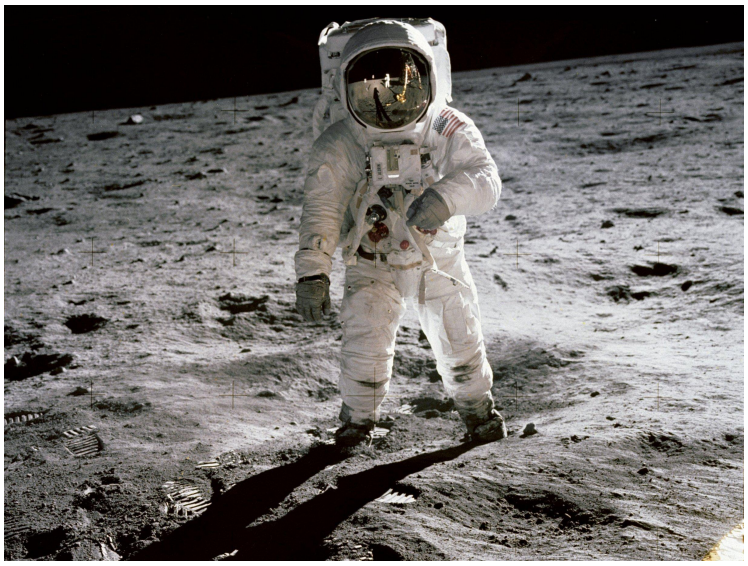
Labs for Foundations of Applied Mathematics

Python Essentials



List of Contributors

E. Evans
Brigham Young University

R. Evans
Brigham Young University

J. Grout
Drake University

J. Humpherys
Brigham Young University

T. Jarvis
Brigham Young University

J. Whitehead
Brigham Young University

J. Adams
Brigham Young University

J. Bejarano
Brigham Young University

Z. Boyd
Brigham Young University

M. Brown
Brigham Young University

T. Christensen
Brigham Young University

M. Cook
Brigham Young University

R. Dorff
Brigham Young University

B. Ehlert
Brigham Young University

M. Fabiano
Brigham Young University

A. Frandsen
Brigham Young University

K. Finlinson
Brigham Young University

J. Fisher
Brigham Young University

R. Fuhriman
Brigham Young University

S. Giddens
Brigham Young University

C. Gigena
Brigham Young University

M. Graham
Brigham Young University

F. Glines
Brigham Young University

M. Goodwin
Brigham Young University

R. Grout
Brigham Young University

J. Hendricks
Brigham Young University

A. Henriksen
Brigham Young University

I. Henriksen
Brigham Young University

C. Hettinger
Brigham Young University

S. Horst
Brigham Young University

K. Jacobson
Brigham Young University

J. Leete
Brigham Young University

J. Lytle
Brigham Young University

R. McMurray
Brigham Young University

S. McQuarrie
Brigham Young University

J. Morrise
Brigham Young University

M. Morrise
Brigham Young University

A. Morrow
Brigham Young University

R. Murray
Brigham Young University

J. Nelson
Brigham Young University

M. Proudfoot
Brigham Young University

D. Reber
Brigham Young University

C. Robertson
Brigham Young University

R. Sandberg
Brigham Young University

J. Stewart
Brigham Young University

S. Suggs
Brigham Young University

T. Thompson
Brigham Young University

M. Victors
Brigham Young University

J. Webb
Brigham Young University

R. Webb
Brigham Young University

J. West
Brigham Young University

A. Zaitzeff
Brigham Young University

Preface

This lab manual is designed to accompany the textbook *Foundations of Applied Mathematics* by Humpherys, Jarvis and Evans.

Contents

Preface **iii**

I Labs **1**

1 Introduction to Python **3**

2 The Standard Library **23**

3 Object-Oriented Programming **41**

4 Introduction to NumPy **55**

5 Introduction to Matplotlib **73**

6 Exceptions and File I/O **89**

7 Data Visualization **101**

8 Testing **119**

9 Profiling **133**

II Appendices **147**

A Initial Installation **149**

B Installing and Updating Python Packages **153**

C NumPy Visual Guide **157**

D Matplotlib Customization **161**

Part I

Labs

1 Introduction to Python

Lab Objective: *Python is a powerful general-purpose programming language. It can also be used interactively, allowing for very rapid development. Python has many powerful scientific computing tools, making it an ideal language for applied and computational mathematics.*

In this introductory lab we introduce Python syntax, data types, functions, and control flow tools. These Python basics are an essential part of almost every problem you will solve and almost every program you will write.

Getting Started

Python is quickly gaining momentum as a fundamental tool in scientific computing. It is simple, readable, and easy to learn. *Anaconda* is a free distribution service by Continuum Analytics, Inc., that includes the cross-platform Python *interpreter* (the software that actually executes Python code) and many Python libraries that are commonly used for applied and computational mathematics. To install Python via Anaconda, go to `http://continuum.io/downloads`, download the installer corresponding to your operating system, and follow the on-screen instructions.

Use the installer corresponding to **Python 2.7**. Although later versions of Python are available, they do not yet have many of the libraries and features that we require in this curriculum. For more information on installing Python and various libraries, see Appendices A and B.

Running Python

Python files are saved with a `.py` extension. For beginners, we strongly recommend using a simple text editor[1] for writing Python files, though many free IDEs (Integrated Development Environments—large applications that facilitate code development with some sophisticated tools) are also compatible with Python. For now, the simpler the coding environment, the better.

A plain Python file looks similar to the code on the following page:

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

[1]Some of the more popular text editors are described in Appendix A.

```
if __name__ == "__main__":
    pass                            # 'pass' is a temporary placeholder.
```

The # character creates a single-line *comment*. Comments are ignored by the interpreter and serve as annotations for the accompanying source code. A pair of three quotes, """ """ or ''' ''', creates a multi-line string literal, which may be used as a multi-line comment.

A triple-quoted string literal at the top of the file serves as the *header* for the file. The header typically includes the author, instructions on how to use the file, and so on. Executable Python code comes after the header.

Finally, each statement under the `if __name__ == "__main__"` clause is executed when the file is run. Any "orphan" statements that are not placed under this clause are also executed when the file is run, but it is generally considered bad practice to leave any statements outside of function definitions or under the clause.

Problem 1. Create a new Python file and save it to your file system. Write the file name and a brief header at the top, then add the following code:

```
if __name__ == "__main__":
    print("Hello, world!")
```

Open a Unix shell (often called a *terminal* or a *command line*) and navigate to the directory where the new file is saved. Use the command `ls` to list the files in the current directory, `pwd` to print the working directory, and `cd` to change directories. Below, the `$` character indicates that the command is being run from the shell.

```
$ pwd
/Users/Guest
$ ls
Desktop     Documents       Downloads       Pictures        Music
$ cd Documents
$ pwd
/Users/Guest/Documents
$ ls
python_intro.py
```

Now that we are in the same directory as the Python file, we can execute it using the following command:

```
$ python python_intro.py
```

If "Hello, world!" is displayed on the screen, you have just successfully executed your first Python program!

IPython

Python can be run interactively using several interfaces. The most basic of these is the Python interpreter. In this and subsequent labs, the triple brackets >>> indicate that the given code is being executed one line at a time via the Python interpreter.

```
$ python                            # Start the Python interpreter.
>>> print("This is plain Python.")  # Execute some code.
This is plain Python.
```

There are, however, more useful interfaces. Chief among these is *IPython*. To execute a script in IPython, use the `run` command.

```
>>> quit()                          # Exit the Python interpreter.
$ ipython                           # Start IPython.

In [1]: print("This is IPython!")   # Execute some code.
This is IPython!

In [2]: run python_intro.py         # Run a particular Python script.
Hello, world!
```

One of the biggest advantages of IPython is that it supports *object introspection*, whereas the regular Python interpreter does not. Object introspection allows you to quickly see all methods and attributes associated with an object. IPython also has a built-in `help()` function that provides interactive help.

```
# A list is a basic Python data structure. To see the methods associated with
# a list, type the object name (list), followed by a period, and press tab.
In [1]: list.   # Press 'tab'.
list.append   list.extend   list.insert   list.remove      list.sort
list.count    list.index    list.pop      list.reverse

# To learn more about a specific method, use "?".
In [1]: list.append?
Docstring:    L.append(object) -- append object to end
Type:         method_descriptor

In [2]: help()                      # Start IPython's interactive help utility.

help> list                          # Get documentation on the list class.
Help on class list in module __builtin__:

class list(object)
 |  list() -> new empty list
 |  list(iterable) -> new list initialized from iterable's items
 |  ...                             # Press 'q' to exit the info screen.

help> quit                          # End the interactive help session.
```

Use IPython side-by-side with a text editor to test syntax and small code snippets. Testing code in IPython *before* putting it into a program helps you catch errors beforehand and greatly speeds up the coding process.

If you cannot answer a coding question using these strategies, search the Internet. In particular, stackoverflow.com is often a valuable resource for answering common programming questions.

Python Basics

Arithmetic

We can now dive into the particulars of Python syntax. To begin, Python can be used as a calculator. Use `**` for exponentiation and `%` for modular division.

```
>>> 3**2 + 2*5                      # Python obeys the order of operations.
19

>>> 13 % 3                          # The modulo operator % calculates the
1                                   # remainder: 13 = (3*4) + 1.
```

In most Python interpreters, the underscore character `_` is a variable with the value of the previous command's output, like the ANS button on many calculators.

```
>>> 12 * 3
36
>>> _ / 4
9
```

Data comparisons like `<` and `>` act as expected. The `==` operator checks for numerical equality and the `<=` and `>=` operators correspond to $\leq$ and $\geq$, respectively.

To connect multiple boolean expressions, use the operators `and`, `or`, and `not`.[2]

```
>>> 3 > 2
True
>>> 4 < 4
False
>>> 1 <= 1 or 2 > 3
True
>>> 7 == 7 and not 4 < 4
True
```

Variables

Variables are used to temporarily store data. A *single* equals sign assigns one or more values (on the right) to one or more variable names (on the left). A *double* equals sign is a comparison operator, as in the previous code block.

[2] In many other programming languages, the `and`, `or`, and `not` operators are written as `&&`, `||`, and `!`, respectively. Python's convention is much more readable and does not require parentheses.

Unlike many programming languages, Python does not require a variable's data type to be specified upon initialization.

```
>>> x = 12                          # Initialize x with the integer 12.
>>> y = 2 * 6                       # Initialize y with the integer 2*6 = 12.
>>> x == y                          # Compare the two variable values.
True

>>> x, y = 2, 4                     # Give both x and y new values in one line.
>>> x == y
False
```

Functions

To define a function, use the `def` keyword followed by the function name, a parenthesized list of formal parameters, and a colon. Then indent the function body using exactly **four** spaces.

```
# Indent the lines of a function with four spaces.
>>> def add(x, y):
...     result = x + y
...     return result
```

> ACHTUNG!
>
> Many other languages use curly braces to delimit blocks, but Python uses whitespace indentation. Mixing tabs and spaces confuses the interpreter and causes problems. Most text editors will allow you to set the indentation type to spaces so you can use the tab key on your keyboard to insert four spaces.

Functions are defined with *parameters* and called with *arguments*, though the terms are often used interchangeably. Below, `width` and `height` are parameters for the function `area()`. The values `2` and `5` are the arguments that are passed when calling the function.

```
>>> def area(width, height):
...     return width * height
...
>>> area(2, 5)
10
```

It is also possible to specify default values for a function's formal parameters. In the following example, the function `printer()` has three formal parameters, and the value of `c` defaults to 0. That is, if it is not specified in the function call, the variable `c` will contain the value 0 when the function is executed.

```
>>> def printer(a, b, c=0):
...     print a, b, c
```

```
...
>>> printer(1, 2, 3)                # Specify each parameter.
1 2 3
>>> printer(1, 2)                   # Specify only non-default parameters.
1 2 0
```

Arguments are passed to functions based on position or name, and positional arguments must be defined before named arguments. For example, `a` and `b` must come before `c` in the function definition of `printer()`. Examine the following code blocks demonstrating how positional and named arguments are used to call a function.

```
# Try defining printer with a named argument before a positional argument.
>>> def printer(c=0, a, b):
...     print a, b, c
...
SyntaxError: non-default argument follows default argument
```

```
# Correctly define printer() with the named argument after positional arguments↩
    .
>>> def printer(a, b, c=0):
...     print a, b, c
...

# Call printer() with 3 positional arguments.
>>> printer(2, 4, 6)
2 4 6

# Call printer() with 3 named arguments. Note the change in order.
>>> printer(b=3, c=5, a=7)
7 3 5

# Call printer() with 2 named arguments, excluding c.
>>> printer(b=1, a=2)
2 1 0

# Call printer() with 1 positional argument and 2 named arguments.
>>> printer(1, c=2, b=3)
1 3 2
```

Python functions can return more than one value. Simply separate the values by commas after the `return` statement.

```
>>> def arithmetic(a, b):
...     difference = a - b
...     product = a * b
...     return difference, product
...
>>> x, y = arithmetic(5, 2)         # x is the difference, y is the product.
```

```
>>> print x, y
3 10
```

Note

The functions `area()` and `printer()` defined above have an important difference: `area()` ends with a `return` statement, but `printer()` lacks a `return` statement.

The `return` statement instantly ends the function call and passes the return value to the function caller. The `print` statement does nothing more than display the value of a given object (or objects) in the terminal. A function without a return statement implicitly returns the Python constant `None`, which is similar to the special value `null` of many other languages.

```
>>> x = area(2, 4)              # Calling area() doesn't print anything,
>>> y = printer(1, 2, 3)        # but calling printer() does.
1 2 3

>>> print(x)                    # However, x now contains a value,
8
>>> print(y)                    # whereas y only contains 'None'.
None
```

Problem 2. In your Python file from Problem 1, define a function called `sphere_volume()`. This function should accept a single parameter `r` and return (not print) the volume of the sphere of radius `r`. For now, use 3.14159 as an approximation for π.

To test your function, call it under the `if __name__ == "__main__"` clause and print the returned value. Run your file to see if your answer is what you expect it to be.

Data Types

Numerical Types

Python has four numerical data types: `int`, `long`, `float`, and `complex`. Each stores a different kind of number. The built-in function `type()` identifies an object's data type.

```
>>> type(3)                     # Numbers without periods are integers.
int

>>> type(3.0)                   # Floats have periods (3. is also a float).
float
```

Python has two types of division: integer and float. Integer division rounds the the result down to the next integer, and float division performs true, fractional division. Both are executed with the / operator. When one or both of the operands are non-integers, / performs float division. If both operands are integers, integer division is performed.

```
>>> 15.0 / 4.0                    # Float division performs as expected.
3.75
>>> 15 / 4                        # Integer division rounds the result to 3.
3
```

Achtung!

Using integer division unintentionally is an incredibly common programming mistake. To avoid this, cast at least one operand as a float before dividing.

```
>>> x, y = 7, 5
>>> x / y
1
>>> float(x) / y
1.4
>>> x / float(y)
1.4
```

Python also supports complex number computations. Use the letter j, not i, for the imaginary part.

```
>>> x = complex(2,3)              # Create a complex number this way...
>>> y = 4 + 5j                    # ...or this way, using j (not i).
>>> x.real                        # Access the real part of x.
2
>>> y.imag                        # Access the imaginary part of y.
5
```

Strings

In Python, strings are created with either single or double quotes. To concatenate two or more strings, use the + operator between string variables or literals.

```
>>> str1 = "Hello"
>>> str2 = 'world'
>>> my_string = str1 + " " + str2 + '!'
>>> my_string
'Hello world!'
```

We can access parts of a string using *slicing*, indicated by square brackets `[ ]`. Slicing syntax is `[start:stop:step]`. The parameters `start` and `stop` default to the beginning and end of the string, respectively. The parameter `step` defaults to 1.

```
>>> my_string = "Hello world!"
>>> my_string[4]                # Indexing begins at 0.
'o'
>>> my_string[-1]               # Negative indices count backward from the end.
'!'

# Slice from the 0th to the 5th character (not including the 5th character).
>>> my_string[:5]
'Hello'

# Slice from the 6th character to the end.
>>> my_string[6:]
'world!'

# Slice from the 3rd to the 8th character (not including the 8th character).
>>> my_string[3:8]
'lo wo'

# Get every other character in the string.
>>> my_string[::2]
'Hlowrd'
```

Problem 3. Write two new functions, called `first_half()` and `backward()`.

1. `first_half()` should accept a parameter and return the first half of it, excluding the middle character if there is an odd number of characters.
   (Hint: the built-in function `len()` returns the length of the input.)

2. The `backward()` function should accept a parameter and reverse the order of its characters using slicing, then return the reversed string.
   (Hint: The `step` parameter used in slicing can be negative.)

Use IPython to quickly test your syntax for each function.

Lists

A Python `list` is created by enclosing comma-separated values with square brackets `[ ]`. Entries of a list do *not* have to be of the same type. Access entries in a list with the same indexing or slicing operations used with strings.

```
>>> my_list = ["Hello", 93.8, "world", 10]
>>> my_list
['Hello', 93.8, 'world!', 10]
```

```
>>> my_list[0]
'Hello'
>>> my_list[-2]
'world!'
>>> my_list[:2]
['Hello', 93.8]
```

Common list methods (functions) include `append()`, `insert()`, `remove()`, and `pop()`. Consult IPython for details on each of these methods using object introspection.

```
>>> my_list = [1, 2]                # Create a simple list of two integers.
>>> my_list.append(4)               # Append the integer 4 to the end.
>>> my_list.insert(2, 3)            # Insert 3 at location 2.
>>> my_list
[1, 2, 3, 4]
>>> my_list.remove(3)               # Remove 3 from the list.
>>> my_list.pop()                   # Remove (and return) the last entry.
4
>>> my_list
[1, 2]
```

Slicing is also very useful for replacing values in a list.

```
>>> my_list = [10, 20, 30, 40, 50]
>>> my_list[0] = -1
>>> my_list[3:] = [8, 9]
[-1, 20, 30, 8, 9]
```

The `in` operator quickly checks if a given value is in a list (or a string).

```
>>> my_list = [1, 2, 3, 4, 5]
>>> 2 in my_list
True
>>> 6 in my_list
False
```

Tuples

A Python `tuple` is an ordered collection of elements, created by enclosing comma-separated values with parentheses `(` and `)`. Tuples are similar to lists, but they are much more rigid, have less built-in operations, and cannot be altered after creation. Lists are therefore preferable for managing dynamic ordered collections of objects.

When multiple objects are returned in by a function, they are returned as a tuple. For example, recall that the `arithmetic()` function returns two values.

```
>>> x, y = arithmetic(5,2)                    # Get each value individually,
>>> print x, y
3 10
```

```
>>> both = arithmetic(5,2)                    # or get them both as a tuple.
>>> print(both)
(3, 10)
```

Problem 4. Write a function called `list_ops()`. Define a list with the entries `"bear"`, `"ant"`, `"dog"`, and `"cat"`, in that order. Then perform the following operations on the list:

1. Append `"eagle"`.
2. Replace the entry at index 2 with `"fox"`.
3. Remove the entry at index 1.
4. Sort the list in reverse alphabetical order.

Return the resulting list.

Sets

A Python `set` is an unordered collection of distinct objects. Objects can be added to or removed from a set after its creation. Initialize a set with curly braces `{ }`, separating the values by commas, or use `set()` to create an empty set.

```
# Initialize some sets. Note that repeats are not added.
>>> gym_members = {"Doe, John", "Doe, John", "Smith, Jane", "Brown, Bob"}
>>> print(gym_members)
set(['Smith, Jane', 'Brown, Bob', 'Doe, John'])

>>> gym_members.add("Lytle, Josh")      # Add an object to the set.
>>> gym_members.discard("Doe, John")    # Delete an object from the set.
>>> print(gym_members)
set(['Lytle, Josh', 'Smith, Jane', 'Brown, Bob'])
```

Like mathematical sets, Python sets have operations like union and intersection.

```
>>> gym_members.intersection({"Lytle, Josh", "Henriksen, Amelia" "Webb, Jared"↩
    })
set(['Lytle, Josh'])
```

Dictionaries

Like a set, a Python `dict` (dictionary) is an unordered data type. A dictionary stores key-value pairs, called *items*. The values of a dictionary are indexed by its keys. Dictionaries are initialized with curly braces, colons, and commas. Use `dict()` or `{}` to create an empty dictionary.

```
>>> my_dictionary = {"business": 4121, "math": 2061, "visual arts": 7321}
>>> print(my_dictionary["math"])
```

```
2061

# Add a value indexed by 'science' and delete the 'business' keypair.
>>> my_dictionary["science"] = 6284
>>> my_dictionary.pop("business")       # Use 'pop' or 'popitem' to remove.
4121
>>> print(my_dictionary)
{'visual arts': 7321, 'math': 2016, 'science': 6284}

# Display the keys and values.
>>> my_dictionary.keys()
['visual arts', 'math', 'science']
>>> my_dictionary.values()
[7321, 2016, 6284]
```

The keys of a dictionary must be *immutable*, which means that they must be objects that cannot be modified after creation. We will discuss mutability more thoroughly in the next lab.

Type Casting

The names of each of Python's data types can be used as functions to cast a value as that type. This is particularly useful with integers and floats.

```
# Cast numerical values as different kinds of numerical values.
>>> x = int(3.0)
>>> y = float(3)
>>> z = complex(3)
>>> print x, y, z
3, 3.0, (3+0j)

# Cast a list as a set and vice versa.
>>> a = set([1, 2, 3, 4, 4])
>>> b = list({'a', 'a', 'b', 'b', 'c'})
>>> print(a)
set([1, 2, 3, 4])
>>> print(b)
['a', 'b', 'c']

# Cast other objects as strings.
>>> s = str(['a', str(1), 'b', float(2)])
>>> t = str(list(set([complex(float(3))])))
>>> s
"['a', '1', 'b', 2.0]"
>>> t
'[(3+0j)]'
```

Control Flow Tools

Control flow blocks dictate the order in which code is executed. Python supports the usual control flow statements including `if` statements, `while` loops and `for` loops.

The If Statement

An `if` statement executes the indented code *if* (and only if) the given condition holds. The `elif` statement is short for "else if" and can be used multiple times following an if statement, or not at all. The `else` keyword may be used at most once at the end of a series of `if`/`elif` statements.

```
>>> food = "bagel"
>>> if food == "apple":              # As with functions, the colon denotes
...     print("72 calories")         # the start of each code block.
... elif food == "banana" or food == "carrot":
...     print("105 calories")
... else:
...     print("calorie count unavailable")
...
calorie count unavailable
```

Problem 5. Write a function called `pig_latin()`. Accept a parameter `word`, translate it into Pig Latin, then return the translation. Specifically, if `word` starts with a vowel, add "hay" to the end; if `word` starts with a consonant, take the first character of `word`, move it to the end, and add "ay".
(Hint: use the `in` operator to check if the first letter is a vowel.)

The While Loop

A `while` loop executes an indented block of code *while* the given condition holds.

```
>>> i = 0
>>> while i < 10:
...     print(i),
...     i = i+1
...
0 1 2 3 4 5 6 7 8 9
```

There are two additional useful statements to use inside of loops:

1. The `break` statement manually exits the loop, regardless of which iteration the loop is on or if the termination condition is met.

2. The `continue` statement skips the current iteration and returns to the top of the loop block if the termination condition is still not met.

```
>>> i = 0
>>> while True:
...     print(i),
...     i += 1
...     if i >= 10:
...         break                   # Exit the loop.
...
0 1 2 3 4 5 6 7 8 9

>>> i = 0
>>> while i < 10:
...     i += 1
...     if i % 3 == 0:
...         continue                # Skip multiples of 3.
...     print(i),
1 2 4 5 7 8 10
```

The For Loop

A `for` loop iterates over the items in any *iterable*. Iterables include (but are not limited to) strings, lists, sets, and dictionaries.

```
>>> colors = ["red", "green", "blue", "yellow"]
>>> for entry in colors:
...     print(entry + "!")
...
red!
green!
blue!
yellow!
```

The `break` and `continue` statements also work in for loops, but a `continue` in a for loop will automatically increment the index or item, whereas a `continue` in a while loop makes no automatic changes to any variable.

In addition, Python has some very useful built-in functions that can be used in conjunction with the `for` statement:

1. `range(start, stop, step)`: Produces a list of integers, following slicing syntax. If only one argument is specified, it produces a list of integers from 0 to the argument, incrementing by one. In Python 2.7, we use `xrange()` more often because it is faster than `range()` and does not explicitly create a list in memory.

2. `zip()`: Joins multiple sequences so they can be iterated over simultaneously.

3. `enumerate()`: Yields both a count and a value from the sequence. Typically used to get both the index of an item and the actual item simultaneously.

4. `sorted()`: Returns a new list of sorted items that can then be used for iteration.

5. `reversed()`: Reverses the order of the iteration.

```
>>> vowels = "aeiou"
>>> colors = ["red", "yellow", "white", "blue", "purple"]
>>> for i in xrange(5):
...     print i, vowels[i], colors[i]
...
0 a red
1 e yellow
2 i white
3 o blue
4 u purple

>>> for letter, word in zip(vowels, colors):
...     print letter, word
...
a red
e yellow
i white
o blue
u purple

>>> for i, color in enumerate(colors):
...     print i, color
...
0 red
1 yellow
2 white
3 blue
4 purple

>>> for item in sorted(colors):
...     print item,
...
blue purple red white yellow

>>> for item in reversed(colors):
...     print item,
...
yellow white red purple blue

# Range arguments follow slicing syntax.
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(4, 8)
[4, 5, 6, 7]
>>> range(2, 20, 3)
[2, 5, 8, 11, 14, 17]
```

Problem 6. A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is $9009 = 91 \times 99$.
Write a function called `palindrome()` that finds and returns the largest palindromic number made from the product of two 3-digit numbers.

This problem originates from `https://projecteuler.net`, an excellent resource for math-related coding problems.

List Comprehension

A *list comprehension* uses for loop syntax between square brackets to create a list. This is a powerful, efficient way to build lists. The code is concise and runs quickly.

```
>>> [float(n) for n in range(5)]
[0.0, 1.0, 2.0, 3.0, 4.0]
```

List comprehensions can be thought of as "inverted loops", meaning that the body of the loop comes before the looping condition. The following loop and list comprehension produce the same list, but the list comprehension takes only about two-thirds the time to execute.

```
>>> loop_output = []
>>> for i in range(5):
...     loop_output.append(i**2)
...
>>> list_output = [i**2 for i in range(5)]
```

Tuple, set, and dictionary comprehensions can be done in the same way as list comprehensions by using the appropriate style of brackets on the end.

```
>>> colors = ["red", "blue", "yellow"]
>>> {c[0]:c for c in colors}
{'y': 'yellow', 'r': 'red', 'b': 'blue'}

>>> {"bright " + c for c in colors}
set(['bright blue', 'bright red', 'bright yellow'])
```

Problem 7. The alternating harmonic series is given by the following:

$$\sum_{n=1}^{\infty} \frac{(-1)^{(n+1)}}{n} = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \ldots = \ln(2)$$

Write a function called `alt_harmonic()`. Use a list comprehension to quickly compute the first n terms of this series, given the parameter `n`. The sum of the first 500,000 terms of this series approximates $\ln(2)$ to five decimal places.
(Hint: consider using Python's built-in `sum()` function.)

Refer back to this and other introductory labs often as you continue getting used to Python syntax and data types. As you continue your study of Python, we strongly recommend the following readings:

1. Chapters 3, 4, and 5 of the Official Python Tutorial
   (`http://docs.python.org/2.7/tutorial/introduction.html`).

2. Section 1.2 of the SciPy Lecture Notes
   (`http://scipy-lectures.github.io/`).

3. PEP8 - Python Style Guide
   (`http://www.python.org/dev/peps/pep-0008/`).

Additional Material

Lambda Functions

The keyword `lambda` is a shortcut for creating one-line functions. For example, we can create Python functions for the polynomials $f(x) = 6x^3 + 4x^2 - x + 3$ and $g(x, y, z) = x + y^2 - z^3$ using only a single line for each definition.

```
>>> f = lambda x: 6*x**3 + 4*x**2 - x + 3
>>> g = lambda x, y, z: x + y**2 - z**3
```

Generalized Function Input

On rare occasion, it is necessary to define a function without knowing exactly what the parameters will be like or how many there will be. This is usually done by defining the function with the parameters `*args` and `**kwargs`. Here `*args` is a list of the positional arguments and `**kwargs` is a dictionary mapping the keywords to their argument. This is the most general form of a function definition.

```
>>> def report(*args, **kwargs):
...     for i, arg in enumerate(args):
...         print "Argument " + str(i) + ":", arg
...     for key in kwargs:
...         print "Keyword", key, "->", kwargs[key]
...
>>> report("TK", 421, exceptional=False, missing=True)
Argument 0: TK
Argument 1: 421
Keyword missing -> True
Keyword exceptional -> False

>>> average = lambda *args: sum(args) / float(len(args))
>>> average(4, 5, 6)
5.0
>>> average(1, 2, 3, 4, 5, 6, 7, 8, 9)
5.0
```

See `https://docs.python.org/2.7/tutorial/controlflow.html` for more on this topic.

Function Decorators

A *function decorator* is a special function that "wraps" other functions. It takes in a function as input, then usually pre-processes its inputs or post-processes its outputs.

```
>>> def typewriter(func):
...     """Decorator for printing the type of output a function returns"""
...     def wrapper(*args, **kwargs):
...         output = func(*args, **kwargs)      # Call the decorated function.
```

```
...         print "output type:", type(output)  # Do something before finishing↩
    .
...         return output                       # Return the function output.
...     return wrapper
```

The outer function, `typewriter()`, returns the new function `wrapper()`. Since `wrapper()` accepts `*args` and `**kwargs` as arguments, the input function `func()` could accepts any number of positional or keyword arguments.

Apply a decorator to a function by tagging the function's definition with an @ symbol and the decorator name.

```
>>> @typewriter
... def combine(a, b, c):
...     return a*b/float(c)
...
```

Placing the @ tag above the definition is equivalent to adding the following line of code after the function definition:

```
>>> combine = typewriter(combine)
```

Now when we call `combine()`, we are really calling `wrapper()`, which then calls the originally defined `combine()`.

```
>>> combine(3, 4, 6)
output type: <type 'float'>
2.0
```

Function decorators can also accept arguments. This requires another level of nesting: the outermost function must define and return a decorator that defines and returns a wrapper.

```
>>> def repeat(times=2):
...     """Decorator for calling a function several times."""
...     def decorator(func):
...         def wrapper(*args, **kwargs):
...             for _ in xrange(times):
...                 output = func(*args, **kwargs)
...             return output
...         return wrapper
...     return decorator
...
>>> @repeat(times=3)
... def hello_world():
...     print "Hello, world!"
...
>>> hello_world()
Hello, world!
Hello, world!
Hello, world!
```

See `https://www.python.org/dev/peps/pep-0318/` for more details.

2 The Standard Library

Lab Objective: *Python is designed to make it easy to implement complex tasks with little code. To that end, every 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 with code that can be imported is called a* module*. All Python distributions include a collection of modules for accomplishing a variety of tasks, collectively called the* Python Standard Library*. In this lab we explore some built-in functions, learn how to create, import, and use modules, and become familiar with the standard library.*

Built-in Functions

Every Python installation comes with several built-in functions that may be used at any time. IPython's object introspection feature makes it easy to learn about these functions. Start IPython from the command line 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
```

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.
`round()`	A float rounded to a given precision in decimal digits.
`sum()`	The sum of a sequence of numbers.

Table 2.1: Some common built-in functions for numerical calculations.

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

# min() and max() can be used on a list, string, or several arguments.
# String characters are ordered lexicographically.
>>> print min([4, 2, 6]), min("aXbYcZ"), min(1, 'a', 'A')
2 X 1
>>> 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.
>>> print len([2, 7, 1]), len("abcdef"), len({1, 'a', 'a'})
3 6 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

# round() is useful for formatting data to be printed.
>>> round(3.14159265358979323, 2)
3.14
```

More detailed documentation on all of Python's built-in functions can be found at `https://docs.python.org/2/library/functions.html`.

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

Names

All Python objects reside somewhere in computer memory. These objects may be numbers, data structures, functions, or any other sort of Python object. A *name* (or variable) 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 a name to an object. If a name is assigned to another name, that new name refers to the same object that the original name refers to (or a copy of it—see the next section).

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

To see all of the names in the current namespace, use the built-in function `dir()`. To delete a variable from the namespace, use the `del` keyword (be careful!).

```
>>> subjects = ["Statistics", "Technology", "Engineering", "Mathematics"]
>>> num = 4
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'num', 'subjects']
>>> del num
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'subjects']
>>> print(num)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'num' is not defined
```

> **Note**
>
> Many programming languages distinguish between *variables* and *pointers*. A pointer holds a memory location where an object is stored. Python names are essentially pointers. Objects in memory that have no names pointing to them are automatically deleted (this is called *garbage collection*).

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.

> **Achtung!**
>
> Making a copy of a mutable object is possible, but doing so incorrectly can cause subtle problems. For example, suppose we have a dictionary mapping items to their base prices, and we want make a similar dictionary that accounts for a small sales tax.

```
>>> holy = {"moly": 1.99, "hand_grenade": 3, "grail": 1975.41}
>>> tax_prices = holy               # Try to make a copy for processing.
>>> for item, price in tax_prices.items():
...     # Add a 7 percent tax, rounded to the nearest cent.
...     tax_prices[item] = round(1.07 * price, 2)
...
# Now the base prices have been updated to the total price.
>>> print(tax_prices)
{'moly': 2.13, 'hand_grenade': 3.21, 'grail': 2113.69}

# However, dictionaries are mutable, so 'holy' and 'tax_prices' actually
# refer to the same object. The original base prices have now been lost.
>>> print(holy)
{'moly': 2.13, 'hand_grenade': 3.21, 'grail': 2113.69}
```

To avoid this problem, explicitly create a copy of the object by casting it as a new structure. Changes made to the copy will not change the original object. In the above code, we replace the second line with the following:

```
>>> tax_prices = dict(holy)
```

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.

Following the example given above, determine which object types are mutable and which are immutable. Use the following operations to modify each of the given types.

numbers	num += 1
strings	word += 'a'
lists	list.append(1)
tuples	tuple += (1,)
dictionaries	dict[1] = 'a'

Print a statement of your conclusions.

Modules

A Python *module* is a file containing Python code that is meant to be used in some other setting, and not necessarily run directly.[1] The `import` statement loads code from a specified Python file. Importing a module containing some functions, classes, or other objects makes those functions, classes, or objects available for use.

All import statements should occur at the top of the file, 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

if __name__ == "__main__":
```

[1] Python files that are not meant to be imported are often called *scripts*.

```
    pass                            # 'pass' is a temporary placeholder.
```

> **Note**
>
> The modules imported in this example are some of the most important modules for this curriculum. 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. `import <module>` makes the specified 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. `import <module> as <name>` creates an alias for an imported module. 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. `from <module> import <object>` loads the specified object into the namespace without loading anything else in the module or the module name itself. This is used most often to access specific functions from a module. The `as` statement can also be tacked on to create an alias.

```
>>> 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 and Importing

Consider the following simple Python module, saved as `example1.py`.

```
data = range(4)

def display():
    print "Data:", data

if __name__ == "__main__":
    display()
    print "This file was executed from the command line or an interpreter."
else:
    print "This file was imported."
```

Executing the file from the command line executes the file line by line, including the code under the `if __name__ == "__main__"` clause.

```
$ python example1.py
Data: [1, 2, 3, 4]
This file was executed from the command line or an interpreter.
```

Executing the file with IPython's `run` command executes each line of the file and also adds the module's names to the current namespace. *This is the quickest way to test individual functions via IPython.*

```
In [1]: run example1.py
Data: [1, 2, 3, 4]
This file was executed from the command line or an interpreter.

In [2]: display()
Data: [1, 2, 3, 4]
```

Importing the file also executes each line, but only adds the indicated alias to the namespace. Also, code under the `if __name__ == "__main__"` clause is *not* executed when a file is imported.

```
In [1]: import example1 as ex
This file was imported.

# The module's names are not direcly available...
In [2]: display()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-795648993119> in <module>()
----> 1 display()

NameError: name 'display' is not defined

# ...unless accessed via the module's alias.
```

```
In [3]: ex.display()
Data: [1, 2, 3, 4]
```

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 main solutions file, import your new 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.

Achtung!

If a module has been imported in IPython and the source code then changes, using `import` again does **not** refresh the name in the IPython namespace. Use `run` instead to correctly refresh the namespace. Consider this example where we test the function `sum_of_squares()`, saved in the file `example2.py`.

```
# example2.py

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

In IPython, run the file and test `sum_of_squares()`.

```
# Run the file, adding the function sum_of_squares() to the namespace.
In [1]: run example2

In [2]: sum_of_squares(3)
Out[2]: 5                           # Should be 14!
```

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

```
# example2.py

def sum_of_squares(x):
    """Return the sum of the squares of all positive integers
    less than or equal to x.
    """
    return sum([i**2 for i in range(1,x+1)])    # Include the final term.
```

```
# Run the file again to refresh the namespace.
In [3]: run example2

# Now sum_of_squares() is updated to the new, corrected version.
In [4]: sum_of_squares(3)
Out[4]: 14                          # It works!
```

Remember that running or importing a file executes any freestanding code snippets, but any code under an `if __name__ == "__main__"` clause will *only* be executed when the file is run (not when it is imported).

Python Standard Library

All Python distributions include a collection of modules for accomplishing a variety of common tasks, collectively called the *Python standard library*. Some commonly standard library modules are listed below.

Module	Description / Purpose
`csv`	Comma Separated Value (CSV) file parsing and writing.
`math`	Standard mathematical functions.
`random`	Random variable generators.
`sys`	Tools for interacting with the interpreter.
`time`	Time value generation and manipulation.
`timeit`	Measuring execution time of small code snippets.

Using IPython's object introspection, we can learn about how to use the various modules and functions in the standard library very quickly. Use `?` or `help()` for information on the module or one of its names. To see the entire module's namespace, use the `tab` key.

```
In [1]: import math

In [2]: math?
Type:        module
String form: <module 'math' from '/anaconda/lib/python2.7/lib-dynload/math.so'>
File:        /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 the module name, a period, then press tab to see the module's namespace.
In [3]: math.   # Press 'tab'.
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 a Python file. Consider the following simple script:

```
# echo.py
import sys

print(sys.argv)
```

If this file 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
['echo', 'I', 'am', 'the', 'walrus']
```

Command line arguments can be used to control a script's behavior, as in the following example:

```
# example3.py
"""Read a single command line argument and print it in all caps."""
import sys

if len(sys.argv) != 2:
    print("One extra command line argument is required")
else:
    print(sys.argv[1].upper())
```

Now specify the behavior of the script with command line arguments.

```
$ python example3.py
An extra command line argument is required
```

```
$ python example3.py hello
HELLO
```

In IPython, command line arguments are specified after the `run` command.

```
In [1]: run example3.py hello
HELLO
```

Another way to get input from the program user is to prompt the user for text. The built-in function `raw_input()` pauses the program and waits for the user to type something. Like command line arguments, the user's input is parsed as a string.

```
In [1]: x = raw_input("Enter a value for x: ")
Enter a value for x: 20             # Type '20' and press 'enter.'

In [2]: x
Out[2]: '20'                        # Note that x contains a string.

In [3]: y = int(raw_input("Enter an integer for y: "))
Enter an integer for y: 16          # Type '16' and press 'enter.'

In [4]: y
Out[4]: 16                          # Note that y contains an integer.
```

The Random Module

Many real-life events can be simulated by taking random samples from a probability distribution. For example, a coin flip can be simulated by randomly choosing between the integers 1 (for heads) and 0 (for tails). The `random` module includes functions for sampling from probability distributions and generating random data.

Function	Description
`choice()`	Choose a random element from a non-empty sequence, such as a list.
`randint()`	Choose a random integer integer over a closed interval.
`random()`	Pick a float from the interval [0, 1).
`sample()`	Choose several unique random elements from a non-empty sequence.
`seed()`	Seed the random number generator.
`shuffle()`	Randomize the ordering of the elements in a list.

```
>>> import random
>>> numbers = range(1,11)           # Get the integers from 1 to 10.
>>> print(numbers)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> random.shuffle(numbers)         # Mix up the ordering of the list.
```

```
>>> print(numbers)                  # Note that random.shuffle() returns ↩
    nothing.
[5, 9, 1, 3, 8, 4, 10, 6, 2, 7]

>>> random.choice(numbers)          # Pick a single element from the list.
5

>>> random.sample(numbers, 4)       # Pick 4 unique elements from the list.
[5, 8, 3, 2]

>>> random.randint(1,10)            # Pick a random number between 1 and 10.
10
```

Problem 4. *Shut the box* is a popular British pub game that is used to help children learn arithmetic. The player starts with the numbers 1 through 9, and the goal of the game is to eliminate as many of these numbers as possible. At each turn the player roll two dice. Any single set of integers from the player's remaining numbers that sum up to the sum of the dice roll may then be removed from the game. The dice are then rolled again. The game ends when none of the remaining integers can be combined to the sum of the dice roll, and the player's final score is the sum of the numbers that could not be eliminated. See `https://www.youtube.com/watch?v=vLlZGBQ6TKs` for a visual explanation.

Modify your solutions file so that when the file is run (but **not** when it is imported), the user plays a game of shut the box. The provided module `box.py` contains some functions that will be useful in your implementation of the game. You do not need to know how the functions work, but you do need to be able to import and use them correctly.

Your game should match the following specifications:

- Before starting the game, record the player's name.
  - If the user provides a single command line argument after the filename, use that argument for their name.
  - If there are no command line arguments, use `raw_input()` to prompt the user for their name.
- Keep track of the remaining numbers.
- Use the `random` module to simulate rolling two six-sided dice. If the sum of the remaining numbers is 6 or less, role only one die.
- Print the remaining numbers and the sum of the dice roll at each turn.
- If the game is not over, prompt the user for numbers to eliminate after each dice roll. The input should be one or more of the remaining integers, separated by spaces. If the user's input is invalid, prompt them for input again before rolling the dice again.
- When the game is over, print the player's name and score.

Your game should look similar to the following examples. The characters in red are typed inputs from the user.

```
$ python solutions.py TA

Numbers left: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Roll: 5
Numbers to eliminate: 5

Numbers left: [1, 2, 3, 4, 6, 7, 8, 9]
Roll: 5
Numbers to eliminate: 4 1

Numbers left: [2, 3, 6, 7, 8, 9]
Roll: 9
Numbers to eliminate: 9

Numbers left: [2, 3, 6, 7, 8]
Roll: 8
Numbers to eliminate: 8

Numbers left: [2, 3, 6, 7]
Roll: 10
Numbers to eliminate: 3 7

Numbers left: [2, 6]
Roll: 8
Numbers to eliminate: 2 6

Score for player TA: 0 points
Congratulations!! You shut the box!
```

```
$ python solutions.py
Player name: Math TA

Numbers left: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Roll: 7
Numbers to eliminate: Seven
Invalid input
Numbers to eliminate: 1, 2, 4
Invalid input
Numbers to eliminate: 10
Invalid input
Numbers to eliminate: 1 2 4

Numbers left: [3, 5, 6, 7, 8, 9]
Roll: 4
```

```
Game over!

Score for player Math TA: 38 points
```

```
$ python solutions.py
Player name: TA

Numbers left: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Roll: 7
Numbers to eliminate: 7

Numbers left: [1, 2, 3, 4, 5, 6, 8, 9]
Roll: 9
Numbers to eliminate: 1 2 3 4
Invalid input
Numbers to eliminate: 1 2 6

Numbers left: [3, 4, 5, 8, 9]
Roll: 7
Numbers to eliminate: 7
Invalid input
Numbers to eliminate: 3 4

Numbers left: [5, 8, 9]
Roll: 2
Game over!

Score for player TA: 22 points
```

Additional Material

More Built-in Functions

Experiment with the following built-in functions:

Function	Description
`all()`	Return `True` if `bool(entry)` evaluates to `True` for *every* entry in the input iterable.
`any()`	Return `True` if `bool(entry)` evaluates to `True` for *any* entry in the input iterable.
`bool()`	Evaluate a single input object as `True` or `False`.
`eval()`	Execute a string as Python code and return the output.
`input()`	Prompt the user for input and evaluate it (`eval(raw_input())`).
`map()`	Apply a function to every item of the input iterable and return a list of the results.
`reduce()`	Apply a function accepting two arguments cumulatively to the items of the input iterable from left to right (thus reducing it to a single output).

The `all()` and `any()` functions are particularly useful for creating readable code when used in conjunction with conditionals or list comprehensions.

```
>>> from random import randint
# Get 5 random numbers between 1 and 10, inclusive
>>> numbers = [randint(1,10) for _ in xrange(5)]

# If all of the numbers are less than 8, print the list.
>>> if all([num < 8 for num in numbers]):
...     print(numbers)
...
[1, 5, 6, 3, 3]

# If none of the numbers are divisible by 3, print the list.
>>> if not any([num % 3 == 0 for num in numbers]):
...     print(numbers)
...
```

Two-Player Shut the Box

Consider modifying your shut the box program so that it pits two players against each other (one player tries to shut the box while the other tries to keep it open). The first player plays a regular round as described in Problem 4. Suppose he or she eliminates every number but 2, 3, and 6. The second player then begins a round with the numbers 1, 4, 5, 7, 8, and 9, the numbers that the first player had eliminated. If the second player loses, the first player gets another round to try to shut the box with the numbers that the second player had eliminated. Play continues until one of the players eliminates their entire list.

Python Packages

Large programming projects often have code spread throughout several folders and files. In order to get related files in different folders to communicate properly, we must make the associated directories into Python *packages*. This is a common procedure when creating smart phone applications and other programs that have graphical user interfaces (GUIs).

A package is simply a folder that contains a file called `__init__.py`. This file is always executed first whenever the package is used. A package must also have a file called `__main__.py` to be executable. Executing the package will run `__init__.py` and then `__main__.py`, but importing the package will only run `__init__.py`.

Suppose we are working on a project with the following file directories.

```
ssb/
        __init__.py
        __main__.py
        characters/
                __init__.py
                link.py
                mario.py
        gui/
                __init__.py
                character_select.py
                main_menu.py
                stage_select.py
        play/
                __init__.py
                damage.py
                sounds.py
        spoilers/
                __init__.py
                __main__.py
                jigglypuff.py
```

Use the regular syntax to import a module or subpackage that is in the current package, and use `from <subpackage.module> import <object>` to load a module within a subpackage. Once a name has been loaded into a package's `__init__.py`, other files in the same package can load the same name with `from . import <object>`. Thus the files `ssb/__init__.py` and `ssb/__main__.py` might be as follows:

```
print("ssb/__init__.py")

# Load a subpackage.
import characters
# Import the start() function from the main_menu module in the gui/ subpackage.
from gui.main_menu import start
```

```
print("ssb/__main__.py")
```

```
# Load a name that was already imported in ssb/__init__.py
from . import start
print("\tStarting game...")
start()
```

To access code in the directory one level above the current directory, use the syntax `from .. import <object>` This tells the interpreter to go up one level and import the object from there. This is called an *explicit relative import* and cannot be done in files that are executed directly (like `__main__.py`).

For instance, `ssb/gui/__init__.py` might need to access `ssb/` subpackages.

```
print("ssb/gui/__init__.py")

from .. import characters           # Import a subpackage from the ssb/ package
    .
from ..play import sounds           # Import a module from the play/ subpackage
    .
print("\tInitializing GUI...")
```

To execute a package, run Python from the shell with the flag `-m` (for "module-name") and exclude the extension `.py`. Suppose each file prints out its own name.

```
$ python -m ssb
ssb/__init__.py                     # First run ssb/__init__.py.
ssb/characters/__init__.py
ssb/gui/__init__.py
ssb/play/__init__.py
ssb/play/sounds.py
    Initializing GUI...
ssb/gui/main_menu.py
ssb/__main__.py                     # Then run ssb/__main__.py.
    Starting game...
```

Separate directories by a period to run a subpackage.

```
print("ssb/spoilers/__main__.py")

print("\tChallenger Approaching!")
import jigglypuff
```

```
# Run spoilers/ as a subpackage.
$ python -m ssb.spoilers
ssb/__init__.py                     # First run ssb/__init__.py.
ssb/characters/__init__.py
ssb/gui/__init__.py
ssb/play/__init__.py
ssb/play/sounds.py
    Initializing GUI...
```

```
ssb/gui/main_menu.py
ssb/spoilers/__init__.py            # Second run ssb/spoilers/__init__.py.
ssb/spoilers/__main__.py            # Finally run ssb/spoilers/__main__.py.
    Challenger Approaching!
ssb/spoilers/jigglypuff.py

# Change directories to ssb/ to run spoilers/ without running ssb/.
$ cd ssb
$ python -m spoilers
ssb/spoilers/__init__.py            # First run ssb/spoilers/__init__.py.
ssb/spoilers/__main__.py            # Then run ssb/spoilers/__main__.py.
    Challenger Approaching!
ssb/spoilers/jigglypuff.py
```

See `https://docs.python.org/2/tutorial/modules.html#packages` for more details and examples on this topic.

3 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 a program. 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` key word 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*: `name` 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 oop import Backpack
>>> my_backpack = Backpack("Fred")

# Access the object's attributes with a period and the attribute name.
>>> 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 belongs to a specific class is called a *method*. Below 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 called `dump()` that resets the contents of the backpack to an empty list. This method should not receive any arguments (except `self`).

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():
    testpack = Backpack("Barry", "black")       # Instantiate the object.
    if testpack.max_size != 5:                  # Test an attribute.
        print("Wrong default max_size!")
    for item in ["pencil", "pen", "paper", "computer"]:
        testpack.put(item)                      # Test a method.
    print(testpack.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 kind of 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): the maximum number of items that can fit
                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 the `closed` attribute 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. For example, the `dump()` method is available even though it is not defined explicitly in the `Knapsack` class.

```
>>> from oop 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. Change the default value of `max_size` to 2, and set the default value of fuel to 10. Store the fuel as an attribute.

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. Magic methods begins and ends with two underscores, like the constructor `__init__()`. Every Python object is automatically endowed with several magic methods, but they are normally hidden from IPython's object introspection because they begin with an underscore. To see an object's magic methods, type an underscore before pressing tab.

```
In [1]: run oop.py                  # Load the names from oop.py.

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

In [3]: b.      # Press 'tab' to see standard methods and attributes.
b.name      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.__    # Now 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. Modules also have a `__doc__` attribute for describing the purpose of the file. Writing detailed docstrings is critical so that others can utilize your code correctly (and so that you don't forget how to use your own code!).

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 class's `__add__()` method. The object on the left side of the + is passed in to `__add__()` as `self` and the object on the right side of the + is passed in as `other`.

```
>>> pack1 = Backpack("Rose", "red")
>>> pack2 = Backpack("Carly", "cyan")

# Put some items in the backpacks.
>>> pack1.put("textbook")
>>> pack2.put("water bottle")
>>> pack2.put("snacks")

# Now add the backpacks like numbers
>>> pack1 + pack2                   # Equivalent to pack1.__add__(pack2).
3
```

Subtraction, multiplication, division, and other standard operations may be similary defined with their corresponding magic methods (see Table 3.1).

Comparisons

Magic methods also facilitate object comparisons. For example, the `__lt__()` method corresponds to the < operator. Suppose one backpack is considered "less" than another if it has fewer items in its list of 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)
```

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

```
>>> pack1 = Backpack("Maggy", "magenta")
>>> pack2 = Backpack("Yolanda", "yellow")

>>> pack1.put('book')
>>> pack2.put('water bottle')
>>> pack1 < pack2
False

>>> pack2.put('pencils')
>>> pack1 < pack2                    # Equivalent to pack1.__lt__(pack2).
True
```

Other standard comparison operators also have corresponding magic methods and should be implemented similarly (see Table 3.1). Note that comparison methods should return either `True` or `False`, while arithmetic methods like `__add__()` might return a numerical value or another kind of object.

Method	Operation	Operator
`__add__()`	Addition	`+`
`__sub__()`	Subtraction	`-`
`__mul__()`	Multiplication	`*`
`__div__()`	Division	`/`
`__lt__()`	Less than	`<`
`__le__()`	Less than or equal to	`<=`
`__gt__()`	Greater than	`>`
`__ge__()`	Greater than or equal to	`>=`
`__eq__()`	Equal	`==`
`__ne__()`	Not equal	`!=`

Table 3.1: Common magic methods for arithmetic and comparisons. What each of these operations do, or should do, is up to the programmer and should be carefully documented. See `https://docs.python.org/2/reference/datamodel.html#special-method-names` for more methods and details.

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 number of contents.

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 tab and newline characters `'\t'` and `'\n'` to help align output nicely.)

ACHTUNG!

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, perhaps by calling `__eq__()` itself:

```
def __ne__(self, other):
    return not self == other
```

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` type. How can your class be improved to act more like true complex numbers?

Additional Material

Static Attributes

Attributes that are accessed through `self` are called *instance* attributes because they are bound to a particular instance of the class. In contrast, a *static* attribute is one that is shared between all instances of the class. To make an attribute static, declare it inside of the `class` block but outside of any of the class's functions, and do not use `self`. Since the attribute is not tied to a specific instance of the class, it should be accessed or changed via the class name.

For example, suppose our Backpacks all have the same brand name.

```
class Backpack(object):
    # ...
    brand = "Adidas"
```

Changing the brand name changes it on every backpack instance.

```
>>> pack1 = Backpack("Bill", "blue")
>>> pack2 = Backpack("William", "white")
>>> print pack1.brand, pack2.brand
Adidas Adidas

# Change the brand name for the class to change it for all class instances.
>>> print Backpack.brand
Adidas
>>> Backpack.brand = "Nike"
>>> print pack1.brand, pack2.brand
Nike Nike
```

Static Methods

Individual class methods can also be static. A static method can have no dependence on the attributes of individual instances of the class, so there can be no references to `self` inside the body of the method and `self` is **not** listed as an argument in the function definition. Thus only static attributes and other static methods are available within the body of a static method. Include the tag `@staticmethod` above the function definition to designate a method as static.

```
class Backpack(object):
    # ...
    @staticmethod
    def origin():
        print "Manufactured by " + Backpack.brand + ", inc."
```

```
# We can call static methods before instantiating the class.
Backpack.origin()
Manufactured by Nike, inc.

# The method can also be accessed through class instances.
```

```
>>> pack = Backpack("Larry", "lime")
>>> pack.origin()
Manufactured by Nike, inc.
```

To practice these principles, consider adding a static attribute to the `Backpack` class to serve as a counter for a unique ID. In the constructor for the `Backpack` class, add an instance variable called `self.ID`. Set this ID based on the static ID variable, then increment the static ID so that the next `Backpack` object will have a new ID.

More Magic Methods

Explore the following magic methods and consider how they could be implemented for the `Backpack` class.

Method	Operation	Trigger Function
`__repr__()`	Object representation	`repr()`
`__nonzero__()`	Truth value	`bool()`
`__len__()`	Object length or size	`len()`
`__getitem__()`	Indexing and slicing	`self[index]`
`__setitem__()`	Assignment via indexing	`self[index] = x`
`__iter__()`	Iteration over the object	`iter()`
`__reversed__()`	Reverse iteration over the object	`reversed()`
`__contains__()`	Membership testing	`in`

See `https://docs.python.org/2/reference/datamodel.html` for more details and documentation on all magic methods.

Hashing

A *hash value* is an integer that identifies an object. The built-in `hash()` function calculates an object's hash value by calling its `__hash__()` magic method.

In Python, the built-in `set` and `dict` structures use hash values to store and retrieve objects in memory quickly. Thus if an object is unhashable, it cannot be put in a set or be used as a key in a dictionary.

```
# Get the hash value of a hashable object.
>>> hash("math")
-8321016616855971138

# Create a dictionary and attempt to get its hash value.
>>> example = {"math": 320}
>>> hash(example)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

If the `__hash__()` method is not defined, the default hash value is the object's memory address (accessible via the built-in function `id()`) divided by 16, rounded down to the nearest integer. However, two objects that compare as equal via the `__eq__()` magic method must have the same hash value. A simple `__hash__()` method for the `Backpack` class that conforms to this rule and returns an integer might be the following.

```
class Backpack(object):
    # ...
    def __hash__(self):
        return hash(self.name) ^ hash(self.color) ^ hash(len(self.contents))
```

The caret operator `^` is a bitwise XOR (exclusive or). The bitwise AND operator `&` and the bitwise OR operator `|` are also good choices to use.

Achtung!

If a class does not have an `__eq__()` method it should **not** have a `__hash__()` method either. Furthermore, if a class defines `__eq__()` but not `__hash__()`, its instances may be usable in hashed collections like sets and dictionaries (because of the default hash value), but two instances that compare as equal may or may not be recognized as distinct in the collection.

4 Introduction to NumPy

Lab Objective: *NumPy is a powerful Python package for manipulating data with multi-dimensional vectors. Its versatility and speed makes Python an ideal language for applied and computational mathematics.*

In this lab we introduce basic NumPy data structures and operations as a first step to numerical computing in Python.

Arrays

In many algorithms, data can be represented mathematically as a *vector* or a *matrix*. Conceptually, a vector is just a list of numbers and a matrix is a two-dimensional list of numbers. Therefore, we might try to represent a vector as a Python list and a matrix as a list of lists. However, even basic linear algebra operations like matrix multiplication are cumbersome to implement and slow to execute when we store data this way. The *NumPy* module[1] offers a much better solution.

The basic object in NumPy is the *array*, which is conceptually similar to a matrix. The NumPy array class is called `ndarray` (for "n-dimensional array"). The simplest way to explicitly create a 1-D `ndarray` is to define a list, then cast that list as an `ndarray` with NumPy's `array()` function.

```
>>> import numpy as np
# Create a 1-D array by passing a list into NumPy's array() function.
>>> np.array([8, 4, 6, 0, 2])
array([8, 4, 6, 0, 2])
```

The alias "`np`" is standard in the Python community.

An `ndarray` can have arbitrarily many dimensions. A 2-D array is a 1-D array of 1-D arrays, and more generally, an n-dimensional array is a 1-D array of $(n-1)$-dimensional arrays. Each dimension is called an *axis*. For a 2-D array, the 0-axis indexes the rows and the 1-axis indexes the columns. Elements are accessed using brackets and indices (like a regular list), and the axes are separated by commas.

```
# Create a 2-D array by passing a list of lists into array().
>>> A = np.array( [ [1, 2, 3],[4, 5, 6] ] )
```

[1] NumPy is *not* part of the standard library, but it is included in most Python distributions.

```
>>> print(A)
[[1, 2, 3],
 [4, 5, 6]]

# Access elements of the array with brackets.
>>> print A[0, 1], A[1, 2]
2 6

# The elements of a 2-D array are 1-D arrays.
>>> A[0]
array([1, 2, 3])
```

Problem 1. NumPy's `dot()` function performs matrix multiplication. Write a function that defines the following matrices as NumPy arrays.

$$A = \begin{bmatrix} 3 & -1 & 4 \\ 1 & 5 & -9 \end{bmatrix} \qquad B = \begin{bmatrix} 2 & 6 & -5 & 3 \\ 5 & -8 & 9 & 7 \\ 9 & -3 & -2 & -3 \end{bmatrix}$$

Return the matrix product AB.

NumPy has excellent documentation. For examples of array initialization and matrix multiplication, use IPython's object introspection feature to look up the documentation for `np.ndarray`, `np.array()` and `np.dot()`.

```
In [1]: import numpy as np
In [2]: np.array?       # press 'Enter'
```

Basic Array Operations

NumPy arrays behave differently with respect to the binary arithmetic operators `+` and `*` than Python lists do. For lists, `+` concatenates two lists and `*` replicates a list by a scalar amount (strings also behave this way).

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]

# Addition performs list concatenation.
>>> x + y
[1, 2, 3, 4, 5, 6]

# Mutliplication concatenates a list with itself a given number of times.
>>> x * 4
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

In contrast, these operations are component-wise for NumPy arrays. Thus arrays behave the same way that vectors and matrices behave mathematically.

```
>>> x = np.array([1, 2, 3])
>>> y = np.array([4, 5, 6])

# Addition or multiplication by a scalar acts on each element of the array.
>>> x + 10
array([11, 12, 13])
>>> x * 4
array([ 4,  8, 12])

# Add two arrays together (component-wise).
>>> x + y
array([5, 7, 9])

# Multiply two arrays together (component-wise).
>>> x * y
array([ 4, 10, 18])
```

Problem 2. Write a function that defines the following matrix as a NumPy array.

$$A = \begin{bmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ -5 & 3 & 1 \end{bmatrix}$$

Return the matrix $-A^3 + 9A^2 - 15A$.

In this context, $A^2 = AA$ (the matrix product). The somewhat surprising result is a demonstration of the Cayley-Hamilton theorem.

Array Attributes

An `ndarray` object has several attributes, some of which are listed below.

Attribute	Description
`dtype`	The type of the elements in the array.
`ndim`	The number of axes (dimensions) of the array.
`shape`	A tuple of integers indicating the size in each dimension.
`size`	The total number of elements in the array.

```
>>> A = np.array([[1, 2, 3],[4, 5, 6]])
>>> A.ndim
2
>>> A.shape
(2, 3)
>>> A.size
6
```

Note that the `ndim` is the number of entries in `shape`, and the the `size` of the array is the product of the entries of `shape`.

Array Creation Routines

In addition to casting other structures as arrays via `np.array()`, NumPy provides efficient ways to create certain commonly-used arrays.

Function	Returns
`arange()`	An array of evenly spaced values within a given interval (like `range()`).
`eye()`	A 2-D array with ones on the diagonal and zeros elsewhere.
`ones()`	A new array of given shape and type, filled with ones.
`ones_like()`	An array of ones with the same shape and type as a given array.
`zeros()`	A new array of given shape and type, filled with zeros.
`zeros_like()`	An array of zeros with the same shape and type as a given array.
`full()`	A new array of given shape and type, filled with a specified value.
`full_like()`	A full array with the same shape and type as a given array.

Each of these functions accepts the keyword argument `dtype` to specify the data type. Common types include `np.bool_`, `np.int64`, `np.float64`, and `np.complex128`.

```
# A 1-D array of 5 zeros.
>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])

# A 2x5 matrix (2-D array) of integer ones.
>>> np.ones((2,5), dtype=np.int)    # The shape is specified as a tuple.
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]])

# The 2x2 identity matrix.
>>> I = np.eye(2)
>>> print(I)
[[ 1.  0.]
 [ 0.  1.]]

# An array of 3s the same size as 'I'.
>>> np.full_like(I, 3)              # Equivalent to np.full(I.shape, 3).
array([[ 3.,  3.],
       [ 3.,  3.]])
```

All elements of a NumPy array must be of the same data type. To change an existing array's data type, use the array's `astype()` method.

```
# A list of integers becomes an array of integers.
>>> x = np.array([0, 1, 2, 3, 4])
>>> print(x)
[0 1 2 3 4]
>>> x.dtype
```

```
dtype('int64')

# Change the data type to one of NumPy's float types.
>>> x = x.astype(np.float64)
>>> print(x)
[ 0.  1.  2.  3.  4.]
>>> x.dtype
dtype('float64')
```

Once we have an array, we might want to extract its diagonal or get the upper or lower portion of the array. The following functions are very helpful for this.

Function	Description
`diag()`	Extract a diagonal or construct a diagonal array.
`tril()`	Get the lower-triangular portion of an array by replacing entries above the diagonal with zeros.
`triu()`	Get the upper-triangular portion of an array by replacing entries below the diagonal with zeros.

```
>>> A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> print(A)
[[1 2 3]
 [4 5 6]
 [7 8 9]]

# Get the diagonal entries of 'A' as a 1-D array.
>>> np.diag(A)
array([1, 5, 9])

# Get only the upper triangular entries of 'A'.
>>> np.triu(A)
array([[1, 2, 3],
       [0, 5, 6],
       [0, 0, 9]])

# diag() can also be used to create a diagonal matrix from a 1-D array.
>>> np.diag([1, 11, 111])
array([[  1,   0,   0],
       [  0,  11,   0],
       [  0,   0, 111]])
```

See `http://docs.scipy.org/doc/numpy/reference/routines.array-creation.html` for the official documentation on NumPy's array creation routines.

Problem 3. Write a function that defines the following matrices as NumPy arrays. Use the functions presented in this section instead of `np.array()` to construct the matrices, then calculate the matrix product ABA. Change the data type of the resulting matrix to `np.int64`, then return it.

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} -1 & 5 & 5 & 5 & 5 & 5 & 5 \\ -1 & -1 & 5 & 5 & 5 & 5 & 5 \\ -1 & -1 & -1 & 5 & 5 & 5 & 5 \\ -1 & -1 & -1 & -1 & 5 & 5 & 5 \\ -1 & -1 & -1 & -1 & -1 & 5 & 5 \\ -1 & -1 & -1 & -1 & -1 & -1 & 5 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

Data Access

Array Slicing

Indexing for a 1-D NumPy array works exactly like indexing for a Python list. To access a single entry of a multi-dimensional array, say a 3-D array, use the syntax `f[i, j, k]`. While the syntax `f[i][j][k]` will also work, it is significantly slower because each bracket returns an array slice.

In slicing syntax, the colon `:` separates the arguments `start`, `stop`, and `step`. If there is no colon, a single entry of that dimension is accessed. With a colon, a range of values is accessed.

```
# Make an array of the integers from 0 to 10 (exclusive).
>>> x = np.arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# Access a elements of the array with slicing syntax.
>>> x[3]                            # The element at index 3.
3
>>> x[:3]                           # Everything up to index 3.
array([0, 1, 2])
>>> x[3:]                           # Everything from index 3 on.
array([3, 4, 5, 6, 7, 8, 9])
>>> x[3:8]                          # The elements from index 3 to 8.
array([3, 4, 5, 6, 7])

>>> A = np.array([[0,1,2,3,4],[5,6,7,8,9]])
>>> A
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])

# Use a comma to separate the dimensions for multi-dimensional arrays.
>>> A[1, 2]                         # The element at row one, column 2.
7
>>> A[:, 2:]                        # All of the rows, from column 2 on.
```

```
array([[2, 3, 4],
       [7, 8, 9]])
```

See Appendix C for visual examples of slicing.

> **Note**
>
> NumPy has two ways of returning an array: as a *view* and as a *copy*. A view of an array is distinct from the original array in Python, but it references the same place in memory. Thus changing a array view also change the array it references. In other words, **arrays are mutable**.
>
> A copy of an array is a separate array with its own place in memory. Changes to a copy of an array does not affect the original array, but copying an array uses more time and memory than getting a view. An array can be copied using `np.copy()` or the array's `copy()` method.

Fancy Indexing

So-called "fancy indexing" is a second way to access elements of an array. Instead of providing indices to obtain a slice of an array, we provide either an array of integers or an array of boolean values (called a *mask*).

```
# Make an array of every 10th integer from 0 to 50 (exclusive).
>>> x = np.arange(0, 50, 10)
>>> x
array([ 0, 10, 20, 30, 40])

# An array of integers extracts the entries of 'x' at the given indices.
>>> index = np.array([3, 1, 4])
>>> x[index]                        # Same as np.array([x[i] for i in index]).
array([30, 10, 40])

# A boolean array extracts the elements of 'x' at the same places as 'True'.
>>> mask = np.array([True, False, False, True, False])
>>> x[mask]
array([ 0, 30])
```

Fancy indexing is especially useful for extracting or changing the values of an array that meet some sort of criterion. Comparison operators like < and == may be used to create masks.

```
# Make an array of every other integer form 10 to 20 (exclusive).
>>> y = np.arange(10, 20, 2)
>>> y
array([10, 12, 14, 16, 18])

# Extract the values of 'y' larger than 15.
>>> mask = y > 15                   # Same as np.array([num > 15 for num in y])↩
    .
>>> mask
```

```
array([False, False, False,  True,  True], dtype=bool)
>>> y[mask]
array([16, 18])

# If the mask doesn't need to be saved, use this very readable syntax.
>>> y[y > 15]
array([16, 18])

# Change the values of 'y' larger than 15 to 0.
>>> y[mask] = 0
>>> print(y)
[10 12 14  0  0]
```

Note that slice operations and indexing always return a view and fancy indexing always returns a copy.

Problem 4. Write a function that accepts a single array as input. Make a copy of the array, then use fancy indexing to set all negative entries of the copy to 0. Return the copy.

Array Manipulation

Shaping

Recall that arrays have a `shape` attribute that describes the dimensions of the array. We can change the shape of an array with `np.reshape()` or the array's `reshape()` method. The total number of entries in the old array and the new array must be the same in order for the shaping to work correctly. A -1 in the new shape tuple to makes the specified dimension as long as necessary.

```
# Make an array of the integers from 0 to 12 (exclusive).
>>> A = np.arange(12)
>>> print(A)
[ 0  1  2  3  4  5  6  7  8  9 10 11]

# 'A' has 12 entries, so it can be reshaped into a 3x4 matrix.
>>> A.reshape((3,4))                # The new shape is specified as a tuple.
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

# Reshape 'A' into an array with 2 rows and the appropriate number of columns.
>>> A.reshape((2,-1))
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

Use `np.ravel()` to flatten a multi-dimensional array into a (flat) 1-D array and `np.transpose()` to transpose a 2-D array (in the matrix sense). Transposition can also be done with an array's `T` attribute.

Function	Description
`reshape()`	Return a view of the array with a changed shape.
`ravel()`	Make a flattened version of an array, return a view if possible.
`transpose()`	Permute the dimensions of the array (also `ndarray.T`).
`hstack()`	Stack arrays in sequence horizontally (column wise).
`vstack()`	Stack arrays in sequence vertically (row wise).
`column_stack()`	Stack 1-D arrays as columns into a 2-D array.

```
>>> A = np.arange(12).reshape((3,4))
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

# Flatten 'A' into a one-dimensional array.
>>> np.ravel(A)                      # Equivalent to A.reshape(A.size)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

# Transpose the matrix 'A'.
>>> A.T                              # Equivalent to np.transpose(A).
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

Note

By default, all NumPy arrays that can be represented by a single dimension, including column slices, are automatically reshaped into "flat" 1-D arrays. Therefore an array will usually have 10 elements instead of 10 arrays with one element each. Though we usually represent vectors vertically in mathematical notation, NumPy methods such as `dot()` are implemented purposefully to play nicely with 1-D "row arrays".

```
>>> A = np.arange(10).reshape((2,5))
>>> A
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])

# Slicing out a column of A still produces a "flat" 1-D array.
>>> x = A[:,1]
>>> x
array([1, 6])
>>> x.shape
(2,)
>>> x.ndim
```

```
1
```

Occasionally it is necessary to change a 1-D array into a "column array". Use `np.reshape()`, `np.vstack()`, or slice the array and put `np.newaxis` on the second axis. Note that `np.transpose()` does not alter 1-D arrays.

```
>>> x = np.arange(3)
>>> x
array([0, 1, 2])

>>> x.reshape((3,1))                # Or np.vstack(x) or x[:,np.newaxis].
array([[0],
       [1],
       [2]])
```

Stacking

Suppose we have two or more matrices that we would like to join together into a single block matrix. NumPy has functions for *stacking* arrays with similar dimensions together for this very purpose. Each of these methods takes in a single tuple of arrays to be stacked in sequence.

```
>>> A = np.arange(6).reshape((3,2))
>>> B = np.ones((3,4))

# hstack() stacks arrays horizontally (column-wise).
>>> np.hstack((A,B,A))
array([[ 0.,  1.,  1.,  1.,  1.,  1.,  0.,  1.],
       [ 2.,  3.,  1.,  1.,  1.,  1.,  2.,  3.],
       [ 4.,  5.,  1.,  1.,  1.,  1.,  4.,  5.]])
```

```
>>> A = np.arange(6).reshape((2,3))
>>> B = np.zeros((4,3))

# vstack() stacks arrays vertically (row-wise).
>>> np.vstack((A,B,A))
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
```

See Appendix C for more visual examples of stacking, and visit `http://docs.scipy.org/doc/numpy-1.10.1/reference/routines.array-manipulation.html` for more array manipulation routines and documentation.

Problem 5. Write a function that defines the following matrices as NumPy arrays.

$$A = \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix} \qquad B = \begin{bmatrix} 3 & 0 & 0 \\ 3 & 3 & 0 \\ 3 & 3 & 3 \end{bmatrix} \qquad C = \begin{bmatrix} -2 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & -2 \end{bmatrix}$$

Use NumPy's stacking functions to create and return the block matrix:

$$\begin{bmatrix} \mathbf{0} & A^\mathsf{T} & I \\ A & \mathbf{0} & \mathbf{0} \\ B & \mathbf{0} & C \end{bmatrix},$$

where I is the identity matrix of the appropriate size and each $\mathbf{0}$ is a matrix of all zeros, also of appropriate sizes.

A block matrix of this form is used in the Interior Point method for linear optimization.

Array Broadcasting

Many matrix operations make sense only when the two operands have the same shape, such as element-wise addition. *Array broadcasting* extends such operations to accept some (but not all) operands with different shapes, and occurs automatically whenever possible.

Suppose, for example, that we would like to add different values to the different columns of an $m \times n$ matrix A. Adding a 1-D array x with the n entries to A will automatically do this correctly. To add different values to the different rows of A, we must first reshape a 1-D array of m values into a column array. Broadcasting then correctly takes care of the operation.

Broadcasting can also occur between two 1-D arrays, once they are reshaped appropriately.

```
>>> A = np.arange(12).reshape((4,3))
>>> x = np.arange(3)
>>> A
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
>>> x
array([0, 1, 2])

# Add the entries of 'x' to the corresponding columns of 'A'.
>>> A + x
array([[ 0,  2,  4],
       [ 3,  5,  7],
       [ 6,  8, 10],
       [ 9, 11, 13]])
```

```
>>> y = np.arange(0, 40, 10).reshape((4,1))
>>> y
array([[ 0],
       [10],
       [20],
       [30]])

# Add the entries of 'y' to the corresponding rows of 'A'.
>>> A + y
array([[ 0,  1,  2],
       [13, 14, 15],
       [26, 27, 28],
       [39, 40, 41]])

# Add 'x' and 'y' together with array broadcasting.
>>> x + y
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

Numerical Computing with NumPy

Universal Functions

A universal function is one that operates on an entire array element-wise. Using a universal function is almost always significantly more efficient than iterating through the array.

Function	Description
`abs()` or `absolute()`	Calculate the absolute value element-wise.
`exp()` / `log()`	Exponential (e^x) / natural log element-wise.
`maximum()` / `minimum()`	Element-wise maximum / minimum of two arrays.
`sqrt()`	The positive square-root, element-wise.
`sin()`, `cos()`, `tan()`, etc.	Element-wise trigonometric operations.

Many scalar functions from the Python standard library have a corresponding universal function in NumPy. If you have a simple operation to perform element-wise on an array, check if NumPy has a universal function for it (it probably does). See `http://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs` for a more comprehensive list of universal functions.

Other Array Methods

Often the easiest way to compute information is by using methods of the `np.ndarray` class on existing arrays.

Method	Returns
`all()`	`True` if all elements evaluate to `True`.
`any()`	`True` if any elements evaluate to `True`.
`argmax()`	Index of the maximum value.
`argmin()`	Index of the minimum value.
`argsort()`	Indices that would sort the array.
`max()`	The maximum element of the array.
`mean()`	The average value of the array.
`min()`	The minimum element of the array.
`sort()`	Return nothing; sort the array in-place.
`std()`	The standard deviation of the array.
`sum()`	The sum of the elements of the array.
`var()`	The variance of the array.

Each of these `np.ndarray` methods has a corresponding (and equivalent) NumPy function. Thus `A.max()` and `np.max(A)` are equivalent. The one exception is the `sort()` function: `np.sort()` returns a sorted copy of the array, while `A.sort()` sorts the array in-place and returns nothing.

Every method listed above has the option to operate *along an axis* via the keyword argument `axis`. If `axis` is specified for a method on an n-D array, the return value is an $(n-1)$-D array, the specified axis having been collapsed in the evaluation process. If `axis` is not specified, the return value is usually a scalar.

```
>>> A = np.arange(9).reshape((3,3))
>>> A
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

# Find the maximum value in the entire array.
>>> A.max()
8

# Find the minimum value of each column.
>>> A.min(axis=0)                   # np.array([min(A[:,i]) for i in xrange(3)↩
    ])
array([0, 1, 2])

# Compute the sum of each row.
>>> A.sum(axis=1)                   # np.array([sum(A[i,:]) for i in xrange(3)↩
    ])
array([3, 12, 21])
```

A more comprehensive list of array methods can be found at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html`. See Appendix C for visual examples of the `axis` keyword argument.

Problem 6. A matrix is called *row-stochastic* if its rows each sum to 1.[a] Stochastic matrices are fundamentally important for finite discrete random processes and some machine learning algorithms.

Write a function than accepts a matrix (as a 2-D array). Divide each row of the matrix by the row sum and return the new row-stochastic matrix. Use array broadcasting instead of a loop, and be careful to avoid integer division.

[a]Similarly, a matrix is called *column-stochastic* if its columns each sum to 1.

Problem 7. This problem comes from `https://projecteuler.net`.

In the 20×20 grid below, four numbers along a diagonal line have been marked in red.

```
08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08
49 49 99 40 17 81 18 57 60 87 17 40 98 43 69 48 04 56 62 00
81 49 31 73 55 79 14 29 93 71 40 67 53 88 30 03 49 13 36 65
52 70 95 23 04 60 11 42 69 24 68 56 01 32 56 71 37 02 36 91
22 31 16 71 51 67 63 89 41 92 36 54 22 40 40 28 66 33 13 80
24 47 32 60 99 03 45 02 44 75 33 53 78 36 84 20 35 17 12 50
32 98 81 28 64 23 67 10 26 38 40 67 59 54 70 66 18 38 64 70
67 26 20 68 02 62 12 20 95 63 94 39 63 08 40 91 66 49 94 21
24 55 58 05 66 73 99 26 97 17 78 78 96 83 14 88 34 89 63 72
21 36 23 09 75 00 76 44 20 45 35 14 00 61 33 97 34 31 33 95
78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 53 56 92
16 39 05 42 96 35 31 47 55 58 88 24 00 17 54 24 36 29 85 57
86 56 00 48 35 71 89 07 05 44 44 37 44 60 21 58 51 54 17 58
19 80 81 68 05 94 47 69 28 73 92 13 86 52 17 77 04 89 55 40
04 52 08 83 97 35 99 16 07 97 57 32 16 26 26 79 33 27 98 66
88 36 68 87 57 62 20 72 03 46 33 67 46 55 12 32 63 93 53 69
04 42 16 73 38 25 39 11 24 94 72 18 08 46 29 32 40 62 76 36
20 69 36 41 72 30 23 88 34 62 99 69 82 67 59 85 74 04 36 16
20 73 35 29 78 31 90 01 74 31 49 71 48 86 81 16 23 57 05 54
01 70 54 71 83 51 54 69 16 92 33 48 61 43 52 01 89 19 67 48
```

The product of these numbers is $26 \times 63 \times 78 \times 14 = 1788696$. Write a function that returns the greatest product of four adjacent numbers in the same direction (up, down, left, right, or diagonally) in the grid.

For convenience, this array has been saved in the file `grid.npy`. Use the following syntax to extract the array:

```
>>> grid = np.load("grid.npy")
```

One way to approach this problem is to iterate through the rows and columns of the array, checking small slices of the array at each iteration and updating the current largest product. Array slicing, however, provides a much more efficient solution.

The naïve method for computing the greatest product of four adjacent numbers in a horizontal row might be as follows:

```
>>> winner = 0
>>> for i in xrange(20):
...     for j in xrange(17):
...         winner = max(np.prod(grid[i,j:j+4]), winner)
...
```

```
>>> winner
48477312
```

Instead, use array slicing to construct a single array where the (i, j)th entry is the product of the four numbers to the right of the (i, j)th entry in the original grid. Then find the largest element in the new array.

```
>>> np.max(grid[:,:-3] * grid[:,1:-2] * grid[:,2:-1] * grid[:,3:])
48477312
```

Use slicing to similarly find the greatest products of four vertical, right diagonal, and left diagonal adjacent numbers.
(Hint: Consider drawing the portions of the grid that each slice in the above code covers, like the examples in the visual guide. Then draw the slices that produce vertical, right diagonal, or left diagonal sequences, and translate the pictures into slicing syntax.)

Additional Material

Random Sampling

The submodule `np.random` holds many functions for creating arrays of random values chosen from probability distributions such as the uniform, normal, and multinomial distributions. It also contains some utility functions for getting non-distributional random samples, such as random integers or random samples from a given array.

Function	Description
`choice()`	Take random samples from a 1-D array.
`random()`	Uniformly distributed floats over [0, 1).
`randint()`	Random integers over a half-open interval.
`random_integers()`	Random integers over a closed interval.
`randn()`	Sample from the standard normal distribution.
`permutation()`	Randomly permute a sequence / generate a random sequence.

Function	Distribution
`beta()`	Beta distribution over [0, 1].
`binomial()`	Binomial distribution.
`exponential()`	Exponential distribution.
`gamma()`	Gamma distribution.
`geometric()`	Geometric distribution.
`multinomial()`	Multivariate generalization of the binomial distribution.
`multivariate_normal()`	Multivariate generalization of the normal distribution.
`normal()`	Normal / Gaussian distribution.
`poisson()`	Poisson distribution.
`uniform()`	Uniform distribution.

```
# 5 uniformly distributed values in the interval [0, 1).
>>> np.random.random(5)
array([ 0.21845499,  0.73352537,  0.28064456,  0.66878454,  0.44138609])

# A 2x5 matrix (2-D array) of integers in the interval [10, 20).
>>> np.random.randint(10, 20, (2,5))
array([[17, 12, 13, 13, 18],
       [16, 10, 12, 18, 12]])
```

Saving and Loading Arrays

It is often useful to save an array as a file. NumPy provides several easy methods for saving and loading array data.

Function	Description
`save()`	Save a single array to a `.npy` file.
`savez()`	Save multiple arrays to a `.npz` file.
`savetxt()`	Save a single array to a `.txt` file.
`load()`	Load and return an array or arrays from a `.npy` or `.npz` file.
`loadtxt()`	Load and return an array from a text file.

```
# Save a 100x100 matrix of uniformly distributed random values.
>>> x = np.random.random((100,100))
>>> np.save("uniform.npy", x)       # Or np.savetxt("uniform.txt", x).

# Read the array from the file and check that it matches the original.
>>> y = np.load("uniform.npy")      # Or np.loadtxt("uniform.txt").
>>> np.allclose(x, y)               # Check that x and y are close entry-wise.
True
```

To save several arrays to a single file, specify a keyword argument for each array in `np.savez()`. Then `np.load()` will return a dictionary-like object with the keyword parameter names from the save command as the keys.

```
# Save two 100x100 matrices of normally distributed random values.
>>> x = np.random.randn(100,100)
>>> y = np.random.randn(100,100)
>>> np.savez("normal.npz", first=x, second=y})

# Read the arrays from the file and check that they match the original.
>>> arrays = np.loadz("normal.npz")
>>> np.allclose(x, arrays["first"])
True
>>> np.allclose(y, arrays["second"])
True
```

Iterating Through Arrays

Iterating through an array (using a `for` loop) negates most of the advantages of using NumPy. Avoid iterating through arrays as much as possible by using array broadcasting and universal functions. When absolutely necessary, use `np.nditer()` to create an efficient iterator for the array. See `http://docs.scipy.org/doc/numpy/reference/arrays.nditer.html` for details.

5 Introduction to Matplotlib

Lab Objective: *Matplotlib is the most commonly-used data visualization library in Python. Being able to visualize data helps to determine patterns, to communicate results, and is a key component of applied and computational mathematics. In this lab we introduce techniques for visualizing data in 1, 2, and 3 dimensions. The plotting techniques presented here will be used in the remainder of the labs in the manual.*

Line Plots

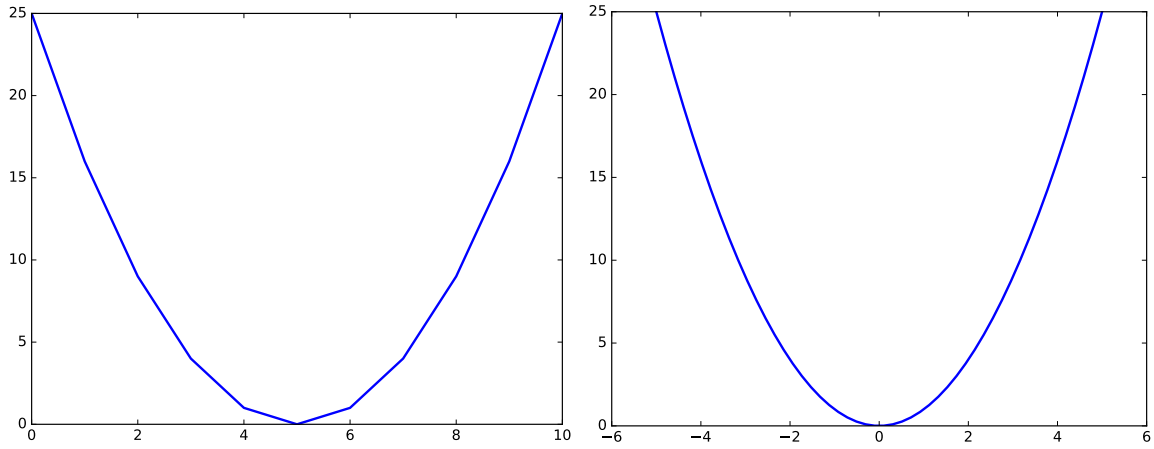
The quickest way to visualize a simple 1-dimensional array is via a *line plot*. The following code creates an array of outputs of the function $f(x) = x^2$, then visualizes the array using the `matplotlib` module.[1]

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt

>>> y = np.arange(-5,6)**2
>>> y
array([25, 16,  9,  4,  1,  0,  1,  4,  9, 16, 25])

# Visualize the plot.
>>> plt.plot(y)                     # Draw the line plot.
[<matplotlib.lines.Line2D object at 0x1084762d0>]
>>> plt.show()                      # Reveal the resulting plot.
```

The result is shown in Figure 5.1a. Just as `np` is a standard alias for NumPy, `plt` is a standard alias for `matplotlib.pyplot` in the Python community.

The call `plt.plot(y)` creates a figure and draws straight lines connecting the entries of `y` relative to the y-axis. The x-axis is by default the index of the array, namely the integers from 0 to 10. Calling `plt.show()` then displays the figure.

[1] Like NumPy, Matplotlib is *not* part of the Python standard library, but it is included in most Python distributions.



(a) `plt.plot(y)` uses the indices of the array for the x-axis.

(b) `plt.plot(x,y)` specifies both the domain and the range.

Figure 5.1: Simple plots of $f(x) = x^2$ over the interval $x \in [-5, 5]$.

Problem 1. Write a function that accepts an integer n as input.

1. Use `np.random.randn()` or `np.random.normal()` to create an $n \times n$ array of values randomly sampled from the standard normal distribution.
2. Compute the mean of each row of the array.
   (Hint: use `np.mean()` and specify the `axis` keyword argument.)
3. Return the variance of these means.
   (Hint: use `np.var()` to calcualte the variance).

Define a new function that creates an array of the results of the first function with inputs $n = 100,\ 200,\ \ldots,\ 1000$. Plot (and show) the resulting array.

This result illustrates one version of the *Law of Large Numbers*.

Specifying a Domain

An obvious problem with Figure 5.1a is that the x-axis does not correspond correctly to the y-axis for the function $f(x) = x^2$ that is being drawn. To correct this, we need an array for the domain as well as one for the range. First define an array `x` for the domain, then use that array to calculate the range `y` of f. The command `plt.plot(x,y)` then plots `x` against `y`. That is, each point `(x[i], y[i])` is drawn and consecutive points are connected.

Another problem with Figure 5.1a is its poor resolution; the curve is visibly bumpy, especially near the bottom of the curve. NumPy's `np.linspace()` function makes it easy to get a higher-resolution domain. Recall that `range()` and `np.arange()` return a list or array of evenly-spaced values in a given interval, where the *spacing* between the entries is specified. In contrast, `np.linspace()` creates an array of evenly-spaced values in a given interval where the *number of elements* is specified.

```
# 4 evenly-spaced values between 0 and 32 (including endpoints).
>>> np.linspace(0, 32, 4)
array([  0.        ,  10.66666667,  21.33333333,  32.        ])

# Get 50 evenly-spaced values from -5 to 5 (including endpoints).
>>> x = np.linspace(-5, 5, 50)
>>> y = x**2                          # Calculate the range of f(x) = x**2.
>>> plt.plot(x, y)
>>> plt.show()
```

The resulting plot is shown in Figure 5.1b. Note that this time, the x-axis is correctly aligned with the y-axis. The resolution is also much better because `x` and `y` have 50 entries each instead of only 10.

All calls to `plt` functions modify the same figure until `plt.show()` is executed, which displays the current figure and resets the system.[2] The next time a `plt` function is called a new figure is created. This makes it possible to plot several lines in a single figure.

Problem 2. Write a function that plots the functions $\sin(x)$, $\cos(x)$, and $\arctan(x)$ on the domain $[-2\pi, 2\pi]$ (use `np.pi` for π). Make sure the domain is refined enough to produce a figure with good resolution.

Note

Plotting can seem a little mystical because the actual plot doesn't appear until `plt.show()` is executed. Matplotlib's *interactive mode* allows the user to see the plot be constructed one piece at a time. Use `plt.ion()` to turn interactive mode on and `plt.ioff()` to turn it off. This is very useful for quick experimentation.

Try executing the following commands in IPython:

```
In [1]: import numpy as np
In [2]: from matplotlib import pyplot as plt

# Turn interactive mode on and make some plots.
In [3]: plt.ion()
In [4]: x = np.linspace(1, 4, 100)
In [5]: plt.plot(x, np.log(x))
In [6]: plt.plot(x, np.exp(x))

# Clear the figure, then turn interactive mode off.
In [7]: plt.clf()
In [8]: plt.ioff()
```

[2] Use `plt.figure()` to manually create several figures at once.

Use interactive mode **only** with IPython. Using interactive mode in a non-interactive setting may freeze the window or cause other problems.

Plot Customization

`plt.plot()` receives several keyword arguments for customizing the drawing. For example, the color and style of the line are specified by the following string arguments.

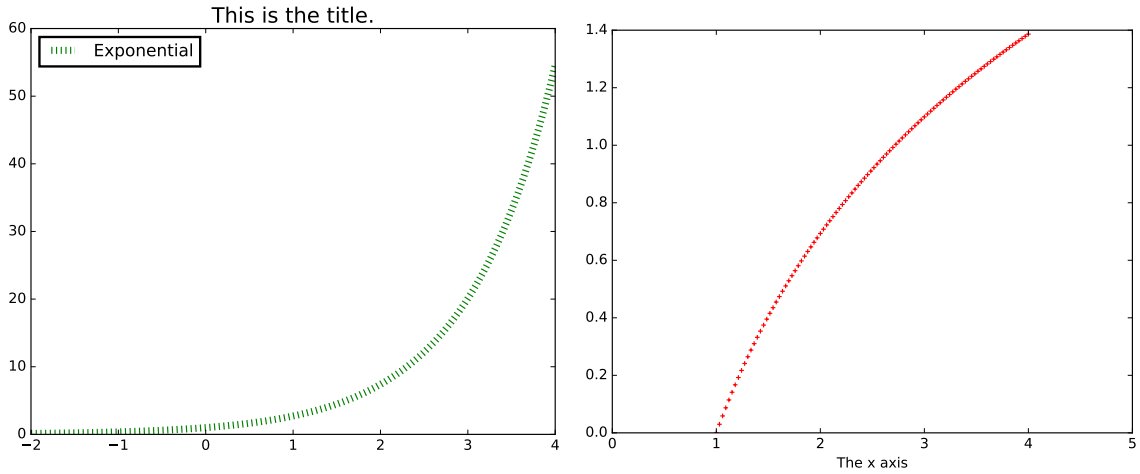
Key	Color
`'b'`	blue
`'g'`	green
`'r'`	red
`'c'`	cyan
`'m'`	magenta
`'k'`	black

Key	Style
`'-'`	solid line
`'--'`	dashed line
`'-.'`	dash-dot line
`':'`	dotted line
`'o'`	circle marker
`'+'`	plus marker

Specify one or both of these string codes as an argument to `plt.plot()` to change from the default color and style. Other `plt` functions further customize a figure.

Function	Description
`legend()`	Place a legend in the plot
`title()`	Add a title to the plot
`xlim()`	Set the limits of the x-axis
`ylim()`	Set the limits of the y-axis
`xlabel()`	Add a label to the x-axis
`ylabel()`	Add a label to the y-axis

```
>>> x1 = np.linspace(-2, 4, 100)
>>> plt.plot(x1, np.exp(x1), 'g:', linewidth=6, label="Exponential")
>>> plt.title("This is the title.", fontsize=18)
>>> plt.legend(loc="upper left")    # plt.legend() uses the 'label' argument of
>>> plt.show()                      # plt.plot() to create a legend.

>>> x2 = np.linspace(1, 4, 100)
>>> plt.plot(x2, np.log(x2), 'r+', markersize=4)
>>> plt.xlim(0, 5)                  # Set the visible limits of the x axis.
>>> plt.xlabel("The x axis")        # Give the x axis a label.
>>> plt.show()
```

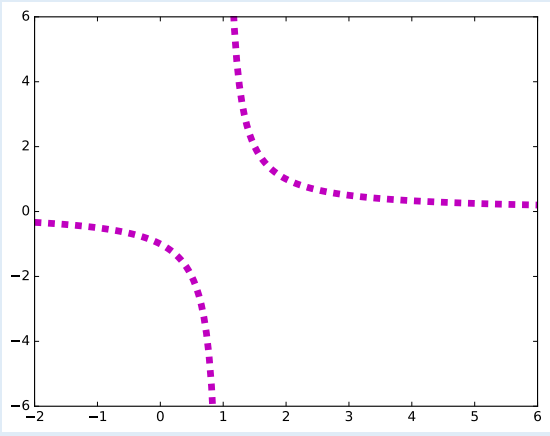


See Appendix D for more comprehensive lists of colors, line styles, and figure customization routines.

Problem 3. Write a function to plot the curve $f(x) = \frac{1}{x-1}$ on the domain $[-2, 6]$. Although $f(x)$ has a discontinuity at $x = 1$, a single call to `plt.plot()` will attempt to make the curve look continuous.

1. Split up the domain and plot the two sides of the curve separately so that the graph looks discontinuous at $x = 1$.
2. Plot both curves with a thick, dashed magenta line.
   The keyword arguments `linewidth` or `lw` specify the line thickness.
3. Change the range of the y-axis to be $[-6, 6]$.

The plot should resemble the figure below.



Subplots

Subplots are non-overlapping plots arranged in a grid within a single figure. To create a figure with a grid of subplots, use `plt.subplot(numrows, numcols, fignum)`. Here, `numrows` is the number of rows of subplots in the figure, `numcols` is the number of columns, and `fignum` specifies which subplot to modify. See Figure 5.3.

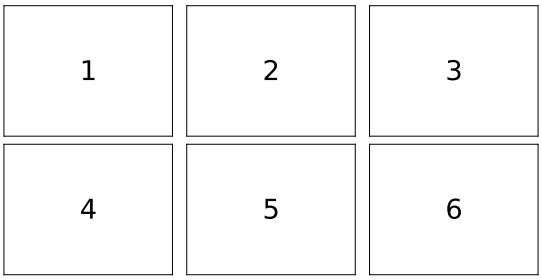
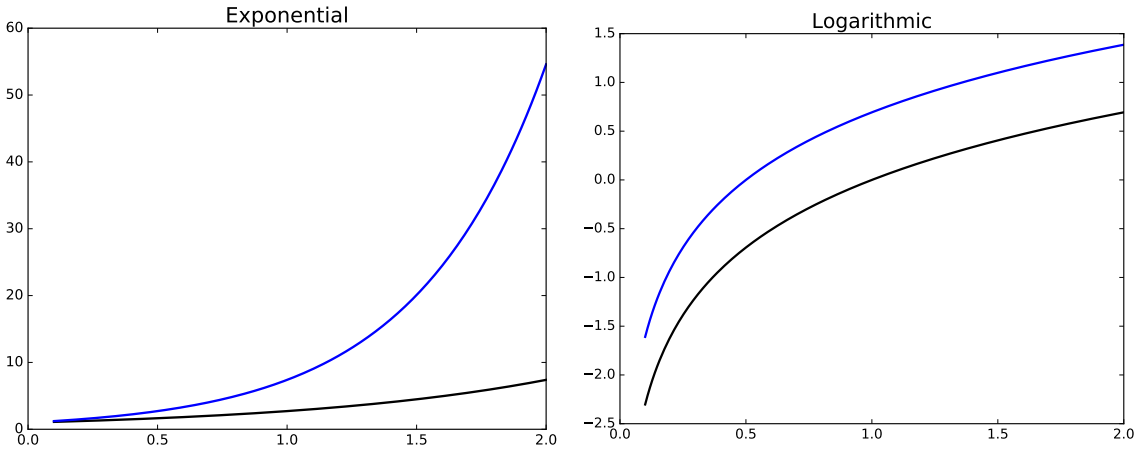


Figure 5.3: The layout of subplots with `plt.subplot(2,3,i)` (2 rows, 3 columns), where `i` is the index pictured above.

If the inputs for `plt.subplot()` are all integers, the commas between the entries can be omitted. For example, `plt.subplot(3,2,2)` and `plt.subplot(322)` are equivalent.

```
>>> x = np.linspace(.1, 2, 200)

>>> plt.subplot(121)                # Start drawing the first subplot.
>>> plt.plot(x, np.exp(x), 'k', lw=2)
>>> plt.plot(x, np.exp(2*x), 'b', lw=2)
>>> plt.title("Exponential", fontsize=18)

>>> plt.subplot(122)                # Start drawing the second subplot.
>>> plt.plot(x, np.log(x), 'k', lw=2)
>>> plt.plot(x, np.log(2*x), 'b', lw=2)
>>> plt.title("Logarithmic", fontsize=18)

>>> plt.show()
```



Problem 4. Write a function that plots the functions $\sin(x)$, $\sin(2x)$, $2\sin(x)$, and $2\sin(2x)$ on the domain $[0, 2\pi]$, each in a separate subplot.

1. Arrange the plots in a square grid of 4 subplots.
2. Set the limits of each subplot to $[0, 2\pi] \times [-2, 2]$.
   (Hint: `plt.axis()` can do this in one line, instead of using both `plt.xlim()` and `plt.ylim()`.)
3. Use `plt.title()` to give each subplot an appropriate title.
4. Use `plt.suptitle()` to give the overall figure a title.
5. Use the following colors and line styles.

   $\sin(x)$: green solid line. $\sin(2x)$: red dashed line.

   $2\sin(x)$: blue dashed line. $2\sin(2x)$: magenta dotted line.

Other Kinds of Plots

Line plots are not always the most illuminating choice graph to describe a set of data. Matplotlib provides several other easy ways to visualize data.

- A *scatter plot* plots two 1-dimensional arrays against each other without drawing lines between the points. Scatter plots are particularly useful for data that is not inherently correlated or ordered.

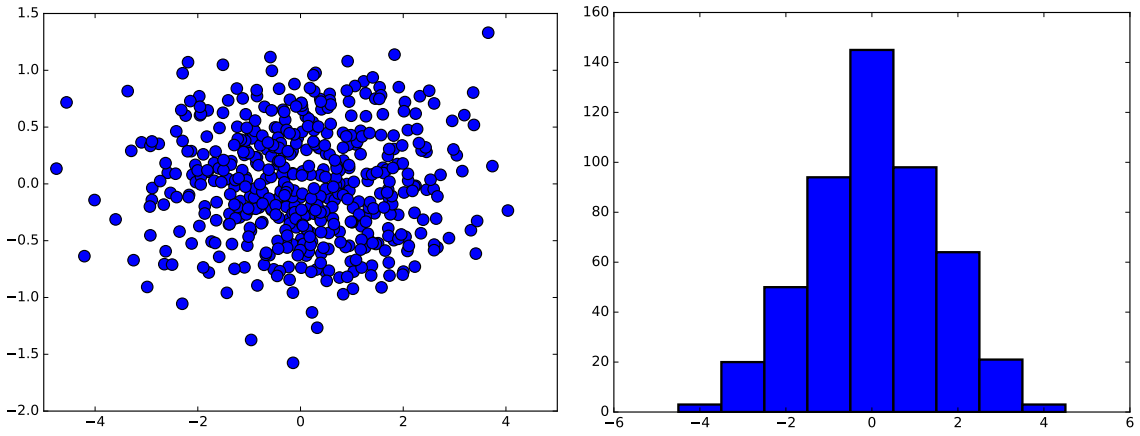
  To create a scatter plot, use `plt.plot()` and specify a point marker (such as `'o'` or `'+'`) for the line style.[3]

[3] `plt.scatter()` can also be used to create scatter plots, but it accepts slightly different arguments than `plt.plot()`. We will explore the appropriate usage of this function in a later lab.

- A *histogram* groups entries of a 1-dimensional data set into a given number of intervals, called *bins*. Each bin has a bar whose height indicates the number of values that fall in the range of the bin. Histograms are best for displaying distributions, relating data values to frequency.

  To create a histogram, use `plt.hist()` instead of `plt.plot()`. Use the argument `bins` to specify the edges of the bins, or to choose a number of bins. The `range` argument specifies the outer limits of the first and last bins.

```
# Get 500 random samples from two normal distributions.
>>> x = np.random.normal(scale=1.5, size=500)
>>> y = np.random.normal(scale=0.5, size=500)

# Draw a scatter plot of x against y, using a circle marker.
>>> plt.subplot(121)
>>> plt.plot(x, y, 'o', markersize=10)

# Draw a histogram to display the distribution of the data in x.
>>> plt.subplot(122)
>>> plt.hist(x, bins=np.arange(-4.5, 5.5))      # Or, equivalently,
#   plt.hist(x, bins=9, range=[-4.5, 4.5])

>>> plt.show()
```



Problem 5. The Fatality Analysis Reporting System (FARS) is a nationwide census providing yearly data regarding fatal injuries suffered in motor vehicle traffic crashes.[a] The array contained in `FARS.npy` is a small subset of the FARS database from 2010–2014. Each of the 148,206 rows in the array represents a different car crash; the columns represent the hour (in military time, as an integer), the longitude, and the latitude, in that order.

Write a function to visualize the data in `FARS.npy`. Use `np.load()` to load the data, then create a single figure with two subplots:

1. A scatter plot of longitudes against latitudes. Because of the large number of data points, use black pixel markers (use `"k,"` as the third argument to `plt.plot()`). Label both axes.
   (Hint: Use `plt.axis("equal")` to fix the axis ratio on the scatter plot).

2. A histogram of the hours of the day, with one bin per hour. Set the limits of the x-axis appropriately. Label the x-axis.

[a] See `http://www.nhtsa.gov/FARS`.

Visualizing 3-D Surfaces

To plot a function $f : \mathbb{R}^2 \to \mathbb{R}$, we must choose and construct a 2-dimensional domain, then calculate the function at each point of that domain. The standard tool for creating a 2-dimensional domain in the Cartesian plane is `np.meshgrid()`. Given two 1-dimensional coordinate arrays, `np.meshgrid()` creates two corresponding coordinate matrices.

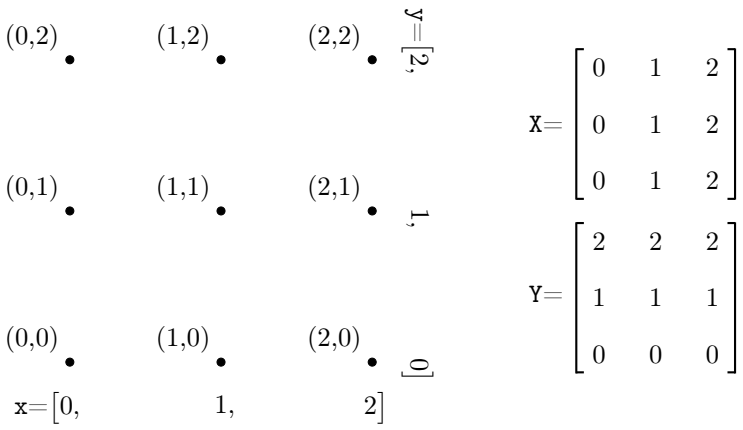


Figure 5.6: `np.meshgrid(x, y)`, returns the arrays `X` and `Y`. The returned arrays give the x- and y-coordinates of the points in the grid formed by `x` and `y`. Specifically, the arrays `X` and `Y` satisfy `(X[i,j], Y[i,j]) = (x[i],y[j])`.

With a 2-dimensional domain, we usually visualize f with two kinds of plots.

- A *heat map* assigns a color to each entry in the matrix, producing a 2-dimensional picture describing a 3-dimensional shape. Darker colors typically correspond to lower values while lighter colors typically correspond to higher values.

  Use `plt.pcolormesh()` to create a heat map.

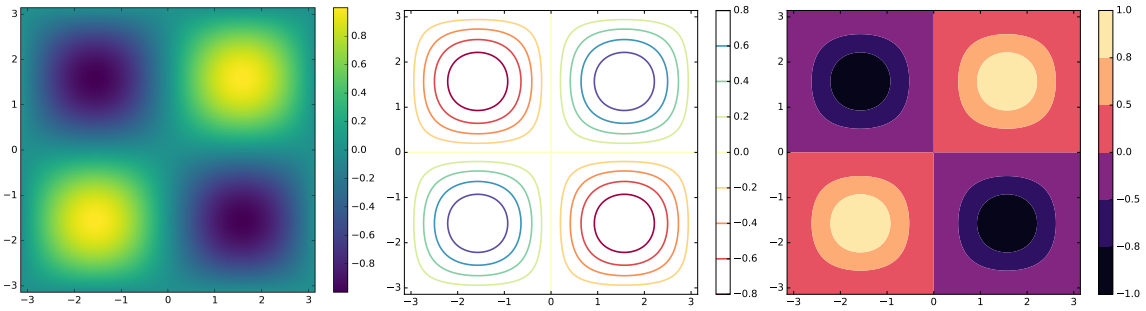
- A *contour map* draws several *level curves* of f. A level curve corresponding to the constant c is the collection of points $\{(x, y) \mid c = f(x, y)\}$. Coloring the space between the level curves produces a discretized version of a heat map. Including more and more level curves makes a filled contour plot look more and more like the complete, blended heat map.

  Use `plt.contour()` to create a contour plot and `plt.contourf()` to create a filled contour plot. Specify either the number of level curves to draw, or a list of constants corresponding to specific level curves.

These three functions all receive the keyword argument `cmap` to specify a color scheme (some of the better schemes are `"viridis"`, `"magma"`, and `"Spectral"`). See `http://matplotlib.org/examples/color/colormaps_reference.html` for the list of all Matplotlib color schemes.

Finally, to see how the colors in these plots relate to the values of the function, use `plt.colorbar()` to draw the color scale beside the plot.

```
# Create a 2-D domain with np.meshgrid().
>>> x = np.linspace(-np.pi, np.pi, 100)
>>> y = x.copy()
>>> X, Y = np.meshgrid(x, y)

# Calculate z = f(x,y) = sin(x)sin(y) using the meshgrid coordinates.
>>> Z = np.sin(X) * np.sin(Y)

# Plot the heat map of f over the 2-D domain.
>>> plt.subplot(131)
>>> plt.pcolormesh(X, Y, Z, cmap="viridis")
>>> plt.colorbar()
>>> plt.xlim(-np.pi, np.pi)
>>> plt.ylim(-np.pi, np.pi)

# Plot a contour map of f with 10 level curves.
>>> plt.subplot(132)
>>> plt.contour(X, Y, Z, 10, cmap="Spectral")
>>> plt.colorbar()

# Plot a filled contour map, specifying the level curves.
>>> plt.subplot(133)
>>> plt.contourf(X, Y, Z, [-1, -.8, -.5, 0, .5, .8, 1], cmap="magma")
>>> plt.colorbar()

>>> plt.show()
```



Problem 6. Write a function to plot $f(x, y) = \frac{\sin(x)\sin(y)}{xy}$ on the domain $[-2\pi, 2\pi] \times [-2\pi, 2\pi]$.

1. Create 2 subplots: one with a heat map of f, and one with a contour map of f. Choose an appropriate number of level curves, or specify the curves yourself.
2. Set the limits of each subplot to $[-2\pi, 2\pi] \times [-2\pi, 2\pi]$.
3. Choose a non-default color scheme.
4. Include the color scale bar for each subplot.

Note

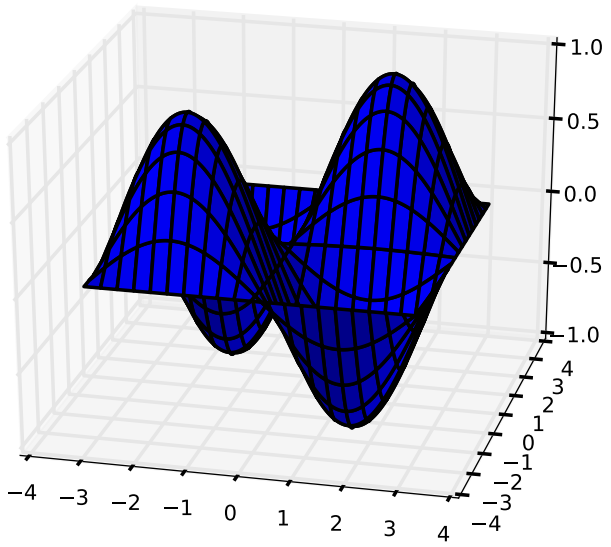
Images are usually stored as either a 2-dimensional array (for black-and-white pictures) or a 3-dimensional array (a stack of 2-dimensional arrays, one for each RGB value). This kind of data does not require a domain, and is easily visualized with `plt.imshow()`.

Additional Material

3-D Plotting

Matplotlib can also be used to plot 3-dimensional surfaces. The following code produces the surface corresponding to $f(x, y) = \sin(x)\sin(y)$.

```
# Create the domain and calculate the range like usual.
>>> x = np.linspace(-np.pi, np.pi, 200)
>>> y = np.copy(x)
>>> X, Y = np.meshgrid(x, y)
>>> Z = np.sin(X)*np.sin(Y)

# Draw the corresponding 3-D plot using some extra tools.
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = plt.figure()
>>> ax = fig.gca(projection='3d')
>>> ax.plot_surface(X, Y, Z)
>>> plt.show()
```



Animations

Lines and other graphs can be altered dynamically to produce animations. Follow these steps to create a Matplotlib animation:

1. Calculate all data that is needed for the animation.

2. Define a figure explicitly with `plt.figure()` and set its window boundaries.

3. Draw empty objects that can be altered dynamically.
4. Define a function to update the drawing objects.
5. Use `matplotlib.animation.FuncAnimation()`.

The submodule `matplotlib.animation` contains the tools putting together and managing animations. The function `matplotlib.animation.FuncAnimation()` accepts the figure to animate, the function that updates the figure, the number of frames to show before repeating, and how fast to run the animation (lower numbers mean faster animations).

```
from matplotlib.animation import FuncAnimation

def sine_animation():
    # Calculate the data to be animated.
    x = np.linspace(0, 2*np.pi, 200)[:-1]
    y = np.sin(x)

    # Create a figure and set its window boundaries.
    fig = plt.figure()
    plt.xlim(0, 2*np.pi)
    plt.ylim(-1.2, 1.2)

    # Draw an empty line. The comma after 'drawing' is crucial.
    drawing, = plt.plot([],[])

    # Define a function that updates the line data.
    def update(index):
        drawing.set_data(x[:index], y[:index])
        return drawing,                    # Note the comma!

    a = FuncAnimation(fig, update, frames=len(x), interval=10)
    plt.show()
```

Try using the following function in place of `update()`. Can you explain why this animation is different from the original?

```
def wave(index):
    drawing.set_data(x, np.roll(y, index))
    return drawing,
```

To animate multiple objects at once, define the objects separately and make sure the update function returns both objects.

```
def sine_cosine_animation():
    x = np.linspace(0, 2*np.pi, 200)[:-1]
    y1, y2 = np.sin(x), np.cos(x)

    fig = plt.figure()
    plt.xlim(0, 2*np.pi)
```

```
    plt.ylim(-1.2, 1.2)

    sin_drawing, = plt.plot([],[])
    cos_drawing, = plt.plot([],[])

    def update(index):
        sin_drawing.set_data(x[:index], y1[:index])
        cos_drawing.set_data(x[:index], y2[:index])
        return sin_drawing, cos_drawing,

    a = FuncAnimation(fig, update, frames=len(x), interval=10)
    plt.show()
```

Animations are very useful for describing parametrized curves, as the "speed" of the curve is displayed. The code below animates the rose curve, parametrized by the angle $\theta \in [0, 2\pi]$, given by the following equations:

$$x(\theta) = \cos(\theta)\cos(6\theta), \qquad y(\theta) = \sin(\theta)\cos(6\theta)$$

```
def rose_animation():
    # Calculate the parametrized data.
    theta = np.linspace(0, 2*np.pi, 200)
    x = np.cos(theta)*np.cos(6*theta)
    y = np.sin(theta)*np.cos(6*theta)

    fig = plt.figure()
    plt.xlim(-1.2, 1.2)
    plt.ylim(-1.2, 1.2)
    plt.gca().set_aspect("equal")           # Make the figure exactly square.

    drawing, = plt.plot([],[])

    # Define a function that updates the line data.
    def update(index):
        drawing.set_data(x[:index], y[:index])
        return drawing,

    a = FuncAnimation(fig, update, frames=len(x), interval=10, repeat=False)
    plt.show()              # repeat=False freezes the animation at the end.
```

Animations can also be 3-dimensional. The only major difference is an extra operation to set the 3-dimensional component of the drawn object. The code below animates the space curve parametrized by the following equations:

$$x(\theta) = \cos(\theta)\cos(6\theta), \qquad y(\theta) = \sin(\theta)\cos(6\theta), \qquad z(\theta) = \tfrac{\theta}{10}$$

```
def rose_animation_3D():
    theta = np.linspace(0, 2*np.pi, 200)
```

```
x = np.cos(theta) * np.cos(6*theta)
y = np.sin(theta) * np.cos(6*theta)
z = theta / 10

fig = plt.figure()
ax = fig.gca(projection='3d')           # Make the figure 3-D.
ax.set_xlim3d(-1.2, 1.2)                # Use ax instead of plt.
ax.set_ylim3d(-1.2, 1.2)
ax.set_aspect("equal")

drawing, = ax.plot([],[],[])            # Provide 3 empty lists.

# Update the first 2 dimensions like usual, then update the 3-D component.
def update(index):
    drawing.set_data(x[:index], y[:index])
    drawing.set_3d_properties(z[:index])
    return drawing,

a = FuncAnimation(fig, update, frames=len(x), interval=10, repeat=False)
plt.show()
```

6 Exceptions and File I/O

Lab Objective: *In Python, an* exception *is an error detected during execution. Exceptions are important for regulating program usage and for correctly reporting problems to the programmer and end user. Understanding exceptions allows us to safely read data from and export data to external files, and being able to read from and write to files is important to analyzing data and communicating results. In this lab we present exception syntax and file interaction protocols.*

Exceptions

Every programming language has a formal way of indicating and handling errors. In Python, we raise and handle *exceptions*. Some of the more common exceptions are listed below, along with the kinds of problems that they typically indicate.

Exception	Indication
`AttributeError`	An attribute reference or assignment failed.
`IndexError`	A sequence subscript was out of range.
`NameError`	A local or global name was not found.
`SyntaxError`	The parser encountered a syntax error.
`TypeError`	An operation or function was applied to an object of inappropriate type.
`ValueError`	An operation or function received an argument that had the right type but an inappropriate value.

```
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined

>>> [1, 2, 3].fly()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'fly'
```

See `https://docs.python.org/2/library/exceptions.html` for the complete list of Python's built-in exception classes.

Raising Exceptions

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

Exception objects can be initialized with any number of arguments. These arguments are stored as a tuple attribute called `args`, which serves as the string representation of the object. We typically provide a single string detailing the reasons for the error.

```
# Raise a generic exception, without an error message.
>>> if 7 is not 7.0:
...     raise Exception
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception

# Now raise a more specific exception, with an error message included.
>>> for x in range(10):
...     if x > 5:
...         raise ValueError("'x' should not exceed 5.")
...     print(x),
...
0 1 2 3 4 5
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ValueError: 'x' should not exceed 5.
```

Problem 1. Consider the following arithmetic "magic" trick.

1. Choose a 3-digit number where the first and last digits differ by 2 or more (for example, 123).
2. Reverse this number by reading it backwards (321).
3. Calculate the positive difference of these numbers ($321 - 123 = 198$).
4. Add the reverse of the result to itself ($198 + 891 = 1089$).

The result of the last step will *always* be 1089, regardless of the original number chosen in step 1 (can you explain why?).

The following function prompts the user for input at each step of the magic trick, but does not check that the user's inputs are correct.

```
def arithmagic():
    step_1 = raw_input("Enter a 3-digit number where the first and last "
                                            "digits differ by 2 or more: ")
    step_2 = raw_input("Enter the reverse of the first number, obtained "
                                              "by reading it backwards: ")
    step_3 = raw_input("Enter the positive difference of these numbers: ")
    step_4 = raw_input("Enter the reverse of the previous result: ")
    print str(step_3) + " + " + str(step_4) + " = 1089 (ta-da!)"
```

Modify `arithmagic()` so that it verifies the user's input at each step. Raise a `ValueError` with an informative error message if any of the following occur:

1. The first number (`step_1`) is not a 3-digit number.
2. The first number's first and last digits differ by less than 2.
3. The second number (`step_2`) is not the reverse of the first number.
4. The third number (`step_3`) is not the positive difference of the first two numbers.
5. The fourth number (`step_4`) is not the reverse of the third number.

(Hint: `raw_input()` always returns a string, so each variable is a string initially. Use `int()` to cast the variables as integers when necessary. The built-in function `abs()` may also be useful.)

Handling Exceptions

To prevent an exception from halting the program, it must be handled by placing the problematic lines of code in a `try` block. An `except` block then follows.

```
# The 'try' block should hold any lines of code that might raise an exception.
>>> try:
...     raise Exception("for no reason")
...     print "No exception raised"
... # The 'except' block is executed when an exception is raised.
... except Exception as e:
...     print "Exception raised", e
...
Exception raised for no reason
>>> # The program then continues on.
```

In this example, the name `e` represents the exception within the except block. Printing `e` displays its error message.

The try-except control flow can be expanded with two other blocks. The flow proceeds as follows:

1. The `try` block is executed until an exception is raised, if at all.

2. An `except` statement specifying the same kind of exception that was raised in the try block "catches" the exception, and the block is then executed. There may be multiple except blocks following a single try block (similiar to having several `elif` statements following a single `if` statement), and a single except statement may specify multiple kinds of exceptions to catch.

3. The optional `else` block is executed if an exception was *not* raised in the try block. Thus either an except block or the else block is executed, but not both.

4. Lastly, the optional `finally` block is always executed if it is included.

```
>>> try:
...     raise ValueError("The house is on fire!")
... # Check for multiple kinds of exceptions using parentheses.
... except (ValueError, TypeError) as e:
...     house_on_fire = True
... else:                           # Skipped due to the exception.
...     house_on_fire = False
... finally:
...     print "The house is on fire:", house_on_fire
...
The house is on fire: True

>>> try:
...     house_on_fire = False
... except Exception as e:
...     house_on_fire = True
... else:                           # Executed because there was no exception.
...     print "The house is probably okay."
... finally:
...     print "The house is on fire:", house_on_fire
...
The house is probably okay.
The house is on fire: False
```

The code in the `finally` block is *always* executed, even if a `return` statement or an uncaught exception occurs in any block following the try statement.

```
>>> try:
...     raise ValueError("The house is on fire!")
... finally:                        # Executes before the error is reported.
...     print "The house may be on fire."
...
The house may be on fire.
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: The house is on fire!
```

See `https://docs.python.org/2/tutorial/errors.html` for more examples.

Problem 2. A *random walk* is a path created by a series of random steps. The following function simulates a random walk where at every step, we either step forward (adding 1 to the total) or backward (adding -1).

```
from random import choice

def random_walk(max_iters=1e12):
    walk = 0
    direction = [1, -1]
    for i in xrange(int(max_iters)):
        walk += choice(direction)
    return walk
```

A `KeyboardInterrupt` is a special exception that can be triggered at any time by entering `ctrl c` (on most systems) in the keyboard. Modify `random_walk()` so that if the user raises a `KeyboardInterrupt`, the function handles the exception and prints "Process interrupted at iteration *i*". If no `KeyboardInterrupt` is raised, print "Process completed". In both cases, return `walk` as before.

Note

The built-in Python exceptions are organized into a class hierarchy. This can lead to some confusing behavior.

```
>>> try:
...     raise ValueError("This is a ValueError!")
... except StandardError as e:
...     print(e)
...
This is a ValueError!                       # The exception was caught.

>>> try:
...     raise StandardError("This is a StandardError!")
... except ValueError as e:
...     print(e)
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
StandardError: This is a StandardError!     # The exception wasn't caught!
```

It turns out that the `ValueError` class inherits from the `StandardError` class. Thus a `ValueError` *is* a `StandardError`, so the first except statement was able to catch the exception, but a `StandardError` is *not* a `ValueError`, so the second except statement could not catch the exception.

The complete exception class hierarchy is documented at `https://docs.python.org/2/library/exceptions.html#exception-hierarchy`.

File Input and Output

Python has a useful `file` object that acts as an interface to all kinds of different file streams. The built-in function `open()` creates a file object. It accepts the name of the file to open and an editing mode. The mode determines the kind of access to use when opening the file. There are three common options:

`'r'`: **read**. This is the default mode. The file must already exist.

`'w'`: **write**. This mode creates the file if it doesn't already exist and **overwrites everything** in the file if it does already exist.

`'a'`: **append**. New data is written to the end of the file. This mode also creates a new file if it doesn't already exist.

```
>>> myfile = open("in.txt", 'r')        # Open 'in.txt' with read-only access.
>>> print(myfile.read())                # Print the contents of the file.
Hello World!                            # (it's a really small file.)
>>> myfile.close()                      # Close the file connection.
```

The With Statement

An `IOError` indicates that some input or output operation has failed. If a file cannot be opened for any reason, an `IOError` is raised and the file object is not initialized. A simple `try-finally` control flow can ensure that a file stream is closed safely.

The `with` statement provides an alternative method for safely opening and closing files. Use `with open(<filename>, <mode>) as <alias>:` to create an indented block in which the file is open and available under the specified alias. At the end of the block, the file is automatically closed. This is the preferred file-reading method when a file only needs to be accessed briefly. The more flexible `try-finally` approach is typically better for working with several file streams at once.

```
>>> myfile = open("in.txt", 'r')        # Open 'in.txt' with read-only access.
>>> try:
...     contents = myfile.readlines()   # Read in the content by line.
... finally:
...     myfile.close()                  # Explicitly close the file.

# Equivalently, use a 'with' statement to take care of errors.
>>> with open("in.txt", 'r') as myfile: # Open 'in.txt' with read-only access.
...     contents = myfile.readlines()   # Read in the content by line.
...                                     # The file is closed automatically.
```

In both cases, if the file `in.txt` does not exist in the current directory, an `IOError` will be raised. However, errors in the `try` or `with` blocks will not prevent the file from being safely closed.

Reading and Writing

Open file objects have an implicit *cursor* that determines the location in the file to read from or write to. After the entire file has been read once, either the file must be closed and reopened, or the cursor must be reset to the beginning of the file with `seek(0)` before it can be read again.

Some of more important file object attributes and methods are listed below.

Attribute	Description
`closed`	`True` if the object is closed.
`mode`	The access mode used to open the file object.
`name`	The name of the file.

Method	Description
`close()`	Close the connection to the file.
`read()`	Read a given number of bytes; with no input, read the entire file.
`readline()`	Read a line of the file, including the newline character at the end.
`readlines()`	Call `readline()` repeatedly and return a list of the resulting lines.
`seek()`	Move the cursor to a new position.
`tell()`	Report the current position of the cursor.
`write()`	Write a single string to the file (spaces are *not* added).
`writelines()`	Write a list of strings to the file (newline characters are *not* added).

Only strings can be written to files; to write a non-string type, first cast it as a string with `str()`. Be mindful of spaces and newlines to separate the data.

```
>>> with open("out.txt", 'w') as outfile:    # Open 'out.txt' for writing.
...     for i in xrange(10):
...         outfile.write(str(i**2)+' ')    # Write some strings (and spaces).
...
>>> outfile.closed                          # The file is closed automatically.
True
```

Executing this code replaces whatever used to be in `out.txt` (whether or not it existed previously) with the following:

```
0 1 4 9 16 25 36 49 64 81
```

Problem 3. Define a class called `ContentFilter`. Implement the constructor so that it accepts the name of a file to be read.

1. If the filename argument is not a string, raise a `TypeError`.

   (Hint: The built-in functions `type()` and `isinstance()` may be useful.)

2. Read the file and store its name and contents as attributes (store the contents as a single string). Securely close the file stream.

String Formatting

Python's `str` type class has several useful methods for parsing and formatting strings. They are particularly useful for processing data from a source file and for preparing data to be written to an external file.

Method	Returns
`count()`	The number of times a given substring occurs within the string.
`find()`	The lowest index where a given substring is found.
`isalpha()`	`True` if all characters in the string are alphabetic (a, b, c, ...).
`isdigit()`	`True` if all characters in the string are digits (0, 1, 2, ...).
`isspace()`	`True` if all characters in the string are whitespace (`" "`, `'\t'`, `'\n'`).
`join()`	The concatenation of the strings in a given iterable with a specified separator between entries.
`lower()`	A copy of the string converted to lowercase.
`upper()`	A copy of the string converted to uppercase.
`replace()`	A copy of the string with occurrences of a given substring replaced by a different specified substring.
`split()`	A list of segments of the string, using a given character or string as a delimiter.
`strip()`	A copy of the string with leading and trailing whitespace removed.

The `join()` method translates a list of strings into a single string by concatenating the entries of the list and placing the principal string between the entries. Conversely, `split()` translates the principal string into a list of substrings, with the separation determined by the a single input.

```
# str.join() puts the string between the entries of a list.
>>> words = ["state", "of", "the", "art"]
>>> "-".join(words)
'state-of-the-art'

>>> " o_O ".join(words)
'state o_O of o_O the o_O art'

# str.split() creates a list out of a string, given a delimiter.
>>> "One fish\nTwo fish\nRed fish\nBlue fish\n".split('\n')
['One fish', 'Two fish', 'Red fish', 'Blue fish', '']

# If no delimiter is provided, the string is split by its whitespace characters
    .
>>> "One fish\nTwo fish\nRed fish\nBlue fish\n".split()
['One', 'fish', 'Two', 'fish', 'Red', 'fish', 'Blue', 'fish']
```

Can you tell the difference between the following routines?

```
>>> with open("in.txt", 'r') as myfile:
...     contents = myfile.readlines()
...
>>> with open("in.txt", 'r') as myfile:
```

```
...     contents = myfile.read().split('\n')
```

Problem 4. Add the following methods to the `ContentFilter` class for writing the contents of the original file to new files. Each method should accept a the name of a file to write to and a keyword argument `mode` that specifies the file access mode, defaulting to `'w'`. If `mode` is not either `'w'` or `'a'`, raise a `ValueError` with an informative message.

1. `uniform()`: write the data to the outfile with uniform case. Include an additional keyword argument `case` that defaults to `"upper"`.

   If `case="upper"`, write the data in upper case. If `case="lower"`, write the data in lower case. If `case` is not one of these two values, raise a `ValueError`.

2. `reverse()`: write the data to the outfile in reverse order. Include an additional keyword argument `unit` that defaults to `"line"`.

   If `unit="word"`, reverse the ordering of the words in each line, but write the lines in the same order as the original file. If `unit="line"`, reverse the ordering of the lines, but do not change the ordering of the words on each individual line. If `unit` is not one of these two values, raise a `ValueError`.

3. `transpose()`: write a "transposed" version of the data to the outfile. That is, write the first word of each line of the data to the first line of the new file, the second word of each line of the data to the second line of the new file, and so on. Viewed as a matrix of words, the rows of the input file then become the columns of the output file, and vice versa. You may assume that there are an equal number of words on each line of the input file.

Also implement the `__str__()` magic method so that printing a `ContentFilter` object yields the following output:

```
Source file:              <filename>
Total characters:         <The total number of characters in the file>
Alphabetic characters:    <The number of letters>
Numerical characters:     <The number of digits>
Whitespace characters:    <The number of spaces, tabs, and newlines>
Number of lines:          <The number of lines>
```

Additional Material

Custom Exception Classes

Custom exceptions can be defined by writing a class that inherits from some existing exception class. The generic `Exception` class is typically the parent class of choice.

```
>>> class TooHardError(Exception):
...     pass
...
>>> raise TooHardError("This lab is impossible!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.TooHardError: This lab is impossible!
```

This may seem like a trivial extension of the `Exception` class, but it is useful to do because the interpreter never automatically raises a `TooHardError`. Any `TooHardError` must have originated from a hand-written `raise` command, making it easier to identify the exact source of the problem.

Assertions

An `AssertionError` is a special exception that is used primarily for software testing. The `assert` statement is a shortcut for testing the truthfulness of an expression and raising an `AssertionError`.

```
>>> assert <expression>, <message>

# Equivalently...
>>> if not <expression>:
...     raise AssertionError(<message>)
...
```

The CSV Module

The CSV format (comma separated value) is a common file format for spreadsheets and grid-like data. The `csv` module in the standard library contains functions that work in conjunction with the built-in `file` object to read from or write to CSV files. See `https://docs.python.org/2/library/csv.html` for details.

More String Formatting

Concatenating string values with non-string values is often cumbersome and tedious. Consider the problem of printing a simple date:

```
>>> day, month, year = 14, "March", 2015
>>> print("Is today " + str(day) + str(month) + ", " + str(year) + "?")
Is today 14 March, 2015?
```

The `str` class's `format()` method makes it easier to insert non-string values into the middle of a string. Write the desired output in its entirety, replacing non-string values with curly braces `{}`. Then use the `format()` method, entering each replaced value in order.

```
>>> print("Is today {} {}, {}?".format(day, month, year))
Is today 14 March, 2015?
```

This method is extremely flexible and provides many convenient ways to format string output nicely. Suppose, for example, that we would like to visualize the progress that a program is making through a loop. The following code prints out a simple status bar.

```
>>> from sys import stdout

>>> iters = int(1e7)
>>> chunk = iters // 20
>>> for i in xrange(iters):
...     print("\r[{:<20}] i = {}".format('='*((i//chunk)+1), i)),
...     stdout.flush()
...
```

Here the string `"\r[{:<20}]"` used in conjunction with the `format()` method tells the cursor to go back to the beginning of the line, print an opening bracket, then print the first argument of `format()` left-aligned with at least 20 total spaces before printing the closing bracket. The comma after the print command suppresses the automatic newline character, keeping the output of each individual print statement on the same line. Finally, `sys.stdout.flush()` flushes the internal buffer so that the status bar is printed in real time.

Of course, printing at each iteration dramatically slows down the progression through the loop. How does the following code solve that problem?

```
>>> for i in xrange(iters):
...     if not i % chunk:
...         print "\r[{:<20}] i = {}".format('='*((i//chunk)+1), i),
...         stdout.flush()
...
```

See `https://docs.python.org/2/library/string.html#format-string-syntax` for more examples and specific syntax for using `str.format()`.

7 Data Visualization

Lab Objective: *Correctly presenting and interpreting data with visualizations is both a science and an art. In this lab we discuss and demonstrate the principles of good data visualization, including the key characteristics of good graphs, common mistakes, and how to avoid unethical visualizations.*

We strongly recommend completing this lab as a Jupyter Notebook.

The Importance of Visualizations

Visualizations of data often reveal insights that may not immediately obvious from simple statistics. The following data set, known as *Anscombe's quartet*, is a famous example of the importance of graphing data.

I		II		III		IV	
x	y	x	y	x	y	x	y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

Each section of Anscombe's quartet shares the following statistical properties:

- The means are 9 for x and 7.5 for y.
- The sample variances are 11 for x and 3.75 for y.
- The correlation between x and y is .816.
- The linear least squares regression line is $y = \frac{1}{2}x + 3$.

Despite these similarities, each section is quite different from the others.

Problem 1. The file `anscombe.npy` contains Anscombe's quartet in the same format as displayed above. Plot each section of the quartet separately as a scatter plot. Also plot the regression line $y = \frac{1}{2}x + 3$ on the domain $x \in [0, 20]$ over each scatter plot.

Write a few sentences describing what makes each section unique.

Principles of Good Data Visualization

Understanding a data set through visualizations is an iterative process. Examine the data, start with an initial visualization, then adjust the original visualization or create new ones. Ask the following questions while searching for insights:

1. Does the visualization represent the data accurately?
2. Would a different visualization communicate more information?
3. Would visualizing a subset of the data provide more information?
4. Would transforming the data reveal a hidden pattern?

Effectively visualizing data involves technical expertise combined with design knowledge. No visualization is perfect, but every good visualization must contain the following essential elements.

- **Clarity**. Good visualizations are as self-explanatory as possible. Always use specific titles and axis labels, and include units of measure. Use a legend or other annotations where appropriate.
- **Simplicity**. A good visualization communicates everything that it needs to, but nothing more. Anything in a plot that fails to communicate information or that misrepresents the data is called *chartjunk*, a term coined by Edward Tufte. Make visualizations as simple, clean, and readable as possible.
- **Integrity**. Tell the truth, the whole truth, and nothing but the truth. It is usually alarmingly easy to manipulate a visualization so that it supports a specific agenda. Resist the temptation to morph a visualization into something that misrepresents the true nature of the data, even though that misrepresentation might support your hypotheses about the data.

  Every visualization should be presented together with information on who created it, where the data was obtained, how it was collected, whether it was cleaned or transformed, and whether there are conflicts of interest or possible biases present. Cite your sources.

This list could be expanded, but virtually every good data visualization principle fits into these three categories in one way or another.

Improving Specific Types of Visualizations

Data can be visualized in many forms and styles. However, Most data sets are more naturally described with one type of visualization than another. In the following sections, we explore how the plots we commonly use can be improved and refined.

Line Plots

A line plot connects ordered (x, y) points with straight lines, and is therefore best for visualizing one or two ordered arrays, such as functional outputs over an ordered domain or a sequence of values over time.

When creating a line plot, consider the following details.

- What do the axes represent? How should they be labeled?
- Would a linear scale or a logarithmic scale most clearly reveal patterns?
- What should the window limits be?
- Is the line an appropriate thickness and color?
- Should each point be distinctly marked, or is a smooth line preferable?
- Should multiple lines be in the same plot, or in separate subplots?

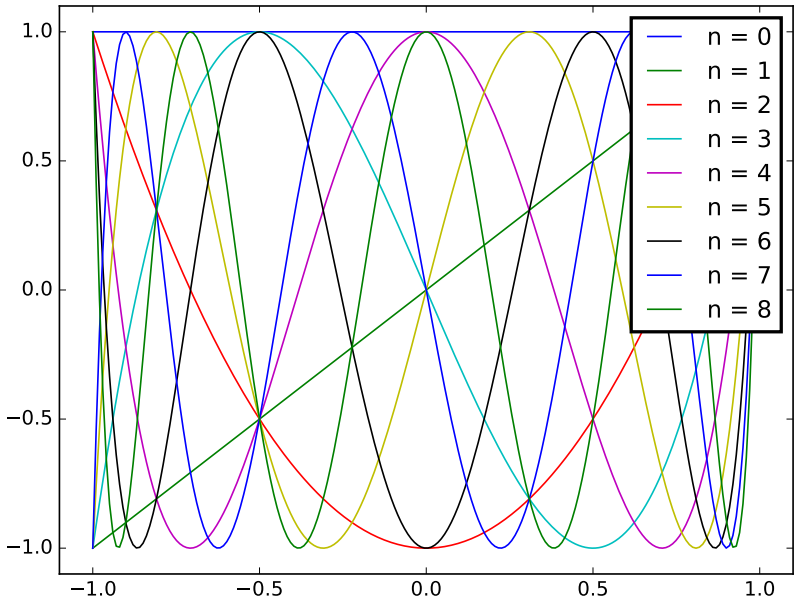
The *Chebyshev polynomials* are a family of orthogonal polynomials that are recursively defined as follows.

$$T_0(x) = 1 \qquad T_1(x) = x \qquad T_{n+1} = 2xT_n(x) - T_{n-1}(x)$$

NumPy's `polynomial` module has a convenient tool for constructing these and other important polynomials.[1] However, plotting several the polynomials on top of each other, especially with a legend, results in a very cluttered visual.

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt
>>> % matplotlib inline                    # Display notebook plots inline.

# Plot the first 9 Chebyshev polynomials in the same plot.
>>> T = np.polynomial.Chebyshev.basis
>>> x = np.linspace(-1, 1, 200)
>>> for n in range(9):
...     plt.plot(x, T(n)(x), label="n = "+str(n))
...
>>> plt.axis([-1.1, 1.1, -1.1, 1.1])       # Set the window limits.
>>> plt.legend()
```

[1] `numpy.polynomial` also has tools for computing other important polynomial families, including the Legendre, Hermite, and Laguerre polynomials.



This line plot can be improved in several easy ways.

1. Use subplots to split the visualization into smaller, comparable pieces. Instead of using a legend, give each subplot a title. This method, called *small multiples*, was made famous by Edward Tufte.

2. Increase the line thicknesses to 2 or 3 (the default is 1).

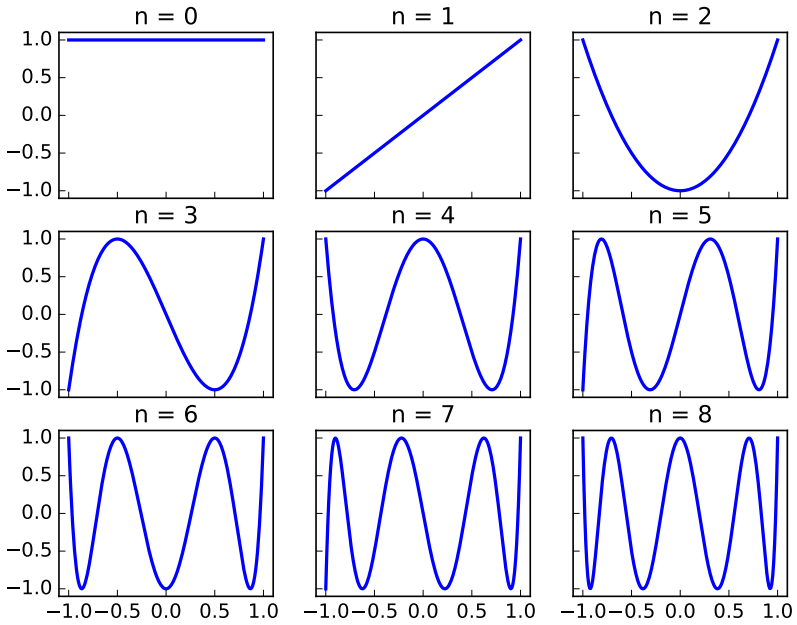
3. Remove extra tick marks and axis labels.

Matplotlib's `plt.tick_params()`, summarized below, controls which tick marks and labels are displayed.

Argument	Options	Description
`axis`	`'x'`, `'y'`, `"both"`	Axis on which to operate.
`which`	`"major"`, `"minor"`, `"both"`	Operate on major or minor ticks.
`color`	Any Matplotlib color	Tick color.
`labelcolor`	Any Matplotlib color	Tick label color.
`bottom`, `top`, `left`, `right`	`"on"`, `"off"`	Turn ticks on or off.
`labelbottom`, `labeltop`, `labelleft`, `labelright`	`"on"`, `"off"`	Turn tick labels on or off.

```
>>> for n in range(9):
...     plt.subplot(3, 3, n+1)
...     plt.plot(x, T(n)(x), lw=2)
...     plt.axis([-1.1, 1.1, -1.1, 1.1])
...
...     # Turn off extra tick marks and axis labels.
...     plt.tick_params(which="both", top="off", right="off")
...     if n < 6:                   # Remove x-axis label on upper plots.
```

```
...         plt.tick_params(labelbottom="off")
...     if n % 3:                   # Remove y-axis label on right plots.
...         plt.tick_params(labelleft="off")
...     plt.title("n = "+str(n))
```



Note

Matplotlib titles and annotations can be formatted with LaTeX, a system for creating technical documents.[a] To do so, use an 'r' before the string quotation mark and surround the text with dollar signs. For example, try replacing the final line of code in the previous example with the following line.

```
...     plt.title(r"$T_{}(x)$".format(n))
```

The string's `format()` method inserts the input n at the curly braces. The title of the sixth subplot, instead of being "n = 5," will then be "$T_5(x)$."

[a] See `http://www.latex-project.org/` for more information.

Problem 2. The $n+1$ Bernstein basis polynomials of degree n are defined as follows.

$$b_{v,n}(x) = \binom{n}{v} x^v (1-x)^{n-v}, \qquad v = 0,\ 1,\ \ldots,\ n$$

Plot at least the first 10 Bernstein basis polynomials ($n = 0,\ 1,\ 2,\ 3$) as small multiples on the domain $[0, 1] \times [0, 1]$. Label the subplots for clarity, adjust tick marks and labels for simplicity, and set the window limits of each plot to be the same. Consider arranging the subplots so that the rows correspond with n and the columns with v.
(Hint: The constant $\binom{n}{v} = \frac{n!}{v!(n-v)!}$ is called the *binomial coefficient* and can be efficiently computed with `scipy.special.binom()` or `scipy.misc.comb()`.)

Scatter Plots

A scatter plot draws (x, y) points without connecting them. Connecting the points would imply an order or relation between the points, so scatter plots are best for displaying data sets without a natural order, or where each point is a distinct, individual instance.

Consider the following questions when making a scatter plot. Note that some of them are the same questions that should be asked when creating a line plot.

- What do the axes represent? How should they be labeled?
- Would a linear scale or a logarithmic scale most clearly reveal patterns?
- What should the window limits be?
- Which marker is best? Are the markers an appropriate size and color?

A scatter plot can be drawn with either `plt.plot()` (specify a point marker such as `'.'`, `','`, `'o'`, or `'+'`) or `plt.scatter()`. While `plt.plot()` is the more flexible function in general, `plt.scatter()` provides a few extra tools. Most useful are the keywords `s` and `c`, which correspond to marker size and marker color, respectively. Each keyword can either be a single entry or an array. Using an array specifies the sizes or colors of each individual marker, allowing a scatter plot to have up to four dimensions of information.

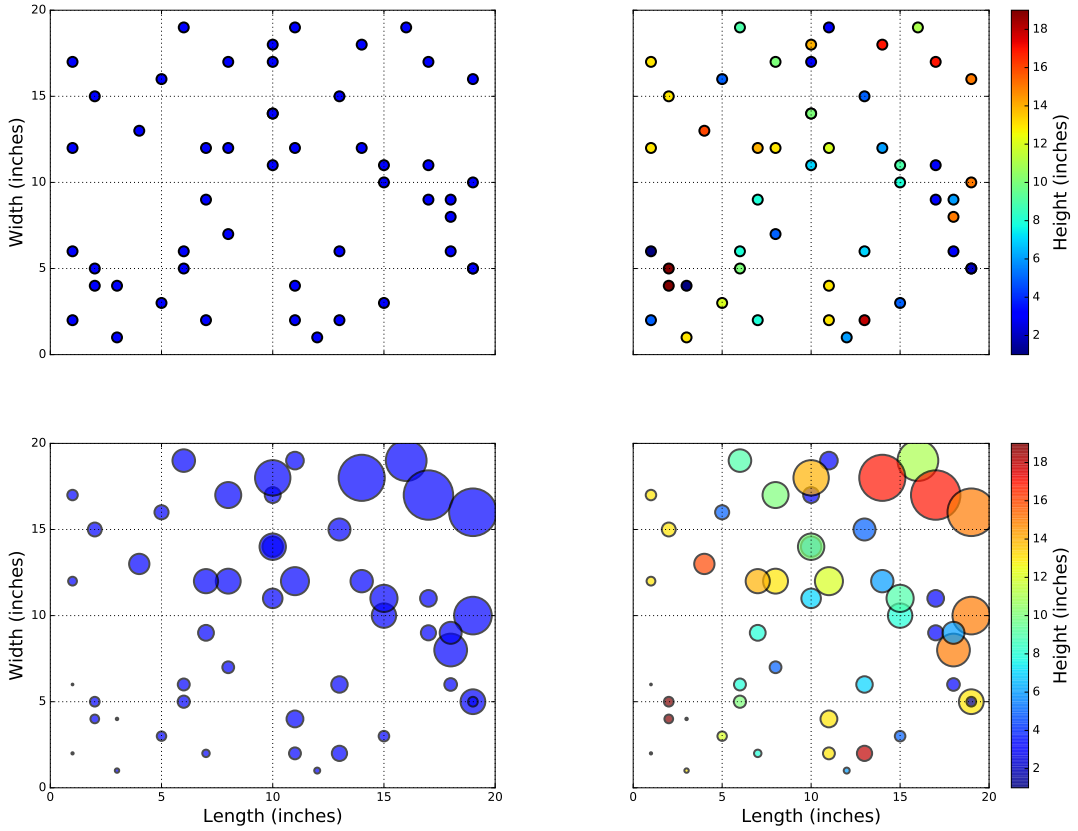
Consider a collection of rectangular boxes where the lengths, widths, and heights are given. A scatter plot of length against width mostly describes the sizes of the boxes; tying the third dimension (height) to the color of the points can provide the additional information. Setting the marker size as the volume of the boxes also adds some depth to the visualization, though modifying both the color and the size might be considered overkill.

Since adjusting the marker size may lead to overlapping points, we specify the *alpha value* of the color to make the markers slightly transparent. The keyword `alpha` accepts a value in the interval $[0, 1]$; 0 makes the markers completely transparent, while a 1 makes the markers completely opaque.

Finally, as with heat maps and contour plots, a color bar can be added with `plt.colorbar()` to indicate the values that the colors represent. This color bar can be given a label as well.

```
>>> length, width, height = np.random.randint(1, 20, (3,50))

>>> plt.subplot(221)                # Plot length against width.
>>> plt.scatter(length, width, s=100)
>>> plt.grid()
>>> plt.ylabel("Width (inches)")
>>> plt.tick_params(labelbottom="off")
>>> plt.axis([0, 20, 0, 20])
```

```
>>> plt.subplot(222)                # Set the marker color to the height.
>>> plt.scatter(length, width, c=height, s=100)
>>> cbar = plt.colorbar()
>>> cbar.set_label("Height (inches)")
>>> plt.grid()
>>> plt.tick_params(labelbottom="off", labelleft="off")
>>> plt.axis([0, 20, 0, 20])

>>> plt.subplot(223)                # Set the marker size to half the volume.
>>> plt.scatter(length, width, s=length*width*height/2., alpha=.7)
>>> plt.grid()
>>> plt.xlabel("Length (inches)")
>>> plt.ylabel("Width (inches)")
>>> plt.axis([0, 20, 0, 20])

>>> plt.subplot(224)                # Use color and marker size together.
>>> plt.scatter(length, width, c=height, s=length*width*height/2., alpha=.7)
>>> cbar = plt.colorbar()
>>> cbar.set_label("Height (inches)")
>>> plt.grid()
>>> plt.tick_params(labelleft="off")
>>> plt.xlabel("Length (inches)")
>>> plt.axis([0, 20, 0, 20])
```



In scatter plots, connecting the points into a line plot usual results in extreme clutter. A regression line, however, highlights a pattern in the data without overshadowing the actual data points.[2]

Problem 3. The file `MLB.npy` contains measurements from over 1,000 recent Major League Baseball players, compiled by UCLA.[a] Each row in the array represents a different player; the columns are the player's height (in inches), weight (in pounds), and age (in years), in that order.

Describe the data with at least one scatter plot. Your graph(s) should demonstrate whether height, weight, or age correlated with each other in the MLB. Consider plotting linear regression lines to indicate trends.

[a] See `http://wiki.stat.ucla.edu/socr/index.php/SOCR_Data_MLB_HeightsWeights`.

[2] See the Least Squares lab (QR 2), especially Problems 2âĂŞ–4, for a refresher on regression lines.

Histograms

A histogram partitions an interval into a number of bins and counts the number of values that fall into each bin. Histograms are ideal for visualizing how unordered data in a single array is distributed over an interval. If the data are draws from a probability distribution, a histogram approximates the distribution's probability density function (PDF).

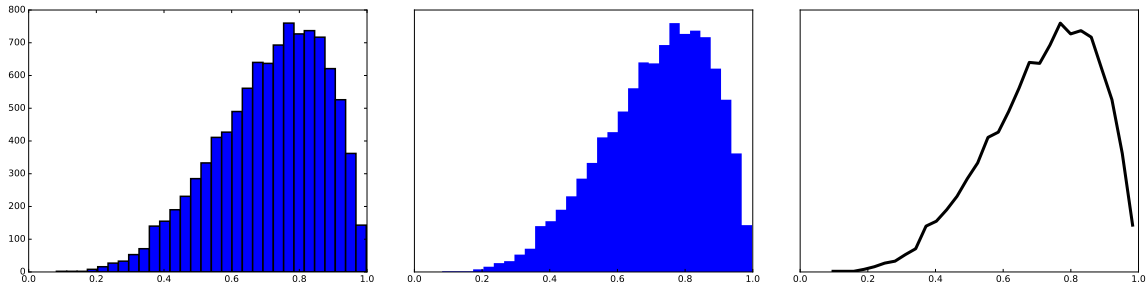
The following are important factors to consider when constructing a histogram.

- What does the x-axis represent? How should it be labeled?
- Is a linear or a logarithmic scale more appropriate for the frequency axis?
- How many bins should be used? Over what range should the bins be? This is perhaps the most important question, as a histogram with too few or too many bins usually fails to give a clear view of the distribution.

For most histograms, the most desirable insight is the general shape of the distribution. The lines separating the bins, axis tick marks, and even the labels on the y axis are all unnecessary (and potentially distracting) details. Removing the lines between bins is easy: set the line width to 0. To ensure the gaps where the lines used to be are filled, specify the `histtype` as `"stepfilled"` (using `histtype="step"` will draw the outline without filling it in). Finally, to get rid of extra markings on the axes, use `plt.tick_params()`.

A histogram can be converted into a line plot by using `np.histogram()`. This function returns the number of values in each bin and the locations of the edges of the bins (in fact, `plt.hist()` employs this function). Use the edges of the bins to calculate the center of the bins, then plot the bin centers against the frequency.

```
# Get 10,000 random samples from a Beta distribution.
>>> data = np.random.beta(a=5, b=2, size=N)

>>> plt.subplot(131)                # Draw a regular histogram.
>>> plt.hist(data, bins=30)

>>> plt.subplot(132)                # Draw a clean histogram.
>>> plt.hist(data, bins=30, lw=0, histtype="stepfilled")
>>> plt.tick_params(left="off", top="off", right="off", labelleft="off")

>>> plt.subplot(133)                # Convert the histogram to a line plot.
>>> freq, bin_edges = np.histogram(data, bins=30)
>>> bin_centers = (bin_edges[:-1] + bin_edges[1:])/2.
>>> plt.plot(bin_centers, freq, 'k-', lw=4)
>>> plt.tick_params(left="off", top="off", right="off", labelleft="off")
```

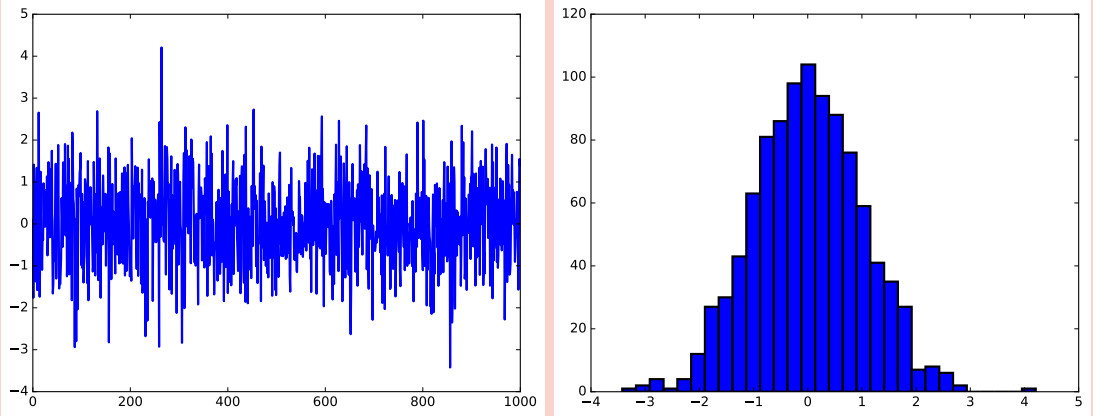


Finally, if the frequency domain is better visualized on a logarithmic scale, use `log=True` as an argument to `plt.hist()`. This is the histogram equivalent of using `plt.semilogy()` for line or scatter plots.

Achtung!

Line plots should **not** be used for data that has no natural progression. For example, consider a collection of random draws from a statistical distribution. In this case, a plain line plot is completely useless, because consecutive random draws are completely unrelated. A histogram, on the other hand, provides an approximation of the distribution's probability density function.

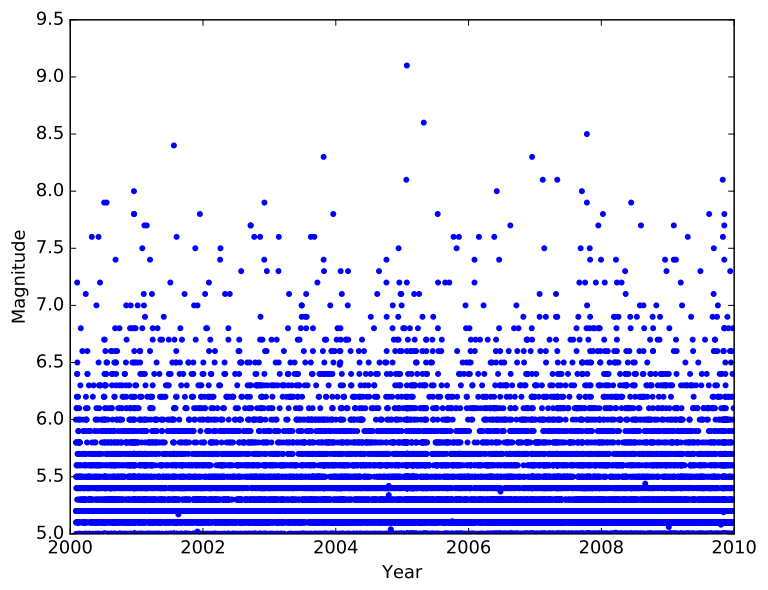
```
# Get 1,000 random samples from the standard normal distribution.
>>> data = np.random.normal(size=1000)

>>> plt.subplot(121)                # Regular line plot of the data.
>>> plt.plot(data)

>>> plt.subplot(122)                # Histogram of the data.
>>> plt.hist(data, bins=30)
```



Problem 4. The file `earthquakes.npy` contains data from over 17,000 earthquakes between 2000 and 2010 that were at least a 5 on the Richter scale.[a] Each row in the array represents a different earthquake; the columns are the earthquake's date (as a fraction of the year), magnitude (on the Richter scale), longitude, and latitude, in that order.

Because each earthquake is a distinct event, a good way to start visualizing this data might be a scatter plot of the years versus the magnitudes of each earthquake.

```
>>> year, magnitude, longitude, latitude = np.load("earthquakes.npy").T
>>> plt.plot(year, magnitude, '.')
>>> plt.xlabel("Year")
>>> plt.ylabel("Magnitude")
```



Unfortunately, this plot communicates very little information because the data is so cluttered. Describe the data with two or three better visualizations, including line plots, scatter plots, and histograms as appropriate. Your plots should clearly answer the following questions:

1. How many earthquakes happened every year?
2. How often do stronger earthquakes happen compared to weaker ones?
3. Where do earthquakes happen? Where do the strongest earthquakes happen?
   (Hint: Use `plt.axis("equal")` to fix the aspect ratio, which may improve comparisons between longitude and latitude.)

[a] See `http://earthquake.usgs.gov/earthquakes/search/`.

Heat Maps and Contour Plots

Let $f : \mathbb{R}^2 \to \mathbb{R}$ be a scalar-valued function on a 2-dimensional domain. A heat map of f assigns a color to each (x, y) point in the domain based on the value of $f(x, y)$, while a contour plot is a drawing of the *level curves* of f. The level curve corresponding to the constant c is the set $\{(x, y) \mid c = f(x, y)\}$. A filled contour plot, which colors in the sections between the level curves, is a discretized version of a heat map.

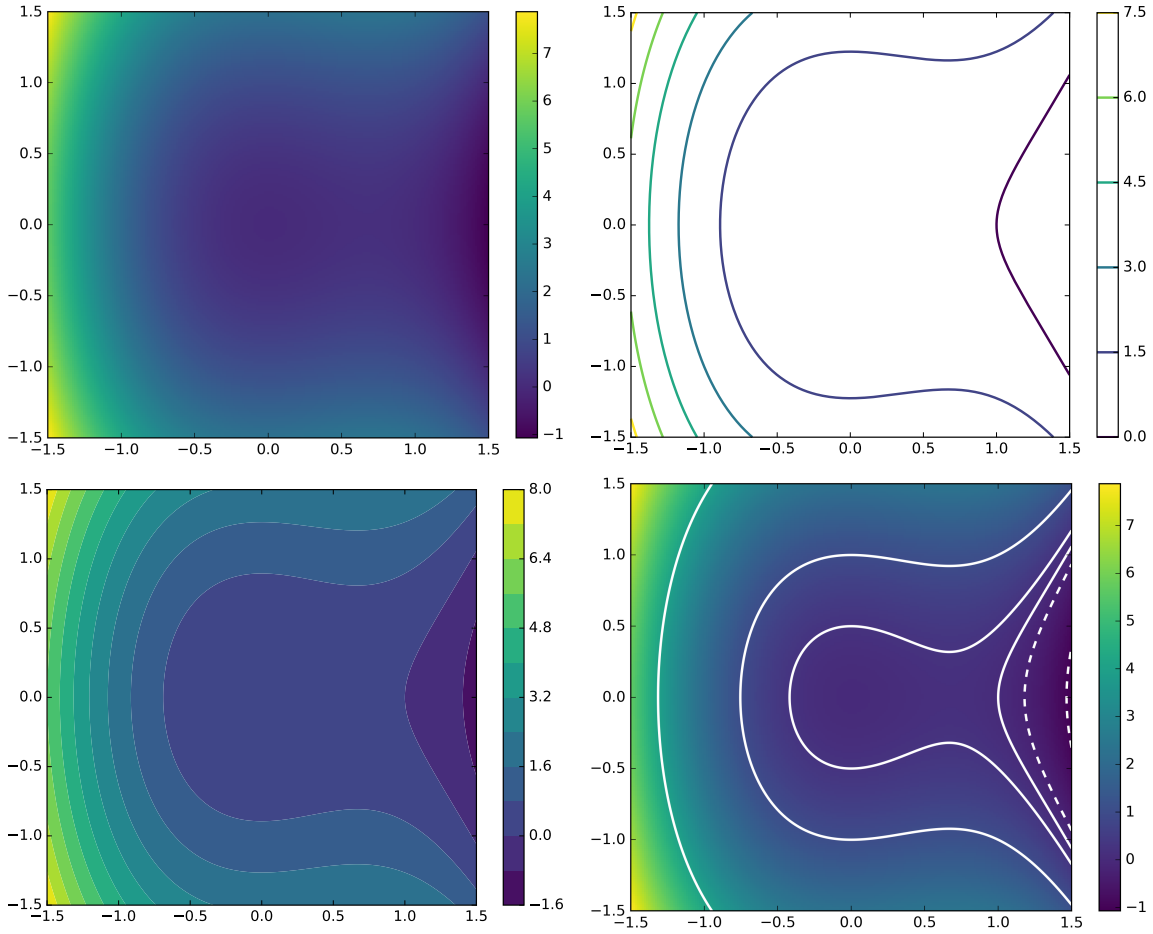
Consider the following questions when plotting a heat map or contour plot:

- Is the domain sufficiently refined?
- Which color scheme is most clear and effective?
- How many / which contour lines should be drawn, if any?
- Is a linear or a logarithmic scale more appropriate for the color?

It is often sufficient to choose a fixed number of level curves. In this case, the values of c corresponding to the level curves are automatically chosen to be evenly spaced over the range of values of f on the domain. However, it is sometimes better to strategically specify the curves by providing a list of c constants.

Consider the function $f(x, y) = y^2 - x^3 + x^2$ on the domain $[-\frac{3}{2}, \frac{3}{2}] \times [-\frac{3}{2}, \frac{3}{2}]$. A heat map of f reveals that it has a large basin around the origin. Then since $f(0, 0) = 0$, choosing several level curves close to 0 more closely describes the topography of the basin. The fourth subplot in the following example uses the curves with $c = -1$, $-\frac{1}{4}$, 0, $\frac{1}{4}$, 1, and 4.

```
# Construct a 2-D domain with np.meshgrid() and calculate f on the domain.
>>> x = np.linspace(-1.5, 1.5, 200)
>>> X, Y = np.meshgrid(x, x.copy())
>>> Z = Y**2 - X**3 + X**2

>>> plt.subplot(221)                # Plot a heat map of f.
>>> plt.pcolormesh(X, Y, Z, cmap="viridis")
>>> plt.colorbar()

>>> plt.subplot(222)                # Plot a contour map with 6 level curves.
>>> plt.contour(X, Y, Z, 6, cmap="viridis")
>>> plt.colorbar()

>>> plt.subplot(223)                # Plot a filled contour map with 12 levels.
>>> plt.contourf(X, Y, Z, 12, cmap="viridis")
>>> plt.colorbar()

>>> plt.subplot(224)                # Plot specific level curves and a heat map↩
    .
>>> plt.contour(X, Y, Z, [-1, -.25, 0, .25, 1, 4], colors="white")
>>> plt.pcolormesh(X, Y, Z, cmap="viridis")
>>> plt.colorbar()
```

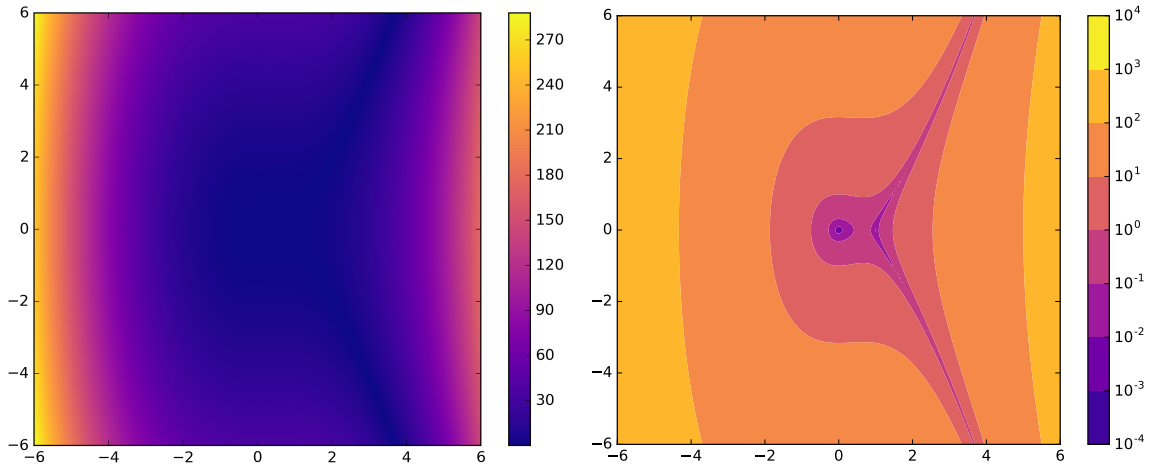


There are two main kinds of color maps: sequential and diverging. Sequential color maps, like `"hot"` and `"cool"`, transition very gradually between two colors; diverging color maps, like `"seismic"` and `"coolwarm"`, transition very rapidly from one color to another at the mean value. When in doubt, use `"viridis"` or `"plasma"`, two specialized sequential color schemes. For the complete list of Matplotlib color maps, see `http://matplotlib.org/examples/color/colormaps_reference.html`.

The color map can be changed to a logarithmic scale by using the keyword argument `norm=matplotlib.colors.LogNorm()` (this works for `plt.scatter()` as well). As an example, consider the same f defined above on the larger domain $[-6, 6] \times [-6, 6]$. Log scaling can only be done on arrays of all positive values, so we visualize $|f|$.

```
>>> from matplotlib.colors import LogNorm

>>> x = np.linspace(-6, 6, 200)
>>> X, Y = np.meshgrid(x, x.copy())
>>> Z = np.abs(Y**2 - X**3 + X**2)

>>> plt.subplot(121)              # Plot a regular heat map of |f|.
>>> plt.pcolormesh(X, Y, Z, cmap="plasma")
>>> plt.colorbar()
```

```
>>> plt.subplot(122)              # Plot a filled contour plot with log scaling↩
    .
>>> plt.contourf(X, Y, Z, 6, cmap="plasma", norm=LogNorm())
>>> plt.colorbar()
```



Problem 5. The *Rosenbrock function* is defined as follows.

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

The minimum value of f is 0, which occurs at the point $(1, 1)$ at the bottom of a steep, banana-shaped valley of the function.

Use heat maps and contour plots to visualize the Rosenbrock function in such a way that the minimum value, and the valley that it lies in, is apparent. Consider plotting the minimizer $(1, 1)$ on top of the plot as well.

Bar Charts

A bar chart plots categorical data in a sequence of bars. They are best for small, discrete, one-dimensional data sets. Use `plt.bar()` (vertical) or `plt.barh()` (horizontal) to create a bar chart in Matplotlib. These functions receive the locations of each bar, then the height of each bar (as lists or arrays).

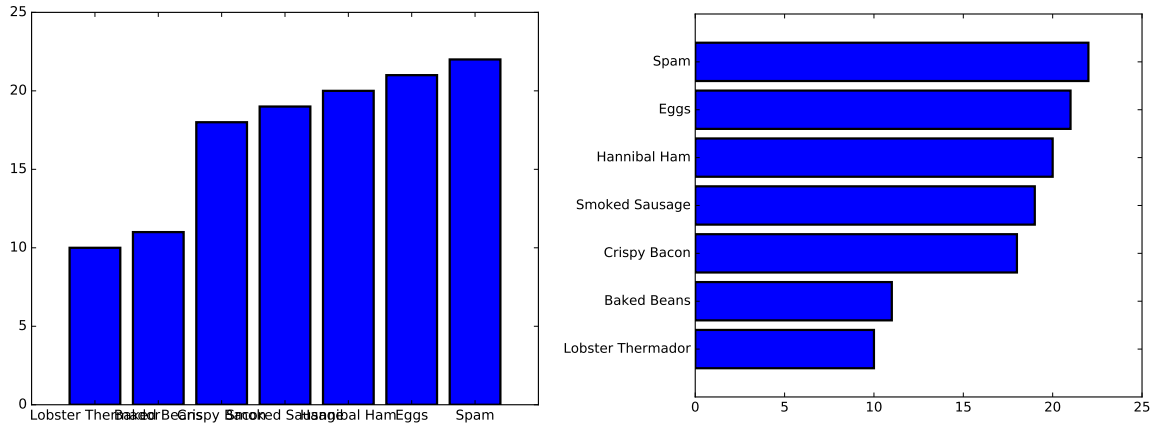
Consider the following questions when constructing a bar chart.

- What are the different categories in the data?
- How should the data be sorted?
- Should the value axis have a linear or a logarithmic scale?

Horizontal bar charts are usually preferable to vertical bar charts because horizontal labels are easier to read than vertical labels. Labels can be rotated, but it is better when the reader doesn't have to turn his or her head.

Data in a bar chart should also be sorted in a logical way. Sort the categories by bar size, alphabetize the labels, or use some other intuitive ordering.

```
>>> labels = ["Lobster Thermador", "Baked Beans", "Crispy Bacon",
...              "Smoked Sausage", "Hannibal Ham", "Eggs", "Spam"]
>>> values = [10, 11, 18, 19, 20, 21, 22]
>>> positions = np.arange(len(labels))

>>> plt.subplot(121)
>>> plt.bar(positions, values, align="center")
>>> plt.xticks(positions, labels)

>>> plt.subplot(122)
>>> plt.barh(positions, values, align="center")
>>> plt.yticks(positions, labels)
```



Practices to Avoid

Bad Types of Visualization

Some kinds of visualizations, especially for categorical data, are popular even though they interfere with the reader's ability to interpret the displayed information. For example, it is more difficult to see the differences in a pie chart than on a bar chart since differences in area are more difficult to detect than differences in length. By the same logic, bar chart with several bars per category are almost always better with the bars side by side rather than stacked.

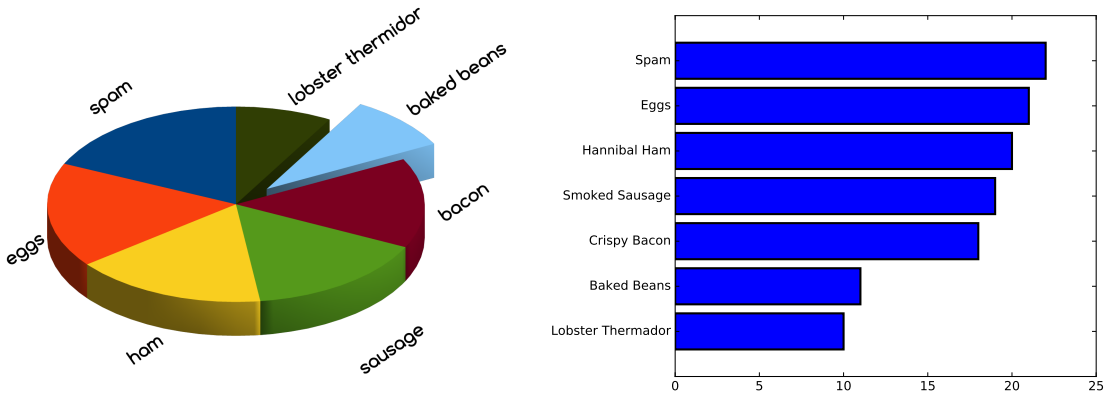
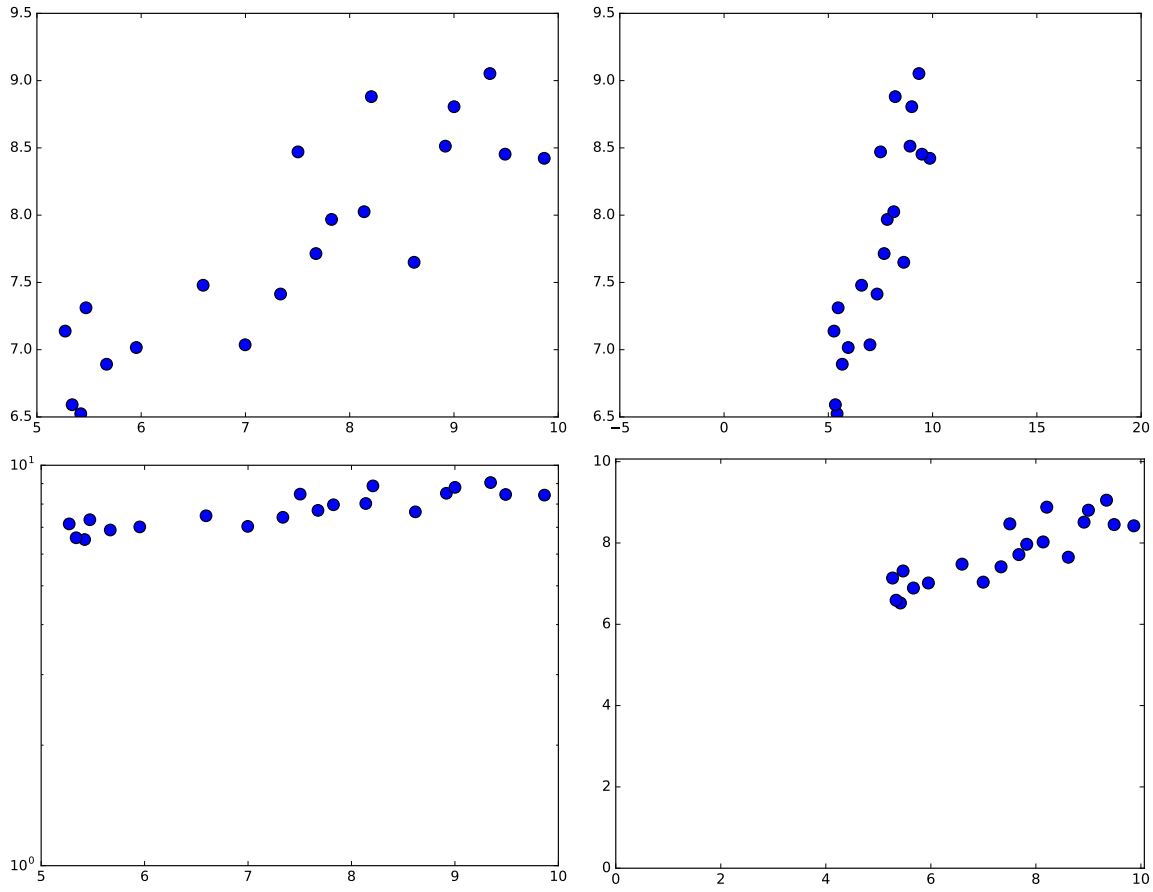


Figure 7.7: Transforming the horrendous 3-D pie chart on the left into the horizontal bar chart on the right both simplifies and clarifies the information.

Misrepresenting Data

To conclude, we return to the issue of integrity. It is very easy to misrepresent the true nature of a set of data with a visualization. Perhaps the easiest way to do so is with poorly chosen window limits or axis scales. Consider the following example of a scatter plot, presented four different ways. The fourth subplot is the only one that correctly shows the position of the points in relation to the origin and that has a reasonable scales on both axes.

```
>>> x = np.linspace(5, 10, N) + np.random.normal(size=N)/3.
>>> y = .5*x + 4 + np.random.normal(size=N)/2.

>>> plt.subplot(221)                # Plot a default scatter plot of the data.
>>> plt.plot(x, y, 'o', ms=10)

>>> plt.subplot(222)                # Change the window limits so that the
>>> plt.plot(x, y, 'o', ms=10)          # data appears steep in the y direction↩
    .
>>> plt.xlim(-5,20)

>>> plt.subplot(223)                # Change the y-axis scale so that the data
>>> plt.semilogy(x, y, 'o', ms=10)      # appears flat in the y direction.

>>> plt.subplot(224)                # Plot the data with equal axis scales and
>>> plt.plot(x, y, 'o', ms=10)          # presenting the relation to the origin↩
    .
>>> plt.xlim(xmin=0)
>>> plt.ylim(ymin=0)
```



Do everything in your power to avoid visualizing data unethically.

Problem 6. The file `countries.npy` contains information from 20 different countries. Each row in the array represents a different country; the columns are the 2015 population (in millions of people), the 2015 GDP (in billions of US dollars), the average male height (in centimeters), and the average female height (in centimeters), in that order.[a]

The countries corresponding are listed below in order.

```
countries = ["Austria", "Bolivia", "Brazil", "China",
             "Finland", "Germany", "Hungary", "India",
             "Japan", "North Korea", "Montenegro", "Norway",
             "Peru", "South Korea", "Sri Lanka", "Switzerland",
             "Turkey", "United Kingdom", "United States", "Vietnam"]
```

Visualize this data set with at least four plots, using at least one scatter plot, one histogram, and one bar chart. Write a few sentences summarizing the major insights that your visualizations reveal.
(Hint: consider using `np.argsort()` and fancy indexing to sort the data for the bar chart.)

[a] See `https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)`, `http://www.averageheight.co/`, and `https://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_population`.

8 Testing

Lab Objective: *Finding and fixing programming errors can be difficult and time consuming, especially in large or complex programs.* Unit testing *is a formal strategy for finding and eliminating errors quickly as a program is constructed and for ensuring that the program still works whenever it is modified. A single unit test checks a small piece code (usually a function or class method) for correctness, independent of the rest of the program. A well-written collection of unit tests can ensure that every unit of code functions as intended, thereby certifying that the program is correct. In this lab, we learn to write unit tests in Python and practice test-driven development. Applying these principles will greatly speed up the coding process and improve your code quality.*

Unit Tests

A *unit test* verifies a piece of code by running a series of test cases and comparing actual outputs with expected outputs. Each test case is usually checked with an `assert` statement, a shortcut for raising an `AssertionError` with an optional error message if a boolean statement is false.

```
# Store the result of a boolean expression in a variable.
>>> result = str(5)=='5'

# Check the result, raising an error if it is false.
>>> if result is False:
...     raise AssertionError("incorrect result")

# Do the same check in one line with an assert statement.
>>> assert result, "incorrect result"

# Asserting a false statement raises an AssertionError.
>>> assert 5=='5', "5 is not a string"
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
AssertionError: 5 is not a string
```

Now suppose we wanted to test a simple `add()` function, located in the file `specs.py`.

```
# specs.py

def add(a, b):
    """Add two numbers."""
    return a + b
```

In a corresponding file called `test_specs.py`, which should contain all of the unit tests for the code in `specs.py`, we write a unit test called `test_add()` to verify the `add()` function.

```
# test_specs.py
import specs

def test_add():
    assert specs.add(1, 3) == 4, "failed on positive integers"
    assert specs.add(-5, -7) == -12, "failed on negative integers"
    assert specs.add(-6, 14) == 8
```

In this case, running `test_add()` raises no errors since all three test cases pass. Unit test functions don't need to return anything, but they should raise an exception if a test case fails.

> **Note**
>
> This style of external testing—checking that certain inputs result in certain outputs—is called *black box testing*. The actual structure of the code is not considered, but what it produces is thoroughly examined. In fact, the author of a black box test doesn't even need to be the person who eventually writes the program! Having one person write tests and another write the code helps detect problems that one developer or the other may not have caught individually.

PyTest

Python's `pytest` module[1] provides tools for building tests, running tests, and providing detailed information about the results. To begin, run `py.test` in the current directory. Without any test files, the output should be similar to the following.

```
$ py.test
============================= test session starts ==============================
platform darwin -- Python 2.7.12, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /Users/Student, inifile:
collected 0 items

========================== no tests ran in 0.02 seconds ==========================
```

Given some test files, say `test_calendar.py` and `test_google.py`, the output of `py.test` identifies failed tests and provides details on why they failed.

[1] Pytest is not part of the standard libray, but it is included in Anaconda's Python distribution. Install pytest with `pip install -U pytest` if needed. The standard library's `unittest` module also provides a testing framework, but is less popular and straightforward than pytest.

```
$ py.test
============================= test session starts ==============================
platform darwin -- Python 2.7.12, pytest-3.0.7, py-1.4.31, pluggy-0.4.0
rootdir: /Users/Student/example_tests, inifile:
collected 12 items

test_calendar.py ........
test_google.py .F..

=================================== FAILURES ===================================
_______________________________ test_subtract ________________________________

    def test_subtract():
>       assert google.subtract(42, 17)==25, "subtract() failed for a > b > 0"
E       AssertionError: subtract() failed for a > b > 0
E       assert 35 == 25
E        +  where 35 = <function subtract at 0x102d4eb90>(42, 17)
E        +    where <function subtract at 0x102d4eb90> = google.subtract

test_google.py:11: AssertionError
==================== 1 failed, 11 passed in 0.02 seconds =====================
```

Each dot represents a passed test and each `F` represents a failed test. They show up in order, so in the example above, only the second of four tests in `test_google.py` failed.

Achtung!

PyTest will not find or run tests if they are not contained in files named `test_*.py` or `*_test.py`, where `*` represents any number of characters. In addition, the unit tests themselves must be named `test_*()` or `*_test()`. If you need to change this behavior, consult the documentation at `http://pytest.org/latest/example/pythoncollection.html`.

Problem 1. The following function contains a subtle but important error.

```
def smallest_factor(n):
    """Return the smallest prime factor of the positive integer n."""
    if n == 1: return 1
    for i in range(2, int(n**.5)):
        if n % i == 0: return i
    return n
```

Write a unit test for this function, including test cases that you suspect might uncover the error (what are the edge cases for this function?). Use `pytest` to run your unit test and discover a test case that fails, then use this information to correct the function.

Coverage

Successful unit tests include enough test cases to test the entire program. *Coverage* refers the number of lines of code that are executed by at least one test case. One tool for measuring coverage is called `pytest-cov`, an extension of `pytest`. This tool must be installed separately, as it does not come bundled with Anaconda.

```
$ conda install pytest-cov
```

Add the flag `--cov` to the `py.test` command to print out code coverage information. Running `py.test --cov` in the same directory as `specs.py` and `test_specs.py` yields the following output.

```
$ py.test --cov
============================= test session starts ==============================
platform darwin -- Python 2.7.12, pytest-3.0.7, py-1.4.31, pluggy-0.4.0
rootdir: /Users/Student/Testing, inifile:
plugins: cov-2.3.1
collected 7 items

test_specs.py .......

---------- coverage: platform darwin, python 2.7.12-final-0 ----------
Name            Stmts   Miss  Cover
-----------------------------------
specs.py           73     34    53%
test_specs.py      46      0   100%
-----------------------------------
TOTAL             119     34    71%


========================== 7 passed in 0.03 seconds ===========================
```

Here, `Stmts` refers to the number of lines of code covered by a unit test, while `Miss` is the number of lines that are not currently covered. Notice that the file `test_specs.py` has 100% coverage while `specs.py` does not. Test files generally have 100% coverage, since `pytest` is designed to run these files in their entirety. However, `specs.py` does not have full coverage and requires additional unit tests. To find out which lines are not yet covered, `pytest-cov` has a useful feature called `cov-report` that creates an HTML file for visualizing the current line coverage.

```
$ py.test --cov-report html:cov_html --cov
============================= test session starts ==============================
# ...
---------- coverage: platform darwin, python 2.7.12-final-0 ----------
Coverage HTML written to dir cov_html
```

Instead of printing coverage statistics, this command creates various files with coverage details in a new directory called `cov_html/`. The file `cov_html/specs_py.html`, which can be viewed in an internet browser, highlights in red the lines of `specs.py` that are not yet covered by any unit tests.

Note

Statement coverage is categorized as *white box testing* because it requires an understanding of the code's structure. While most black box tests can be written before a program is actually implemented, white box tests should be added to the collection of unit tests after the program is completed. By designing unit tests so that they cover every statement in a program, you may discover that some lines of code are unreachable, find that a conditional statement isn't functioning as intended, or uncover problems that accompany edge cases.

Problem 2. With `pytest-cov` installed, check your coverage of `smallest_factor()` from Problem 1. Write additional test cases if necessary to get complete coverage. Then, write a comprehensive unit test for the following (correctly written) function.

```
def month_length(month, leap_year=False):
    """Return the number of days in the given month."""
    if month in {"September", "April", "June", "November"}:
        return 30
    elif month in {"January", "March", "May", "July",
                   "August", "October", "December"}:
        return 31
    if month == "February":
        if not leap_year:
            return 28
        else:
            return 29
    else:
        return None
```

Testing Exceptions

Many programs are designed to raise exceptions in response to bad input or an unexpected error. A good unit test makes sure that the program raises the exceptions that it is expected to raise, but also that it doesn't raise any unexpected exceptions. The `raises()` method in `pytest` is a clean, formal way of asserting that a program raises a desired exception. Consider the following example:

```
# specs.py

def divide(a, b):
    """Divide two numbers, raising an error if the second number is zero."""
    if b == 0:
        raise ZeroDivisionError("second input cannot be zero")
    return a / float(b)
```

The corresponding unit tests checks that the function raises the exception correctly.

```
# test_specs.py
import pytest

def test_divide():
    assert specs.divide(4,2) == 2, "integer division"
    assert specs.divide(5,4) == 1.25, "float division"
    pytest.raises(ZeroDivisionError, specs.divide, a=4, b=0)
```

If calling `divide(a=4, b=0)` results in a `ZeroDivisionError`, `pytest.raises()` catches the exception and the test case passes. On the other hand, if `divide(a=4, b=0)` does not raise a `ZeroDivisionError`, or if it raises a different kind of exception, the test fails.

To ensure that the `ZeroDivisionError` is coming from the written `raise` statement, combine `pytest.raises()` and the `with` statement to check the exception's error message.

```
def test_divide():
    assert specs.divide(4,2) == 2, "integer division"
    assert specs.divide(5,4) == 1.25, "float division"
    with pytest.raises(ZeroDivisionError) as excinfo:
        specs.divide(4, 0)
    assert excinfo.value.args[0] == "second input cannot be zero"
```

Here `excinfo` is an object containing information about the exception; the actual exception object is stored in `excinfo.value`, and hence `excinfo.value.args[0]` is the error message.

Problem 3. Write a comprehensive unit test for the following function. Make sure that each exception is raised properly by explicitly checking the exception message. Use `pytest-cov` and its `cov-report` tool to confirm that you have full coverage for this function.

```
def operate(a, b, oper):
    """Apply an arithmetic operation to a and b."""
    if type(oper) is not str:
        raise TypeError("oper must be a string")
    elif oper == '+':
        return a + b
    elif oper == '-':
        return a - b
    elif oper == '*':
        return a * b
    elif oper == '/':
        if b == 0:
            raise ZeroDivisionError("division by zero is undefined")
        return a / float(b)
    raise ValueError("oper must be one of '+', '/', '-', or '*'")
```

Fixtures

Consider the following class for representing rational numbers as reduced fractions.

```
class Fraction(object):
    """Reduced fraction class with integer numerator and denominator."""
    def __init__(self, numerator, denominator):
        if denominator == 0:
            raise ZeroDivisionError("denominator cannot be zero")
        elif type(numerator) is not int or type(denominator) is not int:
            raise TypeError("numerator and denominator must be integers")

        def gcd(a,b):
            while b != 0:
                a, b = b, a % b
            return a
        common_factor = gcd(numerator, denominator)
        self.numer = numerator // common_factor
        self.denom = denominator // common_factor

    def __str__(self):
        if self.denom != 1:
            return "{} / {}".format(self.numer, self.denom)
        else:
            return str(self.numer)

    def __float__(self):
        return self.numer / float(self.denom)

    def __eq__(self, other):
        if type(other) is Fraction:
            return self.numer==other.numer and self.denom==other.denom
        else:
            return float(self) == other

    def __add__(self, other):
        return Fraction(self.numer*other.numer + self.denom*other.denom,
                                                    self.denom*other.denom)
    def __sub__(self, other):
        return Fraction(self.numer*other.numer - self.denom*other.denom,
                                                    self.denom*other.denom)
    def __mul__(self, other):
        return Fraction(self.numer*other.numer, self.denom*other.denom)

    def __div__(self, other):
        if self.denom*other.numer == 0:
            raise ZeroDivisionError("cannot divide by zero")
        return Fraction(self.numer*other.denom, self.denom*other.numer)
```

```
>>> from specs import Fraction
>>> print(Fraction(8, 12))                          # 8/12 reduces to 2/3.
2 / 3
>>> Fraction(1, 5) == Fraction(3, 15)               # 3/15 reduces to 1/5.
True
>>> print(Fraction(1, 3) * Fraction(1, 4))
1 / 12
```

To test this class, it would be nice to have some ready-made `Fraction` objects to use in each unit test. A *fixture*, a function marked with the `@pytest.fixture` decorator, sets up variables that can be used as mock data for multiple unit tests. The individual unit tests take the fixture function in as input and unpack the constructed tests. Below, we define a fixture that instantiates three `Fraction` objects. The unit tests for the `Fraction` class use these objects as test cases.

```
@pytest.fixture
def set_up_fractions():
    frac_1_3 = specs.Fraction(1, 3)
    frac_1_2 = specs.Fraction(1, 2)
    frac_n2_3 = specs.Fraction(-2, 3)
    return frac_1_3, frac_1_2, frac_n2_3

def test_fraction_init(set_up_fractions):
    frac_1_3, frac_1_2, frac_n2_3 = set_up_fractions
    assert frac_1_3.numer == 1
    assert frac_1_2.denom == 2
    assert frac_n2_3.numer == -2
    frac = specs.Fraction(30, 42)                    # 30/42 reduces to 5/7.
    assert frac.numer == 5
    assert frac.denom == 7

def test_fraction_str(set_up_fractions):
    frac_1_3, frac_1_2, frac_n2_3 = set_up_fractions
    assert str(frac_1_3) == "1 / 3"
    assert str(frac_1_2) == "1 / 2"
    assert str(frac_n2_3) == "-2 / 3"

def test_fraction_float(set_up_fractions):
    frac_1_3, frac_1_2, frac_n2_3 = set_up_fractions
    assert float(frac_1_3) == 1 / 3.
    assert float(frac_1_2) == .5
    assert float(frac_n2_3) == -2 / 3.

def test_fraction_eq(set_up_fractions):
    frac_1_3, frac_1_2, frac_n2_3 = set_up_fractions
    assert frac_1_2 == specs.Fraction(1, 2)
    assert frac_1_3 == specs.Fraction(2, 6)
    assert frac_n2_3 == specs.Fraction(8, -12)
```

Problem 4. Add test cases to the unit tests provided above to get full coverage for the `__init__()`, `__str__()`, `__float__()`, and `__eq__()` methods. You may modify the fixture function if it helps. Also add unit tests for the magic methods `__add__()`, `__sub__()`, `__mul__()`, and `__div__()`. Verify that you have full coverage with `pytest-cov`.

Additionally, **two** of the `Fraction` class's methods are implemented incorrectly. Use your tests to find the issues, then correct the methods so that your tests pass.

See `http://doc.pytest.org/en/latest/index.html` for complete documentation on `pytest`.

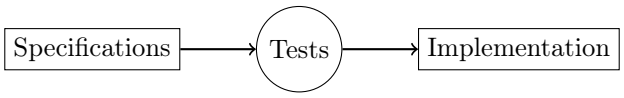
Test-driven Development

Test-driven development (TDD) is the programming style of writing tests **before** implementing the actual code. It may sound tedious at first, but TDD incentivizes simple design and implementation, speeds up the actual coding, and gives quantifiable checkpoints for the development process. TDD can be summarized in the following steps:

1. Define with great detail the program specifications. Write function declarations, class definitions, and (especially) docstrings, determining exactly what each function or class method should accept and return.
2. Write a unit test for each unit of the program (usually black box tests).
3. Implement the program code, making changes until all tests pass.

For adding new features or cleaning existing code, the process is similar.

1. Redefine program specifications to account for planned modifications.
2. Add or modify tests to match the new specifications.
3. Change the code until all tests pass.

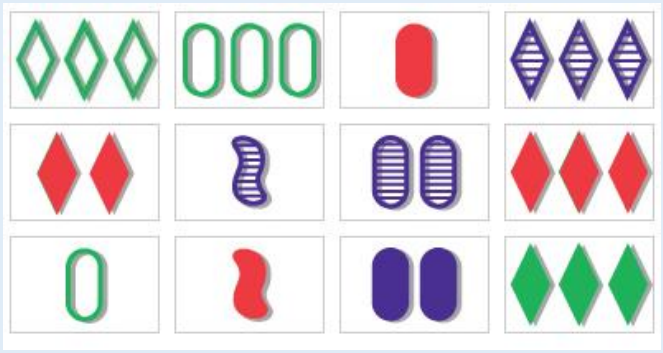


If the test cases are sufficiently thorough, when the tests all pass the program can be considered complete. Remember, however, that it is not sufficient to just have tests, but to have tests that accurately test the code. To check that the test cases are sufficient, examine the test coverage and add additional tests if necessary.

See `https://en.wikipedia.org/wiki/Test-driven_development` for more discussion on TDD and `https://en.wikipedia.org/wiki/Behavior-driven_development` for an overview of Behavior-driven development (BDD), a close relative of TDD.

Problem 5. *Set* is a card game about finding patterns. Each card contains a design with 4 different properties: color (red, green or purple), shape (diamond, oval or squiggly), quantity (one, two, or three) and pattern (solid, striped or outlined). A *set* is a group of three cards which are either all the same or all different for each property. You can try playing Set online at `http://smart-games.org/en/set/start`.

Here is a group of twelve Set cards.



This collection of cards contains six unique sets:

(a) Same in quantity and shape; different in pattern and color

(b) Same in color and pattern; different in shape and quantity

(c) Same in pattern; different in shape, quantity and color

(d) Same in shape; different in quantity, pattern and color

(e) Different in all aspects

(f) Different in all aspects

Each Set card can be uniquely represented by a 4-bit integer in base 3,[a] where each digit represents a different property and each property has three possible values. A full hand in Set is a group of twelve unique cards, so a hand can be represented by a list of twelve 4-digit integers in base 3. For example, the hand shown above could be represented by the following list.

```
hand1 = ["1022", "1122", "0100", "2021",
         "0010", "2201", "2111", "0020",
         "1102", "0210", "2110", "1020"]
```

The following function definitions provide a framework for implementing Set by calculating the number of sets in a given hand.

```
def count_sets(cards):
    """Return the number of sets in the provided Set hand.

    Parameters:
        cards (list(str)) a list of twelve cards as 4-bit integers in
        base 3 as strings, such as ["1022", "1122", ..., "1020"].
    Returns:
        (int) The number of sets in the hand.
    Raises:
        ValueError: if the list does not contain a valid Set hand, meaning
            - there are not exactly 12 cards,
            - the cards are not all unique,
            - one or more cards does not have exactly 4 digits, or
            - one or more cards has a character other than 0, 1, or 2.
    """
    pass

def is_set(a, b, c):
    """Determine if the cards a, b, and c constitute a set.

    Parameters:
        a, b, c (str): string representations of 4-bit integers in base 3.
            For example, "1022", "1122", and "1020" (which is not a set).
    Returns:
        True if a, b, and c form a set, meaning the ith digit of a, b,
            and c are either the same or all different for i=1,2,3,4.
        False if a, b, and c do not form a set.
    """
    pass
```

Write unit tests for these functions, but **do not** implement them yet. Focus on *what* the functions should do rather than *how* they will be implemented.
(Hint: if three cards form a set, then the first digits of the cards are either all the same or all different. Then the sums of these digits can only be 0, 3, or 6. Thus a group of cards forms a set only if for each set of digits—first digits, second digits, etc.—the sum is a multiple of 3.)

[a] A 4-bit integer in base 3 contains four digits that are either 0, 1 or 2. For example, 0000 and 1201 are 4-bit integers in base 3, whereas 000 is not because it has only three digits, and 0123 is not because it contains the number 3.

Problem 6. After you have written unit tests for the functions in Problem 5, implement the actual functions. If needed, add additional test cases to get full coverage.
(Hint: The `combinations()` function from the standard library module `itertools` may be useful in implementing `count_sets()`.)

Additional Material

The Python Debugger

Python has a built in debugger called `pdb` to aid in finding mistakes in code during execution. The debugger can be run either in a terminal or in a Jupyter Notebook.

A *break point*, set with `pdb.set_trace()`, is a spot where the program pauses execution. Once the program is paused, use the following commands to tell the program what to do next.

Command	Description
`n`	**n**ext: executes the next line
`p <var>`	**p**rint: display the value of the specified variable.
`c`	**c**ontinue: stop debugging and run the program normally to the end.
`q`	**q**uit: terminate the program.
`l`	**l**ist: show several lines of code around the current line.
`r`	**r**eturn: return to the end of a subroutine.
`<Enter>`	Execute the most recent command again.

For example, suppose we have a long loop where the value of a variable changes unpredictably.

```
# pdb_example.py
import pdb
from random import randint

i = 0
pdb.set_trace()
while i < 1000000000:
    i += randint(1, 10)
print("DONE")
```

Run the file in the terminal to begin a debugging session.

```
$ python pdb_example.py
> /Users/Student/pdb_example.py(7)<module>()<<
-> while i < 1000000000:
(Pdb) l                                          # Show where we are.
  2     import pdb
  3     from random import randint
  4
  5     i = 0
  6     pdb.set_trace()
  7  -> while i < 1000000000:
  8         i += randint(1, 10)
  9     print("DONE")
[EOF]>>
```

We can check the value of the variable `i` at any step with `p i`, and we can even change the value of `i` mid-program.

```
(Pdb) n                                                 # Execute a few lines.
> /Users/Student/pdb_example.py(8)<module>()
-> i += randint(1, 10)
(Pdb) n
> /Users/Student/pdb_example.py(7)<module>()
-> while i < 1000000000:
(Pdb) n
> /Users/Student/pdb_example.py(8)<module>()
-> i += randint(1, 10)
(Pdb) p i                                               # Check the value of i.
8
(Pdb) n                                                 # Execute another line.
> /Users/Student/pdb_example.py(7)<module>()
-> while i < 1000000000:
(Pdb) p i                                               # Check i again.
14
(Pdb) i = 999999999                                     # Change the value of i.
(Pdb) c                                                 # Continue the program.
DONE
```

See `https://docs.python.org/2/library/pdb.html` for documentation and examples for the Python debugger.

9 Profiling

Lab Objective: *The best code goes through multiple drafts. In a first draft, you should focus on writing code that does what it is supposed to and is easy to read. Once you have working code, you may need to speed it up to meet the demands of the application. In this lab we learn to identify the parts of code that take the most time to run and how speed up slow code.*

In this lab we will optimize the function `qr1()` that computes the QR decomposition of a matrix via the modified Gram-Schmidt algorithm.

```
import numpy as np
from scipy import linalg as la

def qr1(A):
    ncols = A.shape[1]
    Q = A.copy()
    R = np.zeros((ncols, ncols))
    for i in range(ncols):
        R[i, i] = la.norm(Q[:, i])
        Q[:, i] = Q[:, i]/la.norm(Q[:, i])
        for j in range(i+1, ncols):
            R[i, j] = Q[:, j].dot(Q[:, i])
            Q[:,j] = Q[:,j]-Q[:, j].dot(Q[:, i])*Q[:,i]
    return Q, R
```

Profiling Slow Code

Python provides a *profiler* that can identify where code spends most of its runtime. The output of the profiler will tell you where to begin your optimization efforts. In IPython[1], you can profile a function with `%prun`. Here we profile `qr1()` on a random 300×300 array.

```
In [1]: A = np.random.random((300, 300))
```

[1]If you are not using IPython, you will need to use the `cProfile` module documented here: `https://docs.python.org/2/library/profile.html`.

```
In [2]: %prun qr1(A)
```

On the author's computer, we get the following output.

```
         97206 function calls in 1.343 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.998    0.998    1.342    1.342 profiling.py:4(qr1)
    89700    0.319    0.000    0.319    0.000 {method 'dot' of 'numpy.ndarray' objects}
      600    0.006    0.000    0.012    0.000 function_base.py:526(asarray_chkfinite)
      600    0.006    0.000    0.009    0.000 linalg.py:1840(norm)
     1200    0.005    0.000    0.005    0.000 {method 'any' of 'numpy.ndarray' objects}
      600    0.002    0.000    0.002    0.000 {method 'reduce' of 'numpy.ufunc' objects}
     1200    0.001    0.000    0.001    0.000 {numpy.core.multiarray.array}
     1200    0.001    0.000    0.002    0.000 numeric.py:167(asarray)
        1    0.001    0.001    0.001    0.001 {method 'copy' of 'numpy.ndarray' objects}
      600    0.001    0.000    0.022    0.000 misc.py:7(norm)
      301    0.001    0.000    0.001    0.000 {range}
        1    0.001    0.001    0.001    0.001 {numpy.core.multiarray.zeros}
      600    0.001    0.000    0.001    0.000 {method 'ravel' of 'numpy.ndarray' objects}
      600    0.000    0.000    0.000    0.000 {method 'conj' of 'numpy.ndarray' objects}
        1    0.000    0.000    1.343    1.343 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

The first line of the output tells us that executing `qr1(A)` results in almost 100,000 function calls. Then we see a table listing these functions along with data telling us how much time each takes. Here, `ncalls` is the number of calls to the function, `tottime` is the total time spent in the function, and `cumtime` is the amount of time spent in the function including calls to other functions.

For example, the first line of the table is the function `qr1(A)` itself. This function was called once, it took 1.342s to run, and 0.344s of that was spent in calls to other functions. Of that 0.344s, there were 0.319s spent on 89,700 calls to `np.dot()`.

With this output, we see that most time is spent in multiplying matrices. Since we cannot write a faster method to do this multiplication, we may want to try to reduce the number of matrix multiplications we perform.

Speeding Up Code

Once you have identified those parts of your code that take the most time, how do you make them run faster? Here are some of the techniques we will address in this lab:

- Avoid recomputing values
- Avoid nested loops
- Use existing functions instead of writing your own
- Use generators when possible
- Avoid excessive function calls
- Write Pythonic code
- Compiling Using Numba
- Use a more efficient algorithm

You should always use the profiling and timing functions to help you decide when an optimization is actually useful.

Problem 1. In this lab, we will perform many comparisons between the runtimes of various functions. To help with these comparisions, implement the following function:

```
def compare_timings(f, g, *args):
    """Compare the timings of 'f' and 'g' with arguments '*args'.

    Inputs:
        f (func): first function to compare.
        g (func): second function to compare.
        *args: arguments to use when callings functions 'f' and 'g',
            i.e., call f with f(*args).
    Returns:
        comparison (str): The comparison of the runtimes of functions
            'f' and 'g' in the following format:
                Timing for <f>: <time>
                Timing for <g>: <time>
    """
```

(Hint: You can gain access to the name of many functions by using its `func_name()` method. However, this method does not exist for all functions we will be interested in timing. Therefore, even though it is not as clean, use `str(f)` to print a string representation of f.)

Avoid Recomputing Values

In our function `qr1()`, we can avoid recomputing `R[i,i]` in the outer loop and `R[i,j]` in the inner loop. The rewritten function is as follows:

```
def qr2(A):
    ncols = A.shape[1]
    Q = A.copy()
    R = np.zeros((ncols, ncols))
    for i in range(ncols):
        R[i, i] = la.norm(Q[:, i])
        Q[:, i] = Q[:, i]/R[i, i] # this line changed
        for j in range(i+1, ncols):
            R[i, j] = Q[:, j].dot(Q[:, i])
            Q[:,j] = Q[:,j]-R[i, j]*Q[:,i] # this line changed
    return Q, R
```

Profiling `qr2()` on a 300×300 matrix produces the following output.

```
       48756 function calls in 1.047 seconds

Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.863    0.863    1.047    1.047 profiling.py:16(qr2)
 44850    0.171    0.000    0.171    0.000 {method 'dot' of 'numpy.ndarray' objects}
   300    0.003    0.000    0.006    0.000 function_base.py:526(asarray_chkfinite)
   300    0.003    0.000    0.005    0.000 linalg.py:1840(norm)
   600    0.002    0.000    0.002    0.000 {method 'any' of 'numpy.ndarray' objects}
   300    0.001    0.000    0.001    0.000 {method 'reduce' of 'numpy.ufunc' objects}
   301    0.001    0.000    0.001    0.000 {range}
   600    0.001    0.000    0.001    0.000 {numpy.core.multiarray.array}
   600    0.001    0.000    0.001    0.000 numeric.py:167(asarray)
   300    0.000    0.000    0.012    0.000 misc.py:7(norm)
     1    0.000    0.000    0.000    0.000 {method 'copy' of 'numpy.ndarray' objects}
   300    0.000    0.000    0.000    0.000 {method 'ravel' of 'numpy.ndarray' objects}
     1    0.000    0.000    1.047    1.047 <string>:1(<module>)
   300    0.000    0.000    0.000    0.000 {method 'conj' of 'numpy.ndarray' objects}
     1    0.000    0.000    0.000    0.000 {numpy.core.multiarray.zeros}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Our optimization reduced almost every kind of function call by half, and reduced the total run time by 0.295s.

Some less obvious ways to eliminate excess computations include moving computations out of loops, not copying large data structures, and simplifying mathematical expressions.

Avoid Nested Loops

For many algorithms, the temporal complexity of an algorithm is determined by its loops. Nested loops quickly increase the temporal complexity. The best way to avoid nested loops is to use NumPy array operations instead of iterating through arrays. If you must use nested loops, focus your optimization efforts on the innermost loop, which gets called the most times.

Problem 2. The code below is an inefficient implementation of the LU algorithm. Write a function `LU_opt()` that is an optimized version of `LU()`. Look for ways to avoid recomputing values and avoid nested loops by using array slicing instead. Print a comparison of the timing of the original function and your optimized function using your `compare_timings()` function.

```
def LU(A):
    """Return the LU decomposition of a square matrix."""
    n = A.shape[0]
    U = np.array(np.copy(A), dtype=float)
    L = np.eye(n)
    for i in range(1, n):
        for j in range(i):
            L[i,j] = U[i,j] / U[j,j]
            for k in range(j, n):
                U[i,k] -= L[i,j] * U[j,k]
    return L, U
```

Use Built-in Functions

If there is an intuitive operation you would like to perform on an array, chances are that NumPy or another library already has a function that does it. Python and NumPy functions have already been optimized, and are usually many times faster than the equivalent you might write. We saw an example of this in Lab **??** where we compared NumPy array multiplication with our own matrix multiplication implemented in Python.

Problem 3. Without using any builtin functions, implement the following function.

```
def mysum(x):
    """Return the sum of the elements of X without using a built-in ↩
       function.

    Inputs:
        x (iterable): a list, set, 1-d NumPy array, or another iterable.
    """
```

Compare `mysum()` to Python's builtin `sum()` function and NumPy's `np.sum()` using your `compare_timings()` function.

Use Generators

When you are iterating through a list, you can often replace the list with a *generator*. Instead of storing the entire list in memory, a generator computes each item as it is needed. For example, the code

```
>>> for i in range(100):
...     print i
```

stores the numbers 0 to 99 in memory, looks up each one in turn, and prints it. On the other hand, the code

```
>>> for i in xrange(100):
...     print i
```

uses a generator instead of a list. This code computes the first number in the specified range (which is 0), and prints it. Then it computes the next number (which is 1) and prints that.

In our example, replacing each `range()` with `xrange()` does not speed up `qr2()` by a noticeable amount.

Though the example below is contrived, it demonstrates the benefits of using generators.

```
# both these functions will return the first iterate of a loop of length 10^8.
def list_iter():
    for i in range(10**8):              # Use range()
        return i
```

```
def generator_iter():
    for i in xrange(10**8):             # Use xrange()
        return i

>>> compare_timings(list_iter,generator_iter)
Timing for <function list_iter at 0x7f3deb5a4488>: 1.93316888809
Timing for <function generator_iter at 0x7f3deb5a4500>: 1.19209289551e-05
```

It is also possible to write your own generators. Say we have a function that returns an array. And say we want to iterate through this array later in our code. In situations like these, it is valuable to consider turning your function into a generator instead of returning the whole list. The benefits of this approach mirror the benefits of using `xrange()` instead of `range()`. The only thing that needs to be adjusted is to change the `return` statement to a `yield` statement. Here is a quick example:

```
def return_squares(n):
    squares = []
    for i in xrange(1,n+1):
        squares.append(i**2)
    return squares

def yield_squares(n):
    for i in xrange(1,n+1):
        yield i**2
```

The `yield` statement "returns" the single value, but all the local variables for the function are stored away until the next iteration. To iterate step-by-step through a generator, use the builtin `next()` function..

```
>>> squares = yield_squares(3)
>>> next(squares)
1
>>> next(squares)
4
>>> next(squares)
9
```

We can also iterate through a generator using a for loop.

```
>>> for s in yield_squares(3):
...     print s,
...
1 4 9
```

Problem 4. The *Fibonacci sequence* is defined by the recurrence relation $F_{n+1} = F_n + F_{n-1}$, where $F_1 = F_2 = 1$. Write a generator that yields the first n Fibonacci numbers.

If you are interested in learning more about writing your own generators, see `https://docs.python.org/2/tutorial/classes.html#generators` and `https://wiki.python.org/moin/Generators`.

Avoid Excessive Function Calls

Function calls take time. Moreover, looking up methods associated with objects takes time. Removing "dots" can significantly speed up execution time.

For example, we could rewrite our function to reduce the number of times we need to look up the function `la.norm()`.

```
def qr2(A):
    norm = la.norm      # This reduces the number of function lookups.
    ncols = A.shape[1]
    Q = A.copy()
    R = np.zeros((ncols, ncols))
    for i in range(ncols):
        R[i, i] = norm(Q[:, i])
        Q[:, i] = Q[:, i]/R[i, i]
        for j in range(i+1, ncols):
            R[i, j] = Q[:, j].dot(Q[:, i])
            Q[:,j] = Q[:,j]-R[i, j]*Q[:,i]
    return Q, R
```

Once again, an analysis with `%prun` reveals that this optimization does not help significantly in this case.

Write Pythonic Code

Several special features of Python allow you to write fast code easily. First, list comprehensions are much faster than for loops. These are particularly useful when building lists inside a loop. For example, replace

```
>>> mylist = []
>>> for i in xrange(100):
...     mylist.append(math.sqrt(i))
```

with

```
>>> mylist = [math.sqrt(i) for i in xrange(100)]
```

We can accomplish the same thing using the `map()` function, which is even faster.

```
>>> mylist = map(math.sqrt, xrange(100))
```

The analog of a list comprehension also exists for generators, dictionaries, and sets.

Second, swap values with a single assignment.

```
>>> a, b = 1, 2
>>> a, b = b, a
>>> print a, b
2 1
```

Third, many non-Boolean objects in Python have truth values. For example, numbers are `False` when equal to zero and `True` otherwise. Similarly, lists and strings are `False` when they are empty and `True` otherwise. So when `a` is a number, instead of

```
>>> if a != 0:
```

use

```
>>> if a:
```

Lastly, it is more efficient to iterate through lists by iterating over the elements instead of iterating over the indices.

```
# Bad
for i in xrange(len(my_list)):
    print my_list[i],

# Good
for x in my_list:
    print x,
```

However, there are situations where you will need to know the indices of the elements over which you are iterating. In these situations, use `enumerate`.

Problem 5. Consider the following poorly-written function.

```
def foo(n):
    my_list = []
    for i in range(n):
        num = np.random.randint(-9,9)
        my_list.append(num)
    evens = 0
    for j in range(n):
        if my_list[j] % 2 == 0:
            evens += my_list[j]
    return my_list, evens
```

Walk through the code line by line to determine what the code is accomplishing. Using `%prun`, find out which portions of the code below require the most runtime. Rewrite the function to perform the same task in a more efficient way using the optimization techniques we have discussed.

Numba

Though it is much easier to write simple, readable code in Python, it is also much slower than compiled languages such as C. Compiled languages, in general, are much faster. *Numba* is a tool that uses *just-in-time* (JIT) compilation to optimize code, meaning that the code is compiled right before it is executed. We will discuss this process a bit later in this section.

The API for using Numba is incredibly simple. All one has to do is import Numba and add the `@jit` function decorator to your function. The following code would be a Numba equivalent to Problem 3.

```
from numba import jit
@jit
def numba_sum(A):
    total = 0
    for x in A:
        total += x
    return total
```

Though this code looks very simple, a lot is going on behind the scenes. One of the reasons compiled languages like C are so much faster than Python is because they have explicitly defined datatypes. The main strategy used by Numba is to speed up the Python code by assigning datatypes to all the variables. Rather than requiring us to define the datatypes explicitly as we would need to in any compiled language, Numba attempts to *infer* the correct datatypes based on the datatypes of the input.

In the code above, for example, say that our array `A` was an array of integers. Though we have not explicitly defined a datatype for the variable `total`, Numba will infer that the datatype for total should also be an integer.

Once all the datatypes have been inferred and assigned, the code is translated to machine code by the LLVM library. Numba will then cache this compiled version of our code. This means that we can bypass this whole inference and compilation process the next time we run our function.

More Control Within Numba

Though the inference engine within Numba does a good job, it's not always perfect. There are times that Numba is unable to infer all the datatypes correctly.

If you add the keyword argument, `nopython=True` to the `jit` decorator, an error will be raised if Numba was unable to convert everything to explicit datatypes.

If your function is running slower than you would expect, you can find out what is going on under the hood by calling the `inspect_types()` method of the function. Using this, you can see if all the datatypes are being assigned as you would expect.

```
# Due to the length of the output, we will leave it out of the lab text.
>>> numba_sum.inspect_types()
```

If you would like to have more control, you may specify datatypes explicity as demonstrated in the code below.

In this example, we will assume that the input will be doubles. Note that is necessary to import the desired datatype from the Numba module.

```
from numba import double
# The values inside 'dict' will be specific to your function.
@jit(nopython=True, locals=dict(A=double[:], total=double))
def numba_sum(A):
    total = 0
    for i in xrange(len(A)):
        total += A[i]
    return total
```

Notice that the jit function decorator is the only thing that changed. Note also that this means that we will not be allowed to pass an array of integers to this function. If we had not specified datatypes, the inference engine would allow us to pass arrays of any numerical datatype. In the case that our function sees a datatype that it has not seen before, the inference and compilation process would have to be repeated. As before, the new version will also be cached.

Problem 6. The code below defines a Python function which takes a matrix to the nth power.

```
def pymatpow(X, power):
    """Return X^{power}, the matrix product XX...X, 'power' times.

    Inputs:
        X ((n,n) ndarray): A square matrix.
        power (int): The power to which to raise X.
    """
    prod = X.copy()
    temparr = np.empty_like(X[0])
    size = X.shape[0]
    for n in xrange(1, power):
        for i in xrange(size):
            for j in xrange(size):
                tot = 0.
                for k in xrange(size):
                    tot += prod[i,k] * X[k,j]
                temparr[j] = tot
            prod[i] = temparr
    return prod
```

1. Create a function `numba_matpow()` that is the compiled version of `pymatpow()` using Numba.

2. Write a function `numpy_matpow()` that performs the same task as `pymatpow` but uses `np.dot()`. Compile this function using Numba.

3. Compare the speed of `pymatpow()`, `numba_matpow()` and the `numpy_matpow()` function. Remember to time `numba_matpow()` and `numpy_matpow()` on the second pass so the compilation process is not part of your timing. Perform your comparisons using your `compare_timings()` function.

NumPy takes products of matrices by calling BLAS and LAPACK, which are heavily optimized linear algebra libraries written in C, assembly, and Fortran.

ACHTUNG!

NumPy's array methods are often faster than a Numba equivalent that you could code yourself. If you are unsure which method is fastest, time them.

Use a More Efficient Algorithm

The optimizations discussed thus far will speed up your code at most by a constant. They will not change the complexity of your code. In order to reduce the complexity (say from $O(n^2)$ to $O(n\log(n))$), you typically need to change your algorithm. We will address the benefits of using more efficient algorithms in Problem 7.

A good algorithm written with a slow language (like Python) is faster than a bad algorithm written in a fast language (like C). Hence, focus on writing fast algorithms with good Python code, and only Numba when and where it is necessary. In other words, Numba will not always save you from a poor algorithm design.

The correct choice of algorithm is more important than a fast implementation. For example, suppose you wish to solve the following tridiagonal system.

$$A\mathbf{x} = \begin{bmatrix} b_1 & c_1 & 0 & 0 & \cdots & \cdots & 0 \\ a_2 & b_2 & c_2 & 0 & \cdots & \cdots & 0 \\ 0 & a_3 & b_3 & c_3 & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & c_{n-1} \\ 0 & 0 & 0 & 0 & \cdots & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ \vdots \\ d_n \end{bmatrix} = \mathbf{d}$$

One way to do this is with the `solve()` method in SciPy's `linalg` module. Alternatively, you could use an algorithm optimized for tridiagonal matrices. The code below implements one such algorithm in Python called the *Thomas algorithm*.

```
def pytridiag(a,b,c,d):
    """Solve the tridiagonal system Ax = d where A has diagonals a, b, and c.

    Inputs:
        a ((n-1,) ndarray): first subdiagonal of A.
        b ((n,) ndarray): main diagonal of A.
        c ((n-1,) ndarray): first superdiagonal of A.
        d ((n,) ndarray): the right side of the linear system.
```

```
    Returns:
        x ((n,) array): solution to the tridiagonal system Ax = d.
    """
    n = len(b)

    # Make copies so the original arrays remain unchanged.
    aa = np.copy(a)
    bb = np.copy(b)
    cc = np.copy(c)
    dd = np.copy(d)

    # Forward sweep.
    for i in xrange(1, n):
        temp = aa[i-1] / bb[i-1]
        bb[i] = bb[i] - temp*cc[i-1]
        dd[i] = dd[i] - temp*dd[i-1]

    # Back substitution.
    x = np.zeros_like(b)
    x[-1] = dd[-1] / bb[-1]
    for i in reversed(xrange(n-1)):
        x[i] = (dd[i] - cc[i]*x[i+1]) / bb[i]

    return x
```

Problem 7.

1. Write a function that is a compiled version of `pytridiag()`.
2. Compare the speed of your new function with `pytridiag()` and `scipy.linalg.solve()`. When comparing `numba_tridiag()` and `pytridiag()`, use 1000000×1000000 sized systems. When comparing `numba_tridiag()` and the SciPy algorithm, use a 1000×1000 systems.

You may use the code below to generate the arrays `a`, `b`, and `c`, as well as the matrix A.

```
def init_tridiag(n):
    """Construct a random nxn tridiagonal matrix A by diagonals.

    Inputs:
        n (int): The number of rows / columns of A.

    Returns:
        a ((n-1,) ndarray): first subdiagonal of A.
        b ((n,) ndarray): main diagonal of A.
        c ((n-1,) ndarray): first superdiagonal of A.
```

```
        A ((n,n) ndarray): the tridiagonal matrix.
    """
    a = np.random.random_integers(-9, 9, n-1).astype("float")
    b = np.random.random_integers(-9 ,9, n  ).astype("float")
    c = np.random.random_integers(-9, 9, n-1).astype("float")

    # Replace any zeros with ones.
    a[a==0] = 1
    b[b==0] = 1
    c[c==0] = 1

    # Construct the matrix A.
    A = np.zeros((b.size,b.size))
    np.fill_diagonal(A, b)
    np.fill_diagonal(A[1:,:-1], a)
    np.fill_diagonal(A[:-1,1:], c)

    return a, b, c, A
```

Note that an efficient tridiagonal matrix solver is implemented by `scipy.sparse.linalg.spsolve()`.

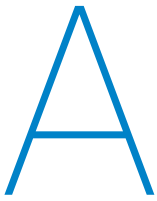
When to Stop Optimizing

You don't need to apply every possible optimization to your code. When your code runs acceptably fast, stop optimizing. There is no need spending valuable time making optimizations once the speed is sufficient.

Moreover, remember not to prematurely optimize your functions. Make sure the function does exactly what you want it to before worrying about any kind of optimization.

Part II

Appendices



Initial Installation

Lab Objective: *Install fundamental package requirements for the labs on a computer.*

> ACHTUNG!
>
> The information provided in this appendix is for convenience only. The reader assumes all the liability and risk involved in making any change to configuration mentioned in this appendix. The authors of this appendix are not responsible for any damage that may result from any changes covered in following material. Some of the changes could cause unexpected behavior in the computer system.

There are a wide variety of ways to install most of the Python packages required for these labs. Installing these libraries can be difficult; the following are ways that have worked for us. They still may not work on every computer. The following steps have been verified by the authors of this guide on Windows 7, Windows 8.1, MacOSX, and Ubuntu 14.04.

Installing Using the Anaconda Python Distribution

Regardless of which operating system you are using, the basic processes needed to install Anaconda are the same:

1. Download the installer from `http://www.continuum.io/downloads`
2. Ensure that the main `Anaconda` directory has been added to the system path
3. Perform some minor tweaks to files in the distribution to prepare for future packages. This step will be the same on all operating systems.

Windows 64-bit Installation

We will provide detailed steps for Windows 64-bit installation, but as already mentioned, the process as a whole is the same on each operating system.

1. Note: you can get an academic license for more of the features of the Anaconda Distribution. The primary benefit is that it will allow many basic linear algebra routines to run much faster. This is not needed for the initial installation but is an option to consider. To obtain an academic license, see `https://store.continuum.io/cshop/academicanaconda`. To activate your academic license, follow the steps in the confirmation email.

2. Install the Anaconda Python Distribution for Windows from `http://www.continuum.io/downloads`

3. Start the installer. Install using the default configuration. As it is installing, make note of the directory where Anaconda is being installed.

4. Ensure the main `Anaconda` directory and the `bin`, `libs`, and `include` subdirectories have been added to your system path if they aren't already there. On Windows, you can find this option by opening the file explorer to "Computer", right clicking on the background and clicking on "Properties", then clicking on "Advanced Settings", then clicking on "Environment Variables". If the Anaconda was installed at `C:\Anaconda\`, add the following to the end of the path if it isn't already there:

   ```
   ;C:\Anaconda\; C:\Anaconda\Scripts; C:\Anaconda\libs; C:\Anaconda\include
   ```

   If the Anaconda directory was installed at `C:\Users\John\Anaconda\`, add the following to the end of the path if it isn't already there:

   ```
   ;C:\Users\John\Anaconda\; C:\Users\John\Anaconda\Scripts; C:\Users\John\Anaconda
   \libs; C:\Users\John\Anaconda\include
   ```

   This should be done to the path variable in the box for "System Variables" (the system path instead of the user path).

Minor Edits to Files

These minor edits are not necessary to get started right away, but will be necessary for some of the libraries we will be using in the future.

1. Open `C:\Anaconda\Lib\site-packages\numpy\distutils\fcompiler\gnu.py` and comment out the two lines that read

   ```
   else:
       raise NotImplementedError("Only MS compiler supported with gfortran on↩
           win64")
   ```

   They are probably somewhere around line 330. Once you have modified them, they should read

   ```
   #else:
   #    raise NotImplementedError("Only MS compiler supported with gfortran ↩
       on win64")
   ```

2. Open the file `C:\Anaconda\Lib\distutils\distutils.cfg` (create it if it doesn't already exist) and make sure it contains the following two definitions

```
[build]
compiler=mingw32

[build_ext]
compiler=mingw32
```

> **Note**
>
> If you update NumPy, you will need to modify the file `C:\Anaconda\Lib\site-packages\numpy\distutils\fcompiler\gnu.py` the same way you did when you first installed everything.

Workflows

There are several different ways to write your programs and run them in Python. It will be valuable for you to try a variety of workflows to find which one you prefer.

Text editor + Python Interpreter/IPython

This is the most basic workflow to use. It involves using your favorite text editor to edit source files and then executing those scripts in your command terminal. Some popular editors for python source code include:

- Vim
- Emacs
- Sublime
- Notepad++ (Windows)
- TextWrangler (OS X)
- Geany (Linux)

IPython is an enhanced interactive interpreter for Python. It has many features that cater to productivity. One very useful feature is object introspection. This feature allows you to examine the properties and methods of any object in the interpreter. Another useful feature is tab completion where the interpreter automaticalls fills in partially typed commands when the tab key is pressed. Using the IPython interpreter with a text editor is a good way to work in Python.

Integrated Development Environments

Integrated Development Environments (IDEs) provide a comprehensive environment for application development. Most IDEs have many tightly integrated tools that are easily accessible. Some popular IDEs for developing in Python are discussed in this section.

IPython Notebook

The IPython Notebook is a browser-based interface to Python. It is similar to the notebook interfaces used by Mathematica, Maple, and Sage. Running a notebook is similar to running a Python interpreter session except that the input is stored in cells and can be modified and re-evaluated as needed. You can also save notebooks and reload them later. The IPython notebook is included as part of the Anaconda Python Distribution, the Enthought Python Distribution, and Python(x,y).

Eclipse + PyDev

Eclipse: `http://www.eclipse.org/`
PyDev: `http://pydev.org/`

Eclipse is a general purpose IDE that supports many languages. The PyDev plugin for Eclipse contains all the tools needed to start working in Python. It includes a built-in debugger, and has a very nice code editor. Eclipse + PyDev is available for Windows, Linux, and Mac OSX.

Spyder

Spyder: `http://code.google.com/p/spyderlib/`

Spyder is a Python IDE that is designed specifically for scientific computing. Its interface is reminiscent of MATLAB and requires PyQT4 to be installed. Spyder is available for Windows and Linux. There is a version available for Mac OSX through MacPorts.

B Installing and Updating Python Packages

Lab Objective: *The process of installing and updating Python packages.*

Installing New Packages

There are a number of ways to install Python packages. We suggest trying the following methods in order.

Anaconda Package Manager

There are some packages that you can install through Anaconda that were not included by default in your initial installation. Look to see if the package you wish to download is found here: `http://docs.continuum.io/anaconda/pkg-docs.html`. If the package you wish to install is not included in this list, this method is not an option.

```
$ conda update conda   # ensures you have the most recent version
$ conda install <packagename>  # Do not include < >
```

You do not need to download anything extra, just run the command. It is not necessary to include the version number in this command. The package name is sufficient.

Pip

`pip` is one of the most popular Python Package Managers and it works similarly to `conda`. Go to the following website to see if the package you would like to install is included: `https://pypi.python.org/pypi?%3Aaction=index`. If the package you wish to install is not included in this list, this method will not be an option.

```
$ pip install <packagename>  # Do not include < >
```

Windows Only: Wheel Files

Wheel files (`.whl`) are becoming more popular. Wheel is a built-package format, so you do not need to worry about compiling anything. Though this method will work with some extra effort on MacOSX and Linux, it is currently easier to perform on Windows. This is because wheel files for MacOSX and Linux are currently hard to find. Trying other methods may be a better use of your time. For Windows, `http://www.lfd.uci.edu/~gohlke/pythonlibs/` contains a very large list of packages for Python. You will notice that there are multiple versions of these packages available. Most will be in a format similar to the following examples:

```
pandas-0.16.2-cp27-none-win_amd64.whl # 64-bit, Python 2.7
pandas-0.16.2-cp27-none-win32.whl # 32-bit, Python 2.7
```

Be sure to download the version compatible with your computer and Python 2.7. Once the wheel file is downloaded, navigate to the directory where the file is saved and run the following:

```
$ pip install wheel   # Only need to do this once
$ pip install <filename>  # Do not include < >
```

If All Else Fails...

If all else fails, you can download the package you need from the source. There are often installation instructions included in a package, but it can be difficult to set up everything correctly. If you are having trouble, it is likely others have had difficulties as well. A quick internet search may provide some direction.

> **Note**
>
> The best way to ensure a package has been installed correctly is to try importing it in a Python Interactive Shell.

Updating packages

> **Note**
>
> If you update NumPy, you will need to modify the file `C:\Anaconda\Lib\site-packages\numpy\distutils\fcompiler\gnu.py` as outlined in Appendix A.

Anaconda

As a general rule, you update your packages using the same system you used to install them. If you want to update a package included by using in Anaconda, you can run (in the command terminal) the command

```
$ conda update <packagename>   # Do not include < >
```

If you want to update all of the packages that are included in Anaconda, you can run the two commands

```
$ conda update conda
$ conda update anaconda
```

The first of these commands updates Anaconda's package manager. The second updates all packages so that they match with the current release version of Anaconda. Because Anaconda has this capability, it is a good idea to install packages using `conda` whenever possible.

Pip

If you installed a package via `pip`, you can upgrade it using the command

```
$ pip install --upgrade <packagename>  # Do not include < >
```

If you would like to see the version number of all the packages you have installed, run:

```
$ pip freeze
```

Other

If you installed a package via an online installer, get an updated version from the same source and follow the same steps as you did when you first installed it.

Achtung!

Be careful not to try to update a Python package while it is in use. It is safest to update packages while there is no Python interpreter currently running.

C NumPy Visual Guide

Lab Objective: *NumPy operations can be difficult to visualize, but the concepts are straightforward. This appendix provides visual demonstrations of how NumPy arrays are used with slicing syntax, stacking, broadcasting, and axis-specific operations. Though these visualizations are for 1- or 2-dimensional arrays, the concepts can be extended to n-dimensional arrays.*

Data Access

The entries of a 2-D array are the rows of the matrix (as 1-D arrays). To access a single entry, enter the row index, a comma, and the column index. Remember that indexing begins with 0.

$$A[0] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \qquad A[2,1] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

Slicing

A lone colon extracts an entire row or column from a 2-D array. The syntax `[a:b]` can be read as "the a^{th} entry up to (but not including) the b^{th} entry." Similarly, `[a:]` means "the a^{th} entry to the end" and `[:b]` means "everything up to (but not including) the b^{th} entry."

$$\texttt{A[1]} = \texttt{A[1,:]} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \qquad \texttt{A[:,2]} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

$$\texttt{A[1:,:2]} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \qquad \texttt{A[1:-1,1:-1]} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

Stacking

`np.hstack()` stacks sequence of arrays horizontally and `np.vstack()` stacks a sequence of arrays vertically.

$$A = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \qquad B = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}$$

$$\texttt{np.hstack((A,B,A))} = \begin{bmatrix} \times & \times & \times & * & * & * & \times & \times & \times \\ \times & \times & \times & * & * & * & \times & \times & \times \\ \times & \times & \times & * & * & * & \times & \times & \times \end{bmatrix}$$

$$\texttt{np.vstack((A,B,A))} = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ * & * & * \\ * & * & * \\ * & * & * \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}$$

Because 1-D arrays are flat, `np.hstack()` concatenates 1-D arrays and `np.vstack()` stacks them vertically. To make several 1-D arrays into the columns of a 2-D array, use `np.column_stack()`.

$$x = \begin{bmatrix} \times & \times & \times & \times \end{bmatrix} \qquad y = \begin{bmatrix} * & * & * & * \end{bmatrix}$$

$$\texttt{np.hstack((x,y,x))} = \begin{bmatrix} \times & \times & \times & \times & * & * & * & * & \times & \times & \times & \times \end{bmatrix}$$

$$\texttt{np.vstack((x,y,x))} = \begin{bmatrix} \times & \times & \times & \times \\ * & * & * & * \\ \times & \times & \times & \times \end{bmatrix} \qquad \texttt{np.column\_stack((x,y,x))} = \begin{bmatrix} \times & * & \times \\ \times & * & \times \\ \times & * & \times \\ \times & * & \times \end{bmatrix}$$

Broadcasting

NumPy automatically aligns arrays for component-wise operations whenever possible. See `http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html` for more in-depth examples and broadcasting rules.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \qquad x = \begin{bmatrix} 10 & 20 & 30 \end{bmatrix}$$

$$\texttt{A + x} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 & 20 & 30 \end{bmatrix} = \begin{bmatrix} 11 & 22 & 33 \\ 11 & 22 & 33 \\ 11 & 22 & 33 \end{bmatrix}$$

$$\texttt{A + np.vstack(x)} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$$

Operations along an Axis

Most array methods have an `axis` argument that allows an operation to be done along a given axis. To compute the sum of each column, use `axis=0`; to compute the sum of each row, use `axis=1`.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$\texttt{A.sum(axis=0)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 8 & 12 & 16 \end{bmatrix}$$

$$\texttt{A.sum(axis=1)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 10 & 10 \end{bmatrix}$$

D Plot Customization and Matplotlib Syntax Guide

Lab Objective: *The documentation for Matplotlib can be a little difficult to maneuver and basic information is sometimes difficult to find. This appendix condenses and demonstrates some of the more applicable and useful information on plot customizations. For an introduction to Matplotlib, see lab 5.*

Colors

By default, every plot is assigned a different color specified by the "color cycle". It can be overwritten by specifying what color is desired in a few different ways.

-

Matplotlib recognizes some basic built-in colors.

Code	Color
`'b'`	blue
`'g'`	green
`'r'`	red
`'c'`	cyan
`'m'`	magenta
`'y'`	yellow
`'k'`	black
`'w'`	white

The following displays how these colors can be implemented. The result is displayed in Figure D.1.

```
import numpy as np
from matplotlib import pyplot as plt

colors = np.array(["k", "g", "b", "r", "c", "m", "y", "w"])
x = np.linspace(0, 5, 1000)
y = np.ones(1000)
```

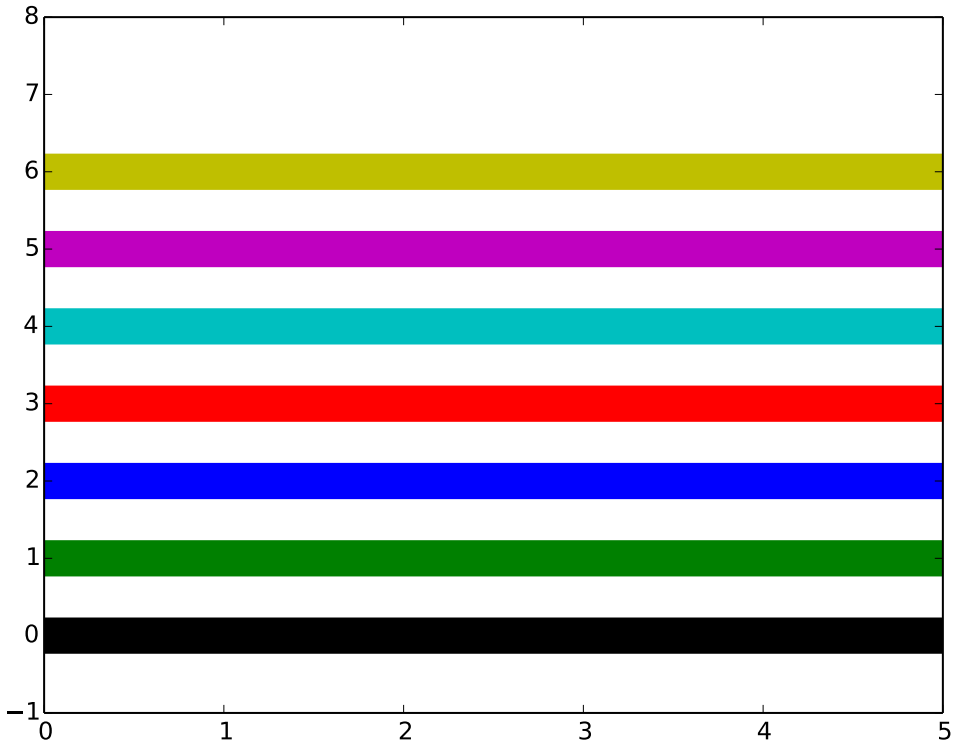


Figure D.1: A display of all the built-in colors.

```
for i in xrange(8):
    plt.plot(x, i*y, colors[i], linewidth=18)

plt.ylim([-1, 8])
plt.savefig("colors.pdf", format='pdf')
plt.clf()
```

colors.py

There are many other ways to specific colors. A popular method to access colors that are not built-in is to use a `RGB` tuple. Colors can also be specified using an html hex string or its associated html color name like `"DarkOliveGreen"`, `"FireBrick"`, `"LemonChiffon"`, `"MidnightBlue"`, `"PapayaWhip"`, or `"SeaGreen"`.

Window Limits

You may have noticed the use of `plt.ylim([ymin, ymax])` in the previous code. This explicitly sets the boundary of the y-axis. Similarily, `plt.xlim([xmin, xmax])` can be used to set the boundary of the x-axis. Doing both commands simultaneously is possible with the `plt.axis([xmin, xmax, ymin, ymax])`. Remember that these commands must be executed after the plot.

Lines

Thickness

You may have noticed that the width of the lines above seemed thin considering we wanted to inspect the line color. `linewidth` is a keyword argument that is defaulted to be `None` but can be given any real number to adjust the line width.

The following displays how `linewidth` is implemented. It is displayed in Figure D.2.

```
lw = np.linspace(.5, 15, 8)

for i in xrange(8):
    plt.plot(x, i*y, colors[i], linewidth=lw[i])

plt.ylim([-1, 8])
plt.show()
```

linewidth.py

Style

By default, plots are drawn with a solid line. The following are accepted format string characters to indicate line style.

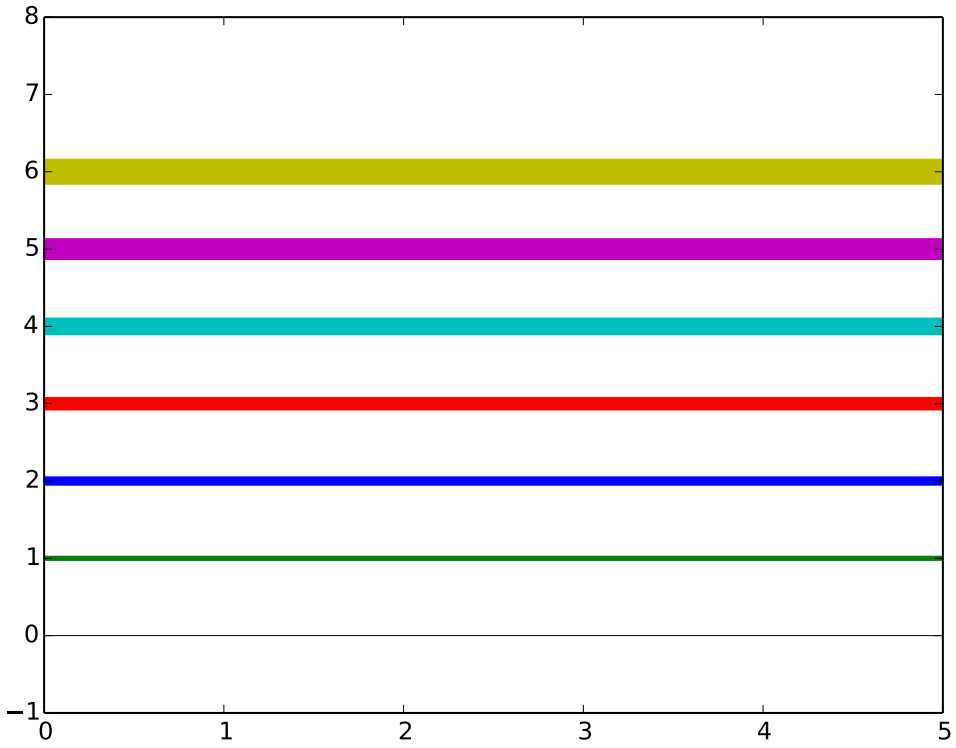


Figure D.2: plot of varying linewidths.

character	description
-	solid line style
--	dashed line style
-.	dash-dot line style
:	dotted line style
.	point marker
,	pixel marker
o	circle marker
v	triangle_down marker
^	triangle_up marker
<	triangle_left marker
>	triangle_right marker
1	tri_down marker
2	tri_up marker
3	tri_left marker
4	tri_right marker
s	square marker
p	pentagon marker
*	star marker
h	hexagon1 marker
H	hexagon2 marker
+	plus marker
x	x marker
D	diamond marker
d	thin_diamond marker
|	vline marker
_	hline marker

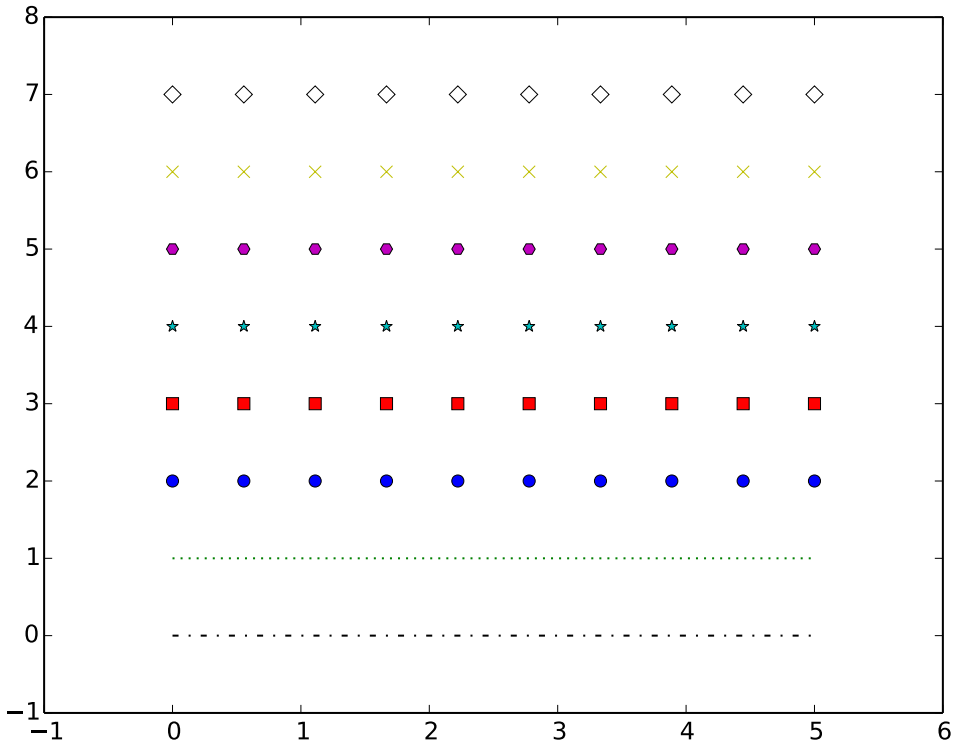


Figure D.3: plot of varying linestyles.

The following displays how `linestyle` can be implemented. It is displayed in Figure D.3.

```
x = np.linspace(0, 5, 10)
y = np.ones(10)
ls = np.array(['-.', ':', 'o', 's', '*', 'H', 'x', 'D'])

for i in xrange(8):
    plt.plot(x, i*y, colors[i]+ls[i])

plt.axis([-1, 6, -1, 8])
plt.show()
```

linestyle.py

Text

It is also possible to add text to your plots. To label your axes, the `plt.xlabel()` and the `plt.ylabel()` can both be used. The function `plt.title()` will add a title to a plot. If you are working with subplots, this command will add a title to the subplot you are currently modifying. To add a title above the entire figure, use `plt.suptitle()`.

All of the `text()` commands can be customized with `fontsize` and `color` keyword arguments.

We can add these elements to our previous example. It is displayed in Figure D.4.

```
for i in xrange(8):
    plt.plot(x, i*y, colors[i]+ls[i])

plt.title("My Plot of Varying Linestyles", fontsize = 20, color = "gold")
plt.xlabel("x-axis", fontsize = 10, color = "darkcyan")
plt.ylabel("y-axis", fontsize = 10, color = "darkcyan")

plt.axis([-1, 6, -1, 8])
plt.show()
```

text.py

See `http://matplotlib.org` for Matplotlib documentation.

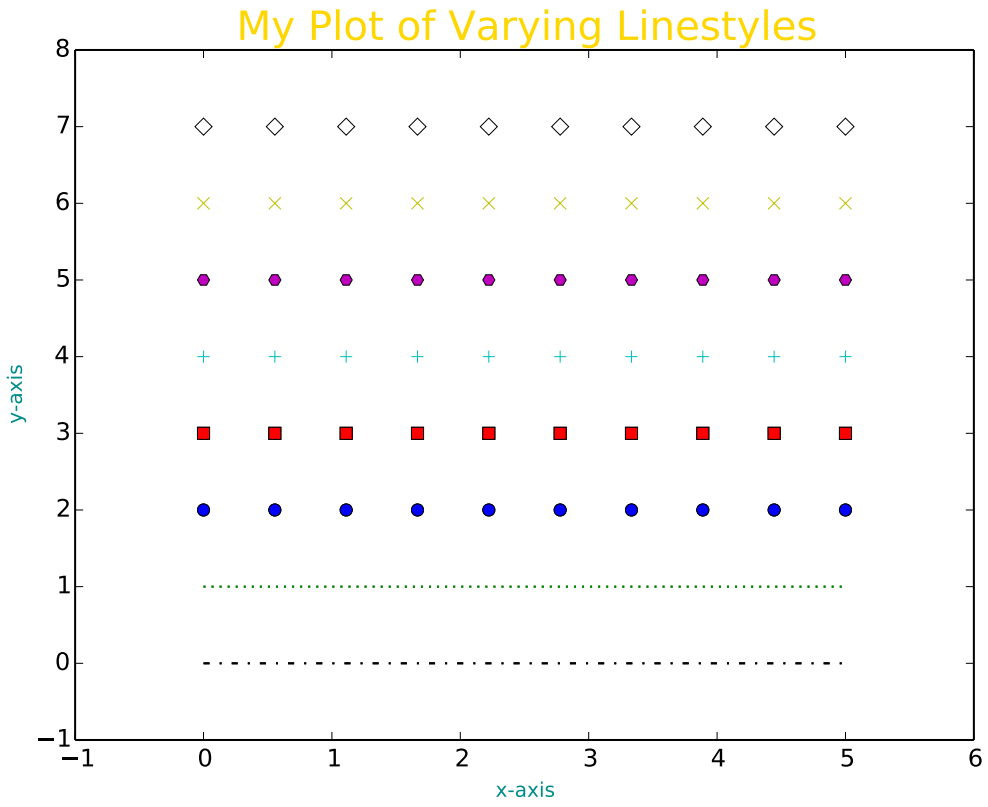


Figure D.4: plot of varying linestyles using text labels.