib.pyplot as plt

def main():
    """This is a docstring. It provides information about the function."""
    print("Hello, world!")

if __name__ == "__main__":
    main()
```

The modules imported in this example are some of the most important modules used in Python for scientific computing. The NumPy, SciPy, and MatPlotLib modules will be presented in detail in subsequent labs.

Importing Syntax

There are several ways to use the `import` statement.

1. Using `import` alone makes the module available under the alias of its own name. For example, the `math` module has a function called `sqrt` that computes the square root of the input.

```
>>> import math                 # The name 'math' now gives access to
>>> math.sqrt(2)                # the built-in math module.
1.4142135623730951
```

2. It is often convenient to create an alias for an imported module using the `as` statement. Then the alias is added to the namespace, but the module name itself is not.

```
>>> import math as m            # The name 'm' gives access to the math
>>> m.sqrt(2)                   # module, but the name 'math' does not.
1.4142135623730951
>>> math.sqrt(2)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'math' is not defined
```

3. Often we only need access to a particular function or other object in a module. Use `from` to specifiy where the object is located, then import it individually.

```
>>> from math import sqrt       # The name 'sqrt' gives access to the
>>> sqrt(2)                     # square root function, but the rest of
1.4142135623730951              # the math module is unavailable.
>>> math.sin(2)
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
NameError: name 'math' is not defined
```

In each case, the far right word of the import statement is the name that is added to the namespace.

Running Vs. Importing

In addition to expanding the namespace of the current workspace, the `import` command also executes any freestanding code in the file to be imported. Carefully consider the following example file.

```
# example.py
"""Illustrate the difference between executing and importing a file."""

data = [1, 2, 3, 4]
print "Data: ", data

if __name__ == '__main__':
    print "This file was executed from the terminal or an interpreter."
else:
    print "This file was imported."
```

In IPython, note the difference between when the file is run and when it is imported.

```
# First, run the file. The freestanding code is executed, as well
# as everything under the 'if __name__ == "__main__"' clause.
In [1]: run example.py
Data: [1, 2, 3, 4]
This file was executed from the terminal or an interpreter.

# However, the file has not been imported.
In [2]: print(example.data)
NameError                                 Traceback (most recent call last)
<ipython-input-2-6bc16431f322> in <module>()
----> 1 print(example.data)

NameError: name 'example' is not defined

# Now when the file is imported, the freestanding code and everything
# under the 'else' clause is executed, and the names become available.
In [3]: import example
Data: [1, 2, 3, 4]
This file was imported.

In [4]: print(example.data)
[1, 2, 3, 4]
```

Beware of stray code that might be executed unintentionally when a file is imported.

Reloading and Testing

In IPython, importing provides a quick way to test code. However, if a module has been imported and the source code then changes (you test your code, discover and error, then fix it), the `reload` function must be used to access the changes. Using the `import` command again will **not** change the module.

Consider this example where we test a file containing a single function.

```
# example2.py
def sum_of_squares(x):
    """Return the sum of the squares of all integers less than or equal to x."""
    return sum([i**2 for i in range(x)])
```

In IPython, import the file and test the `sum_of_squares` function.

```
In [1]: import example2 as test

In [2]: test.sum_of_squares(3)
Out[2]: 5
```

Since $1^2 + 2^2 + 3^2 = 14$, not 5, something has gone wrong. We modify the file to correct the mistake:

```
# example2.py
def sum_of_squares(x):
    """Return the sum of the squares of all integers less than or equal to x."""
    return sum([i**2 for i in range(1,x+1)])    # Be sure to include x.
```

```
# Using import again doesn't change the loaded module, even though
# the source file was modified.
In [3]: import test

In [4]: test.sum_of_squares(3)
Out[4]: 5

# Using reload, however, updates the loaded module with the changes.
In [5]: reload(test)
Out[5]: <module 'example2' from 'example2.py'>

In [6]: test.sum_of_squares(3)
Out[6]: 14
```

Problem 3. Create a module called `calculator.py`. Write a function that returns the sum of two arguments, a function that returns the product of two arguments, and a function that returns the square root of a single argument. When the file is either run or imported, nothing should be executed.

In your solutions file, import the `calculator` module. Using only the functions defined in the module, write a new function that calculates the length of the hypotenuse of a right triangle given the lengths of the other two sides.

Python Standard Library

All Python distributions include a collection of modules for accomplishing a variety of common tasks, collectively called the *Python standard library*. A summary of the documentation for these modules, called the *docstring*, is stored in the `__doc__` attribute. Individual functions also have docstrings.

```
>>> import math
>>> print(math.__doc__)
This module is always available.  It provides access to the
mathematical functions defined by the C standard.

>>> print(math.cos.__doc__)
cos(x)

Return the cosine of x (measured in radians).
```

More extensive documentation is available via the `help` built-in function.

Using IPython's object introspection, we can learn about how to use the various modules and functions in the standard library very quickly. Similarly, we can list all of the functions included in a module with the `tab` key.

```
In [1]: import math

In [2]: math?
Type:        module
```

```
String form: <module 'math' from '/Users/ACME/anaconda/lib/python2.7/lib-dynload/↩
     math.so'>
File:        /Users/ACME/anaconda/lib/python2.7/lib-dynload/math.so
Docstring:
This module is always available.  It provides access to the
mathematical functions defined by the C standard.

# Type 'math.' then press tab to lists the available functions.
In [3]: math.
math.acos       math.atanh      math.e          math.factorial
math.hypot      math.log10      math.sin        math.acosh
math.ceil       math.erf        math.floor      math.isinf
math.log1p      math.sinh       math.asin       math.copysign
math.erfc       math.fmod       math.isnan      math.modf
math.sqrt       math.asinh      math.cos        math.exp
math.frexp      math.ldexp      math.pi         math.tan
math.atan       math.cosh       math.expm1      math.fsum
math.lgamma     math.pow        math.tanh       math.atan2
math.degrees    math.fabs       math.gamma      math.log
math.radians    math.trunc

In [4]: math.sqrt?
Type:        builtin_function_or_method
String form: <built-in function sqrt>
Docstring:
sqrt(x)

Return the square root of x.
```

The Sys Module

The `sys` (system) module includes methods for interacting with the Python interpreter. The module has a name `argv` that is a list of arguments passed to the interpreter when it runs Python scripts.

```
# echo.py
import sys

print(sys.argv)
```

If this script is run from the command line with additional arguments it will print them out. Note that command line arguments are parsed as strings.

```
$ python echo.py I am the walrus
['test_script', 'I', 'am', 'the', 'walrus']
```

This method can be used to point a Python script to a file to be analyzed. It can also be used to control a script's behavior, as in the following example.

```
# cipher.py
import sys

if len(sys.argv) < 3:
    print("Three arguments are required")
    sys.exit(1)             # Manually quit the program early.
```

```
arg = sys.argv[2]

if sys.argv[1] == '1':
    print("-".join(arg))

elif sys.argv[1] == '2':
    print(arg.upper())
```

Now provide command line arguments to specify the behavior of the script.

```
$ python cipher.py 1
Three arguments are required

$ python cipher.py 1 first
f-i-r-s-t

$ python cipher.py 2 second
SECOND
```

The Time Module

The `time` module includes functions for dealing with time. In particular, functions in `time` access the computer's system clock. This is useful for precisely measuring how long it takes for code to run.

The `time` module includes a function also called `time` that measures the number of seconds from a fixed starting point, the "Epoch". For most machines, this starting point will be January 1, 1970.

```
>>> import time

# time.time() returns the number of seconds from January 1, 1970 to the
# time of execution. This command gives a new time every time it is run.
>>> time.time()
1436832057.321525
```

In order to measure how long it takes to execute some Python code, a measurement is taken right before and right after it is run. Subtracting the first measurement from the second gives the amount of time in seconds that have passed.

```
def time_for_loop():
    # Time how long it takes to go through 10000 iterations using 'range'.
    start = time.time()             # Clock the starting time.
    for i in range(10000): pass     # Perform the operation.
    end = time.time()               # Clock the ending time.
    return end - start              # Report the difference.
```

The standard library also has a module called `timeit`. This library is built to time Python code and has more tools than `time`. In IPython, `timeit` can be used like a built-in function any time with the `%timeit` command.

```
# Time how long it takes to go through 10000 iterations using 'xrange'.
In [0]: %timeit for i in xrange(10000): pass
1000 loops, best of 3: 303   s  per loop
```

Problem 4. Download `matrix_multiply.py` and `matrices.npz`. The Python file `matrix_multiply.py` is a module that has three methods for multiplying two matrices together, called `method1`, `method2`, and `method3`. It also has a `load_matrices` method that returns two matrices from `matrices.npz`.

Modify your solutions function so that when it is run from a Python interpreter (but not when it is imported), the following is executed:

1. If no command line argument is given, print "No Input."
2. If anything other than "matrices.npz" is given, print "Incorrect Input."
3. If "matrices.npz" is given as a command line argument, load two matrices from `matrices.npz`. Time (separately) how long each method takes to multiply the two matrices together, then print the results to the terminal.

   (Hint: Read the code in `matrix_multiply.py`, especially the function docstrings, to determine how to use each function.)

Lab 2

Object-Oriented Programming

Lab Objective: *Python is a class-based language. A* class *is a blueprint for an object that binds together specified variables and routines. Creating and using custom classes is often a good way to clean and speed up code. In this lab we learn how to define and use Python classes. In subsequents labs, we will create customized classes often for use in algorithms.*

Python Classes

A Python *class* is a code block that defines a custom object and determines its behavior. The `class` command defines and names a new class. Other statements follow, indented below the class name, to determine the behavior of objects instantiated by the class.

A class needs a method called a *constructor* that is called whenever the class instantiates a new object. The constructor specifies the initial state of the object. In Python, a class's constructor is always named `__init__()`. For example, the following code defines a class for storing information about backpacks.

```
class Backpack(object):
    """A Backpack object class. Has a name and a list of contents.

    Attributes:
        name (str): the name of the backpack's owner.
        contents (list): the contents of the backpack.
    """
    def __init__(self, name):           # This function is the constructor.
        """Set the name and initialize an empty contents list.

        Inputs:
            name (str): the name of the backpack's owner.

        Returns:
            A Backpack object wth no contents.
        """
        self.name = name                # Initialize some attributes.
        self.contents = []
```

This `Backpack` class has two *attributes*: `color` and `contents`. Attributes are variables stored within the class object. In the body of the class definition, attributes are accessed via the name `self`. This name refers to the object internally once it has been created.

Instantiation

The `class` code block above only defines a blueprint for backpack objects. To create a backpack object, we "call" the class like a function. An object created by a class is called an *instance* of the class. It is also said that a class *instantiates* objects.

Classes may be imported in the same way as modules. In the following code, we import the `Backpack` class and instantiate a `Backpack` object.

```
# Import the Backpack class and instantiate an object called 'my_backpack'.
>>> from Packs import Backpack
>>> my_backpack = Backpack("Fred")

# Access the attributes of the object with a period.
>>> my_backpack.name
'Fred'
>>> my_backpack.contents
[]

# The object's attributes can be modified dynamically.
>>> my_backpack.name = "George"
>>> print(my_backpack.name)
George
```

> **Note**
>
> Many programming languages distinguish between *public* and *private* attributes. In Python, all attributes are automatically public. However, an attribute can be hidden from the user in IPython by starting the name with an underscore.

Methods

In addition to storing variables as attributes, classes can have functions attached to them. A function that belong to a specific class is called a *method*. In the following code, we define two simple methods in the `Backpack` class.

```
class Backpack(object):
    # ...
    def put(self, item):
        """Add 'item' to the backpack's list of contents."""
        self.contents.append(item)

    def take(self, item):
        """Remove 'item' from the backpack's list of contents."""
        self.contents.remove(item)
```

The first argument of each method must be `self`, to give the method access to the attributes and other methods of the class. The `self` argument is only included in the declaration of the class methods, **not** when calling the methods.

```
# Add some items to the backpack object.
>>> my_backpack.put("notebook")
>>> my_backpack.put("pencils")
>>> my_backpack.contents
['notebook', 'pencils']

# Remove an item from the backpack.
>>> my_backpack.take("pencils")
>>> my_backpack.contents
['notebook']
```

Problem 1. Expand the `Backpack` class to match the following specifications.

1. Modify the constructor so that it accepts a name, a color, and a maximum size, in that order. Make `max_size` a default argument with default value 5. Store each input as an attribute.
2. Modify the `put()` method to ensure that the backpack does not go over capacity. If the user tries to put in more than `max_size` items, print "No Room!" and do not add the item to the contents list.
3. Add a new method to the backpack called `dump()` that empties the contents of the backpack. This method should receive no arguments.

   (Hint: this method can be implemented in a single line.)

To ensure that your class works properly, consider writing a test function outside outside of the `Backpack` class that instantiates and analyzes a `Backpack` object. Your function may look something like this:

```
def test_backpack():
    backpack_1 = Backpack("Barry", "black")     # Instantiate the object.
    if backpack_1.max_size != 5:                # Test an attribute.
        print("Wrong default max_size!")
    for item in ["pencil", "pen", "paper", "computer"]:
        backpack_1.put(item)                    # Test a method.
    print(backpack.contents)
```

Inheritance

Inheritance is an object-oriented programming tool for code reuse and organization. To create a new class that is similar to one that already exists, it is often better to "inherit" the already existing methods and attributes rather than create a new class from scratch. This is done by including the existing class as an argument in the class definition (where the word `object` is in the definition of the `Backpack` class).

This creates a *class hierarchy*: a class that inherits from another class is called a *subclass*, and the class that a subclass inherits from is called a *superclass*.

For example, since a knapsack is a backpack (but not all backpacks are knapsacks), we create a special `Knapsack` subclass that inherits the structure and behaviors of the `Backpack` class, and adds some extra functionality.

```
# Inherit from the Backpack class in the class definition.
class Knapsack(Backpack):
    """A Knapsack object class. Inherits from the Backpack class.
    A knapsack is smaller than a backpack and can be tied closed.

    Attributes:
        name (str): the name of the knapsack's owner.
        color (str): the color of the knapsack.
        max_size (int): the maximum number of items that can fit
            in the knapsack.
        contents (list): the contents of the backpack.
        closed (bool): whether or not the knapsack is tied shut.
    """
    def __init__(self, name, color, max_size=3):
        """Use the Backpack constructor to initialize the name, color,
        and max_size attributes. A knapsack only holds 3 item by default
        instead of 5.

        Inputs:
            name (str): the name of the knapsack's owner.
            color (str): the color of the knapsack.
            max_size (int, opt): the maximum number of items that can
                be stored in the knapsack. Defaults to 3.

        Returns:
            A knapsack object with no contents.
        """
        Backpack.__init__(self, name, color, max_size)
        self.closed = True
```

A subclass may have new attributes and methods that are unavailable to the superclass, such as `self.closed` in the `Knapsack` class. If methods in the new class need to be changed, they are overwritten as is the case of the constructor in the `Knapsack` class. New methods can be included normally. As an example, we modify the `put()` and `take()` methods in `Knapsack` to check if the knapsack is shut.

```
class Knapsack(Backpack):
    # ...
    def put(self, item):
        """If the knapsack is untied, use the Backpack.put() method."""
        if self.closed:
            print "I'm closed!"
        else:
            Backpack.put(self, item)

    def take(self, item):
        """If the knapsack is untied, use the Backpack.take() method."""
        if self.closed:
            print "I'm closed!"
        else:
            Backpack.take(self, item)
```

Since `Knapsack` inherits from `Backpack`, a knapsack object is a backpack object. All methods defined in the `Backpack` class are available to instances of the `Knapsack` class. Thus, in this example, the `dump()` method is available even though it is not defined explicitly in the `Knapsack` class.

```
>>> from Packs import Knapsack
>>> my_knapsack = Knapsack("Brady", "brown")
>>> isinstance(my_knapsack, Backpack)       # A Knapsack is a Backpack.
True

# The put and take method now require the knapsack to be open.
>>> my_knapsack.put('compass')
I'm closed!

# Open the knapsack and put in some items.
>>> my_knapsack.closed = False
>>> my_knapsack.put("compass")
>>> my_knapsack.put("pocket knife")
>>> my_knapsack.contents
['compass', 'pocket knife']

# The dump method is inherited from the Backpack class, and
# can be used even though it is not defined in the Knapsack class.
>>> my_knapsack.dump()
>>> my_knapsack.contents
[]
```

Problem 2. Create a `Jetpack` class that inherits from the `Backpack` class.

1. Overwrite the constructor so that in addition to a name, color, and maximum size, it also accepts an amount of fuel. Store the fuel as an attribute. Also change the default value of `max_size` to 2, and set the default value of fuel to 10.
2. Add a `fly()` method that accepts an amount of fuel to be burned and decrements the fuel attribute by that amount. If the user tries to burn more fuel than remains, print "Not enough fuel!" and do not decrement the fuel.
3. Overwrite the `dump()` method so that both the contents and the fuel tank are emptied.

Magic Methods

In Python, a *magic method* is a special method used to make an object behave like a built-in data type.[1]Magic methods begins and ends with two underscores, like `__init__()`. Every Python object is automatically endowed with several magic

[1]A complete list of magic methods can be found at `https://docs.python.org/2/reference/datamodel.html`

methods, but they are hidden because they begin with an underscore (this is a way of hiding attributes or methods from the user; for example, try hiding the `closed` attribute in the `Knapsack` class by changing it to `_closed`).

We can see details on the `Backpack` object, including its magic methods, using IPython's object instrospection feature:

```
In [1]: import Packs

In [2]: b = Packs.Backpack("Oscar", "green")

In [3]: b.      # Press 'tab' to see standard methods and attributes.
b.color     b.contents  b.put       b.take

In [4]: b.put?
Signature: b.put(item)
Docstring: Add 'item' to the backpack's content list.
File:      ~/Downloads/Packs.py
Type:      instancemethod

In [5]: b.__    # Press 'tab' to see magic methods.
b.__add__           b.__getattribute__  b.__reduce_ex__
b.__class__         b.__hash__          b.__repr__
b.__delattr__       b.__init__          b.__setattr__
b.__dict__          b.__lt__            b.__sizeof__
b.__doc__           b.__module__        b.__str__
b.__eq__            b.__new__           b.__subclasshook__
b.__format__        b.__reduce__        b.__weakref__

In [6]: b?
Type:           Backpack
File:           ~/Downloads/Packs.py
Docstring:
A Backpack object class. Has a name and a list of contents.

Attributes:
    name (str): the name of the backpack's owner.
    contents (list): the contents of the backpack.
Init docstring:
Set the name and initialize an empty contents list.

Inputs:
    name (str): the name of the backpack's owner.

Returns:
    A backpack object wth no contents.
```

Note

In all of the preceding examples, the comments enclosed by sets of three double quotes are the object's *docstring*, stored as the `__doc__` attribute. A good docstring typically includes a summary of the class or function, information about the inputs and returns, and other notes. Writing detailed docstrings is critical so that others can utilize your code correctly.

Now, suppose we wanted to add two backpacks together. How should addition be defined for backpacks? A simple option is to add the number of contents. Then if backpack A has 3 items and backpack B has 5 items, $A + B$ should return 8.

```
class Backpack(object):
    # ...
    def __add__(self, other):
        """Add the number of contents of each Backpack."""
        return len(self.contents) + len(other.contents)
```

Using the + binary operator on two `Backpack` objects calls the `__add__()` method. The object on the left side of the + is passed in as `self` and the object on the right side of the + is passed in as `other`.

```
>>> from Packs import Backpack
>>> backpack1 = Backpack("Rose", "red")
>>> backpack2 = Backpack("Carly", "cyan")

# Put some items in the backpacks.
>>> backpack1.put("textbook")
>>> backpack2.put("water bottle")
>>> backpack2.put("snacks")

# Now add the backpacks like numbers.
>>> backpack1 + backpack2
3
```

Subtraction, multiplication, and division may be similary defined with the magic methods `__sub__()`, `__mul__()`, and `__div__()`. What each of these operations do, or should do, is up to the programmer and should be carefully documented.

Comparisons

Magic methods also allow for object comparisons. The `__lt__()` and `__gt__()` methods correspond to $<$ and $>$, respectively. Suppose we decide that one backpack should be considered "less" than another if it has fewer items in contents.

```
class Backpack(object)
    # ...
    def __lt__(self, other):
        """Compare two backpacks. If 'self' has fewer contents
        than 'other', return True. Otherwise, return False.
        """
        return len(self.contents) < len(other.contents)
```

Since $<$ is a boolean binary operator, the `__lt__()` method should returns either `True` or `False`, as should other comparison operators.

Now using the $<$ binary operator on two `Backpack` objects calls the `__lt__()` function. Again, the object on the left side of the operator is passed in as `self`, and the object on the right side is passed in as `other`.

Other comparison operators have corresponding magic methods as well. The `__le__()`, `__ge__()`, `__eq__()`, and `__ne__()` methods correspond to $<=$, $>=$, $==$, and $!=$, respectively.

```
>>> from Packs import Backpack
>>> backpack1 = Backpack("Maggy", "magenta")
>>> backpack2 = Backpack("Yolanda", "yellow")

>>> backpack1.put('book')
>>> backpack2.put('water bottle')
>>> backpack1 < backpack2
False

>>> backpack2.put('pencils')
>>> backpack1 < backpack2
True
```

Problem 3. Endow the `Backpack` class with two additional magic methods:

1. The `__eq__()` magic method is used to determine if two objects are equal, and is invoked by the `==` operator. Implement the `__eq__()` magic method for the `Backpack` class so that two `Backpack` objects are equal if and only if they have the same name, color, and contents. The two contents lists do not have to have their items in the same order to be considered equal.

2. The `__str__()` magic method is used to produce the string representation of an object. This method is invoked when an object is cast as a string with the `str` function, or when using the `print` statement. Implement the `__str__()` method in the `Backpack` class so that printing a `Backpack` object yields the following output:

   ```
   Owner:      <name>
   Color:      <color>
   Size:       <number of items in contents>
   Max Size:   <max_size>
   Contents:   [<item1>, <item2>, ...]
   ```

   (Hint: Use the '\t' tab and '\n' newline characters to help align output.)

Warning

Comparison operators are not automatically related. For example, for two backpacks A and B, if $A == B$ is `True`, it does not automatically imply that $A != B$ is `False`. Accordingly, when defining `__eq__()`, one should also define `__ne__()` so that the operators will behave as expected.

Problem 4. Create your own `ComplexNumber` class that supports the basic operations of complex numbers.

1. Complex numbers have a real and an imaginary part. The constructor should therefore accept two numbers. Store the first as `self.real` and the second as `self.imag`.

2. Implement a `conjugate` method that returns the object's complex conjugate (as a new `ComplexNumber` object). Recall that $\overline{a+bi} = a-bi$.

3. Add the following magic methods:

   (a) The `__abs__()` magic method determines the output of the built-in `abs` function (absolute value). Implement `__abs__()` so that it returns the magnitude of the complex number. Recall that $|a+bi| = \sqrt{a^2+b^2}$.

   (b) Implement `__lt__()` and `__gt__()` so that `ComplexNumber` objects can be compared by their magnitudes. That is, $(a+bi) < (c+di)$ if and only if $|a+bi| < |c+di|$, and so on.

   (c) Implement `__eq__()` and `__ne__()` so that two `ComplexNumber` objects are equal if and only if they have the same real and imaginary parts.

   (d) Implement `__add__()`, `__sub__()`, `__mul__()`, and `__div__()` appropriately. Each of these should return a new `ComplexNumber` object. (Hint: use the complex conjugate to implement division).

Compare your class to Python's built-in `complex` class. How can your class be improved to act more like complex numbers?

Part II

Data Structures and Graph Algorithms

Lab 3

Public Key Cryptography

Lab Objective: *Implement the RSA cryptosystem as an example of public key cryptography and learn to use Python's RSA implementation.*

A *public key cryptosystem* uses separate keys for encryption and decryption. If Alice wishes to send Bob a message, she encrypts it using Bob's *public key*, which is available to everyone. Then Bob decrypts it with his *private key*, which only he knows. As long as it is difficult to find the private key from the public key, this is a secure system. Public key cryptosystems are advantageous because for n people to have secure communications with each other, they only need exactly n public-private key pairs.

As an analogy, consider a locked box with two keys. One key, called the public key, is available to anyone and can lock the box but not unlock it. The other key, called the private key, is only available to one person and can unlock the box. To send a message to someone with the private key, the sender puts the message in the box and locks it. Since the only person that can unlock the box is the intended recipient, the message is safe until it arrives.

One of the oldest and most popular public key cryptosystems is called *RSA*.

The RSA system

Suppose Alice wants to receive secret messages using RSA. To do so, she first needs to generate a public key (for encryption) and a private key (for decryption). Alice does this by choosing two prime numbers, p and q, and setting $n = pq$. Then she sets $\phi(n) = (p-1)(q-1)$.[1] Alice chooses her encryption exponent to be some integer e that is relatively prime to $\phi(n)$. Then she uses the Extended Euclidean Algorithm to find d' such that $ed' + \phi(n)x' = 1$, and adds or subtracts multiples of $\phi(n)$ until $d = d' + k\phi(n)$ is between 0 and $\phi(n)$. Alice publishes her public key (e, n) for the world to see and keeps her private key (d, n) a secret (e for encryption, d for decryption).

[1] The function $\phi : \mathbb{Z} \to \mathbb{N}$ is called the *Euler phi function*. In general $\phi(n)$ is the number of positive integers less than n that are relatively prime to n.

Now imagine Bob wants to send Alice a message, which he represents as an integer $m < n$ (say, using the A=01, B=02 scheme). Bob computes

$$c \equiv m^e \pmod{n}$$

and sends the ciphertext c to Alice.

To decrypt the message, Alice computes

$$m' \equiv c^d \pmod{n}.$$

At this point, she uses the following theorem, which is easily proved with a combination of ring and group theory.

Theorem 3.1. *For any integers m and n such that n does not divide m, the following equality holds:*

$$m^{\phi(n)} \equiv 1 \pmod{n}.$$

Then Alice concludes that

$$m' = c^d \equiv m^{ed} = m^{\phi(n)(ek-x')+1} \equiv m \pmod{n}.$$

Now Alice can read Bob's message.

As an example, let us encrypt and decrypt the message SECRET=190503180520. First we define p, q, n, and $\phi(n)$.

```
>>> p = 1000003
>>> q = 608609
>>> n = p*q
>>> phi_n = (p-1)*(q-1)
```

Now we choose an encryption exponent $e = 1234567891$ and compute $d = 589492582555$ using the Extended Euclidean Algorithm.

```
>>> e = 1234567891
>>> d = 589492582555
```

Finally we are ready to encrypt the message. Note that $m < n$. If this were not the case, we would need to break up m into shorter pieces. Also, we force m to be a `long` integer so that the exponentiation operation does not overflow. The function `pow(a, b, n)` computes $a^b \pmod{n}$.

```
>>> m = long(190503180520)
>>> c = pow(m, e, n)
```

We decrypt the message by raising c to the d^{th} power, modulo n.

```
>>> m == pow(c, d, n)
True
```

Logistical considerations

The cryptosystem described is not particularly easy to use, since the message must be converted to an integer and back again by hand. The provided `rsa_tools` module contains some functions to fix this problem. The function `string_to_int()` turns any string into an integer (using a mapping more complicated than $A = 01, B = 02$), and the function `int_to_string()` changes it back again.

```
>>> import rsa_tools as r
>>> r.string_to_int('SECRET')
91556947314004
>>> r.int_to_string(91556947314004)
'SECRET'
```

At this point we have a problem, because the message 91556947314004 is larger than $n = 608610825827$. We can only use RSA to encrypt messages that are smaller than n. In fact, the function `string_size()` provided in `rsa_tools` tells us the maximum number of characters we can encrypt with this choice of n.

```
>>> r.string_size(608610825827)
4
```

The function `partition()`, also in the `rsa_tools` will break our message into pieces of length 4. We specify the "fill value" `x` that will be used to make all pieces the same length.

```
>>> r.partition('SECRET', 4, 'x')
['SECR', 'ETxx']
```

Now we can proceed to encrypt and decrypt the strings `'SECR'` and `'ETxx'` as before.

Problem 1. Write a class called `myRSA` that can generate keys, encrypt messages, and decrypt messages. Use methods from the provided `rsa_tools.py` module to convert strings to ints and vice-versa.

Write a method called `generate_keys` that accepts a pair of primes and an encryption exponent and sets the `public_key` and `_private_key` attributes. (Starting `_private_key` with an underscore hides the attribute from the user.) Also include an `encrypt` method that accepts a string and encrypts it, using `public_key` as the default encryption key. If a different public key is provided, then the message should be encrypted with the provided key. Finally, write a `decrypt` method that decrypts an encrypted message with `_private_key`.

In the next problem we will write a test function for this class.

Security of RSA

Suppose an enemy Eve wants to read the message that Bob sent to Alice. Like Bob, she has access to Alice's public key (e, n). Let's assume she has also intercepted the ciphertext c.

One way for Eve to read Bob's message is to directly find m such that $m^e \equiv c \pmod{n}$. Such a computation is known as taking a *discrete logarithm*. When n is very large, this computation is essentially impossible.

Another option is for Eve to compute (d, n) and then find c^d. However, computing d means computing $\phi(n)$. Computing $\phi(n)$ from n directly requires factoring n. When n has hundreds of digits, finding its factors with known algorithms can take years.

Thus, the security of RSA depends on selecting large primes so that n has many digits.

Exceptions

Every programming language has a formal way of handling errors. In Python, we `raise` and handle *Exceptions*. There are different kinds of exceptions, each with its appropriate usage. In Python, as in most languages, exceptions are organized into a class hierarchy. A complete list of Python exceptions can be found here. The following code displays some examples of common exceptions:

```
# A 'NameError' exception indicates that a nonexistant name was used.
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined

# An 'AttributeError' exception indicates that a nonexistant method or
# attribute was called on some object.
>>> x = list()
>>> x.fly()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'fly'

# A 'SyntaxError' exception indicates bad coding syntax.
>>> def myFunction(a, b)
  File "<stdin>", line 1
    def myFunction(a, b)
                       ^
SyntaxError: invalid syntax
```

Many exceptions, like the ones listed above, are due to coding mistakes and typos. Exceptions can also be used programmatically to indicate a problem to the user. To raise an exception, use the keyword `raise`, followed by the name of the exception. As soon as an exception is raised, the program stops running unless the exception is handled.

```
# raise a specific type of exception, with an error message included.
>>> x = 7
>>> if x > 5:
...     raise ValueError("'x' should not exceed 5.")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: 'x' should not exceed 5.
```

Handling Exceptions

To prevent an exception from stopping the program, it must be handled by putting the problematic lines of code in a `try` block. An `except` block then follows. If an exception is thrown, the compiler exits the `try` block and enters the `except` block.

```
# the 'try' block should hold any lines of code that might raise the exception.
>>> try:
...     raise Exception
...     print("No exception raised")
... # the 'except' block is entered into after the exception is thrown
... except:
...     print("Exception raised!")
...
Exception raised!

# Specific types of exceptions can also be caught explicitly.
>>> try:
...     bad = 0 / 0
... except ZeroDivisionError:
...     print("Divided by zero!")
...
Divided by zero!

# Finally, the exception object can be used to get information.
>>> try:
...     import magic
... except ImportError as e:
...     print("Sorry! " + e.message)
...
Sorry! No module named magic
```

Documentation on handling exceptions can be found here.

Problem 2. Write a `test_myRSA` function that accepts a message, two primes, and an encryption exponent. If the first argument is not a string, raise a `TypeError` with error message "message must be a string." If the final three arguments are not integers, raise a `TypeError` with error message "p, q, and e must be integers." (Hint: use the built-in `type` function.)

Create an instance of the `myRSA` class. Encrypt and decrypt the message using the keys generated by the integer arguments. If the original message and the decryption are not exacly the same, raise a `ValueError` with error message "decrypt(encrypt(message)) failed." Otherwise, return `True`.

Note

Custom exception are made by writing a class that inherits from some existing exception (the `Exception` class is the most generic). They can be very useful for handling problems that you expect, but that the compiler won't catch.

Making RSA Secure

Since the security of RSA depends on the use of large prime numbers for p and q, we need a fast way to find such numbers for RSA to be a practical cryptosystem. It is very slow to check that a large number is prime by finding its factors. In fact, it is the difficulty of factoring that makes RSA secure.

However, there exist fast algorithms that determine when a number is "probably prime." One such algorithm comes from a special case of Theorem **??** known as *Fermat's Little Theorem*:

$$a^{p-1} \equiv 1 \pmod{p}$$

for all primes p and integers a that are not divisible by p. Therefore, to check if a fixed number p is likely a prime, we can compute $a^{p-1} \pmod{p}$ for several values of a. If we ever get something different than 1, then we know that p is composite. The value of a that proved p was composite is called a *witness number*. In fact, the converse is true: if $a^{p-1} \equiv 1 \pmod{p}$ for every $a < p$ then p is prime.

With a few exceptions, if p is composite then more than half the integers in $[2, p-1]$ are witness numbers. Thus, if we run the above test on p just a few times and don't find a witness number, there is a high probability that p is prime. This test is called Fermat's test for primality.

In practice, a probabilistic algorithm like Fermat's test is used to identify numbers that have a high chance of being prime. Then, deterministic algorithms are used to verify the primality of a candidate.

Problem 3. Write a function called `is_prime` that implements Fermat's test for primality. Run the test at most five times, using integers randomly chosen from $[2, n-1]$ as possible witnesses. If a witness number is found, return the number of tries it took to find it. If no witness number is found after five tries, return 0. (Hint: use the built-in `pow` function so the exponentiation operation does not overflow.)

For most composite values of n, if you call `is_prime(n)` one hundred times, you should expect to get 0 at most five times. However, some composite numbers do not follow this rule. For example, how many times do you have to call `is_prime()` on 340561 to get an answer of 0?

PyCrypto: RSA in Python

PyCrypto is a professional implementation of RSA in Python. The package is called `Crypto`, and is included with most Python distributions. It can be downloaded separately here. Documentation for PyCrypto can be found here. This library contains many random number generators and encryption classes. Many programs use PyCrypto for their security needs.

Warning

Make certain that you are using the latest version of PyCrypto. Security software is updated often to fix security vulnerablilities and bugs. The current version of PyCrypto at the time of writing is version 2.6.1.

The RSA module in PyCrypto is located in the PublicKey module. The library allows us to explicitly construct a key, or generate a key automatically.

```
>>> from Crypto.PublicKey import RSA

# generate a 2048-bit RSA key
>>> keypair = RSA.generate(2048)

# Save the public key as a string for distribution.
>>> public_key = keypair.publickey()
>>> share_this = public_key.exportKey()
```

The `share_this` variable may be distributed as a public key.

```
>>> encrypter = RSA.importKey(share_this)
>>> ciphertext = encrypter.encrypt('abcde', 2048)
>>> ciphertext
('\xb1\t\xc6L\xd1u\x80.C@\x07$#\x8e\xca\x8a\x05*\xdf\x1f.N\xa9\x80
\x08\xcb*8~7\x1d\x87&&Ke\xd5\xed_H\xb9\xd0x\xac!\xf3\xa9\xdc\xbfy5
s\x92\x8d\x15\xf7vY\x99G\xb6\x03j[\xa3\xc6\x92a\n\x91\x08N\xbc\xe4
\xcd\xe2\x9b\xeb\x1eT\xe5\xef\x96\x83\x10\xb7\x0c\xd1\x9bK]z\xa5!\
xcc\xe0/\xd3L\xd4\xa9xx?*\xfb\xf6\xcaM8\xe6\x9d\xd4u\xbd\xda\xa8tf
X\x02\xfa\xff\x99\xff\xbb"\xd1\x87\'\xb9d\x1c\x1b\x9fcWd\x83\xea}\
x1f\xff\xd3\x9b0\x8e\x0f\x91\n\x16\r\r\xa5\xa5S\xafw\x07N^$\x9c]\x
ac\x96\xe3\x80l\xc9\xe5\xe4Od\xa5\t>\x16j\xa1\xb9\x9c5\xc0\xfe\xe3
\xe5i\xcd\xaf\xdc\xcad\x82\x10u\x91\xa0"\xcf\xe3A\x11\x82\x87\xb9\
xdf\xd7\x86\x87\xff\x11#-\xb5!Q)\xf7\xf1a\x94\xb3e?\xd0\x96W4\xb4\
xca\xcf\x18\xd1I\xcd\x1dl\xd7\xe0Y\xdf}F\x93\x92\xe1(d\xcc\xdc\xa7
 x\xc5`',)
>>> keypair.decrypt(ciphertext)
'abcde'
```

Problem 4. Write a new RSA class called `PyCrypto` that acts like the `myRSA` class, but implement it with methods from PyCrypto's `RSA` package.

Store both of your RSA keys in a single attribute called `_keypair`, and store a sharable string representation of the public key in the `public_key` attribute. Initialize `_keypair` and `public_key` in the constructor. The `encrypt` method accepts a string and encrypts it, using `_keypair` as the default encryption key. If the string representation of a different public key is provided, then the message should be encrypted with the provided key. The `decrypt` method decrypts an encrypted message using `_keypair`.

Try testing your class with a partner by exchanging public keys.

Warning

The following warning comes from the PyCrypto website:

"The export of cryptography software is (still) governed by arms control regulations in Canada, the United States, and elsewhere. The export or re-export of this software may be regulated by law in your country."

Lab 4

Data Structures I: Lists

Lab Objective: *Implement linked lists as an introduction to data structures.*

Introduction

Analyzing and manipulating data are essential skills in scientific computing. Storing, retrieving, and manipulating data take time. As a dataset grows, so does the amount of time it takes to access and manipulate it. The structure of how the data are stored determines how efficiently the data may be processed.

Data structures are specialized objects for organizing data. There are many kinds, each with specific strengths and weaknesses. For example, some data structures take a long time to build, but once built their data are quickly accessible. Others are built quickly, but are not as efficiently accessible. Different applications require different structures for optimal performance.

Nodes

Booleans, strings, floats, and integers are some of the built-in data types in Python. Most data in applications take one of these forms. However, as the size of a dataset increases, these types prove inefficient. Many data structures use *nodes* to overcome these inefficiencies.

If we thought of data as several types of objects that need to be stored in a warehouse, a node would be a standard size box that can hold all the different types of objects. Suppose a warehouse needs to store lamps of various sizes. Rather than trying to stack lamps of different shapes on top of each other efficiently, it is preferable to put them in the boxes of standard size. Then adding new boxes and retrieving stored ones becomes much easier.

A `Node` class is usually simple. In Python, the data in the Node is stored as an attribute. Other attributes may be added (or inherited) specific to a particular data structure. The data structure links the nodes together in a way that is efficient for its particular application.

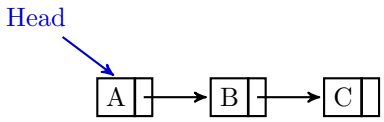

Figure 4.1: A Singly-linked List. The head attribute tracks the first node.

```
# LinkedLists.py
class Node(object):
    """A Node class for storing data."""
    def __init__(self, data):
        """Construct a new node that stores some data."""
        self.data = data
```

```
# Import the Node class from LinkedLists.py
>>> from LinkedLists import Node

# Create some nodes. Note that any data type may be stored.
>>> int_node = Node(1)
>>> str_node = Node('abc')
>>> list_node = Node([1,'abc'])

# Access a node's data.
>>> list_node.data
[1, 'abc']
```

Problem 1. Add extra functionality to the `Node` class by implementing the `__str__` magic method so that it returns a string representation of its data. Also implement the `__lt__`, `__eq__`, and `__gt__` comparison magic methods so that the data stored inside of two Nodes is compared. For example, a Node `x` is less than a Node `y` if and only if the data contained in `x` is less than the data contained in `y`.

Note

Often the data stored in a node is actually a *key* value. The key could be a pointer, a dictionary key, or the index of an array where the true desired information resides. However, for simplicity, in this and the following lab we store actual data in node objects, not references to data located elsewhere.

Linked Lists

A linked list is a data type that chains nodes together. Each node instance in a linked list stores a reference to the next link in the chain. A linked list class also stores a reference to the first node in the chain, called the `head`. See Figure 4.1.

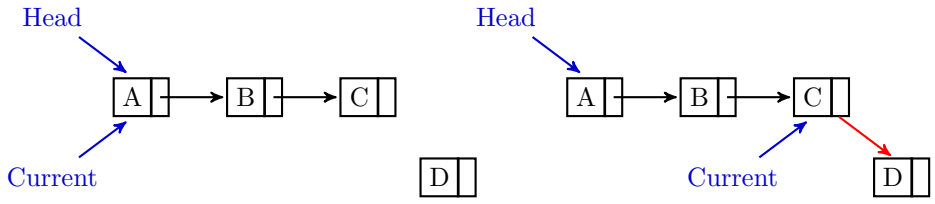

Figure 4.2: To add a new node to the end of the list, create a new node and start at the head. Iterate to the last node in the list and connect it to the new node.

```
class LinkedListNode(Node):
    """A Node class for linked lists. Inherits from the 'Node' class.
    Contains a reference to the next node in the list.
    """
    def __init__(self, data):
        """Construct a Node and initialize an
        attribute for the next node in the list.
        """
        Node.__init__(self, data)
        self.next = None
```

A basic implementation of a linked list will have a constructor and a method for adding new nodes to the end of the list. To get to the end of the list, start at the `head` of the list. Then traverse the list by going from node to node until the end is reached. Then, set the `next` attribute of the last node to be the new node. This is done in the following class. See Figure 4.2 for an illustration.

```
class LinkedList(object):
    """Singly-linked list data structure class.
    The first node in the list is referenced to by 'head'.
    """
    def __init__(self):
        """Create a new empty linked list. Create the head
        attribute and set it to None since the list is empty.
        """
        self.head = None

    def add(self, data):
        """Add a new Node containing 'data' to the end of the list."""
        new_node = LinkedListNode(data)
        if self.head is None:
            # If the list is empty, point the head attribute to the new node.
            self.head = new_node
        else:
            # If the list is not empty, traverse the list
            # and place the new_node at the end.
            current_node = self.head
            while current_node.next is not None:
                # Move current_node to the next node if it is nonempty.
                current_node = current_node.next
            # current_node now points to the last node in the list.
            current_node.next = new_node
```

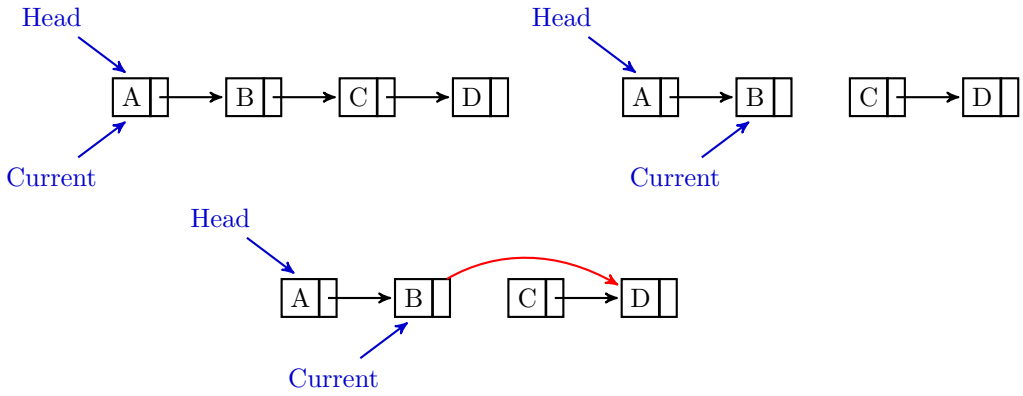

Figure 4.3: By disconnecting B from C, C and D are deleted since nothing points to C. To keep D from being lost, connect B to D first. Then only C is deleted.

Problem 2. Write the `__str__` method for the `LinkedList` class so that when a `LinkedList` instance is printed, its output matches that of a Python list.

In addition to adding new nodes to the end of a list, it is also useful to remove nodes and insert new nodes at specified locations. To delete a node, all references to the node must be removed. Then Python will automatically delete the object, since there is no way for the user to access it. Naïvely, this might be done by finding the previous node to the one being removed, and setting its `next` attribute to none.

```
class LinkedList(object):
    def __init__(self):
        # ...
    def add(self, data):
        # ...

    def remove(self, data):
        """Remove the Node containing 'data'."""

        # Find the node whose next node contains data
        current_node = self.head
        while current_node.next.data != data:
            current_node = current_node.next

        # Remove the next reference to the target node
        current_node.next = None
```

Since the only reference to the node that is deleted is the previous node's next attribute, this will delete the node. However, since the only reference to the next node came from the deleted node, it also will deleted. This will continue to the end of the list. Thus, deleting one node in this manner deletes the remainder of the list. This can be remedied by pointing the previous node's next attribute to the node after the deleted node. Then there will be no reference to the removed node and it will be deleted. See Figure 4.3 for an illustration.

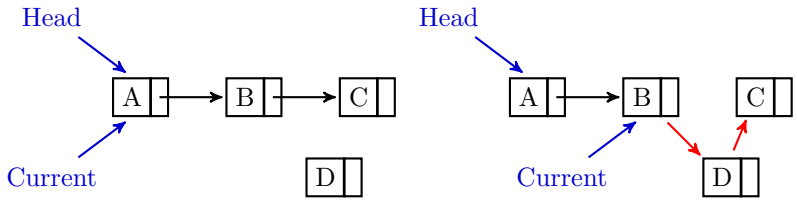

Figure 4.4: To insert D before C, find the node before C and set the connections.

```
class LinkedList(object):
    def __init__(self):
        # ...
    def add(self, data):
        # ...

    def remove(self, data):
        """Remove the Node containing 'data'."""
        # First, check if the head is the node to be removed. If so, set the
        # new head to be the first node after the old head. This removes
        # the only reference to the old head, so it is then deleted.
        if self.head.data == data:
            self.head = self.head.next
        else:
            current_node = self.head
            # Move current_node through the list until it points
            # to the node that precedes the target node.
            while current_node.next.data != data:
                current_node = current_node.next

            # Point current_node to the node after the target node.
            new_next_node = current_node.next.next
            current_node.next = new_next_node
```

Warning

Python keeps track of the variables in use and automatically deletes a variable if there is no access to it. In many other languages, leaving a reference to an object without explicitly deleting it could cause a serious memory leak. See here for more information on python's auto-cleanup system.

Problem 3. Though the above code works to remove specified nodes, it is not quite complete. Modify the `remove` method to account for possible errors: if the list is empty, or if the target node is not in the list, raise a `ValueError` with error message "`<data>` is not in the list."

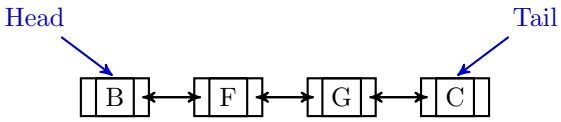

Figure 4.5: A Doubly-linked List. The tail attribute tracks the last node.

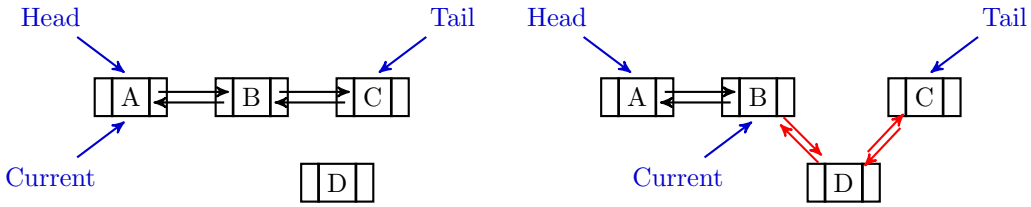

Figure 4.6: Insertion for Doubly-linked Lists.

Problem 4. Add an `insert` method to the `LinkedList` class that inserts a new node before the first node in the list that contains the data specified by the user. Accept data for the new node (`data`) and data for the node before which the new node will be inserted (`place`). If the list is empty, or there is no node containing `place` in the list, raise a `ValueError` with error message "`<place>` is not in the list."

See Figure 4.4 for an illustration of the `insert` method. Note that since `insert` inserts a node before a specified node that is already in the list, it is not possible to `insert` at the end of the list or to an empty list.

Doubly-Linked Lists

A doubly-linked list is a linked list where each node keeps track the node that precedes it as well as the node that follows. The end of the list is also typically kept track of with a `tail` attribute. See Figure 4.5 for an illustration.

```
# LinkedLists.py

class DoublyLinkedListNode(LinkedListNode):
    """A Node class for doubly-linked lists. Inherits from the 'Node' class.
    Contains references to the next and previous nodes in the list.
    """
    def __init__(self,data):
        """Set the next and prev attributes."""
        Node.__init__(self,data)
        self.next = None
        self.prev = None
```

All of the methods for linked lists can be implemented for doubly-linked lists. See Figures 4.6 and 4.7 for illustrations of the insert and remove methods.

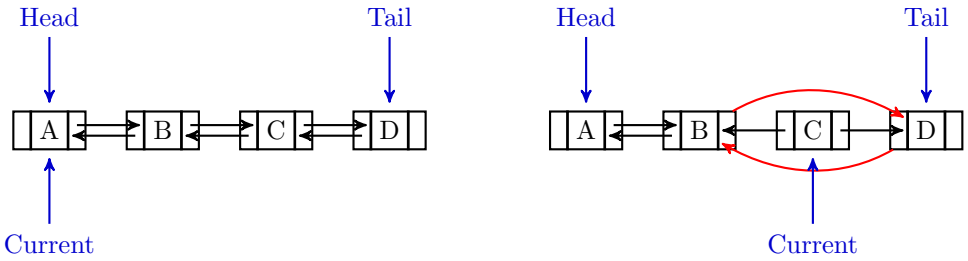

Figure 4.7: Removal for Doubly-linked Lists.

Problem 5. Implement a `DoublyLinkedList` class that inherits from `LinkedList` and uses `DoublyLinkedListNode` instances to build the list. Add a `tail` attribute that keeps track of the node at the end of the list. Overwrite `add`, `remove`, and `insert`. Raise the same exceptions as before.

Problem 6. Implement a sorted linked list. This data structure adds new nodes strategically so that the data is always kept in order. Inherit this class from `DoublyLinkedList`, and override the `add` and `insert` methods. When a new node is added, traverse the list until the data in the next node is greater than or equal to the data for the new node. Then insert the new node, thereby preserving the ordering. Also override the `insert` method with the following:

```
def insert(self, *args):
    raise NotImplementedError("insert() has been disabled for this class."↩
        )
```

This effectively disables `insert` for the `SortedLinkedList` class and prevents the user from accidentally inserting a node in a location that would disrupt the ordering. The `*args` argument allows `insert` to receive any number of arguments without raising a `TypeError` exception.

To test this data structure, import the provided `WordList` module. This includes a method called `create_word_list` that reads each line of text from a file and returns it as a randomly-ordered list of strings. Write a function called `sort_words` that sorts the list generated by `create_word_list` by adding them to a `SortedLinkedList` object. Then return the object.

Warning

`English.txt`, the default source file for `create_word_list`, contains over 58,000 English words. Sorting the entire data set should take about 15 minutes. Test your data structure on small data sets first.

NOTE

Python has many quick sorting methods. Even on the seemingly large data set of over 58,000 words used in the preceding problem, the `sort` method for Python lists is almost instantaneous. In the next lab we turn our attention to *trees*, special kinds of linked lists that allow for much quicker sorting and data retrieval.

Restricted-Access Lists

Often it is wise to restrict the user's access to the some of the data within a structure. The three most common and basic restricted-access structures are stacks, queues, and deques. Each structure restricts the user's access differently, making them ideal for different situations. These structures will reappear in many future labs and applications.

- A *stack* is a *Last In, First Out* structure (LIFO): only the last item that was inserted can be accessed. A stack is like a pile of plates: the last plate put on the pile is the first one to be taken off. Stacks usually have two main methods: `push`, to insert new data, and `pop`, to remove and return the last piece of data inserted.

- A *queue* (pronounced "cue") is a *First In, First Out* structure (FIFO): new nodes are added to the end of the queue, but an existing node can only be removed or accessed if it is at the front of the queue. A queue is like a line at the bank: the person at the front of the line is served next, while newcomers add themselves to the back of the line. Queues also usually have a `push` and a `pop` method, but `push` inserts data to the end of the queue while `pop` removes and returns the data at the front of the queue (`push` and `pop` for queues are sometimes called `enqueue` and `dequeue`, respectively).

- A *deque* (pronounced "deck") is a double-ended queue: data can be inserted or removed from either end, but data in the middle is inaccessible. A deque is like a deck of cards, where only the top and bottom cards are readily accessible. A deque has two methods for insertion and two for removal, usually called `append`, `appendleft`, `pop`, and `popleft`.

  In practice, a deque can also be used as a stack or a queue. If we restrict our usage to `append` and `pop` (or to `appendleft` and `popleft`), we effectively have a stack. Similarly, if we are restricted to `append` and `popleft` (or to `appendleft` and `pop`), we effectively have a queue.

The `collections` module in the Python standard library has a `deque` object, implemented as a doubly-linked list. This is an excellent object to use in practice

instead of a Python `list` when speed is of the essence and data only needs to be accessed from the ends of the list.

Problem 7. (Optional) Write `Stack`, `Queue`, and `Deque` classes.

The `Deque` class should inherit from the `DoublyLinkedList` class. Use inheritance to implement the `append`, `appendleft`, `pop`, and `popleft` methods as described in the preceding section. The `append` and `appendleft` methods should accept a single parameter (the data to be added) and return nothing, while the `pop` and `popleft` methods should accept no parameters and return a single value (the data removed). Disable all other methods to restrict data access.

The `Stack` and `Queue` classes should inherit from the `Deque` class. Add a `push` method and overload the `pop` method in each class to match the behaviors described in the preceding section. Disable any other methods.

Lab 5

Data Structures II: Trees

Lab Objective: *Implement tree data structures and understand their relative strengths and weaknesses.*

Recursion

Recursion is an important problem solving technique in computer programming. A recursive function is one that calls itself. When the function is executed, it continues calling itself until it reaches a specified base case. Then the function exits without calling itself again, and each previous function call is resolved. As a simple example, suppose we want to recursively sum all positive integers from 1 to some integer n.

```
def recursive_sum(n):
    """Calculate the sum of all positive integers in [1, n] recursively."""

    # Typically the base case comes first. There are no positive integers less
    # than 1, so if 'n' is 1 we stop the recursion and return 1 (since the sum of
    # all integers in [1, 1] is 1).
    if n == 1:
        return 1

    # If the base case hasn't been reached, the function recurses by calling
    # itself on the next smallest integer and adding 'n'.
    else:
        return n + recursive_sum(n-1)
```

The computer calculates `recursive_sum(5)` with a sequence of function calls.

```
# To find the recursive_sum(5), we need to calculate recursive_sum(4).
# But to find recursive_sum(4), we need to calculate recursive_sum(3).
# This continues until the base case is reached.

recursive_sum(5)        # return 5 + recursive_sum(4)
    recursive_sum(4)        # return 4 + recursive_sum(3)
        recursive_sum(3)        # return 3 + recursive_sum(2)
            recursive_sum(2)        # return 2 + recursive_sum(1)
                recursive_sum(1)        # Base case: return 1.
```

Now that we've reached a base case, we can unwind the recursion. Reading from bottom to top, we substitute the values that result from each function call.

```
recursive_sum(5)        # 5 + 10 = 15
    recursive_sum(4)        # 4 + 6 = 10
        recursive_sum(3)        # 3 + 3 = 6
            recursive_sum(2)        # 2 + 1 = 3
                recursive_sum(1)        # Base case: return 1.
```

So `recursive_sum(5)` returns 15 (which is correct, since $1 + 2 + 3 + 4 + 5 = 15$). Many problems that can be solved by iterative methods can also be solved (often more efficiently) with a recursive approach. Compare, for example, the following two methods for calculating the n^{th} Fibonacci number.

```
def iterative_fib(n):
    """Calculate the nth Fibonacci number iteratively."""
    fibonacci = list()      # Initialize an empty list.
    fibonacci.append(0)     # append 0 (the 0th Fibonacci number).
    fibonacci.append(1)     # append 1 (the 1st Fibonacci number).
    for i in range(1, n):
        # Starting at the third entry, calculate the next number
        # by adding the last two entries in the list.
        fibonacci.append(fibonacci[-1] + fibonacci[-2])
    # When the entire list has been loaded, return the nth entry.
    return fibonacci[n]

def recursive_fib(n):
    """Calculate the nth Fibonacci number recursively."""
    # The base cases are the first two Fibonacci numbers.
    if n == 0:              # Base case 1: the 0th Fibonacci number is 0.
        return 0
    elif n == 1:            # Base case 2: the 1st Fibonacci number is 1.
        return 1
    # If we haven't reached a base case, the function recurses by calling
    # itself on the previous two Fibonacci numbers.
    else:
        return recursive_fib(n-1) + recursive_fib(n-2)
```

This time, the sequence of function calls is slightly more complicated because `recursive_fib` calls itself twice until a base case is reached.

```
recursive_fib(5)        # The original call makes two additional calls:
    recursive_fib(4)        # this one...
        recursive_fib(3)
            recursive_fib(2)
                recursive_fib(1)        # Base case 2: return 1
                recursive_fib(0)        # Base case 1: return 0
            recursive_fib(1)        # Base case 2: return 1
        recursive_fib(2)
            recursive_fib(1)        # Base case 2: return 1
            recursive_fib(0)        # Base case 1: return 0
    recursive_fib(3)        # ...and this one.
        recursive_fib(2)
            recursive_fib(1)        # Base case 2: return 1
            recursive_fib(0)        # Base case 1: return 0
        recursive_fib(1)        # Base case 2: return 1
```

The sum of all of the base case results, from top to bottom, is $1+0+1+1+0+1+0+1=5$, so `recursive_fib(5)` returns 5 (correctly). The key to recursion is understanding the base cases correctly and making correct recursive calls.

Problem 1. Rewrite the following iterative function for finding data in a linked list using recursion. Use the basic linked list object from the previous lab to test the function (copy `LinkedLists.py` into your folder and import the `LinkedList` class).

```
# solutions.py

def iterative_search(linkedlist, data):
    current = linkedlist.head
    while current is not None:
        if current.data == data:
            return current
        current = current.next
    raise ValueError(str(data) + " is not in the list.")
```

(Hint: define an inner function to perform the actual recursion)

Warning

It is not always better to rewrite an iterative method recursively. In Python, a function may only call itself 999 times. On the 1000^{th} call, a `RuntimeError` is raised to prevent a stack overflow. Whether or not recursion is appropriate depends on the problem to be solved and the algorithm to solve it.

Trees

A *tree* data structure is a specialized linked list. Trees are more difficult to build than standard linked lists, but they are almost always more efficient. While the computational complexity of finding a node in a linked list is $O(n)$, a well-built, balanced tree will find a node with a complexity of $O(\log n)$. Some types of trees can be constructed quickly but take longer to retrieve data, while others take more time to build and less time to retrieve data.

The first node in a tree is called the *root*. The root node points to other nodes, called children. Each child node in turn points to its children. This continues on each branch until its end is reached. A node with no children is called a *leaf node*.

Mathematically, a tree is a directed graph with no cycles. Therefore a linked lists qualifies as a tree, albeit a boring one. The head node is the root node, and it has one child node. That child node also has one child node, which in turn has one child. This continues until the end of the list, with the last node as the only leaf node.

Other kinds of trees may be more complicated. See Figure 5.1.

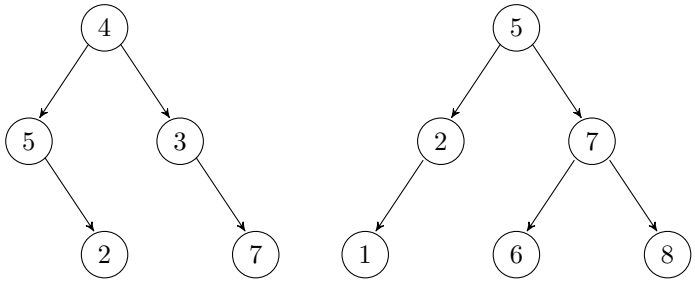

Figure 5.1: Both of these graphs are trees, but only the tree on the right is a binary search tree. How could the graph on the left be altered to make it a BST?

Binary Search Trees

A *binary search tree* (BST) data structure is a tree that allows each node to have up to two children, usually called `left` and `right`. The left child of a node contains data that is less than its parent node's data. The right child's data is greater.

The tree on the right in Figure 5.1 is an example of a of binary search tree. In practice, binary search tree nodes have attributes that keep track of their data, their children, and (in doubly-linked trees) their parent.

```
# Trees.py

class BSTNode(object):
    """A Node class for Binary Search Trees. Contains some data, a
    reference to the parent node, and references to two child nodes.
    """
    def __init__(self, data):
        """Construct a new node and set the data attribute. The other
        attributes will be set when the node is added to a tree.
        """
        self.data = data
        self.prev = None        # A reference to this node's parent node.
        self.left = None        # This node's data will be less than self.data
        self.right = None       # This node's data will be greater than self.data

    def __str__(self):
        """String representation: the data contained in the node."""
        return str(self.data)
```

The actual binary search tree class has an attribute pointing to its root.

```
# Trees.py

class BST(object):
    """Binary Search Tree data structure class.
    The first node is referenced to by 'root'.
    """
    def __init__(self):
        """Initialize the root attribute."""
        self.root = None
```

Finding in a Binary Search Tree

Many tree algorithms are best understood and implemented using recursion. For instance, finding a node in a binary search tree can be done recursively. Starting at the root, we check if the data we are looking for matches the current node. If it does not, then if the data is less than the current node's data we search again on the left child. If the data is greater, we search on the right child. This process continues until the data is found or, if the data is not in the tree, an empty child is searched. Carefully read the following code; similar techniques will be used for subsequent methods.

```
class BST(object):
    # ...

    def find(self, data):
        """Return the node containing 'data'. If there is no such node in the
        tree, raise a ValueError with error message "<data> is not in the tree."
        """
        # First, check to see if the tree is empty.
        if self.root is None:
            raise ValueError(str(data) + " is not in the tree.")

        # Define a recursive function to traverse the tree.
        def _step(current, item):
            """Recursively step through the tree until the node containing
            'item' is found. If there is no such node, raise a Value Error.
            """
            if current is None:                 # Base case 1: dead end.
                raise ValueError(str(data) + " is not in the tree.")
            if item == current.data:            # Base case 2: the data matches.
                return current
            if item < current.data:             # Step to the left
                return _step(current.left,item)
            else:                               # Step to the right
                return _step(current.right,item)

        # Start the recursion on the root of the tree.
        return _step(self.root, data)
```

> **Note**
>
> Conceptually, each node of a BST partitions the data of its subtree into two halves: the data that is less than the parent, and the data that is greater. We can extend this concept to multiple dimensions (see the K-D Trees lab).

Inserting to a Binary Search Tree

To insert new data into a binary search tree, a leaf node is added at the correct location. First, we find the node that should be the parent of the new node. We find the parent recursively, using a similar approach to the `find` method. Once the correct parent is found, the new node is added as the left or right child of the parent. See Figure 5.2 for an example.

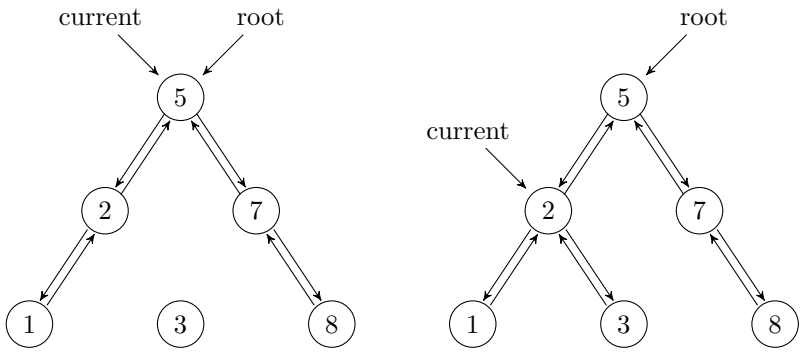

Figure 5.2: To insert a node containing 3 to the BST on the left, start 'current' at the root and recurse down the tree until it points to the node that should be 3's parent. Connect that parent to the child, then the child to its new parent.

Problem 2. Implement the `insert` method in the `BST` class. To accomplish this, write a recursive `_find_parent` method within `insert`. Find the correct parent, then determine whether the new node will be its left or right child. Then double-link the parent and the new child. Be sure to consider the special case of inserting to an empty tree. To test your tree, use (but do not modify) the provided `BST.__str__` method.

Do not allow for duplicates in the tree: if the user executes `insert(x)` and there is already a node in the tree containing `x`, raise a ValueError.

Removing from a Binary Search Tree

Deleting nodes from a binary search tree is more difficult than searching and inserting. Insertion always creates a new leaf node, but removal may delete any kind of node. This leads to several different cases to consider.

Removing a leaf node

In Python, an object is automatically deleted if there are no references to it. Call the node to be removed the *target node*, and suppose it has no children. To remove the target, find the target's parent, then delete the parent's reference to the target. Then there are no references to the target, so the target node is deleted. Since the target is a leaf node, removing it does not affect the rest of the tree structure.

Removing a node with one child

If the target node has one or more children, we must be careful not to delete the children when the target is removed. Simply removing the target as if it were a leaf node would delete the entire subtree originating from the target.

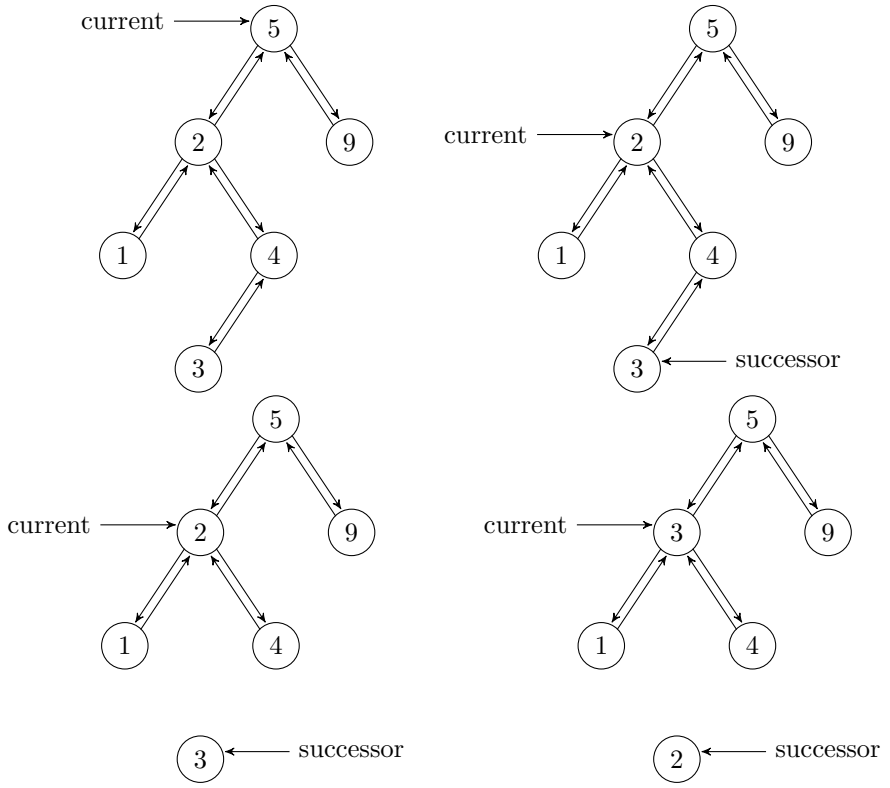

Figure 5.3: To remove the node containing 2 from the top left BST, start 'current' at the root (upper left) and recurse down the tree until it points to the target. Then locate the in-order successor (upper right) and delete it (lower left, recording its data. Finally, swap the data in the target with the data that was in the successor (lower right).

To avoid deleting all of the target's descendents, we point the target's parent to an appropriate successor. If the target has only one child, then that child is the successor. Connect the target's parent to the successor, and double-link by setting the successor's parent to be the target node's parent. Then, since the target has no references pointing to it, it is deleted. The target's successor, however, is pointed to by the target's parent, and so it remains in the tree.

Removing a node with two children

Removal is more complicated if the target node has two children. To delete this kind of node, first we find it's immediate in-order successor. This successor is the node with the smallest data that is larger than the target's data. It may be found by moving to the right child of the target (so that it's value is greater than the target's value), and then to the left for as long as possible (so that it has the smallest such value). Note that because of how the successor is chosen, any in-order successor can only have at most one child.

Once the successor is found, the target and its successor must switch places

in the graph, and then the target must be removed. This can be done by simply switching the data values for the target and its successor. Then the node with the target data has at most one child, and may be deleted accordingly. If the successor was chosen appropriately, then the binary search tree structure and ordering will be maintained once the deletion is finished.

The easiest way to implement this is to use recursion. First, because the successor has at most one child, we may recursivley remove the successor node by calling `remove` on the successor's data. Then set the data stored in the target node as the successor's data. See Figure 5.3.

Removing the root node

In each of the above cases, we must also consider the subcase where the target is the root node. If the root has no children, resetting the root or calling the constructor will do. If the root has one child, that child becomes the new root of the tree. If the root has two children, the successor becomes the new root of the tree.

Problem 3. Implement the `remove` method in the `BST` class. If the tree is empty, or if the target node is not in the tree, raise a `ValueError`.

Make sure to cover all possible cases:

1. The tree is empty (`ValueError`)
2. The target is not in the tree (`ValueError`)
3. The target is the root node:
   (a) the root is a leaf node, hence the only node in the tree
   (b) the root has one child
   (c) the root has two children
4. The target is in the tree but is not the root:
   (a) the target is a leaf node
   (b) the target has one child
   (c) the target has two children

Test your solution thoroughly with each case.

(Hint: use `find` wherever appropriate.)

There are many variations on the binary search tree, each with their own particular advantages and disadvantages. The reader is encouraged to research B-trees, red-black trees, and splay trees. We conclude with a discussion of one of the most famous BST variants: the AVL.

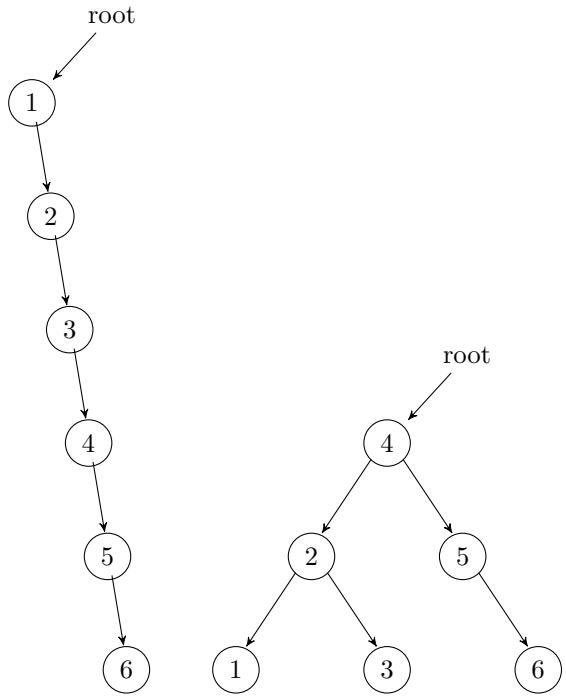

Figure 5.4: On the left, the BST resulting from inserting 1, 2, 3, 4, 5, and 6, in that order. On the right, the balanced AVL tree equivalent.

AVL Trees

Binary search trees are a good way of organizing data so that it is quickly accessible. However, pathologies may arise when certain data sets are stored using a basic binary serch tree. This is best demonstrated by inserting ordered data into a binary search tree. Since the data is already ordered, each node will only have one child, and we essentially end up with a linked list.

```
# Sequentially adding ordered integers destroys the efficiency of a BST.
>>> unbalanced_tree = BST()
>>> for i in xrange(10):
...     unbalanced_tree.insert(i)
...
# The tree is perfectly flat, so it loses its search efficiency.
>>> print(unbalanced_tree)
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
```

Problems also arise when one branch of the tree becomes much longer than the others, leading to longer search times.

An AVL tree (named after Georgy Adelson-Velsky and Evgenii Landis) is a tree that prevents any one branch from getting longer than the others. It accomplishes this by recursively "balancing" the branches as nodes are added. See Figure 5.4. The AVL's balancing algorithm is beyond the scope of this project, but details and exercises on the algorithm can be found in Chapter 2 of the Volume II text.

```
>>> balanced_tree = AVL()
>>> for i in xrange(10):
...     balanced_tree.insert(i)
...
# The AVL tree is balanced, so it retains (and optimizes) its search efficiency.
>>> print(balanced_tree)
[3]
[1, 7]
[0, 2, 5, 8]
[4, 6, 9]
```

Problem 4. Compare the speed of building and searching the different data structures we have implemented so far. Visualize the results by creating a plot with two subplots: one for build times, and one for search times. Repeat the following for n varying from 500 to 5000 at intervals of 500:

Use the `create_word_list` function in the provided `WordList` module to generate a list of n randomized words from the file `English.txt`. Time (separately) how long it takes to load a `LinkedList`, a `BST`, and an `AVL` with the data set. Use `add` to load the `LinkedList` and `insert` to load the trees. Then choose 5 random words from the data set, and time how long it takes to find each word in each object. Use the `iterative_search` function from problem 1 to search the `LinkedList` and `find` to search the trees. Calculate the average search time for each object.

In the first subplot, plot the number of words in each data set against the time it took to build each object. In the second subplot, plot the number of words in each data set against the average time it took to search each object. Your plot should look similar to Figure 5.5.

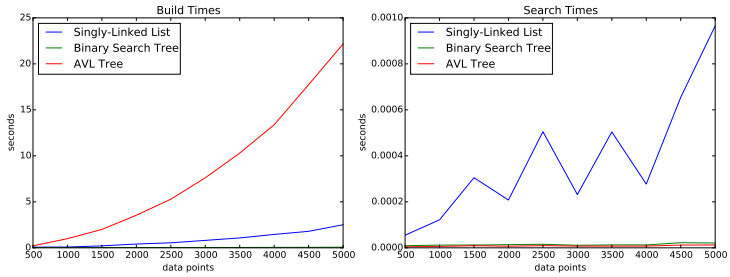

Figure 5.5: Note that the `BST` has the fastest build times, but the `AVL` has the fastest search times. How would the graph change if the data were sorted to begin with?

Lab 6

Nearest Neighbor Search

Lab Objective: *Nearest neighbor search is an optimization problem that arises in applications such as computer vision, pattern recognition, internet marketing, and data compression. In this lab we implement a K-D tree to solve the problem efficiently, then learn use scipy's K-D tree in sklearn to implement a handwriting recognition algorithm.*

The Nearest Neighbor Search Problem

Suppose you move into a new city with several post offices. Since your time is valuable, you wish to know which post office is closest to your home. This is called the nearest neighbor search problem, and it has many applications.

In general, suppose that X is a collection of data, called a *training set*. Let y be any point (often called the *target* point) in the same space as the data in X. The nearest neighbor search problem determines the point in X that is closest to y. For example, in the post office problem the set X could be addresses or latitude and longtitude data for each post office in the city. Then y would be the data that represents your new home, and the task is to find the closest post office in X to y.

Problem 1. In order to solve the nearest neighbor search problem we need a way to measure distance. A function that measures distance between two points is called a *metric*. The euclidean metric measures the distance between two points in $\mathbb{R}^n$ by

$$d(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} = \|x-y\|_2$$

Write a function that accepts two $1 \times k$ numpy arrays and returns the euclidean distance between them. Raise a `ValueError` if the vectors don't have the same dimension.

Consider again the post office example. One way to find out which post office is closest is to drive from home to each post office, measure the mileage, and then choose the post office that is the closest. This is called an exhaustive search. More precisely, we measure the distance of y to each point in X, and choose the point with the smallest distance. This method is ineffecient however, and only feasible if the training set is very small.

Problem 2. Write a function that solves the nearest neighbor search problem by exhaustively checking all of the distances between a given point and each point in a data set. The function should take in a set of data points (as an $m \times k$ numpy array) and a single target point (as a $1 \times k$ numpy array). Return the point in the training set that is closest to the target point and its distance from the target.

The complexity of this algorithm is $O(mk)$, where k is the number of dimensions and m is the number of data points.

K-D Trees

A *k-d tree* is a special kind of binary search tree[1] for high dimensional data (i.e., more dimensions than 1). While a binary search tree excludes regions of the number line from a search until the search point is found, a k-d tree works on regions of $\mathbb{R}^k$. So long as the data in the tree meets certain dimensionality requirements, similar efficiency gains may be made.

Recall that to search for a point in a binary search tree, we start at the root, and if the point we are searching for is less than the root we proceed down the left branch of the tree. If it is larger we proceed down the right branch. By doing this, we exclude a region of the number line (and therefore the subtree in the opposite direction) from our search. By eliminating this region from consideration, we have far fewer points to search and the efficiency of our search is greatly increased.

Like a binary search tree, a k-d tree starts with a root node with a depth, or level, of 0. At the i^{th} level, the nodes to the left of a parent have a lower value in the i^{th} dimension. Nodes to the right have a greater value in the i^{th} dimension. At the next level, we do the same for the next dimension. For example, consider data in $\mathbb{R}^3$. The root node partitions the data according to the first dimension. The children of the root partition according to the second dimension, and the grandchildren along the third. See Figure 6.1 for an example in $\mathbb{R}^2$.

As with any other data structure, the first task is to construct a node class to store data. A `KDTNode` is similar to a `BSTNode`, except it has another attribute called `axis`. The `axis` attribute tells us which dimension of $\mathbb{R}^k$ to split on.

[1] This lab is a sequel to the Data Structures II lab, and should not be attempted until the reader has successfully implemented a binary search tree in Python.

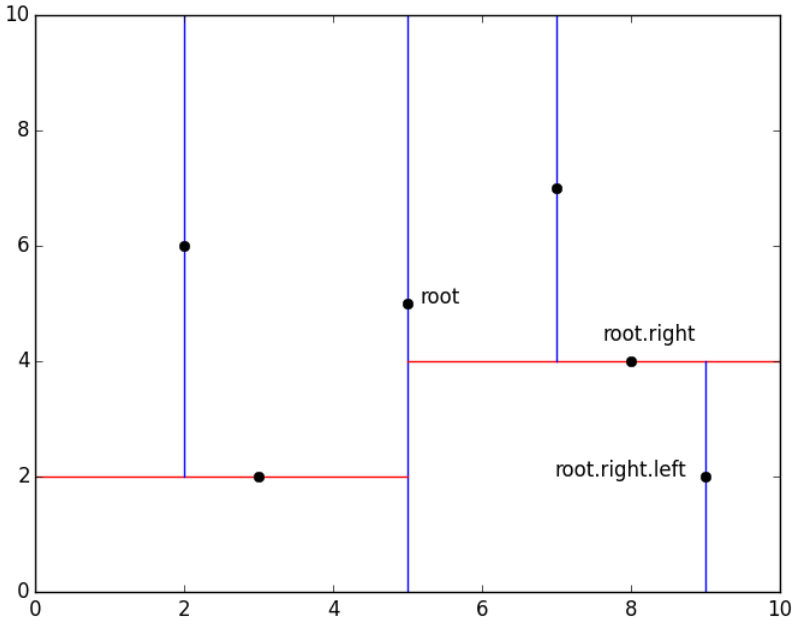

Figure 6.1: A regular binary search tree partitions $\mathbb{R}$, but a k-d tree partitions $\mathbb{R}^k$. The above graph illustrates the partition for a k-d tree loaded with the points (5, 5), (8, 4), (3, 2), (7, 7), (2, 6), and (9, 2), in that order. To find the point (9, 2), we start at the root. Since the x-coordinate of (9, 2) is greater than the x-coordinate of (5, 5), we move into the region to the right of the middle blue line, thus excluding all points (x, y) with $x < 5$. Next we compare (9, 2) to the root's right child, (8, 4). Since the y-coordinate of (9, 2) is less than the y-coordinate of (8, 4), we move below the red line on the right, thus excluding all points (x, y) with $y > 4$. We have now found (9, 2), since it is the left child of (8, 4).

```
from Trees import BSTNode

class KDTNode(BSTNode):
    """Node class for K-D Trees. Inherits from BSTNode.
    Attributes:
        left (KDTNode): a reference to this node's left child.
        right (KDTNode): a reference to this node's right child.
        parent (KDTNode): a reference to this node's parent node.
        data (ndarray): a coordinate in k-dimensional space.
        axis (int): the dimension to make comparisons on.
    """
    def __init__(self, data):
        """Construct a K-D Tree node containing 'data'. The left, right,
        and prev attributes are set in the constructor of BSTNode.
        """
        BSTNode.__init__(self, data)
        self.axis  = 0
```

Problem 3. Import the `BSTNode` class from `Trees.py`Write a `KDTNode` class that inherits from `BSTNode`.

1. Modify the constructor so that a `KDTNode` can only hold a numpy array (`np.ndarray`). If any other data type is given, raise a `TypeError`.
2. Write the `__sub__` magic method so that `x - y` returns the euclidean distance between the data in node `x` and the data in node `y`.
3. Write the `__eq__` magic method so that `x == y` is `True` if and only if `x` and `y` have the same data (Hint: `np.allclose()`)
4. Finally, write the `__lt__` and `__gt__` magic methods so that the $<$ and $>$ operators compare the i^{th} entry of the data, where i is the `axis` attribute of the node on the *right side* of the operator. For example,

```
>>> x = KDTNode(np.array([1,2]))
>>> y = KDTNode(np.array([3,1]))
>>> y.axis = 0          # Compare the '0th' entry of the data
>>> x < y               # True, since 1 < 3
True
>>> x > y
False

>>> y.axis = 1          # Compare the '1st' entry of the data
>>> x < y               # False, since 2 > 1
False
>>> x > y
True
```

Now we construct the k-d tree class. For an optimal k-d tree, the data needs to be inserted in a very particular order. However, inserting at random still usually produces a good tree. Here we simply insert the data in the order that it is given.

The major difference between a k-d tree and a binary search tree is how the data is compared at each depth level. This is simplified by using the magic methods in the `KDTnode` class. Though we don't need to use a `find` method in solving the nearest neighbor problem, we provide the k-d tree version of `find` as an instructive example.

In the `find` method, every comparison in the recursive `_step` function compares the data of `target` and `current` based on the `axis` attribute of `current`, since it is on the right-hand side of the expression. This way if each existing node in the tree has the correct `axis`, the correct comparisons are made as we descend the tree.

```
from Trees import BST

class KDT(BST):
    """A k-dimensional binary search tree object.
    Used to solve the nearest neighbor problem efficiently.
    Attributes:
        root (KDTNode): the root node of the tree. Like all other
            nodes in the tree, the root houses data as a numpy array.
        k (int): the dimension of the tree (the 'k' of the k-d tree).
```

```
    """

    def find(self, data):
        """Return the node containing 'data'.

        Raises:
            ValueError: there is node containing 'data' in the tree,
                or the tree is empty.
        """

        # First check that the tree is not empty.
        if self.root is None:
            raise ValueError(str(data) + " is not in the tree.")

        # Define a recursive function to traverse the tree.
        def _step(current, target):
            """Recursively approach the target node."""

            if current is None:             # Base case: target not found.
                return current
            if current == other:            # Base case: target found!
                return current
            if target < current:            # Recursively search to the left.
                return _step(current.left, target)
            else:                           # Recursively search to the right.
                return _step(current.right, target)

        # Create a new node to use the KDTNode comparison operators.
        n = KDTNode(data)

        # Call the recursive function, starting at the root
        found = _step(self.root, n)
        if found is None:                   # Report the data was not found.
            raise ValueError(str(data) + " is not in the tree.")
        return found                        # Othewise, return the target node.
```

Problem 4. Finish implementing the `KDT` class.

1. Override the `insert` method. To insert a new node, find the correct insertion point by recursively descending through the tree as in the `find` method (see figure 6.2 for an example).

   The `axis` attribute of the new node will be one more than that axis of the parent node. If the last dimension of the data has been reached, start over at the first dimension.

2. To solve the nearest neighbor search problem, we need only create the k-d tree once. Then we can use it multiple times with different target points. To prevent the user from altering the tree, disable the `remove` method. Raise a `NotImplementedError` if the method is called, and allow it to receive any number of arguments.

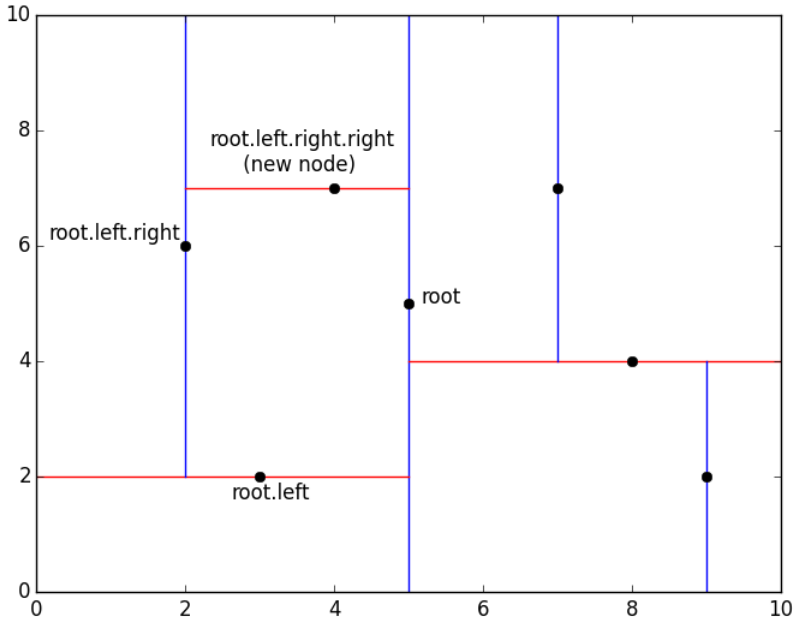

Figure 6.2: To insert the point (4, 7) into the k-d tree of figure 6.1, we find the node that will be the new node's parent. Start at the root, (5, 5). Since the x-coordinate of (4, 7) is less than the x-coordinate of (5, 5), we move into the region to the left of the middle blue line, to the root's left child, (3, 2). The y-coordinate of (4, 7) is greater than the y-coordinate of (3, 2), so we move above the red line on the left, to the right child (2, 6). Now we return to comparing the x-coordinates, and since $4 > 2$ and (2, 6) has no right child, we install (4, 7) as the right child of (2, 6).

Using a k-d tree to solve the nearest neighbor search problem requires some care. At first glance, it appears that a procedure similar to `find` or `insert` will immediately yield the result. However, this is not always the case (see Figure 6.3).

To correctly find the nearest neighbor we will keep track of the target point, the current search node, current best point, and current minimum distance. Start at the root node. Then the current search node and current best point will be root, and the current minimum distance will be the euclidean distance from `root` to `target`. We then proceed recursively as in the `search` method. As we find better current best points, we update the appropriate variables accordingly.

Once we have reached the bottom of the tree, we will have a good guess for the nearest neighbor. However, we are not guaranteed to have arrived at the correct point. One way to ensure that we have arrived at the correct point is to draw a hypersphere with a radius of the current minimum distance around the candidate nearest neighbor. If this hypersphere does not intersect any of the hyperplanes that split the k-d tree, then we know that have found a best point.

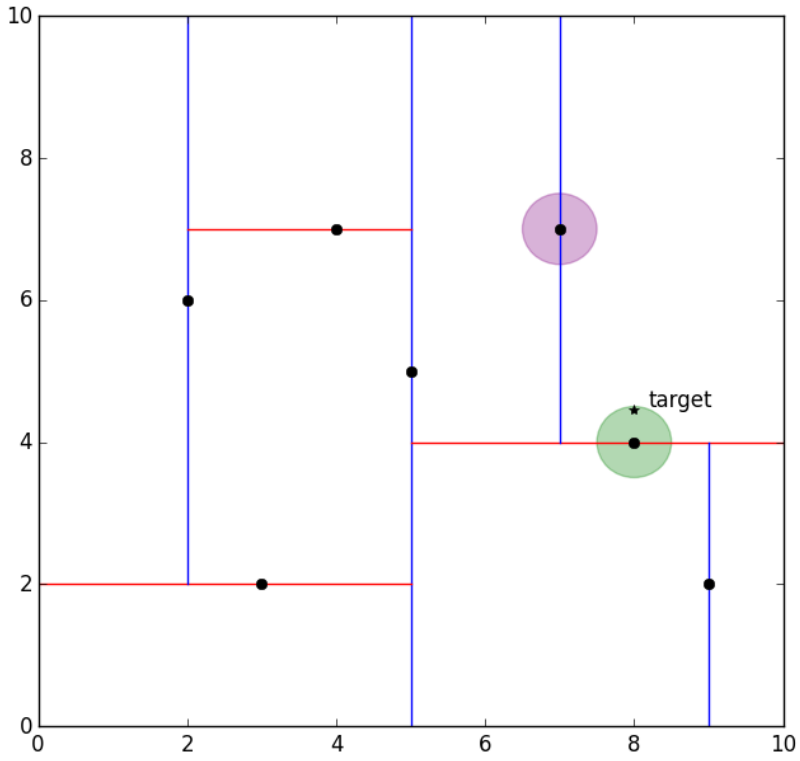

Figure 6.3: Suppose we want to find the point in the k-d tree of figure 6.2 that is closest to (8, 4.5). First we record the distance from the root to the target as the current minimum distance (about 3.04), then travel down the tree to the right. The right child, (8, 4), is only .5 units away from the target (the green circle), so we update the minimum distance. Since (8, 4) is not a leaf in the tree, we could continue down to the left child, (7, 7). However, this leaf node is much further from the target (the purple circle). To ensure that we terminate the algorithm correctly we check to see if the hypersphere of radius .5 around the current node (the green circle) intersects with any other hyperplanes. Since it does not, we stop descending down the tree and conclude (correctly) that (8, 4) is the nearest neighbor.

While we can not easily draw the correct hypersphere, there is an equivalent procedure that has a straightforward implementation in python. Before we finally decide to descend in one direction, we add the minimum distance to the i^{th} entry of the target point's data, where i is the `axis` of the candidate nearest neighbor. If this sum is greater that the i^{th} entry of the current search node, then the hypersphere would necessarily intersect one of the hyperplanes drawn by the tree (why?).

We summarize the algorithm below.

Algorithm 6.1 k-d tree nearest neighbor search

```
1: procedure KDTSearch(current, target, neighbor, distance)
2:     if current is None then
3:         return neighbor, distance (Base Case)
4:     index = current.axis
5:     d = euclidean_distance
6:     if d(current,target) < distance then
7:         neighbor = current
8:         distance = d(current,target)
9:     if target.data[index] < current.data[index] then
10:        neighbor, distance = KDTSearch(current.left, target,
11:        neighbor, distance)
12:        if target.data[index] + distance >= current.data[index] then
13:            neighbor, distance = KDTSearch(current.right, target,
14:            neighbor, distance)
15:    else
16:        neighbor, distance = KDTSearch(current.right, target,
17:        neighbor, distance)
18:        if target.data[index] - distance <= current.data[index] then
19:            neighbor, distance = KDTSearch(current.left, target,
20:            neighbor, distance)
21:    return neighbor, distance
```

Problem 5. Use Algorithm 6.1 to write a function that solves the nearest neighbor search problem by searching through a k-d tree (your `KDT` object). The function should take in a data set and a single target point. Return the nearest neighbor in the data set and the distance from the nearest neighbor to the target point, as in Problem 2.

To test your function, use Scipy's built-in `KDTree` object. This structure behaves like the `KDT` class, but its operations are heavily optimized. To solve the nearest neighbor problem, initialize the tree with data, then 'query' the tree with the target point. The `query` method returns a tuple of the minimun distance and the index of the nearest neighbor in the data.

```
>>> from scipy.spatial import KDTree

# Initialize the tree with data (in this example we use random data).
>>> data = np.random.random((100,5))
>>> target = np.random.random(5)
>>> tree = KDTree(data)

# Query the tree and print the minimum distance.
>>> min_distance, index = tree.query(target)
>>> print(min_distance)
0.309671532426
```

```
# Print the nearest neighbor by indexing into the tree's data.
>>> print(tree.data[index])
[ 0.68001084  0.02021068  0.70421171  0.57488834  0.50492779]
```

Application: Handwriting Recognition

Classification

Suppose that we are given a training set of data as well as a set of *labels* that describe each datum in the training set. For example, suppose that we had a training set containing the incomes and debt levels of N individuals. Along with this data, we have a set N labels that state whether the individual has filed for bankruptcy. The classification problem is to try and assign the correct label to an unlabelled data point.

k-Nearest Neighbors

In our previous work, we used a k-d tree to find the nearest neighbor of a target point. A more general problem is to find the k nearest neighbors to a point (using some metric to measure "distance" between data points). In classification, we find the k nearest neighbors, we let each neighbor "vote" to decide what label to give the new point. For example, consider the bankrupty case in the previous section. If we find the 10 nearest neighbors to a new individual, and 8 of them went bankrupt, then we would predict that the individual will also go bankrupt. On the other hand, if 7 of the nearest neighbors had not filed for bankruptcy, we would predict that the individual was at low risk for bankruptcy.

The Handwriting Recognition Problem

The problem of recognizing handwritten letters and numbers with a computer has many applications. A computer image may be thought of a vector in $\mathbb{R}^n$, where n is the number of pixels in the image and the entries represent how bright each pixel is. If two people write the same number, we would expect the vectors representing a scanned image of those number to be close in the euclidean metric. This insight means that given a training set of scanned images along with correct labels, we may confidently infer the label of a new scanned image.

scipy.sklearn

The `sklearn` module in `scipy` contains powerful tools for solving the nearest neighbor problem. To start nearest neighbors classification, we import the neighbors module from sklearn. This module has a class for setting up a k-nearest neighbors classifier.

```
# Import the neighbors module
>>> from sklearn import neighbors
```

```
# Create an instance of a k-nearest neighbors classifier.
# 'n_neighbors' determines how many neighbors to give votes to.
# 'weights' may be 'uniform' or 'distance.'  The 'distance' option
#     gives nearer neighbors more weight.
# 'p=2' instructs the class to use the euclidean metric.
>>> nbrs = neighbors.KNeighborsClassifier(n_neighbors=8, weights='distance', p=2)
```

The `nbrs` object has two useful methods for classification. The first, `fit`, will take arrays of data (the training set) and labels and put them into a k-d tree. This can then be used to find k-nearest neighbors, much like the `KDT` class that we implemented previously.

```
# 'points' is some numpy array of data
# 'labels' is a numpy array of labels describing the data in points.
>>> nbrs.fit(points, labels)
```

The second method, `predict`, will do a k-nearest neighbor search on the k-d tree and use the result to attach a label to unlabelled points.

```
# 'testpoints' is an array of unlabeled points.
# Perform the search and calculate the accuracy of the classification.
>>> prediction = nbrs.predict(testpoints)
>>> np.average(prediction/testlabels)
```

Problem 6. The United States Postal Service has made a collection of labelled hand written digits available to the public, provided in `PostalData.npz`. We will use this data for k-nearest neighbor classification. This data set may be loaded by using the following command:

```
labels, points, testlabels, testpoints = np.load('PostalData.npz').items()
```

This contains a training set and a test set. The first entry of each array is a name, so `points[1]` and `labels[1]` are the actual points and labels to use. Each point is an image that is represented by a flattened 28×28 matrix of pixels. The corresponding label indicates which number was written.

Classify the testpoints with `n_neighbors` as 1, 4 or 10, and with `weights` as `'uniform'` or `'distance'`. For each trial print a report indicating how your classifier performs in terms of percentage of correct classifications. Which combination gives the most correct classifications? (Hint: define an inner function that takes in `n_neighbors` and `weights` as arguments calls the neighbors functions appropriately)

A similar classification process is used by the United States Postal Service to automatically determine the zip code to send a letter to.

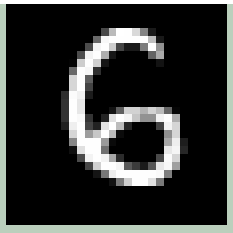

Figure 6.4: An example of the number 6 taken from the data set

Lab 7

Breadth-First Search and the Kevin Bacon Problem

Lab Objective: *Graph theory has many practical applications. A graph may represent a complex system or network, and analyzing the graph often reveals critical information about the network. In this lab we learn to store graphs as adjacency dictionaries, implement a breadth-first search to identify the shortest path between two nodes, then use the NetworkX package to solve the so-called "Kevin Bacon problem."*

Graphs in Python

Computers can represent mathematical graphs using various kinds of data structures. In previous labs, we stored graphs as trees and linked lists. For non-tree graphs, perhaps the most common data structure is an adjacency matrix, where each row of the matrix corresponds to a node in the graph and the entries of the row indicate which other nodes the current node is connected to. For more on adjacency matrices, see chapter 2 of the Volume II text.

Another comon graph data structure is an *adjacency dictionary*, a Python dictionary with a key for each node in the graph. The dictionary values are lists containing the nodes that are connected to the key. For example, the following is an adjacency dictionary for the graph in Figure 7.1:

```
# Python dictionaries are used to store adjacency dictionaries.
>>> adjacency_dictionary = {'A':['B', 'C', 'D', 'E'], 'B':['A', 'C'],
'C':['B', 'A', 'D'], 'D':['A', 'C'], 'E':['A']}

# The nodes are stored as the dictionary keys.
>>> print(adjacency_dictionary.keys())
['A', 'C', 'B', 'E', 'D']

# The values are the nodes that the key is connected to.
>>> print(adjacency_dictionary[A])
>>> ['B', 'C', 'D', 'E']            # A in connected to B, C, D, and E.
```

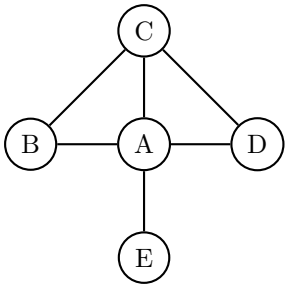

Figure 7.1: A simple graph with five vertices.

Problem 1. Implement the `__str__` method in the provided `Graph` class. Print each node in the graph in sorted order, followed by a sorted list of the neighboring nodes separated by semicolons.

(Hint: consider using the `join` method for strings.)

```
>>> my_dictionary = {'A':['C', 'B'], 'C':['A'], 'B':['A']}
>>> graph = Graph(my_dictionary)
>>> print(graph)
A: B; C
B: A
C: A
```

Breadth-First Search

Many graph theory problems are solved by finding the shortest path between two nodes in the graph. To find the shortest path, we need a way to strategically search the graph. Two of the most common searches are depth-first search (DFS) and breadth-first search (BFS). In this lab, we focus on BFS. For details on DFS, see the chapter 2 of the Volume II text.

A BFS traverses a graph as follows: begin at a starting node. If the starting node is not the target node, explore each of the starting node's neighbors. If none of the neighbors are the target, explore the neighbors of the starting node's neighbors. If none of those neighbors are the target, explore each of their neighbors. Continue the process until the target is found.

As an example, we will do a programmatic BFS on the graph in Figure 7.1 one step at a time. Suppose that we start at node C and we are searching for node E.

```
# Start at node C
>>> start = 'C'
>>> current = start

# The current node is not the target, so check its neighbors
>>> adjacency_dictionary[current]
['B', 'A', 'D']
```

```
# None of these are E, so go to the first neighbor, B
>>> current = adjacency_dictionary[start][0]
>>> adjacency_dictionary[current]
['A', 'C']

# None of these are E either, so move to the next neighbor
# of the starting node, which is A
>>> current = adjacency_dictionary[start][1]
>>> adjacency_dictionary[current]
['B', 'C', 'D', 'E']

# The last entry of this list is our target node, and the search terminates.
```

You may have noticed that some problems in the previous approach that would arise in a more complicated graph. For example, what prevents us from entering a cycle? How can we algorithmically determine which nodes to visit as we explore deeper into the graph?

Implementing Breadth-First Search

We solve these problems using a queue. Recall that a *queue* is a type of limited-access list. Data is inserted to the back of the queue, but removed from the front. Refer to the end of the Data Structures I lab for more details.

A queue is helpful in a BFS to keep track of the order in which we will visit the nodes. At each level of the search, we add the neighbors of the current node to the queue. The `collections` module in the Python standard library has a `deque` object that we will use as our queue.

```
# Import the deque object and start at node C
>>> from collections import deque
>>> current = 'C'

# The current node is not the target, so add its neighbors to the queue.
>>> visit_queue = deque()
>>> for neighbor in adjacency_dictionary[current]:
...     visit_queue.append(neighbor)
...
>>> print(visit_queue)
deque(['B', 'A', 'D'])

# Move to the next node by removing from the front of the queue.
>>> current = visit_queue.popleft()
>>> print(current)
B
>>> print(visit_queue)
deque(['A', 'D'])

# This isn't the node we're looking for, but we may want to explore its
# neighbors later. They should be explored after the other neighbors
# of the first node, so add them to the end of the queue.
>>> for neighbor in adjacency_dictionary[current]:
...     visit_queue.append(neighbor)
...
>>> print(visit_queue)
deque(['A', 'D', 'A', 'C'])
```

We have arrived at a new problem. The nodes A and C were added to the queue to be visited, even though C has already been visited and A is next in line. We can prevent these nodes from being added to the queue again by creating a `set` of nodes to contain all nodes that have already been visited, or that are marked to be visited. Checking set membership is very fast, so this additional data strutcure has minimal impact on the program's speed (and is faster than checking the deque).

In addition, we keep a `list` of the nodes that have actually been visited to track the order of the search. By checking the `set` at each step of the algorithm, the previous problems are avoided.

```
>>> current = 'C'
>>> marked = set()
>>> visited = list()
>>> visit_queue = deque()

# Visit the start node C.
>>> visited.append(current)
>>> marked.add(current)

# Add the neighbors of C to the queue.
>>> for neighbor in adjacency_dictionary[current]:
...     visit_queue.append(neighbor)
...     # Since each neighbor will be visited, add them to marked as well.
...     marked.add(neighbor)
...
# Move to the next node by removing from the front of the queue.
>>> current = visit_queue.popleft()
>>> print(current)
B
>>> print(visit_queue)
['A', 'D']

# Visit B. Since it isn't the target, add B's neighbors to the queue.
>>> visited.append(current)
>>> for neighbor in adjacency_dictionary[current]:
...     visit_queue.append(neighbor)
...     marked.add(neighbor)
...

# Since C is visited and A is in marked, the queue is unchanged.
>>> print(visit_queue)
deque(['A', 'D'])
```

Problem 2. Implement the `traverse` method in the `Graph` class using a BFS. Start from a specified node and proceed until all nodes in the graph have been visited. Return the list of visited nodes. If the starting node is not in the adjacency dictionary, raise a `ValueError`.

Problem 3. (Optional) Create a new method called `DFS` in the `Graph` class that mimics the `traverse` method, but uses a DFS instead of a BFS.

(Hint: this can be done by changing a single line of the BFS code.)

Shortest Path

In a BFS, as few neighborhoods are explored as possible before finding the target. Therefore, the path taken to get to the target must be the shortest path.

Examine again the graph in Figure 7.1. The shortest path from C to E is start at C, go to A, and end at E. During a BFS, A is visited because it is one of C's neighbors, and E is visited because it is one of A's neighbors. If we knew programmatically that A was the node that visited E, and that C was the node that visited A, we could retrace our steps to reconstruct the search path.

To implement this idea, initialize a new dictionary before starting the BFS. When a node is added to the visit queue, add a key-value pair to the dictionary where the key is the node that is visited and the value is the node that is visiting it. When the target node is found, step back through the dictionary until arriving at the starting node, recording each step.

Problem 4. Implement the `shortest_path` method in the `Graph` class using a BFS. Begin at a specified starting node and proceed until a specified target is found. Return a list containing the node values in the shortest path from the start to the target (including the endpoints). If either of the inputs are not in the adjacency graph, raise a `ValueError`.

Network X

NetworkX is a Python package for creating, manipulating, and exploring large graphs. It contains a graph object constructor as well as methods for adding nodes and edges to the graph. It also has methods for recovering information about the graph and its structure.

```
# Create a new graph object using networkX
>>> import networkx as nx
>>> nx_graph = nx.Graph()

# There are several ways to add nodes and edges to the graph.
# One is to use the add_edge method, which creates new edge
# and node objects as needed, ignoring duplicates
>>> nx_graph.add_edge('A', 'B')
>>> nx_graph.add_edge('A', 'C')
>>> nx_graph.add_edge('A', 'D')
>>> nx_graph.add_edge('A', 'E')
>>> nx_graph.add_edge('B', 'C')
>>> nx_graph.add_edge('C', 'D')
```

```
# Nodes and edges are easy to access
>>> print(nx_graph.nodes())
['A', 'C', 'B', 'E', 'D']

>>> print(nx_graph.edges())
[('A', 'C'), ('A', 'B'), ('A', 'E'), ('A', 'D'), ('C', 'B'), ('C', 'D')]

# NetworkX also has its own shortest_path method, implemented
# with a bidirectional BFS (starting from both ends)
>>> nx.shortest_path(nx_graph, 'C', 'E')
['C', 'A', 'E']

# With small graphs, we can visualize the graph with nx.draw()
>>> from matplotlib import pyplot as plt
>>> nx.draw(nx_graph)
>>> plt.show()
```

Problem 5. Write a `convert_to_networkx` function that accepts an adjacency dictionary. Create a `networkx` object, load it with the graph information from the dictionary, and return it.

The Kevin Bacon Problem

"The 6 Degrees of Kevin Bacon" is a well-known parlor game. The game is played by naming an actor, then trying to find a chain of actors that have worked with each other leading to Kevin Bacon. For example, Samuel L. Jackson was in the film *Captain America: The First Avenger* with Peter Stark, who was in *X-Men: First Class* with Kevin Bacon. Thus Samuel L. Jackson has a "Bacon number" of 2. Any actors who have been in a movie with Kevin Bacon have a Bacon number of 1.

Problem 6. Write a `BaconSolver` class to solve the Kevin Bacon problem.

The file `movieData.txt` contains data from about 13,000 movies released over the course of several years. A single movie is listed on each line, followed by a sequence of actors that starred in it. The movie title and actors' names are separated by a '/' character. The actors are listed by last name first, followed by their first name.

The provided `parse` function generates an adjacency dictionary from a specified file. In particular, `parse("movieData.txt")` generates a dictionary where each key is a movie title and each value is a list of the actors that appeared in the movie.

Implement the constructor of `BaconSolver`. Accept a filename to pull data from and get the dictionary generated by calling `parse` with the filename. Store the collection of values in the dictionary (the actors) as a class attribute, avoiding duplicates. Convert the dictionary to a NetworkX graph and store it as another class attribute.

Finally, use NetworkX to implement the `path_to_bacon` method. Accept start and target values (actors' names) and return a list with the shortest path from the start to the target. Set Kevin Bacon as the default target. If either of the inputs are not contained in the stored collection of dictionary values (if either input is not an actor's name), raise a `ValueError`.

```
>>> movie_graph = BaconSolver("movieData.txt")
>>> movie_graph.path_to_bacon("Jackson, Samuel L.")
['Jackson, Samuel L.', 'Captain America: The First Avenger', 'Stark,
Peter', 'X-Men: First Class', 'Bacon, Kevin']
```

Warning

Because of the size of the dataset, **do not** attempt to visualize the graph with `nx.draw`. The visualization tool in NetworkX is only effective on relatively small graphs. In fact, graph visualization in general remains a challenging and ongoing area of research.

Problem 7. Implement the `bacon_number` method in the `BaconSolver` class. Accept start and target values and return the number of actors in the shortest path from start to target. Note that this is different than the number of entries in the shortest path list, since movies do not contribute to an actor's Bacon number.

Also implement the `average_bacon` method. Compute the average Bacon number across all of the actors in the stored collection of actors. Exclude any actors that are not connected to Kevin Bacon (their theoretical Bacon number would be infinity). Return the average Bacon number and the number of actors not connected at all to Kevin Bacon.

As an aside, the prolific Paul Erdős is considered a Kevin Bacon in the mathematical community. Someone with an "Erdős number" of 2 co-authored a paper with someone who co-authored a paper with Paul Erdős.

Problem 8. (Optional) Create a `plot_bacon` method in the `BaconSolver` class that produces a simple histogram displaying the frequency of the Bacon numbers in the data set. The output should look something like Figure 7.2.

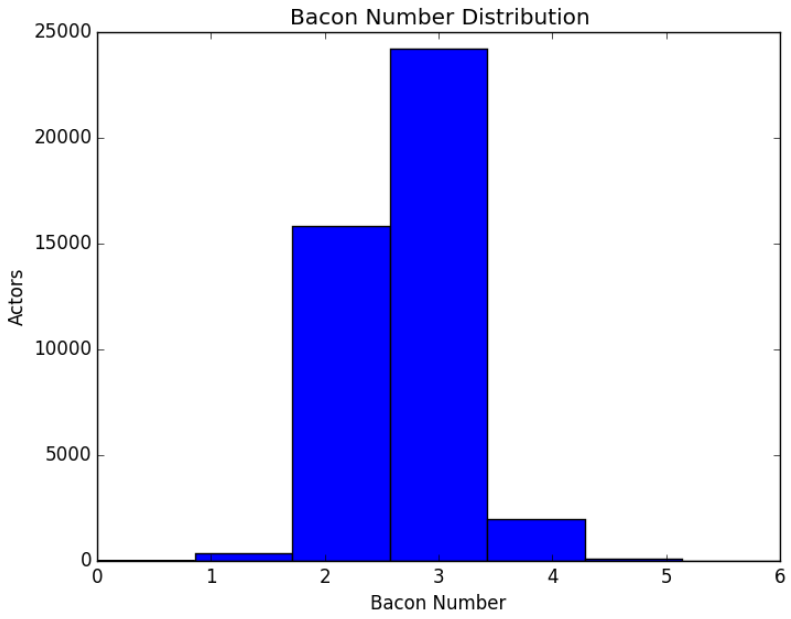

Figure 7.2: The frequency of the bacon numbers.

Part III

Probabilistic Algorithms

Lab 8

Markov Chains

Lab Objective: *A* Markov chain *is a finite collection of states with specified probabilities for transitioning from one state to another. They are characterized by the fact that future behavior of the system depends only on its current state. Markov chains have far ranging applications; in this lab, we create a Markov chain for generating random English sentences.*

Definition and Implementation

Suppose that we wish to model a system that can be described by a finite number of states. A Markov chain is a collection of states, together with the probabilities of moving from one state to another. An example of a Markov chain is a board game where players move around the board based on die rolls. Each space represents a state, and a player is said to be in a state if their piece is currently on the corresponding space. In this case, the probability of moving from one space to another only depends on the players current location. Where the player was on a previous turn does not affect their current turn.

Markov chains have an associated transition matrix that stores all the information about the chain. The $(ij)^{th}$ entry of the matrix gives the probability of moving from state j to state i. Thus the columns of the transition matrix must sum to 1.

Consider a very simple weather model, where the probability of being hot or cold depends on the weather of the previous day. If the probability that tomorrow is hot given that today is hot is 0.7, and the probability that tomorrow is cold given that today is cold is 0.4, then by assigning hot to the 0^{th} column and cold to the 1^{st} column, the Markov chain has the following transition matrix:

$$W = \begin{pmatrix} 0.7 & 0.6 \\ 0.3 & 0.4 \end{pmatrix}$$

We interpret the matrix W as follows. If it is hot today, examine the 0^{th} column of W. There is a 70% chance that tomorrow will be hot (0^{th} row), and a 30% chance that tomorrow will be cold (1^{st} row). Conversely, if it is cold today, here is a 60% chance that tomorrow will be hot, and a 40% chance that tomorrow will be cold.

Problem 1. Transition matrices for Markov chains are efficiently stored as NumPy arrays. Write a function that accepts a dimension n and returns the transition matrix for a random Markov chain with n states.

Simulating State Transitions

We may simulate moving from state to state by sampling from a uniform distribution. In a general Markov chain, if we are in state j then the j^{th} column of the transition matrix gives the probabilities of moving to any other state i. By definition, these probabilities sum to 1. Thus, the entries of each column partition the interval $[0, 1]$, and we can choose the next state to move to by generating a random number between 0 and 1.

Consider again the weather model example from the previous section. Suppose that today is hot, and that we want to simulate tomorrow's weather. The column that corresponds to "hot" in the transition matrix is $(0.7, 0.3)^T$. If we generate a random number and it is smaller than 0.3, then our simulation indicates that tomorrow will be cold. Conversely, if the random number is betwen 0.3 and 1, then the simulation says that tomorrow will be hot. In Python, the programming logic is as follows:

```
import numpy as np

def forecast():
    """Forecast tomorrow's weather given that today is hot."""

    transition_matrix = np.array([[0.7, 0.6], [0.3, 0.4]])
    random_number = np.random.random()
    if random_number < transition_matrix[1,0]:
        print "Cold"
        return 1
    else:
        print "Hot"
        return 0
```

Problem 2. Modify the `forecast` function so that it accepts a parameter `num_days` and runs a simulation of the weather for the number of days given. Return a list containing the day-by-day weather predictions (0 for hot, 1 for cold). Assume the first day is hot, but do not include the data from the first day in the list of predictions. The resulting list should therefore have `num_days` entries.

For Markov chains with very few states, the approach in the `forecast` function is practical and the implementation is fairly simple. However, small Markov chains are typically useless in applications.

Larger Chains

The `forecast` function makes one random draw from a *uniform* distribution to simulate a state change. For larger Markov chains, we draw from a *multinomial* distribution. A multinomial distribution is a mulitvariate generalization of the binomial distribution. A single draw from a binomial distribution with parameter p indicates successes or failure of a single experiment with probability p of success. The classic example is a coin flip, where the p is the probability that the coin lands heads side up. A single draw from a multinomial distribution with parameters $[p_1, p_2, ..., p_n]$ indicates which of n outcomes occurs. In this case the classic example is a dice roll, with 6 possible outcomes instead of the 2 in a coin toss.

```
# To simulate a single dice roll, store the probabilities of each outcome.
>>> probabilities = np.array([1./6, 1./6, 1./6, 1./6, 1./6, 1./6])

# Make a single random draw (roll the die once).
>>> np.random.multinomial(1, probabilities)
array([0, 0, 0, 1, 0, 0])                    # The roll resulted in a 4.
```

Problem 3. Let the following be the transition chain for a Markov chain modeling weather with four states: hot, mild, cold, and freezing.

$$W' = \begin{pmatrix} 0.5 & 0.3 & 0.1 & 0 \\ 0.3 & 0.3 & 0.3 & 0.3 \\ 0.2 & 0.3 & 0.4 & 0.5 \\ 0 & 0.1 & 0.2 & 0.2 \end{pmatrix}$$

with hot, mild, cold, and freezing corresponding to columns 0, 1, 2, and 3, respectively.

Write a new function that accepts a parameter `num_days` and runs the same kind of simulation as `forecast`, but that uses the new four-state transition matrix. This time, assume the first day is mild. Return a list containing the day-to-day results (0 for hot, 1 for mild, 2 for cold, and 3 for freezing).

Problem 4. Write a function that investigates and interprets the results of the simulations in the previous two problems. Specifically, find the average percentage of days that are hot, mild, cold, and freezing in each simulation. Does changing the starting day alter the results? Print a report of your findings to the terminal.

Using Markov Chains to Simulate English

One of the original applications of Markov chains was to study natural languages[1]. In the early 20^{th} century, Markov used his chains to model how Russian switched from vowels to consonants. By midcentury, they had been used to try and model English. It turns out that Markov chains are, by themselves, insufficient to model very good English. However, they can approach a model of bad English, with sometimes amusing results.

A Markov chain model of English has each word as a state. By nature, a Markov chain is only concerned with its current state. Thus, a Markov chain is unaware of context or even previous words in a sentence. For example, a Markov chain's current state may be the word "continuous." Then the chain would say that the next word in the sentence is more likey to be "function" rather than "racoon." However, without the context of the rest of the sentence, even two likely words stringed together may result in gibberish.

To build a Markov chain to simulate English, we need to determine the transition probabilities between words. One way to do this would be to assign every word in English a number. Say there are N of them, and create an $N \times N$ matrix of zeros. Then, read every written work in English and when word b follows word a, we add 1 to the $(b, a)^{th}$ entry of the matrix. Once we have done this for every word of every written work, we normalize the columns and have a transition matrix that we may simulate from.

The main problem with this approach is the sheer enormity of the task at hand. We will restrict ourselves to a subproblem of modeling the English of a specific file. Thus, the transition probabilities of our Markov chains will reflect the sort of English that the source authors speak. For example, the transition matrix built from the Complete Works of William Shakespeare will differ greatly from, say, a collection of academic journals. We will call the source collection of works in the next problems the *training set*.

Problem 5. First we must convert a file of English words to numbers. Each unique word in the file will correspond to a single number, and each of these numbers will correspond to a row and column in the transition matrix (to be built in the next problem).

Write a function that accepts the path to a file containing a training set of English sentences, with one sentence per line. Parse the file, assigning a unique natural number to each unique word. As you parse, write the corresponding sequence of numbers to a new file, maintaining the line break structure.

We provide an example below. On the left is the training set of sentences, and on the right is the corresponding file of numbers. Note that once a word is assigned a number, the same number is used to represent the word throughout the rest of the file, thus preserving the 1-1 relationship of words to numbers.

[1] See `http://langvillea.people.cofc.edu/MCapps7.pdf` for some details

```
Love is patient Love is kind                        1 2 3 1 2 4
It does not envy It does not boast                  5 6 7 8 5 6 7 9
It is not proud It is not rude                      5 2 7 10 5 2 7 11
It is not self-seeking It is not easily angered     5 2 7 5 2 7 12 13 14
It keeps no record of wrongs                        5 15 16 17 18 19
Love does not delight in evil                       1 6 7 20 21 22
but rejoices with the truth                         23 24 25 26 27
It always protects always trusts                    5 28 29 28 30
always hopes always perseveres                      28 31 28 32
Love never fails                                    1 33 34
```

Here the word "Love" is assigned the number 1, "is" is assigned 2, and "kind" is assigned 4 (since it is the 4^{th} unique word in the file).

Also keep track of each word-number pair with a some basic data structure. This could be an ordered list of words, a word-number dictionary, a set of tuples, or any other simple structure (which will be fastest?). Return this data structure.

Starting and Stopping states

Now that we have converted our English text into numbers, we can build the transition matrix for the Markov chain. We will scan the file we created in the previous problem and use it to create the matrix.

In the previous weather model we chose a fixed number of states to simulate. However, in English, sentences are of varying length. One way to simulate this is to create a start state and an end state. To generate a new sentence, we begin in the given start state. The start state may transition to any of the words that are at the beginning of the sentences in the training set. Words that are at the end of the sentences in the training set will have a probability of moving to the end state. Once the chain has moved to the end state, we terminate the sentence.

Problem 6. Write a function that accepts the path to the file created in the previous problem and the number of unique words in the training. Initialize a square zero matrix of zeros whose dimension is the number of unique words in the text, plus 2 (to include the start and stop state). Then, read each line of the file and for each pair of subsequent numbers add one to the corresponding entry of the matrix.

For instance, if we scanned the line 2 6 3 7 9, then we would add one to the $(6, 2)$, $(3, 6)$, $(7, 3)$, and $(9, 7)$ entries of the matrix. In addition, the start state must transition to the first word in of the line and the last word of the line must transition to the end state. Then we would also increment the $(2, 0)$ and $(N, 9)$ entries, where N is the index of the last column in the matrix. Finally, to avoid a column of all zeros, we say that the end state transitions to itself with probability 1. Thus we increment the (N, N) entry as well.

Once the entire file has been read, divide each column by its column sum. Then each column will sum to one, with the ij^{th} entry corresponding to the probability of moving from word j to word i.

Problem 7. Write a function that accepts a file name to read data in from (the training set), a file name to write data out to, and an optional integer argument `num_sentences`. Use Problem 5 to write the file of numbers that corresponds to the training set, and to get the data structure describing the one-to-one relationship between the words and the numbers. Then use Problem 6 to generate the corresponding transition matrix.

Begin at the start state and use the strategy from Problem 3 to transition through the Markov chain. Keep track of the path through the chain and the corresponding path of words. When the end state is reached, stop transitioning and terminate the sentence. Write the resulting sentence to the outfile, followed by a newline character. Write as many sentences to the file as is specified by `num_sentences`.

Additional Exercises

Problem 8. The approach in the previous three problems begins to fail as the training set grows larger. For example, a single Shakespearean play may not be large enough to cause memory problems, but the Complete Works of William Shakespeare certainly will.

Consolidate the functions from the previous three problems into a single function. Then, to accommodate larger data sets, use a sparse matrix for the transition matrix in instead of a regular NumPy array (use the `lil_matrix` from the `scipy.sparse` library). Ensure that the process still works on small training sets, then proceed to larger training sets. How are the resulting sentences different if a very large training set is used instead of a small training set?

Part IV

Fourier Analysis

Lab 9

Discrete Fourier Transform

Lab Objective: *The analysis of periodic functions has many applications in pure and applied mathematics, especially in settings dealing with sound waves. The Fourier transform provides a way to analyze such periodic functions. In this lab, we implement the discrete Fourier transform and explore digital audio signals.*

Sound Waves

Sounds are vibrations in the air around us. The frequency and intensity of these vibrations determine how sound is perceived. Sounds correspond physically to continuous functions, but they may be discretely approximated on a computer. These discrete approximations can be made indistinguishable from a continuous signal.

Digital Audio Signals

There are two components of a digital audio signal: samples from the soundwave and a sample rate. These correspond to amplitude and frequency, respectively. A sample is a measurement of the amplitude of the wave at an instant in time.

If we know at what rate the samples were taken, then we can construct the wave exactly as it was recorded. In most applications, this sample rate will be measured in number of samples taken per second. The standard rate for high quality audio is 44100 equally spaced samples per second.

Problem 1. Write a class called `Signal` for storing digital audio signals. The constructor should accept a sample rate (an integer) and an array of samples (a NumPy array). Store these inputs as attributes.

Write a method called `plot` that generates the graph of the soundwave. Use the sample rate to get the x-axis in terms of seconds. See Figure 9.1 for an example.

Wave File Format

One of the most common audio file formats across operating systems is the `wave` format, also called `wav` after its file extension. It is a lightweight, common standard that is in wide use. SciPy has built-in tools to read and create `wav` files. To read in a `wav` file, we can use the `read` function that returns the file's sample rate and samples. See Figure 9.1.

```
# Read from the sound file.
>>> from scipy.io import wavfile
>>> rate, wave = wavfile.read('tada.wav')

# To visualize the data, use the Signal class's plot function.
>>> sig = Signal(rate, wave)
>>> sig.plot()
```

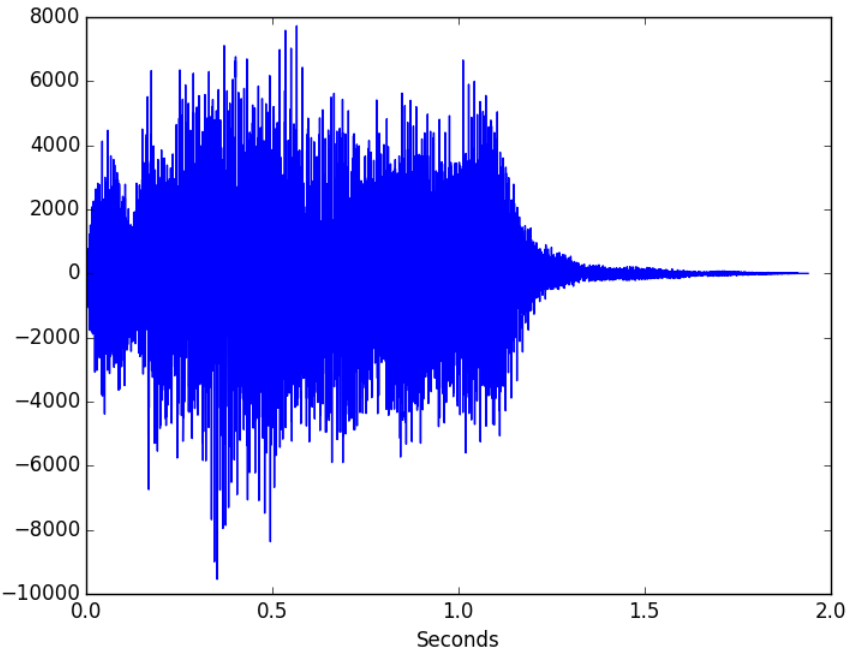

Figure 9.1: The soundwave of `tada.wav`.

Writing a signal to a file is also simple. We use `wavfile.write`, specifing the name of the new file, the sample rate, and the array of samples.

```
# Write a random signal sampled at a rate of 44100 hz to my_sound.wav.
>>> wave = sp.random.randint(-32767, 32767, 30000)
>>> samplerate = 44100
>>> wavfile.write('my_sound.wav', samplerate, wave)
```

Scaling

The `wavfile.write` function expects an array of 16 bit integers for the samples (whole numbers between -32767 and 32767). Therefore, waves may need to be scaled and converted to integers before being written to a file.

```
# Generate random samples between -0.5 and 0.5.
>>> samples = sp.random.random(30000)-.5
# Scale the wave so that the samples are between -32767 and 32767.
>>> samples *= 32767*2
# Cast the samples as 16 bit integers.
>>> samples = sp.int16(samples)
```

The scaling technique in the above example works, but only because we knew beforehand that the values were in the interval $[-\frac{1}{2}, \frac{1}{2}]$. If the entries of a wave are not scaled properly, the operating system may not know how to play the file.

Problem 2. Add a method to the `Signal` class called `export` that accepts a file name and generates a `.wav` file from the sample rate and the array of samples. Scale the array of samples appropriately before writing to the outfile. Ensure that your scaling technique is valid for arbitrary arrays of samples.

Creating Sounds in Python

In order to generate a sound in python, we need to sample the corresponding sine wave and then save it as an audio file. For example, suppose that we want to generate a sound with a frequency of 500 hertz for 10 seconds.

```
>>> samplerate = 44100
>>> frequency = 500
>>> length = 10         # Length in seconds of the desired sound.
```

Recall the the function $\sin(x)$ has a period of 2π. To create sounds, however, we want the period of our wave to be 1, corresponding to 1 second. Thus, we will sample from the function

$$\sin(2\pi x f)$$

where f is our desired frequency.

```
# The lambda keyword is a shortcut for creating a one-line function.
>>> wave_function = lambda x: sp.sin(2*sp.pi*x*frequency)
```

In the following code, we generate a signal using three steps: first, find the correct step size given the sample rate. Next, generate the points at which we wish to sample the wave. Finally, sample the wave by passing the sample points to `wave_function`. Then we can use our `Signal` class to plot the soundwave or write it to a file.

```
# Calculate the step size, the sample points, and the sample values.
>>> stepsize = 1./samplerate
>>> sample_points = sp.arange(0, length, stepsize)
>>> samples = wave_function(sample_points)

# Use the Signal class to write the sound to a file.
>>> sinewave = Signal(samplerate, samples)
>>> sinewave.export("sine.wav")
```

The `export` method should take care of scaling and casting the entries as 16-bit integers.

Problem 3. The 'A' note occurs at a frequency of 440 hertz. Generate the sine wave that corresponds to an 'A' note being played for 5 seconds.

Once you have successfully generated the 'A' note, experiment with different frequencies to generate different notes. The following table shows some frequencies that correspond to common notes.

Note	Frequency
A	440
B	493.88
C	523.25
D	587.33
E	659.25
F	698.46
G	783.99

Implement a function outside of the `Signal` class that accepts a frequency and a duration and returns an instance of the `Signal` class corresponding to the desired soundwave. Sample at a rate of 44100 samples per second to create these sounds.

Discrete Fourier Transform

Some Technicalities

Under the right conditions, a continuous periodic function may be represented as a sum of sine waves:

$$f(x) = \sum_{k=-\infty}^{\infty} c_k \sin kx$$

where the constants c_k are called the *Fourier coefficients*.

Such a transform also exists for discrete periodic functions. Whereas the frequencies present in the continuous case are multiples of a sine wave with a period of 1, the discrete case is somewhat different. The Fourier coefficients in the discrete case represent the amplitudes of sine waves whose periods are multiples of a

"fundamental frequency." The fundamental frequency is a sine wave with a period length equal to the amount of time of the signal.

The k^{th} coefficient of a signal $\{x_0, .., x_{N-1}\}$ is calculated with the following formula:

$$c_k = \sum_{n=0}^{N-1} x_n e^{\frac{2\pi i k n}{N}} \tag{9.1}$$

where i is the square root of -1. This process is done for each k from 0 to $N-1$. Thus there are just as many Fourier coefficients as samples from the orginal signal.

Problem 4. Write a function that accepts a NumPy array and computes the discrete Fourier transform of the array using Equation 9.1. Return the array of calculated coefficients.

SciPy has several methods for calculating the DFT of an array. Use `scipy.fft` or `scipy.fftpack.fft` to check your implementation. The naive method is significantly slower than SciPy's implementation, so test your function only on small arrays. Can you calculate each coefficient c_k in just one line of code?

Plotting the DFT

The graph of the fourier transform of a sound file is useful in applications. While the graph of the original signal gives information about the amplitude of a soundwave at certain points, the graph of the discrete Fourier transform shows which frequencies are present in the signal. Frequencies present in the signal have non-zero coefficients. The magnitude of these coefficients corresponds to how influential the frequency is in the signal. For example, the sounds that we generated in the previous section contained only one frequency. If we created an 'A' note at 440 hz, then the graph of the DFT would appear as in Figure 9.2.

On the other hand, the DFT of a more complicated soundwave will have many frequencies present. Some of these frequencis correspond to the different tones present in the signal. See Figure 9.3 for an example.

Fixing the x-axis

If we take the DFT of a signal and then plot it without any other considerations, the x-axis will correspond to the index of the coefficients in the DFT and not their frequencies. In a previous section, we mention that the "fundamental frequency" for the DFT corresponds to a sine wave whose period is the same as the length of the signal. Thus, if unchanged, the x-axis gives us the number of times a particular

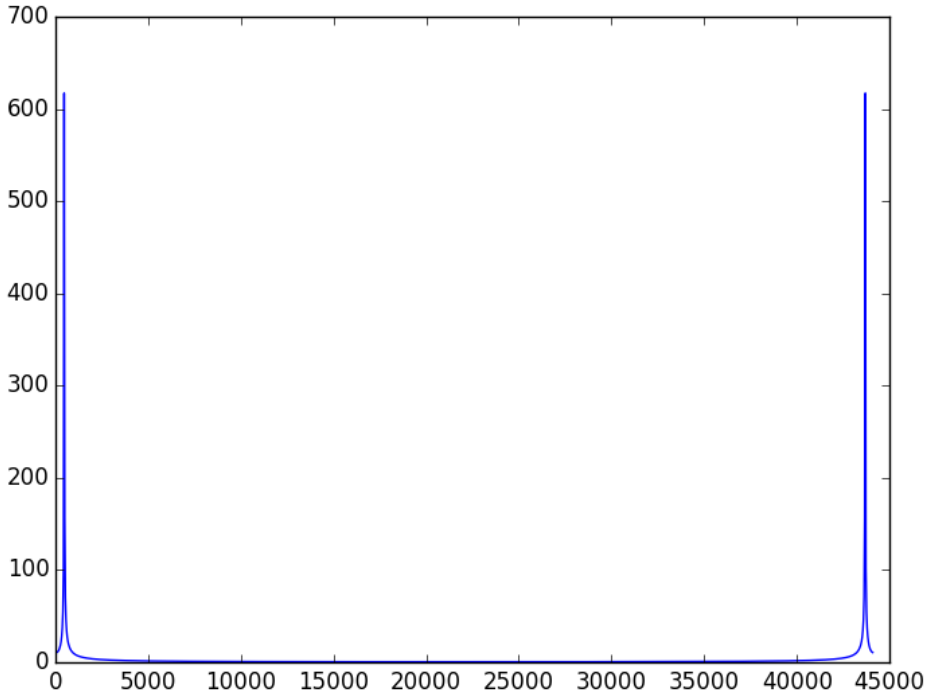

Figure 9.2: The magnitude of the coefficients of the discrete Fourier transform of an 'A' note. Notice that there are two spikes in the graph, the first around 440 on the x-axis. This second spike is due to symmetries inherent in the DFT. For our purposes we will mostly be concerned with the left side of the DFT plot.

sine wave cycles throughout the whole signal. If we want to label the x-axis with the frequencies measured in hertz, or cycles per second, we will need to convert the units. Fortunately, the bitrate is measured in samples per second. Therfore, if we divide the frequency (given by the index) by the number of samples, and multiply by the sample rate, we end up with cycles per second, or hertz.

$$\frac{\text{cycles}}{\text{samples}} \times \frac{\text{samples}}{\text{second}} = \frac{\text{cycles}}{\text{second}}$$

```
# Calculate the DFT and the x-values that correspond to the coefficients. Then
# convert the x-values so that they measure frequencies in hertz.
>>> dft = sp.fft(signal)
>>> x_vals = sp.arange(1,len(dft)+1, 1)*1. # Make them floats

# x_vals now corresponds to frequencies measured in cycles per signal length.
>>> x_vals = x_vals/len(signal)
>>> x_vals = x_vals*rate
```

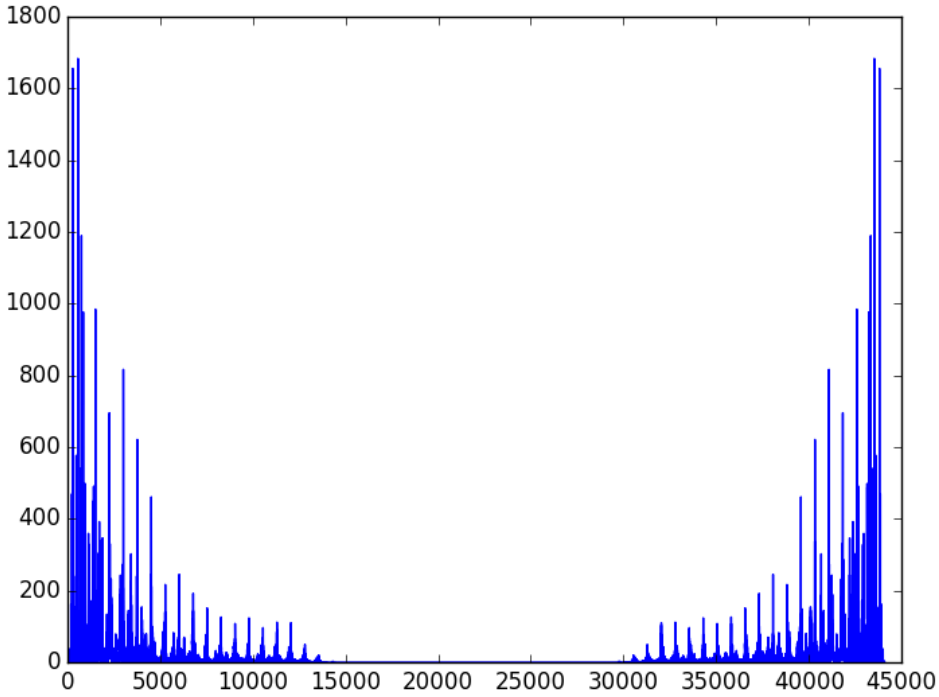

Figure 9.3: The discrete Fourier transform of `tada.wav`. Each spike in the graph corresponds to a frequency that is present in the signal.

Problem 5. Update the `plot` method in the `Signal` class so that it generates a single plot with two subplots: the original soundwave, and the magnitude of the coefficients of the DFT (as in Figure 9.3). Use one of SciPy's FFT implementations to calculate the DFT.

Problem 6. A chord is a conjunction of several notes played together. We can create a chord in Python by adding several sound waves together. For example, to create a chord with 'A', 'C', and 'E' notes, we generate the sound waves for each, as in the prior problem, and then add them together.

Create several chords and observe the plot of their DFT. There should be as many spikes as there are notes in the plot. Then create a sound that changes over time.

(Hints: you may consider implementing the `__add__` magic method for the `Signal` class. The SciPy/NumPy functions `hstack` and `vstack` may also be

helpful.)

Lab 10

Filtering and Convolution

Lab Objective: *The Fourier transform reveals things about an audio signal that are not immediately apparent from the soundwave. In this lab we learn to filter noise out of a signal using the discrete Fourier transform, and explore the effect of convolution on sound files.*

Cleaning up a Noisy Signal

Listen to `Noisysignal1.wav`. This is a mono recording of a (probably familiar) voice with some annoying noise over it. The plot of the soundwave isn't very descriptive; in fact, it looks like static. See Figure 10.1.

However, if we take the Fourier transform of the signal, we see that the static in Figure 10.1 is the result of some concentrated high frequency noise. (In this case, artificially added). See Figure 10.2.

The noise can be removed by setting the coefficients of the high frequencies to zero. Since the discrete Fourier transform is symmetric, if we set coeffecient j to 0, then we must set coefficient $N-j$ to 0 as well, where N is the number of coefficients. Then we calculate the inverse Fourier transform to get a new, clean signal.

```
>>> rate,data = wavfile.read('Noisysignal1.wav')

# Calculate the Fourier transform
>>> fsig = sp.fft(data, axis = 0)

# Coefficients 10000 to 20000 were chosen by inspecting the
# plot of the Fourier transform.
>>> for j in xrange(10000, 20000):
...     # Set the chosen coefficients to 0
...     fsig[j] = 0
...     fsig[-j] = 0

# Calculate the inverse Fourier transform, cast it as real,
# and scale it to be compatible with the wavfile format.
>>> newsig = sp.ifft(fsig)
>>> newsig = sp.real(newsig)
>>> newsig = sp.int16(newsig / sp.absolute(newsig).max() * 32767)
```

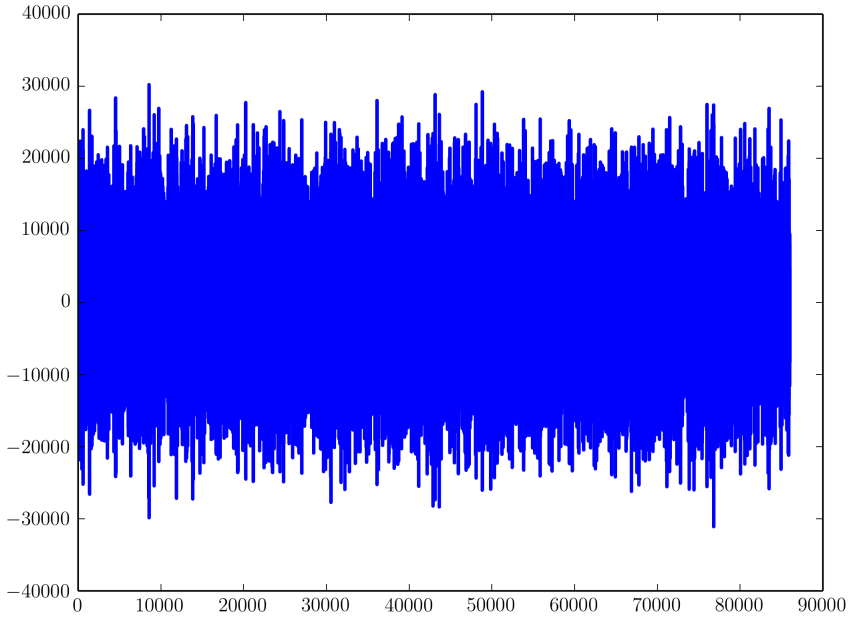

Figure 10.1: The plot of `Noisysignal1.wav`.

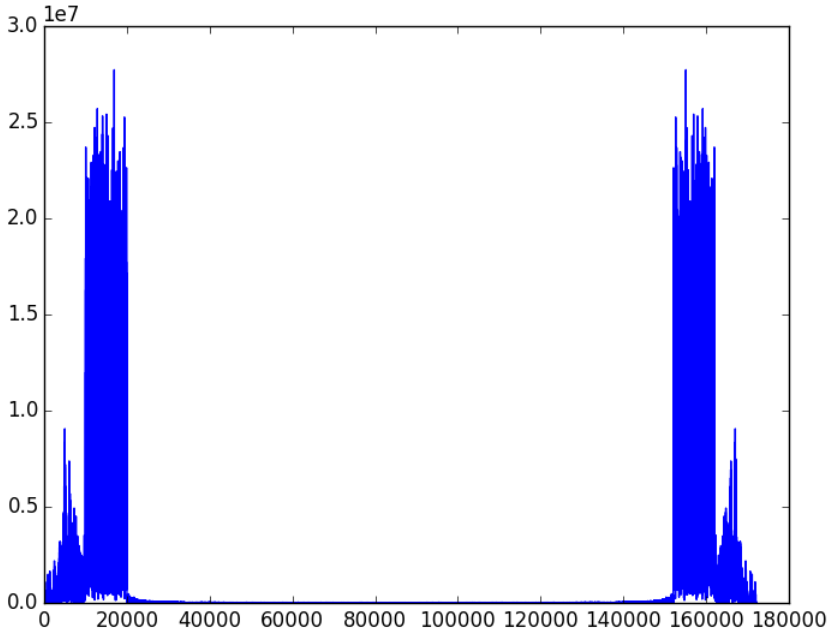

Figure 10.2: Spectrum of `Noisysignal1.wav`

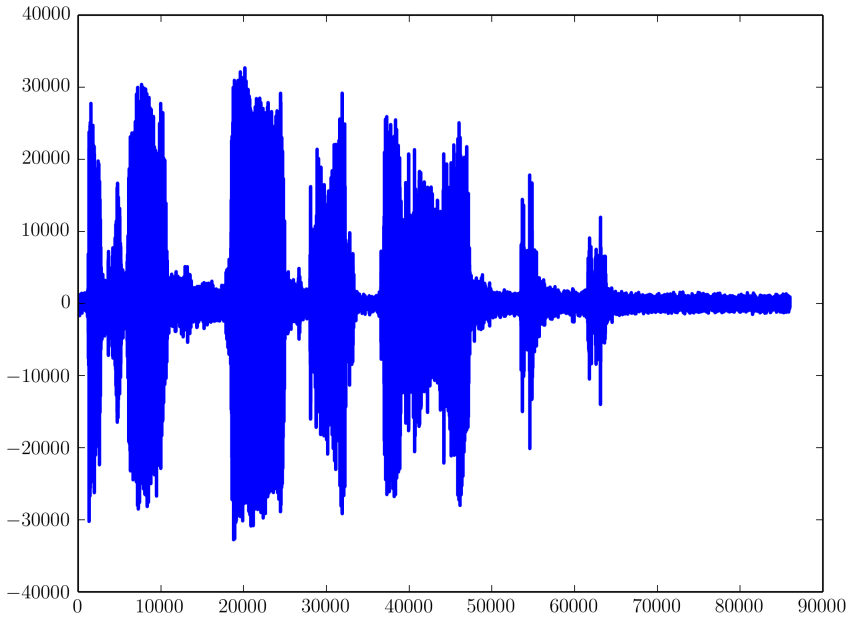

Figure 10.3: The plot of `Noisysignal1.wav` after being cleaned.

Now we can save the resulting cleaned-up signal `newsig` to a `.wav` file. The plot of the wave now reveals individual syllables as they are spoken. See Figure 10.3.

Problem 1. Listen to `Noisysignal2.wav`. You will probably just hear noise. Inspect the discrete Fourier transform to see where there is noise. Remove the noise using the technique described above in order to make the cleaned-up signal audible. What does the voice say? Who is the speaker? (If you don't know the answer to this last question, try a quick Google search.)

The DFT is commonly used in sound filtering, though identifying the particular frequencies to zero out can be difficult.

Filtering and Convolution

The DFT is useful for more than filtering noise out of a signal. Suppose we have a recording of musical piece played in a small carpeted room with essentially no acoustics (little or no echo), and suppose we would like to apply an effect to make it sound as if the piece were played in a large concert hall or some other room. The DFT makes this possible when used together with the idea of *convolution*.

When a balloon is popped in large room, although the sound of the actual pop only lasts a few milliseconds, the sound echoes about the room for up to several

seconds. This echoing sound is called an *impulse response* of the room, and is a way of approximating the acoustics of a room.

So first, we need a recording of how the room responds to a short pulse of sound. Effective ways of producing a loud sound approximating a pulse include firing a (blank) gunshot, popping a balloon, or, if neither of those are available, clapping the hands one time.

Recall that we model sound with discrete samples of a soundwave in rapid succession. When these sounds are played back, the ear percieves them as a continuous soundwave. In other words, sound playback is a series of pulses of varying intensities, similar to the pulse in an impulse response. If we "mix" the individual sounds of an instrument in a carpeted room with the impulse response from a concert hall, then the new soundwave will sound as if the instrument is being played in the concert hall.

Since audio needs to be samples frequently (44100 samples per second is standard) to create smooth playback, a recording of a song can be millions of samples. Each of these samples needs to be combined with the impulse response, which may be several seconds long. This may be starting to seem computationally infeasible or at least very difficult. The key is to recognize that this process can be described as a convolution: namely, the final sound is simply the convolution of the our original sound with the impulse response. We can calculate convolutions quickly using the convolution theorem:

$$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g)$$

where $\mathcal{F}$ is the Fourier Transform, $*$ is convolution, and $\cdot$ is component-wise multiplication. Thus we calculate the convolution of two arrays by simply taking the Fourier transform of each, multiplying them pointwise, and then taking the inverse transform.

Problem 2. (Optional)[a] Find a large room or area with good acoustics, and record (an approximation to) its impulse response using a balloon pop. To record the sound, you will want to use at least a decent microphone. You may want to record it using the program Audacity[b] and a laptop. If you use a unidirectional microphone, be sure the microphone is pointing at the balloon when you pop it, so that the direct sound from the pop is picked up. (If you don't, the result will still be okay. However, after the convolution it will probably sound somewhat distant, as if we were standing somewhere where we couldn't hear the music directly.) If you've chosen a good room, the response should be audible for at least a full second.

Include a plot of both the waveform and spectrum of the impulse response you recorded.

[a] If the instructor does not require this problem then students may use the provided `balloon.wav` file which contains the sound of a balloon pop in a large room.

[b] Audacity is free sound manipulation software and may be downloaded at http://audacity.sourceforge.net

Problem 3. Download and listen to the file `chopin.wav`. You will hear a piano being played in a dead room with little or no acoustics. Using the Convolution Theorem, take the convolution of this signal with the impulse response recorded in the previous problem. The convolution given in the theorem is *circular*, meaning that sounds at the end of the signal will tend to mix with sounds at the beginning of the signal. To avoid this effect, add several seconds of silence to the end of `chopin.wav` by appending zeroes to the end of the signal. Also, keep in mind that the Convolution Theorem requires both signals to have the same length; therefore you will need to pad the smaller of your two signals (namely, the impulse response signal) with zeros in order to make it the same size as the other signal. These zeros should be added to the middles of signals, as we need to maintain its symmetric structure. Describe the resulting sound.

To summarize:

1. Read in `chopin.wav` and the impulse response with `wavfile`,
2. Add several seconds of silence to the signal from `chopin.wav`,
3. Insert zeros into the middle of the impulse response transform so that it is the same length as,
4. Calculate the convolution of the signals,
5. And finally, calculate the inverse Fourier transform.

In some instances, a circular convolution is actually desirable. For instance, an interesting effect is achieved by taking the circular convolution of a long segment of white noise with some other (shorter) sound. We can create white noise using SciPy's `random` module:

```
# Create 10 seconds of mono white noise.
samplerate = 22050
noise = sp.int16(sp.random.randint(-32767, 32767, samplerate * 10))
```

Problem 4. Create white noise and listen to the resulting sound (**CAUTION:** Turn your volume way down. It may be very very loud). This kind of noise is called "white" because it contains all frequencies with the same strength, or rather, with the same expected strength (since the amplitude of a specific frequency is a matter of chance). In order to see this, plot the spectrum of the noise.

Now can take the circular convolution of this noise with some other sound. For instance, let's use `tada.wav`. The result is in `tada-conv.wav`. We notice that the original short sound has been sustained to an indefinite length. The result is not a set of static tones, but rather a rich sound which preserves not only the tones, but the

texture, of the original sound; you can hear different tones fluctuating randomly in amplitude over time. If you were to play this `tada-conv.wav` on repeat, you would find that, because we used a circular convolution, the sound loops seamlessly from the end back to the beginning; however, most sound players are not capable of doing this properly, so you will probably hear a break in the sound. To demonstrate the "seamlessness", we can paste together three copies of the sound consecutively:

```
rate, sig = wavfile.read('tada-conv.wav')
sig = sp.append(sig, sig)
sig = sp.append(sig, sig)
```

Listen to the resulting sound, and notice that we are not able to identify where the sound loops back to the beginning, because there is no break or click.

Lab 11

Introduction to Wavelets

Lab Objective: *In the context of Fourier analysis, one seeks to represent a function as a sum of sinusoids. A drawback to this approach is that the Fourier transform only captures global frequency information, and local information is lost; we can know which frequencies are the most prevalent, but not when or where they occur. The Wavelet transform provides an alternative approach that avoids this shortcoming and is often a superior analysis technique for many types of signals and images.*

The Discrete Wavelet Transform

In wavelet analysis, we seek to analyze a function by considering its *wavelet decomposition*. The wavelet decomposition of a function is a way of expressing the function as a linear combination of a particular family of basis functions. In this way, we can represent a function by the sequence of coefficients (called *wavelet coefficients*) defining this linear combination. The mapping from a function to its sequence of wavelet coefficients is called the *discrete wavelet transform*.

This situation is entirely analogous to the discrete Fourier transform. Instead of using trigonometric functions as our basis, we use a different family of basis functions. In Wavelet analysis, we determine the family of basis functions by first starting off with a function ψ called the *wavelet* and a function ϕ called the *scaling function* (these functions are also called the mother and father wavelets, respectively). We then generate countably many basis functions (sometimes called baby wavelets) from these two functions:

$$\psi_{m,k}(x) = \psi(2^m x - k)$$

$$\phi_{m,k}(x) = \phi(2^m x - k),$$

where $m, k \in \mathbb{Z}$. The historically first, and most basic, wavelet is called the *Haar Wavelet*, given by

$$\psi(x) = \begin{cases} 1 & \text{if } 0 \leq x < \frac{1}{2} \\ -1 & \text{if } \frac{1}{2} \leq x < 1 \\ 0 & \text{otherwise.} \end{cases}$$

The associated scaling function is given by

$$\phi(x) = \begin{cases} 1 & \text{if } 0 \leq x < 1 \\ 0 & \text{otherwise.} \end{cases}$$

In the case of finitely-sampled signals and images, only finitely many wavelet coefficients are nonzero. Depending on the application, we are often only interested in the coefficients corresponding to a subset of the basis functions. Since a given family of wavelets forms an orthogonal set, we can compute the wavelet coefficients by taking inner products (i.e. by integrating). This direct approach is not particularly efficient, however. Just as there are fast algorithms for computing the fourier transform (e.g. the FFT), we can efficiently calculate wavelet coefficients using techniques from signal processing. In particular, we will use an *iterative filterbank* to compute the transform.

Let's launch into an implementation of the one-dimensional discrete wavelet transform. The key operations in the algorithm are the discrete convolution ($*$) and down-sampling (DS). The inputs to the algorithm are a one-dimensional array X (the signal that we want to transform), a one-dimensional array L (called the *low-pass filter*), a one-dimensional array H (the *high-pass filter*), and a positive integer n (controlling to what degree we wish to transform the signal, i.e. how many wavelet coefficients we wish to compute). The low-pass and high-pass filters can be derived from the wavelet and scaling function. The low-pass filter extracts low frequency information, which gives us an approximation of the signal. This approximation highlights the overall (slower-moving) pattern without paying too much attention to the high frequency details, which to the eye (or ear) may be unhelpful noise. However, we also need to extract the high-frequency details with the high-pass filter. While they may sometimes be nothing more than unhelpful noise, there are applications where they are the most important part of the signal; for example, details are very important if we are sharpening a blurry image or increasing contrast.

For the Haar Wavelet, our filters are given by

$$L = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$
$$H = \begin{bmatrix} -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}.$$

See Algorithm 11.1 and Figure 11.1 for the specifications.

Algorithm 11.1 The one-dimensional discrete wavelet transform.

```
1: procedure DWT(X, L, H, n)
2:     i ← 0                                   ▷ Some initialization steps
3:     A_i ← X
4:     while i < n do
5:         D_{i+1} ← DS(A_i * H)               ▷ High-pass filtering
6:         A_{i+1} ← DS(A_i * L)               ▷ Low-pass filtering
7:         i ← i + 1
8:     return A_n, D_n, D_{n−1}, ..., D_1.
```

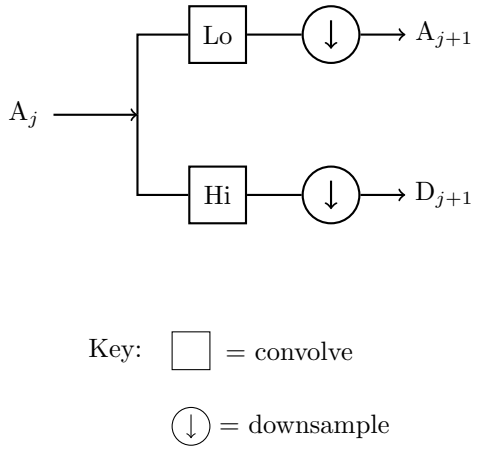

Figure 11.1: The one-dimensional discrete wavelet transform implemented as a filter bank.

At each stage of the algorithm, we filter the signal into an approximation and its details. Note that the algorithm returns a sequence of one dimensional arrays

$$A_n, D_n, D_{n-1}, \ldots, D_1.$$

If the input signal X has length 2^m for some $m \geq n$ and we are using the Haar wavelet, then A_n has length 2^{m-n}, and D_i has length 2^{m-i} for $i = 1, \ldots, n$. The arrays D_i are outputs of the high-pass filter, and thus represent high-frequency details. Hence, these arrays are known as *details*. The array A_n is computed by recursively passing the signal through the low-pass filter, and hence it represents the low-frequency structure in the signal. In fact, A_n can be seen as a smoothed approximation of the original signal, and is called the *approximation*.

As noted earlier, the key mathematical operations are convolution and down-sampling. To accomplish the convolution, we simply use a function in SciPy.

```
>>> import numpy as np
>>> from scipy.signal import fftconvolve
>>> # initialize the filters
>>> L = np.ones(2)/np.sqrt(2)
>>> H = np.array([-1,1])/np.sqrt(2)
>>> # initialize a signal X
>>> X = np.sin(np.linspace(0,2*np.pi,16))
>>> # convolve X with L
>>> fftconvolve(X,L)
[ -1.84945741e-16   2.87606238e-01   8.13088984e-01   1.19798126e+00
   1.37573169e+00   1.31560561e+00   1.02799937e+00   5.62642704e-01
   7.87132986e-16  -5.62642704e-01  -1.02799937e+00  -1.31560561e+00
  -1.37573169e+00  -1.19798126e+00  -8.13088984e-01  -2.87606238e-01
  -1.84945741e-16]
```

The convolution operation alone gives us redundant information, so we down-sample to keep only what we need. In particular, we will down-sample by a factor of two, which means keeping only every other entry:

```
>>> # down-sample an array X
>>> sampled = X[1::2]
```

Putting these two operations together, we can obtain the approximation coefficients in one line of code:

```
>>> A = fft.convolve(X,L)[1::2]
```

Computing the detail coefficients is done in exactly the same way, replacing L with H.

Problem 1. Write a function that calculates the discrete wavelet transform as described above. The output should be a list of one-dimensional NumPy arrays in the following form: $[A_n, D_n, \dots, D_1]$.

The main body of your function should be a loop in which you calculate two arrays: the i-th approximation and detail coefficients. Append the detail coefficients array to your list, and feed the approximation array back into the loop. When the loop is finished, append the approximation array. Finally, reverse the order of your list to adhere to the required return format.

Test your function by calculating the Haar wavelet coefficients of a noisy sine signal for $n = 4$:

```
>>> domain = np.linspace(0,4*np.pi, 1024)
>>> noise =  np.random.randn(1024)*.1
>>> noisysin = np.sin(domain) + noise
>>> coeffs = dwt(noisysin, L, H, 4)
```

Plot your results and verify that they match the plots in Figure 11.2.

We can now transform a one-dimensional signal into its wavelet coefficients, but the reverse transformation is just as important. Luckily, we can reconstruct a signal from the approximation and detail coefficients. We reverse the effects of the filterbank, using slightly modified filters, essentially adding the details back into the signal at each stage until we reach the original. The Haar wavelet filters for the inverse transformation are

$$L = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$
$$H = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}.$$

Suppose we have the wavelet coefficients A_n and D_n. Consulting Figure 11.1, we can recreate A_{n-1} by tracing the schematic backwards: A_n and D_n are first *up-sampled*, then they are convolved with L and H, respectively, and finally added together to obtain A_{n-1}. Up-sampling means doubling the length of an array by inserting a 0 at every other position. In Python, this whole process looks like:

```
>>> # up-sample the coefficient arrays A, D
>>> up_A = np.zeros(2*A.size)
```

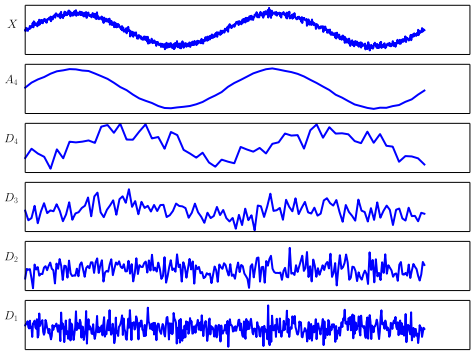

Figure 11.2: A level 4 wavelet decomposition of a signal. The top panel is the original signal, the next panel down is the approximation, and the remaining panels are the detail coefficients. Notice how the approximation resembles a smoothed version of the original signal, while the details capture the high-frequency oscillations and noise.

```
>>> up_A[::2] = A
>>> up_D = np.zeros(2*D.size)
>>> up_D[::2] = D
>>> # now convolve and add, but discard last entry
>>> A = fftconvolve(up_A,L)[:-1] + fftconvolve(up_D,H)[:-1]
```

Now that we have A_{n-1}, we repeat the process with A_{n-1} and D_{n-1} to obtain A_{n-2}. Proceed for a total of n steps (one for each $D_n, D_{n-1}, \ldots, D_1$) until we have obtained A_0. Since A_0 is defined to be the original signal, we have finished the inverse transformation.

Problem 2. Write a function that calculates the inverse wavelet transform as described above. The inputs should be a list of arrays (of the same form as the output of your discrete wavelet transform function), the low-pass filter, and the high-pass filter. The output should be a single array, the recovered signal.

Note that the input list of arrays has length $n+1$ (consisting of A_n together with $D_n, D_{n-1}, \ldots, D_1$), so your code should perform the process given above n times.

In order to check your work, compute the discrete wavelet transform of a random array for different values of n, then compute the inverse transform. Compare the original signal with the recovered signal using `np.allclose`.

The PyWavelets Module

Having implemented our own version of the basic 1-dimensional wavelet transform, we now turn to PyWavelets, a Python library for Wavelet Analysis. It provides

convenient and efficient methods to calculate the one- and two-dimensional discrete Wavelet transform, as well as much more. Assuming that the package has been installed on your machine, type the following to get started:

```
>>> import pywt
```

Performing the discrete Wavelet transform is very simple. Below, we compute the one-dimensional transform for a sinusoidal signal.

```
>>> import numpy as np
>>> f = np.sin(np.linspace(0,8*np.pi, 256)) # build the sine wave
>>> fw = pywt.wavedec(f, 'haar') # compute the wavelet coefficients of f
```

The variable `fw` is now a list of arrays, starting with the final approximation frame, followed by the various levels of detail coefficients, just like the output of the wavelet transform function that you already coded. Plot the level 2 detail and verify that it resembles a blocky sinusoid.

```
>>> from matplotlib import pyplot as plt
>>> plt.plot(fw[-2], linestyle='steps')
>>> plt.show()
```

To reconstruct the signal, we simply call the function `waverec`:

```
>>> f_prime = pywt.waverec(fw, 'haar') # reconstruct the signal
>>> np.allclose(f_prime, f) # compare with the original
True
```

The second positional argument, as you will notice, is a string that gives the name of the wavelet to be used. We first used the Haar wavelet, with which you are already familiar. PyWavelets supports a number of different Wavelets, however, which you can list by executing the following code:

```
>>> # list the available Wavelet families
>>> print pywt.families()
['haar', 'db', 'sym', 'coif', 'bior', 'rbio', 'dmey']
>>> # list the available wavelets in the coif family
>>> print pywt.wavelist('coif')
['coif1', 'coif2', 'coif3', 'coif4', 'coif5']
```

Different wavelets have different properties; the most suitable wavelet is dependent on the specific application. See Figure 11.3 for the plots of a couple of additional wavelets.

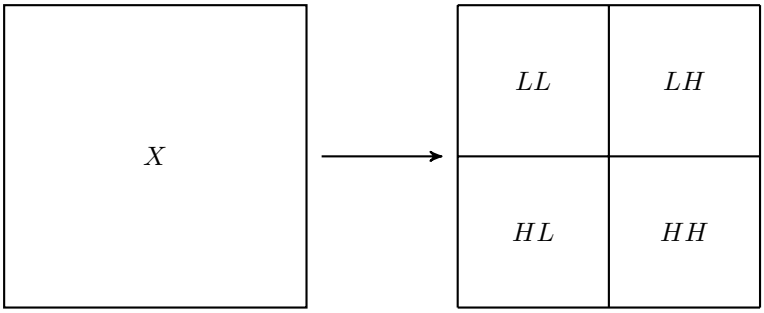

Figure 11.4: The subband pattern for one step in the 2-dimensional wavelet transform.

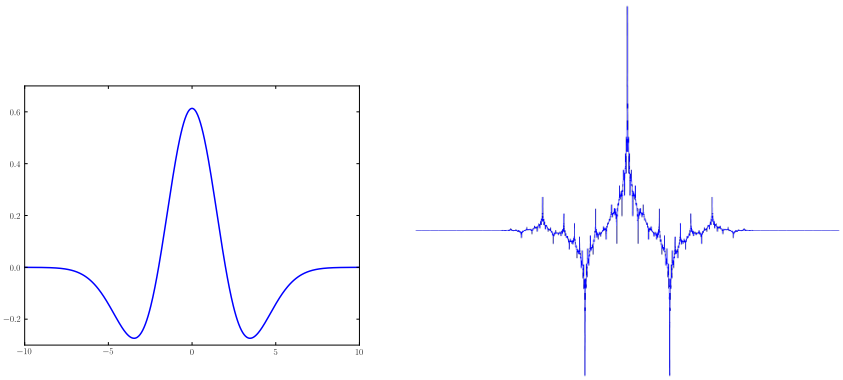

Figure 11.3: Examples of different mother wavelets.

The 2-dimensional Wavelet Transform

We can generalize the wavelet transform for two dimensions much as we generalized the fourier transform. This allows us to perform wavelet analysis on, for example, digital images. In particular, we can calculate the wavelet transform of a two-dimensional array by first transforming the rows, and then the columns of the array.

When implemented as an iterative filterbank, each pass through the filterbank yields an approximation plus three sets of detail coefficients rather than just one. More specifically, if the two-dimensional array X is the input to the filterbank, we obtain arrays LL, LH, HL, and HH, where LL is a smoothed approximation of X and the other three arrays contain wavelet coefficients capturing high-frequency oscillations in vertical, horizontal, and diagonal directions. In the parlance of signal processing, the arrays LL, LH, HL, and HH are called *subbands*. By recursively feeding any or all of the subbands back into the filterbank, we can decompose an input array into a collection of many subbands. This decomposition can be represented schematically by a dyadic partition of a rectangle, called a *subband pattern*. The subband pattern for one pass of the filterbank is shown in Figure 11.4, with a concrete example given in Figure 11.5. The wavelet coefficients that we obtain from a two-dimensional wavelet transform are very useful in a variety of

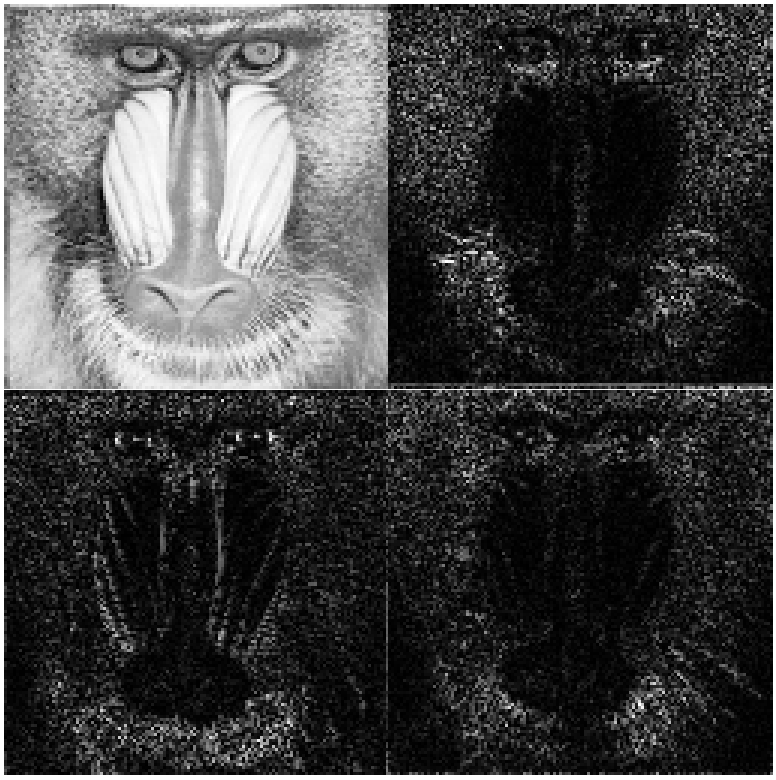

Figure 11.5: Subbands for the Mandrill image after one pass through the filterbank. Note how the upper left subband (LL) is an approximation of the original Mandrill image, while the other three subbands highlight the stark vertical, horizontal, and diagonal changes in the image.
Original image source: `http://sipi.usc.edu/database/`.

image processing tasks. They allow us to analyze and manipulate images in terms of both their frequency and spatial properties, and at differing levels of resolution. Furthermore, wavelet bases often have the remarkable ability to represent images in a very *sparse* manner – that is, most of the image information is captured by a small subset of the wavelet coefficients. This is the key fact for wavelet-based image compression.

PyWavelets provides a simple way to calculate the subbands resulting from one pass through the filterbank.

```
>>> from scipy.misc import imread
>>> fingerprint = imread('finger.pgm')
>>> # use the db4 wavelet with periodic extension
>>> lw = pywt.dwt2(fingerprint, 'db4', mode='per')
```

Note that the `mode` keyword argument determines the type of extension mode (required for the convolution operation). The variable `lw` is a list. The first entry of the list is the LL, or approximation, subband. The second entry of the list is a tuple containing the remaining subbands, LH, HL, and HH (in that order). Plot

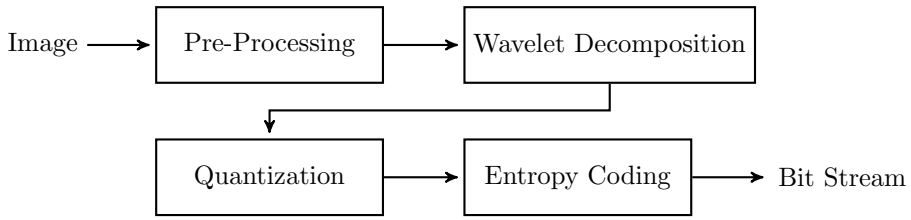

Figure 11.6: Wavelet Image Compression Schematic

these subbands as follows:

```
>>> plt.subplot(221)
>>> plt.imshow(np.abs(lw[0]), cmap=plt.cm.Greys_r, interpolation='none')
>>> plt.subplot(222)
>>> plt.imshow(np.abs(lw[1][0]), cmap=plt.cm.Greys_r, interpolation='none')
>>> plt.subplot(223)
>>> plt.imshow(np.abs(lw[1][1]), cmap=plt.cm.Greys_r, interpolation='none')
>>> plt.subplot(224)
>>> plt.imshow(np.abs(lw[1][2]), cmap=plt.cm.Greys_r, interpolation='none')
>>> plt.show()
```

Compare this with the subbands (of a different image) shown in Figure 11.5.

Image Compression

We now turn to the topic of image compression. Numerous image compression techniques have been developed over the years to reduce the cost of storing large quantities of images. Transform methods based on Fourier and Wavelet analysis have long played an important role in these techniques; for example, the popular JPEG image compression standard is based on the discrete cosine transform. The JPEG2000 compression standard and the FBI Fingerprint Image database, along with other systems, take the wavelet approach.

The general framework for compression is fairly straightforward. First, the image to be compressed undergoes some form of preprocessing, depending on the particular application. Next, the discrete wavelet transform is used to calculate the wavelet coefficients, and these are then *quantized*, i.e. mapped to a set of discrete values (for example, rounding to the nearest integer). The quantized coefficients are then passed through an entropy encoder (such as Huffman Encoding), which reduces the number of bits required to store the coefficients. What remains is a compact stream of bits that can then be saved or transmitted much more efficiently than the original image. All of the above steps are invertible, allowing us to reconstruct the image from the compressed bitstream. See Diagram 11.6.

Lab 12

Gaussian Quadrature

Lab Objective: *Numerical quadrature is an important numerical integration technique. The popular Newton-Cotes quadrature uses uniformly spaced points to approximate the integral, but Gibbs phenomenon prevents Newton-Cotes from being effective for many functions. The Gaussian Quadrature method uses carefully chosen points and weights to mitigate this problem.*

Shifting the Interval of Integration

As with all quadrature methods, we begin by choosing a set of points x_i and weights w_i to approximate an integral.

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i).$$

When we do guassian quadrature, we are required to choose a weight function $W(x)$. This function determines both the $x_i's$ and the $w_i's$. Theoretically, the weight function determines a set of orthogonal polynomials to approximate the function f.

The weight function also determines the interval over which the integration will occur. For example, we choose the weight function as $W(x) = 1$ over $[-1, 1]$ to integrate functions on $[-1, 1]$. To calculate the definite integrate over any interval, we perform a u-substitution. This results in the following formula.

$$\int_a^b f(x)dx = \frac{b-a}{2}\int_{-1}^{1} f(\frac{b-a}{2}z + \frac{a+b}{2})dz.$$

Once we have changed the interval, we may apply quadrature to the integral from -1 to 1 and then scale it appropriately to get the answer we want.

$$\int_a^b f(x)dx \approx \frac{b-a}{2}\sum_i w_i f(\frac{(b-a)}{2}x_i + \frac{(b+a)}{2})$$

Problem 1. Let $f(x) = x^2$ on $[1, 4]$. Then $g(x)$ will be the interval-adjusted version of f on $[-1, 1]$, with $W(x) = 1$, $a = 1$, and $b = 4$. So,

$$\begin{aligned} g(x) &= f(\frac{b-a}{2}x + \frac{b+a}{2}) \\ &= \frac{9}{4}x^2 + \frac{15x}{2} + \frac{25}{4} \end{aligned}$$

and the interval-adjusted integral of $f(x)$ will be

$$\begin{aligned} G(x) &= \frac{b-a}{2}\int f(\frac{b-a}{2}x + \frac{b+a}{2})dx \\ &= \frac{9}{8}x^3 + \frac{45}{8}x^2 + \frac{75}{8}x \end{aligned}$$

Verify that evaluating $G(1) - G(-1) = \int_1^4 f(x)dx$.

Problem 2. Write a function that will accept a function f and an interval $[a, b]$ and return a function g on $[-1, 1]$ that has the same integral (scaled by a constant) as f.

Use your function to plot $f(x) = x^2$ on $[1, 4]$ and the corresponding function $\frac{(b-a)}{2}g$ on $[-1, 1]$. Note that the functions will not look the same plotted, since they are defined over intervals with different lengths, but they integrate to the same value.

Integrating with Given Weights and Points

We now give an example of quadrature with known weights and points. We use the constant weight function $W(x) = 1$ from -1 to 1 (this weight function corresponds to the Legendre polynomials) to calculate the integral of $f(x) = sin(x)$ from $-\pi$ to π, with 5 interpolation points.

First, we change the interval from $[-\pi, \pi]$ to $[-1, 1]$.

```
>>> import numpy as np
>>> a, b = - np.pi, np.pi

# f is the function to integrate.
>>> f = np.sin

# g is the function with the interval changed.
>>> g = lambda x: f((b - a) / 2 * x + (a + b) / 2)
```

The weights(w_i) and points at which f is evaluated (x_i) are given in order in Table 12.1. We put them into an array here.

point x_i	weight w_i
$-\frac{1}{3}\sqrt{5+2\sqrt{\frac{10}{7}}}$	$\frac{322-13\sqrt{70}}{900}$
$-\frac{1}{3}\sqrt{5-2\sqrt{\frac{10}{7}}}$	$\frac{322+13\sqrt{70}}{900}$
0	$\frac{128}{225}$
$\frac{1}{3}\sqrt{5-2\sqrt{\frac{10}{7}}}$	$\frac{322+13\sqrt{70}}{900}$
$\frac{1}{3}\sqrt{5+2\sqrt{\frac{10}{7}}}$	$\frac{322-13\sqrt{70}}{900}$

Table 12.1: Quadrature points and weights on $[-1, 1]$.

```
>>> from math import sqrt
>>> points = np.array([- sqrt(5 + 2 * sqrt(10. / 7)) / 3,
                       - sqrt(5 - 2 * sqrt(10. / 7)) / 3,
                       0,
                       sqrt(5 - 2 * sqrt(10. / 7)) / 3,
                       sqrt(5 + 2 * sqrt(10. / 7)) / 3])
>>> weights = np.array([(322 - 13 * sqrt(70)) / 900,
                        (322 + 13 * sqrt(70)) / 900,
                        128. / 225,
                        (322 + 13 * sqrt(70)) / 900,
                        (322 - 13 * sqrt(70)) / 900])
```

We now calculate the integral

```
>>> integral = (b - a)/2 * np.inner(weights, g(points))
```

Problem 3. Write a function that accepts a function f, an array of points, an array of weights, and limits of integration and returns the integral. Don't forget to adjust the interval as in the above example.

Calculating Weights and Points

Calculating an integral when the weights and points are given is straightforward. But, how are these weights and points found? There are many publications that will give tables of points for various weight functions. We will demonstrate how to find such a list using the Golub-Welsch algorithm.

The Golub-Welsch Algorithm

This Golub-Welsch algorithm builds a tri-diagonal matrix and finds its eigenvalues. These eigenvalues are the points at which a function is evaluated for Guassian quadrature. The weights are the length of $[a, b]$ times the first coordinate of each

eigenvector squared. We note that finding eigenvalues for a tridiagonal matrix is a well conditioned, relatively painless problem. Using a good eigenvalue solver gives the Golub-Welsch algorithm a complexity of $O(n^2)$. A full treatment of the Golub-Welsch algorithm may be found at `http://gubner.ece.wisc.edu/gaussquad.pdf`.

We mentioned that the choice of weight function corresponds to a class of orthogonal polyomials. An important fact about orthogonal polynomials is that any set of orthogonal polynomials $\{u_i\}_{i=1}^N$ satisfies a three term recurrence relation

$$u_i(x) = (\gamma_{i-1}x - \alpha_i)u_{i-1}(x) - \beta_i u_{i-2}(x)$$

where $u_{-1}(x) = 0$ and $u_0(x) = 1$. The coefficients $\{\gamma_k, \alpha_i, \beta_i\}$ have been calculated for several classes of orthogonal polynomials, and may be determined for an arbitrary class using the procedure found in "Calculation of Gauss Quadrature Rules" by Golub and Welsch. Using these coefficients we may create a tri-diagonal matrix

$$J = \begin{bmatrix} a_1 & b_1 & 0 & 0 & \dots & 0 \\ b_1 & a_2 & b_2 & 0 & \dots & 0 \\ 0 & b_2 & a_3 & b_3 & \dots & 0 \\ \vdots & & & & & \vdots \\ \vdots & & & & & \vdots \\ 0 & \dots & & & & b_{N-1} \\ 0 & \dots & & & b_{N-1} & a_N \end{bmatrix}$$

Where $a_i = \frac{-\beta_i}{\alpha_i}$ and $b_i = (\frac{\gamma_{i+1}}{\alpha_i \alpha_{i+1}})^{\frac{1}{2}}$. This matrix is called the Jacobi matrix. The eigenvalues of this matrix give us the points x_i and the length of $[a, b]$ times the squares of the first entries of the corresponding eigenvectors gives the weights.

Problem 4. Write a function that will accept three arrays representing the coefficients $\{\gamma_i, \alpha_i, \beta_i\}$ from the recurrence relation above and return the Jacobi matrix.

Problem 5. The coefficients of the Legendre polynomials (which correspond to the weight function $W(x) = 1$ on $[-1, 1]$ are given by

$$\alpha_i = \frac{2i-1}{i} \qquad \beta_i = 0 \qquad \gamma_i = \frac{i-1}{i}$$

Write a function that accepts an integer n representing the number of points to use in the quadrature. Calculate α, β, and γ as above, calculate the Jacobi matrix, then use it to find the points x_i and weights w_i that correspond to this weight function. When $n = 5$, do they match the ones given in the first part of this lab?

Problem 6. Write a new function that accepts a function f, bounds a and b, and n for the number of points to use. Use the previously defined functions to estimate $\int_a^b f(x)dx$ using the coefficients of the Legendre polynomials.

This completes our implementation of the Gaussian Quadrature for a particular set orthogonal polynomials.

scipy.integrate

There are other techniques for finding the weights and points for a given weighting function. This is, in fact, not even the fastest method. In general practice, we use `scipy.integrate` to calculate integrals. `scipy.integrate.quadrature` offers a reasonably fast Gaussian quadrature implementation.

Another common hallmark of quadrature is that it can be used adaptively. It is common in practice to refine the points of a quadrature estimate on an interval where a function is observed to be changing rapidly. This allows for more accurate computation at a relatively low computational cost. This is the approach used by the function `scipy.integrate.quad`.

Problem 7. The standard normal distribution is an important object of study in probability and statistic. It is defined by the probability density function $p(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$ (here we are assuming a mean of 0 and a variance of 1). This is a function that cannot be integrated symbollically.

The probability that a normally distributed random variable X will take on a value less than (or equal to) a given value x is

$$P(X \leq x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt$$

This function is essentially zero for values of x that lie reasonably far from the mean, so we can estimate this probability by integrating from -5 to x instead of from $-\infty$ to x.

Write a function that uses `scipy.integrate.quad` to estimate the probability that this normally distributed random variable will take a value less than a given number x that lies relatively close to the mean. You can test your result at $x = 1$ by comparing it with the following code:

```
from scipy.stats import norm
N = norm()
N.cdf(1)
```