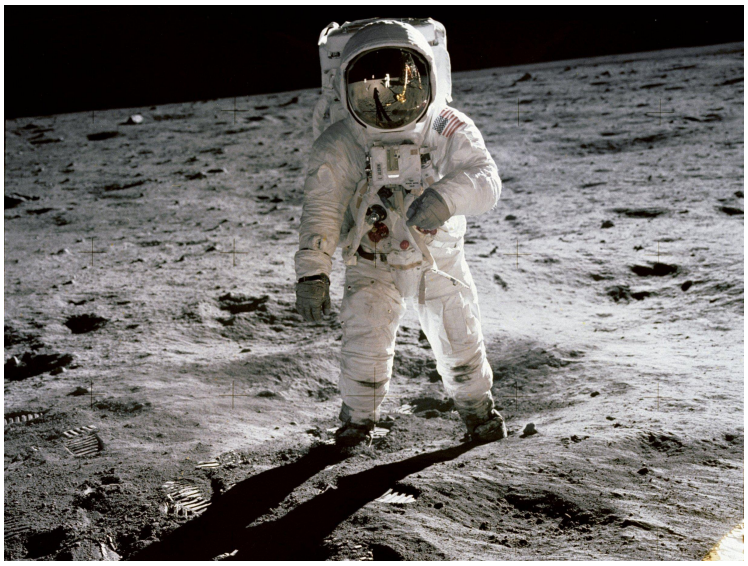
Labs for Foundations of Applied Mathematics

Python Essentials

Jeffrey Humpherys & Tyler J. Jarvis, managing editors



List of Contributors

B. Barker
Brigham Young University

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

K. Baldwin
Brigham Young University

J. Bejarano
Brigham Young University

J. Bennett
Brigham Young University

A. Berry
Brigham Young University

Z. Boyd
Brigham Young University

M. Brown
Brigham Young University

A. Carr
Brigham Young University

C. Carter
Brigham Young University

S. Carter
Brigham Young University

T. Christensen
Brigham Young University

M. Cook
Brigham Young University

M. Cutler
Brigham Young University

R. Dorff
Brigham Young University

B. Ehlert
Brigham Young University

O. Escobar Rodriguez
Brigham Young University

M. Fabiano
Brigham Young University

K. Finlinson
Brigham Young University

J. Fisher
Brigham Young University

R. Flores
Brigham Young University

R. Fowers
Brigham Young University

A. Frandsen
Brigham Young University

R. Fuhriman
Brigham Young University

T. Gledhill
Brigham Young University

S. Giddens
Brigham Young University

C. Gigena
Brigham Young University

M. Graham
Brigham Young University

F. Glines
Brigham Young University

C. Glover
Brigham Young University

M. Goodwin
Brigham Young University

R. Grout
Brigham Young University

D. Grundvig
Brigham Young University

S. Halverson
Brigham Young University

E. Hannesson
Brigham Young University

S. Harding
Brigham Young University

K. Harmer
Brigham Young University

J. Henderson
Brigham Young University

J. Hendricks
Brigham Young University

A. Henriksen
Brigham Young University

I. Henriksen
Brigham Young University

B. Hepner
Brigham Young University

C. Hettinger
Brigham Young University

S. Horst
Brigham Young University

R. Howell
Brigham Young University

E. Ibarra-Campos
Brigham Young University

K. Jacobson
Brigham Young University

R. Jenkins
Brigham Young University

J. Larsen
Brigham Young University

J. Larsen
Brigham Young University

J. Leete
Brigham Young University

Q. Leishman
Brigham Young University

J. Lytle
Brigham Young University

E. Manner
Brigham Young University

M. Matsushita
Brigham Young University

R. McMurray
Brigham Young University

S. McQuarrie
Brigham Young University

E. Mercer
Brigham Young University

D. Miller
Brigham Young University

J. Morrise
Brigham Young University

M. Morrise
Brigham Young University

A. Morrow
Brigham Young University

J. Murphey
Brigham Young University

R. Murray
Brigham Young University

J. Nelson
Brigham Young University

C. Noorda
Brigham Young University

A. Oldroyd
Brigham Young University

J. Oliphant
Brigham Young University

A. Oveson
Brigham Young University

E. Parkinson
Brigham Young University

M. Probst
Brigham Young University

M. Proudfoot
Brigham Young University

D. Reber
Brigham Young University

H. Ringer
Brigham Young University

C. Robertson
Brigham Young University

M. Russell
Brigham Young University

K. Sandall
Brigham Young University

R. Sandberg
Brigham Young University

C. Sawyer
Brigham Young University

N. Schill
Brigham Young University

N. Sill
Brigham Young University

D. Smith
Brigham Young University

J. Smith
Brigham Young University

P. Smith
Brigham Young University

M. Stauffer
Brigham Young University

E. Steadman
Brigham Young University

J. Stewart
Brigham Young University

S. Suggs
Brigham Young University

A. Tate
Brigham Young University

T. Thompson
Brigham Young University

B. Trendler
Brigham Young University

M. Victors
Brigham Young University

E. Walker
Brigham Young University

J. Webb
Brigham Young University

R. Webb
Brigham Young University

J. West
Brigham Young University

R. Wonnacott
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. This manual begins with an introduction to Python [VD10] from scratch, which only requires a basic understanding of general programming concepts (variables, functions, etc.). Later labs introduce several Python packages that are essential for mathematical and scientific computing in Python.

Contents

Part I

Labs

1 Introduction to Python

Lab Objective: *Python is a powerful general-purpose programming language. It can 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. To install Python, see the Getting Started document.

Running Python

Python files are saved with a `.py` extension. For beginners, we strongly recommend using a simple text editor 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 following code.

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

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 also 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 identifies the author and includes instructions on using the file. Executable Python code comes after the header.

Problem 1. Open the file named `python_intro.py` (or create the file in a text editor if you don't have it). Add your information to the header at the top, then add the following code.

```
if __name__ == "__main__":
    print("Hello, world!")          # Indent with four spaces (NOT a tab).
```

Be sure to save your edited file. Open a command prompt (*Terminal* on Linux or Mac and *Command Prompt* or *GitBash* on Windows) and navigate to the directory where the new file is saved. Use the command `ls` (or `DIR` on Windows) to list the files and folders in the current directory, `pwd` (`CD .` on Windows) to print the working directory, and `cd` to change directories.

```
$ pwd                               # Print the working directory.
/Users/Guest
$ ls                                # List the files and folders here.
Desktop     Documents       Downloads       Pictures        Music
$ cd Documents                      # Navigate to a different folder.
$ pwd
/Users/Guest/Documents
$ ls                                # Check to see that the file is here.
python_intro.py
```

Now the Python file can be executed with the following command:

```
$ python python_intro.py
```

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

ACHTUNG!

The `if __name__ == "__main__"` clause is incredibly helpful to create test functions and debug your code. In order to use `if __name__ == "__main__"` to test declared functions, **it must be placed at the end of your file** with the code you desire to run directly following. If you attempt to run that block of code at the beginning of the file with functions that are declared afterwards, you'll get an error of having undefined functions.

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*,[1] [PG07, jup]. To execute a script in IPython, use the `%run` command.

```
>>> exit()                          # 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!
```

Python is an **object-oriented** programming language. As a reminder:

- An **object** refers to a particular instance of a class.
- **Attributes** are the characteristics or properties of an object that store data.
- **Methods** define the actions or behaviors that an object can perform.

One of the biggest advantages of IPython is that it supports *object introspection*, whereas the regular Python interpreter does not. Object introspection quickly reveals 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'.
              append()  count()   insert()  remove()
              clear()   extend()  mro()     reverse()
              copy()    index()   pop()     sort()

# To learn more about a specific method, use a '?' and hit 'Enter'.
In [1]: list.append?
Signature: list.append(self, object, /)
Docstring: Append object to the end of the list.
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 builtins:

class list(object)
 |  list(iterable=(),/)
 |  # ...                           # Press 'q' to exit the info screen.

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

[1] See `https://ipython.org/` and `https://jupyter.org/`.

> **Note**
>
> Use IPython side-by-side with a text editor to test syntax and small code snippets quickly. Testing small pieces of code in IPython **before** putting them into a program reveals errors and greatly speeds up the coding process. Consult the internet with questions; `stackoverflow.com` is a particularly valuable resource for answering common programming questions.
>
> The best way to learn a new coding language is by actually writing code. Follow along with the examples in the yellow code boxes in this lab by executing them in an IPython console. Avoid copy and paste for now; your fingers need to learn the language as well.

Python Basics

Arithmetic

Python can be used as a calculator with the regular `+`, `-`, `*`, and `/` operators. 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
9.0
```

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 > 2.99
True
>>> 1.0 <= 1 or 2 > 3
True
>>> 7 == 7 and not 4 < 4
True

>>> True and True and True and True and True and False
False
>>> False or False or False or False or False or True
```

[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.

```
True
>>> True or not True
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 that returns `True` or `False`, as in the previous code block.

Unlike many programming languages, Python does not require a variable's data type to be specified upon initialization. Because of this, Python is called a *dynamically typed* language.

```
>>> 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 parameters, and a colon. Then indent the function body using exactly **four** spaces.

```
>>> def add(x, y):
...     return x + y                # Indent with four spaces.
```

Achtung!

Many other languages use the curly braces `{}` to delimit blocks, but Python uses whitespace indentation. In fact, whitespace is essentially the only thing that Python is particularly picky about compared to other languages: **mixing tabs and spaces confuses the interpreter and causes problems**. Most text editors have a setting to set the indentation type to spaces so you can use the tab key on your keyboard to insert four spaces (sometimes called *soft tabs*). For consistency, **never** use tabs; **always** use 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):        # Define the function.
...     return width * height
...
```

```
>>> area(2, 5)                          # Call the function.
10
```

Python functions can also return multiple values.

```
>>> def arithmetic(a, b):
...     return a - b, a * b             # Separate return values with commas.
...
>>> x, y = arithmetic(5, 2)             # Unpack the returns into two variables.
>>> print(x, y)
3 10
```

The keyword `lambda` is a shortcut for creating one-line functions. For example, the polynomials $f(x) = 6x^3 + 4x^2 - x + 3$ and $g(x, y, z) = x + y^2 - z^3$ can be defined as functions in one line each.

```
# Define the polynomials the usual way using 'def'.
>>> def f(x):
...     return 6*x**3 + 4*x**2 - x + 3
>>> def g(x, y, z):
...     return x + y**2 - z**3

# Equivalently, define the polynomials quickly using 'lambda'.
>>> f = lambda x: 6*x**3 + 4*x**2 - x + 3
>>> g = lambda x, y, z: x + y**2 - z**3
```

Note

Documentation is important in every programming language. Every function should have a *docstring*—a string literal in triple quotes just under the function declaration—that describes the purpose of the function, the expected inputs and return values, and any other notes that are important to the user. Short docstrings are acceptable for very simple functions, but more complicated functions require careful and detailed explanations.

```
>>> def add(x, y):
...     """Return the sum of the two inputs."""
...     return x + y

>>> def area(width, height):
...     """Return the area of the rectangle with the specified width
...     and height.
...     """
...     return width * height
...
>>> def arithmetic(a, b):
...     """Return the difference and the product of the two inputs."""
...     return a - b, a * b
```

Lambda functions cannot have custom docstrings, so the `lambda` keyword should be only be used as a shortcut for very simple or intuitive functions that need no additional labeling.

Problem 2. The volume of a sphere with radius r is $V = \frac{4}{3}\pi r^3$. In your Python file from Problem 1, define a function called `sphere_volume()` that accepts a single parameter r. Return the volume of the sphere of radius r, using 3.14159 as an approximation for π (for now). Also write an appropriate docstring for your function.

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.

Achtung!

The `return` statement instantly ends the function call and passes the return value to the function caller. However, functions are not required to have a return statement. A function without a return statement implicitly returns the Python constant `None`, which is similar to the special value `null` of many other languages. Calling `print()` at the end of a function does **not** cause a function to return any values.

```
>>> def oops(i):
...     """Increment i (but forget to return anything)."""
...     print(i + 1)
...
>>> def increment(i):
...     """Increment i."""
...     return i + 1
...
>>> x = oops(1999)                  # x contains 'None' since oops()
2000                                # doesn't have a return statement.
>>> y = increment(1999)             # However, y contains a value.
>>> print(x, y)
None 2000
```

If you have any intention of using the results of a function, use a `return` statement.

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

```
>>> def pad(a, b, c=0):
...     """Print the arguments, plus a zero if c is not specified."""
...     print(a, b, c)
...
```

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

It's important to note that positional arguments must precede named arguments in a function call. Additionally, parameters without default values must precede parameters with default values in a function definition. For example, `a` and `b` must come before `c` in the function definition of `pad()`. Examine the following code blocks demonstrating how positional and named arguments are used to call a function.

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

```
# Correctly define pad() with the named argument after positional arguments.
>>> def pad(a, b, c=0):
...     """Print the arguments, plus a zero if c is not specified."""
...     print(a, b, c)
...

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

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

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

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

Problem 3. The built-in `print()` function has the useful keyword arguments `sep` and `end`. It accepts any number of positional arguments and prints them out with `sep` inserted between values (defaulting to a space), then prints `end` (defaulting to the *newline character* `'\n'`).

Write a function called `isolate()` that accepts five arguments. The function should print the first three arguments separated by 5 spaces and then print the last two arguments with a single space separating the last three arguments. For example,

```
>>> isolate(1, 2, 3, 4, 5)
1     2     3 4 5
```

ACHTUNG!

In previous versions of Python, `print()` was a *statement* (like `return`), not a function, and could therefore be executed without parentheses. However, it lacked keyword arguments like `sep` and `end`. If you are using Python 2.7, include the following line at the top of the file to turn the `print` statement into the new `print()` function.

```
>>> from __future__ import print_function
```

Data Types and Structures

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. The `/` operator performs float division (true fractional division), and the `//` operator performs integer division, which rounds the result down to the next integer. If both operands for `//` are integers, the result will be an `int`. If one or both operands are floats, the result will be a `float`. Regular division with `/` always returns a `float`.

```
>>> 15 / 4                           # Float division performs as expected.
3.75
>>> 15 // 4                          # Integer division rounds the result down.
3
>>> 15. // 4
3.0
```

ACHTUNG!

In previous versions of Python, using `/` with two integers performed integer division, even in cases where the division was not even. This can result in some incredibly subtle and frustrating errors. If you are using Python 2.7, always include a `.` on the operands or cast at least one as a float when you want float division.

```
# PYTHON 2.7
>>> 15 / 4                          # The answer should be 3.75, but the
3                                   # interpreter does integer division!

>>> 15. / float(4)                  # 15. and float(4) are both floats, so
3.75                                # the interpreter does float division.
```

Alternatively, including the following line at the top of the file redefines the `/` and `//` operators so they are handled the same way as in Python 3.

```
>>> from __future__ import division
```

Python also supports complex numbers computations by pairing two numbers as the real and imaginary parts. 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.
2.0
>>> y.imag                          # Access the imaginary part of y.
5.0
```

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!'
```

Parts of a string can be accessed 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 4. 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 parameter.
   (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[0]
'Hello'
>>> my_list[-2]
'world'
>>> my_list[:2]
['Hello', 93.8]
```

Common list methods (functions) include `append()`, `insert()`, `remove()`, `extend()`, 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]
>>> my_list.extend([5, 6])           # Append multiple values at once.
>>> my_list
[1, 2, 5, 6]
```

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]
>>> print(my_list)
[-1, 20, 30, 8, 9]
```

The `in` operator quickly checks if a given value is in a list (or another iterable, including strings).

```
>>> my_list = [1, 2, 3, 4, 5]
>>> 2 in my_list
True
>>> 6 in my_list
False
>>> 'a' in "xylophone"               # 'in' also works on strings.
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 fewer 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 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 5. Write a function called `list_ops()`. Define a list with the entries `"bear"`, `"ant"`, `"cat"`, and `"dog"`, 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 (or pop) the entry at index 1.
4. Sort the list in reverse alphabetical order.
5. Replace `"eagle"` with `"hawk"`.
   (Hint: the list's `index()` method may be helpful.)
6. Add the string `"hunter"` to the last entry in the list.

Return the resulting list of strings.

Work out (on paper) what the result should be, then check that your function returns the correct list. Consider printing the list at each step to see the intermediate results.

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

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

>>> 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)
{'Lytle, Josh', 'Brown, Bob', 'Smith, Jane'}

>>> gym_members.intersection({"Lytle, Josh", "Henriksen, Ian", "Webb, Jared"})
{'Lytle, Josh'}
>>> gym_members.difference({"Brown, Bob", "Sharp, Sarah"})
{'Lytle, Josh', 'Smith, Jane'}
```

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)
{'math': 2061, 'visual arts': 7321, 'science': 6284}

# Display the keys and values.
>>> my_dictionary.keys()
dict_keys(['math', 'visual arts', 'science'])
>>> my_dictionary.values()
dict_values([2061, 7321, 6284])
```

As far as data access goes, lists are like dictionaries whose keys are the integers $0, 1, \ldots, n-1$, where n is the number of items in the list. The keys of a dictionary need not be integers, but they must be *immutable*, which means that they must be objects that cannot be modified after creation. We will discuss mutability more thoroughly in the Standard Library 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 for converting between 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.
>>> set([1, 2, 3, 4, 4])
{1, 2, 3, 4}
>>> list({'a', 'a', 'b', 'b', 'c'})
['a', 'c', 'b']

# Cast other objects as strings.
>>> str(['a', str(1), 'b', float(2)])
"['a', '1', 'b', 2.0]"
>>> str(list(set([complex(float(3))])))
'[(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 6. Write a function called `pig_latin()`. Accept a string 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, end=' ')            # Print a space instead of a newline.
...     i += 1                       # Shortcut syntax for 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. `break` manually exits the loop, regardless of which iteration the loop is on or if the termination condition is met.

2. `continue` 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, end=' ')
...     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, end=' ')
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.

```
>>> for word in ["It", "definitely", "looks", "pretty", "bad", "today"]:
...     if word == "definitely":
...         continue
...     elif word == "bad":
...         break
...     print(word, end=' ')
...
It looks pretty
```

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 sequence of integers, following slicing syntax. If only one argument is specified, it produces a sequence of integers from 0 up to (but not including) the argument, incrementing by one. This function is used **very** often.
2. `zip()`: Joins multiple sequences in parallel 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. `reversed()`: Reverses the order of the iteration.
5. `sorted()`: Returns a new list of sorted items that can then be used for iteration.

Each of these functions except for `sorted()` returns an *iterator*, an object that is built specifically for looping but not for creating actual lists. To put the items of the sequence in a collection, use `list()`, `set()`, or `tuple()`.

```
# Strings and lists are both iterables.
>>> vowels = "aeiou"
>>> colors = ["red", "yellow", "white", "blue", "purple"]

# Iterate by index.
>>> for i in range(5):
...     print(i, vowels[i], colors[i])
...
0 a red
1 e yellow
2 i white
3 o blue
4 u purple

# Iterate through both sequences at once.
>>> for letter, word in zip(vowels, colors):
...     print(letter, word)
...
a red
e yellow
i white
o blue
u purple

# Get the index and the item simultaneously.
>>> for i, color in enumerate(colors):  #
...     print(i, color)
...
0 red
1 yellow
2 white
3 blue
4 purple
```

```
# Iterate through the list in sorted (alphabetical) order.
>>> for item in sorted(colors):
...     print(item, end=' ')
...
blue purple red white yellow

# Iterate through the list backward.
>>> for item in reversed(colors):
...     print(item, end=' ')
...
purple blue white yellow red

# range() arguments follow slicing syntax.
>>> list(range(10))                 # Integers from 0 to 10, exclusive.
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(range(4, 8))               # Integers from 4 to 8, exclusive.
[4, 5, 6, 7]

>>> set(range(2, 20, 3))            # Every third integer from 2 to 20.
{2, 5, 8, 11, 14, 17}
```

Problem 7. This problem originates from `https://projecteuler.net`, an excellent resource for math-related coding problems.

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.

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}
{'bright blue', 'bright red', 'bright yellow'}
```

Problem 8. The alternating harmonic series is defined as follows.

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

Write a function called `alt_harmonic()` that accepts an integer n. Use a list comprehension to quickly compute and sum the first n terms of this series (be careful not to sum only $n-1$ terms). 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.)

Additional Material

Further Reading

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.

- The official Python tutorial: `https://docs.python.org/3/tutorial/introduction.html` (especially chapters 3, 4, and 5).
- Section 1.2 of the SciPy lecture notes: `http://scipy-lectures.github.io/`.
- PEP8 - Python style guide: `http://www.python.org/dev/peps/pep-0008/`.

Advanced List Comprehension

As shown in the previous lab, list comprehensions can be used to create lists in a single line of code. There are several ways to supercharge list comprehensions to make them even more powerful. It all depends on what you would like to control within the list. We can add conditionals (i.e `if-else` statements) to the list comprehension to filter out or change certain elements. Moreover, nested looping is applicable for all list comprehensions. You can combine all of the following methods to create a list comprehension that fits your needs. Note that an iterable is an object that is countable and is meant to be traversed through (e.g. lists, tuples, strings, etc.).

- Nested Looping: Just like a nested `for` loop, you can nest the loops within the syntax of your comprehension. This is useful when you want to iterate over multiple lists at the same time. The syntax for this follows a similar structure to a normal nested `for` loop where the outer loop of the iterable goes before all loops it encompasses.
- Filtering the elements of the iterable: If you want to be able to only look at certain elements of the iterable, you can add an `if` statement following the name of the iterable. Note that this typically only works for simple conditionals. If you require a more complex conditional, you may want to create a function for the conditional and call the function in the list comprehension.
- Changing or filtering the elements going into the list: If your goal is to change or filter the actual elements that will go into the list you are making, you can add an `if` conditional following the expression dictating what the value of the element will be and immediately before the start of the `for` statement. All other conditionals that function like the `elif` statement come after the first `if` statement. However, even if you only have one `if` statement, Python syntax requires that you have an `else` statement at the end of all your conditionals, but before the start of the `for` loop, so the list has a default value it can give the element. Note we say "function like the `elif` statement" because list comprehension does not support the `elif` statement. To make conditionals that function like the `elif` statement, you must first state the `else` statement followed by the expression or value that the element will take, and then by the `if` statement and the condition the element must meet. That is, we use [... `else` *expression* `if` *condition* ...] for however many `elif` statements you need after the first `if` statement and always ensuring we end with the default `else` statement.

```
# Nested Looping
>>> [letter for word in ["Hello", "There"] for letter in word]
['H', 'e', 'l', 'l', 'o', 'T', 'h', 'e', 'r', 'e']
```

```
# Filtering the elements of the iterable
# Make a list of only even numbers from 0 to 9 (range is the iterable)
>>> [i for i in range(10) if i % 2 == 0]
[0, 2, 4, 6, 8]

# Changing or filtering the elements going into the list
# Make a list of the modulus of 3 of all numbers from 0 to 9
# using 'z' for 0, 'o' for 1, and 't' for 2. Note 'o' is
# the elif and 't' is the default if no other condition is met
>>> ['z' if i%3==0 else 'o' if i%3==1 else 't' for i in range(10)]
['z', 'o', 't', 'z', 'o', 't', 'z', 'o', 't', 'z']

# Combination of all three (notice the difficulty in code readability)
>>> iterable = ["General", "Kenobi"]
>>> [letter if letter != 'K' else 0 for word in iterable for letter in word if ↩
    letter not in "aeiou"]
['G', 'n', 'r', 'l', 0, 'n', 'b']
```

List comprehensions are actually faster than normal `for` loops. This is because when you append or create lists, Python has to perform a few operations to check that the given variable is a list and then append the new element to the list. This is not the case with list comprehensions. You can view this video for more information on list comprehensions. While list comprehensions are faster, they are not always the best option. They can be hard to read and understand, especially when they use various conditionals and nesting, and cannot give you all the control you may need when doing a normal for loop. So be sure to use them wisely and not overuse or overcomplicate them.

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
```

See `https://docs.python.org/3/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 and returns a new function that pre-processes the inputs or post-processes the outputs of the original function.

```
>>> 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)) # Process 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 accept 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 // c
```

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

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

Now calling `combine()` actually calls `wrapper()`, which then calls the original `combine()`.

```
>>> combine(3, 4, 6)
output type: <class 'int'>
2
>>> combine(3.0, 4, 6)
output type: <class 'float'>
2.0
```

Function decorators can also be customized with 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):
...     """Decorator for calling a function several times."""
...     def decorator(func):
...         def wrapper(*args, **kwargs):
...             for _ in range(times):
...                 output = func(*args, **kwargs)
...             return output
```

```
...         return wrapper
...     return decorator
...
>>> @repeat(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

Python has 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, *[, default=obj, key=func]) -> value
min(arg1, arg2, *args, *[, key=func]) -> value

With a single iterable argument, return its smallest item. The
default keyword-only argument specifies an object to return if
the provided iterable is empty.
With two or more arguments, return the smallest argument.
Type:      builtin_function_or_method

In [2]: len?
Signature: len(obj, /)
Docstring: Return the number of items in a container.
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: 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 particularly useful for formatting data to be printed.
>>> round(3.14159265358979323, 2)
3.14
```

See `https://docs.python.org/3/library/functions.html` for more detailed documentation on all of Python's built-in functions.

Problem 1. Write a function that accepts a list L and returns the minimum, maximum, and average of the entries of L in that order as multiple values (separated by a comma). Can you implement this function in one return statement?

Namespaces

Whenever a Python object—a number, data structure, function, or other entity—is created, it is stored somewhere in computer memory. A *name* (or variable) is a reference to a Python object, and 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_students = 4

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

# Python statements defining a function also 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 as the original name.

```
>>> beatles = ["John", "Paul", "George", "Ringo"]
>>> band_members = beatles             # Assign a new name to the list.
>>> 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 name from the namespace, use the `del` keyword (**with caution!**).

```
# Add 'stem' to the namespace.
>>> stem = ["Science", "Technology", "Engineering", "Mathematics"]
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'stem']

# Remove 'stem' from the namespace.
>>> del stem
>>> "stem" in dir()
False
>>> print(stem)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'stem' is not defined
```

NOTE

Many programming languages distinguish between *variables* and *pointers*. A pointer refers to a variable by storing the address in memory where the corresponding object is stored. Python names are essentially pointers, and traditional pointer operations and cleanup are done automatically. For example, Python automatically deletes objects in memory that have no names assigned to them (no pointers referring to them). This feature is called *garbage collection*.

Mutability

Every Python object type falls into one of two categories: a *mutable* object, which may be altered at any time, or an *immutable* object, which cannot be altered once created. Attempting to change an immutable object creates a new object in memory. If two names refer to the same mutable object, any changes to the object are reflected in both names since they still both refer to that same object. On the other hand, if two names refer to the same immutable object and one of the values is "changed," then one name will refer to the original object, and the other will refer to a new object in memory.

ACHTUNG!

Failing to correctly copy mutable objects can cause subtle problems. For example, consider a dictionary that maps items to their base prices. To make a similar dictionary that accounts for a small sales tax, we might try to make a copy by assigning a new name to the first dictionary.

```
>>> 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 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, since they are distinct objects in memory. To fix the above code, 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. Determine which Python object types are mutable and which are immutable by repeating the following experiment for an `int`, `str`, `list`, `tuple`, and `set`.

1. Create an object of the given type and assign a name to it.
2. Assign a new name to the first name.
3. Alter the object via only one of the names (for tuples, use `my_tuple += (1,)`).
4. Check to see if the two names are equal. If they are, then changing one name also changes the other. Thus, both names refer to the same object and the object type is mutable. Otherwise, the names refer to different objects—meaning a new object was created in step 2—and therefore the object type is immutable.

For example, the following experiment shows that `dict` is a mutable type.

```
>>> 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                                   # Both names changed!
```

Print a statement of your conclusions that clearly indicates which object types are mutable and which are immutable.

Achtung!

Mutable objects cannot be put into Python sets or used as keys in Python dictionaries. However, the values of a dictionary may be mutable or immutable.

```
>>> a_dict = {"key": "value"}              # Dictionaries are mutable.
>>> broken = {1, 2, 3, a_dict, a_dict}     # Try putting a dict in a set.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'

>>> okay = {1: 2, "3": a_dict}             # Try using a dict as a value.
```

Modules

A *module* is a Python file containing 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 by adding their names to the current namespace.

All import statements should occur at the top of the file, below the header but before any other code. There are several ways to use `import`:

1. `import <module>` makes the specified module available under the alias of its own name.

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

2. `import <module> as <name>` creates an alias for an imported module. The alias is added to the current namespace, but the module name itself is not.

```
>>> import numpy as np          # The name 'np' gives access to the numpy
>>> np.sqrt(2)                  # module, but the name 'numpy' does not.
1.4142135623730951
>>> numpy.sqrt(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'numpy' 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 random import randint  # The name 'randint' gives access to the
>>> r = randint(0, 10000)       # randint() function, but the rest of
>>> random.seed(r)              # the random module is unavailable.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'random' is not defined
```

In each case, the final 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`.

```
# example1.py

data = list(range(4))
```

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

```
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: [0, 1, 2, 3]
This file was executed from the command line or an interpreter.
```

Executing the file with IPython's special `%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: [0, 1, 2, 3]
This file was executed from the command line or an interpreter.

In [2]: display()
Data: [0, 1, 2, 3]
```

Importing the file also executes each line,[2] 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 directly 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: [0, 1, 2, 3]
```

[2] Try importing the `this` or `antigravity` modules. Importing these modules actually executes some code.

Problem 3. Create a module called `calculator.py`. Write a function `sum()` that returns the sum of two arguments and a function `product()` that returns the product of two arguments. Also use `import` to add the `sqrt()` function from the `math` module to the namespace. When this file is either run or imported, nothing should be executed.

In your solutions file, import your new custom module. Write a function that accepts two numbers representing the lengths of the sides of a right triangle. Using only the functions from `calculator.py`, calculate and return the length of the hypotenuse of the triangle.

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. 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).

The 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, and the complete list is at `https://docs.python.org/3/library/`.

Module	Description
`cmath`	Mathematical functions for complex numbers.
`itertools`	Tools for iterating through sequences in useful ways.
`math`	Standard mathematical functions and constants.
`random`	Random variable generators.
`string`	Common string literals.
`sys`	Tools for interacting with the interpreter.
`time`	Time value generation and manipulation.

Use IPython's object introspection to quickly learn about how to use the various modules and functions in the standard library. 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' (built-in)>
Docstring:
This module 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'.
    acos()      cos()       factorial() isclose()   log2()      tan()
    acosh()     cosh()      floor()     isfinite()  modf()      tanh()
    asin()      degrees()   fmod()      isinf()     nan         tau
    asinh()     e           frexp()     isnan()     pi          trunc()
    atan()      erf()       fsum()      ldexp()     pow()
    atan2()     erfc()      gamma()     lgamma()    radians()
    atanh()     exp()       gcd()       log()       sin()
    ceil()      expm1()     hypot()     log10()     sinh()
```

```
    copysign()  fabs()      inf         log1p()     sqrt()

In [3]: math.sqrt?
Signature: math.sqrt(x, /)
Docstring: Return the square root of x.
Type:      builtin_function_or_method
```

The Itertools Module

The `itertools` module makes it easy to iterate over one or more collections in specialized ways.

Function	Description
`chain()`	Iterate over several iterables in sequence.
`cycle()`	Iterate over an iterable repeatedly.
`combinations()`	Return successive combinations of elements in an iterable.
`permutations()`	Return successive permutations of elements in an iterable.
`product()`	Iterate over the Cartesian product of several iterables.

```
>>> from itertools import chain, cycle              # Import multiple names.

>>> list(chain("abc", ['d', 'e'], ('f', 'g')))     # Join several
['a', 'b', 'c', 'd', 'e', 'f', 'g']                 # sequences together.

>>> for i,number in enumerate(cycle(range(4))):    # Iterate over a single
...     if i > 10:                                 # sequence over and over.
...         break
...     print(number, end=' ')
...
0 1 2 3 0 1 2 3 0 1 2
```

A *k-combination* is a set of k elements from a collection where the ordering is unimportant. Thus the combination (a, b) and (b, a) are equivalent because they contain the same elements. One the other hand, a *k-permutation* is a sequence of k elements from a collection where the ordering matters. Even though (a, b) and (b, a) contain the same elements, they are counted as different permutations.

```
>>> from itertools import combinations, permutations

# Get all combinations of length 2 from the iterable "ABC".
>>> list(combinations("ABC", 2))
[('A', 'B'), ('A', 'C'), ('B', 'C')]

# Get all permutations of length 2 from "ABC". Note that order matters here.
>>> list(permutations("ABC", 2))
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]
```

Problem 4. The *power set* of a set A, denoted $\mathcal{P}(A)$ or 2^A, is the set of all subsets of A, including the empty set $\emptyset$ and A itself. For example, the power set of the set $A = \{a, b, c\}$ is $2^A = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$.

Write a function that accepts an iterable A. Use an `itertools` function to compute and return the power set of A as a list of sets (why couldn't it be a set of sets in Python?). The empty set should be returned as `set()`.

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 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.

Some of the most common `random` utilities involve picking random elements from iterables.

```
>>> import random

>>> numbers = list(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 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
```

The Time Module

The `time` module in the standard library include functions for dealing with time. In particular, the `time()` function measures the number of seconds from a fixed starting point, called "the Epoch" (January 1, 1970 for Unix machines).

```
>>> import time
>>> time.time()
1495243696.645818
```

The `time()` function is useful for measuring how long it takes for code to run: record the time just before and just after the code in question, then subtract the first measurement from the second to get the number of seconds that have passed.

```
>>> def time_for_loop(iters):
...     """Time how long it takes to iterate 'iters' times."""
...     start = time.time()         # Clock the starting time.
...     for _ in range(int(iters)):
...         pass
...     end = time.time()           # Clock the ending time.
...     return end - start          # Report the difference.
...
>>> time_for_loop(1e5)              # 1e5 = 100000.
0.005570173263549805
>>> time_for_loop(1e7)              # 1e7 = 10000000.
0.26819777488708496
```

The Sys Module

The `sys` (system) module includes methods for interacting with the Python interpreter. For our purposes, we care about the instance that you call a .py file in the command line using 'python' followed by the name of a .py file or in your ipython terminal using '%run' followed by the name of a .py file. Remember, to run a .py file in the command line, use cd to navigate to the folder containing the file before trying to run it. One command we will use is `sys.argv`; which returns a list containing the .py file name and all arguments that were passed to the interpreter. This way we can interact with these initial arguments and execute certain parts of our code if the arguments satisfy certain conditions. This will be implemented in problem 5. Note, command line arguments are passed in after the name of the .py file and are separated by spaces like so:

```
# the 'python' command followed by a .py file followed by any arguments
$ python yourProgram.py argument1 argument2 argument3

# the equivalent form in an ipython terminal
In[1]: %run yourProgram.py argument1 argument2 argument3
```

Now we can execute code based on the command line arguments by inserting the use of the `sys` module in our .py file:

```
# example3.py
"""If there are two command line arguments  after the .py file  name, print a descriptive statment."""
import sys

if len(sys.argv) == 3:  # if there's two arguments and the file name
    print("the first command line argument is " + sys.argv[1])
    print("the second command line argument is " + sys.argv[2])
else:
    print("um... I need two after the .py file. This is not two:")
    print(sys.argv)
```

Now provide command line arguments for the program to process.

```
# No extra command line arguments.
$ python example3.py
Output:
um... I need two after the .py file. This is not two:
['example3.py']

# With two arguments the if statement executes.
$ python example3.py crunchy juicy
Output:
the first command line argument is crunchy
the second command line argument is juicy
```

Note that the first command line argument is always the filename, so that is always the first element of the sys.argv list. This is why we have the if statement execute if the length of the list is 3 even though we want it to execute when there are 2 command line arguments. Also, `sys.argv` is always a list of strings. If a number is provided on the command line, it is converted to a string when it is stored in `sys.argv`. In IPython, command line arguments are specified after the `%run` command.

```
In [1]: %run example3.py 42 too many
Output:
um... I need two after the .py file. This is not two:
['example3.py', '42', 'too', 'many']
```

Another way to get input from the program user is to prompt the user for text. The built-in function `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.

```
>>> x = input("Enter a value for x: ")
Enter a value for x: 20             # Type '20' and press 'enter.'

>>> x
'20'                                # Note that x contains a string.
```

```
>> y = int(input("Enter an integer for y: "))
Enter an integer for y: 16          # Type '16' and press 'enter.'

>>> y
16                                  # Note that y contains an integer.
```

Problem 5. *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 rolls two dice, then chooses a set of integers from the remaining numbers that sum up to the sum of the dice roll. These numbers are removed, and the dice are then rolled again. If the sum of the remaining numbers is 6 or less, then only one die is rolled. 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. For a demonstration, see `https://www.youtube.com/watch?v=mwURQC7mjDI`.

Modify your solutions file so that when the file is run with the correct command line arguments (but **not** when it is imported), the user plays a game of shut the box. The provided module `box.py` contains two functions that will be useful in your implementation of the game. You do not need to understand exactly how the functions work, but you do need to be able to import and use them correctly. Their functionality is outlined at the beginning of each function declaration. Your game should match the following specifications:

- Require three total command line arguments: the file name (included by default), the player's name, and a time limit in seconds. If there are not exactly three command line arguments, do not start the game.
- Track the player's remaining numbers, starting with 1 through 9.
- Use the `random` module to simulate rolling two six-sided dice. However, if the sum of the player's remaining numbers is 6 or less, roll only one die.
- The player wins if they have no numbers left, and they lose if they are out of time or if they cannot choose numbers to match the dice roll.
- If the game is not over, print the player's remaining numbers, the sum of the dice roll, and the number of seconds remaining. Prompt the user for numbers to eliminate. 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.
  (Hint: use `round()` to format the number of seconds remaining nicely.)
- When the game is over, display the player's name, their score, and the total number of seconds since the beginning of the game. Congratulate or mock the player appropriately.

(Hint: **Before you start coding**, write an outline for the entire program, adding one feature at a time. Only start implementing the game after you are completely finished designing it.)

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

```
$ python standard_library.py LuckyDuke 60

Numbers left: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Roll: 12
Seconds left: 60.0
Numbers to eliminate: 3 9

Numbers left: [1, 2, 4, 5, 6, 7, 8]
Roll: 9
Seconds left: 53.51
Numbers to eliminate: 8 1

Numbers left: [2, 4, 5, 6, 7]
Roll: 7
Seconds left: 51.39
Numbers to eliminate: 7

Numbers left: [2, 4, 5, 6]
Roll: 2
Seconds left: 48.24
Numbers to eliminate: 2

Numbers left: [4, 5, 6]
Roll: 11
Seconds left: 45.16
Numbers to eliminate: 5 6

Numbers left: [4]
Roll: 4
Seconds left: 42.76
Numbers to eliminate: 4

Score for player LuckyDuke: 0 points
Time played: 21.82 seconds
Congratulations!! You shut the box!
```

The next two examples show different ways that a player could lose (which they usually do), as well as examples of invalid user input. Use the `box` module's `parse_input()` to detect invalid input.

```
$ python standard_library.py ShakySteve 10

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

```
Seconds left: 10.0
Numbers to eliminate: Seven             # Must enter a number.
Invalid input

Seconds left: 7.64
Numbers to eliminate: 1, 2, 4           # Do not use commas.
Invalid input

Seconds left: 4.55
Numbers to eliminate: 1 2 3             # Numbers don't sum to the roll.
Invalid input

Seconds left: 2.4
Numbers to eliminate: 1 2 4

Numbers left: [3, 5, 6, 7, 8, 9]
Roll: 8
Seconds left: 0.31
Numbers to eliminate: 8
Game over!                              # Time is up!

Score for player ShakySteve: 30 points
Time played: 11.77 seconds
Better luck next time >:)
```

```
$ python standard_library.py SnakeEyesTom 10000

Numbers left: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Roll: 2
Seconds left: 10000.0
Numbers to eliminate: 2

Numbers left: [1, 3, 4, 5, 6, 7, 8, 9]
Roll: 2
Game over!                              # Numbers cannot match roll.

Score for player SnakeEyesTom: 43 points
Time played: 1.53 seconds
Better luck next time >:)
```

Additional Material

More Built-in Functions

The following built-in functions are worth knowing, especially for working with iterables and writing very readable conditional statements.

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.
`map()`	Apply a function to every item of the input iterable and return an iterable of the results.

```
>>> from random import randint
# Get 5 random numbers between 1 and 10, inclusive.
>>> numbers = [randint(1, 10) for _ in range(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 5. 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. In addition, each player should have their own time limit that only ticks down during their turn. If time runs out on your turn, you lose no matter what.

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, the associated directories must be organized into a Python *packages*.

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` in order to be executable. Executing the package will run `__init__.py` and then `__main__.py`, but importing the package will only run `__init__.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>`. 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`).

Finally, to execute a package, run Python from the shell with the flag `-m` (for "module-name") and exclude the extension `.py`.

```
$ python -m package_name
```

See `https://docs.python.org/3/tutorial/modules.html#packages` for examples and more details.

3 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 (a list of lists). However, even basic linear algebra operations like matrix multiplication are cumbersome to implement and slow to execute when data is stored this way. The *NumPy* module[1] [Oli06, ADH+01, Oli07] 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 string representation has no commas or an array() label.
>>> print(np.array([1, 3, 5, 7, 9]))
[1 3 5 7 9]
```

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 (like a list of lists), a 3-D array is a 1-D array of 2-D arrays (a list of lists of lists), and, more generally, an n-dimensional array is a 1-D array of $(n-1)$-dimensional arrays (a list of lists of lists of lists...). 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, with the axes separated by commas.

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

```
# Create a 2-D array by passing a list of lists into array().
>>> A = np.array([[1, 2, 3], [4, 5, 6]])
>>> 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. There are two main ways to perform matrix multiplication in NumPy: with NumPy's `dot()` function (`np.dot(A, B)`), or with the `@` operator (`A @ B`). 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.

For examples of array initialization and matrix multiplication, use object introspection in IPython 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'
```

ACHTUNG!

The `@` operator was not introduced until Python 3.5. It triggers the `__matmul__()` magic method,[a] which for the `ndarray` is essentially a wrapper around `np.dot()`. If you are using a previous version of Python, always use `np.dot()` to perform basic matrix multiplication.

[a] See the lab on Object Oriented Programming for an overview of magic methods.

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).

```
# Addition concatenates lists together.
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]

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

NumPy arrays act like mathematical vectors and matrices: `+` and `*` perform component-wise addition or multiplication.

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

# Addition or multiplication by a scalar acts on each element of the array.
>>> x + 10                          # Add 10 to each entry of x.
array([11, 12, 13])
>>> x * 4                           # Multiply each entry of x by 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, not the component-wise square). 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' is a 2-D array with 2 rows, 3 columns, and 6 entries.
>>> print(A.ndim, A.shape, A.size)
2 (2, 3) 6
```

Note that `ndim` is the number of entries in `shape`, and that 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()`	Array of sequential integers (like `list(range())`).
`eye()`	2-D array with ones on the diagonal and zeros elsewhere.
`ones()`	Array of given shape and type, filled with ones.
`ones_like()`	Array of ones with the same shape and type as a given array.
`zeros()`	Array of given shape and type, filled with zeros.
`zeros_like()`	Array of zeros with the same shape and type as a given array.
`full()`	Array of given shape and type, filled with a specified value.
`full_like()`	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`. To learn more about NumPy datatypes, see `https://numpy.org/devdocs/user/basics.types.html`.

```
# 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.int64)    # 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.]]

# 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.]])
```

Unlike native Python data structures, **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)        # Equivalent to x = np.float64(x).
>>> print(x)
[ 0.  1.  2.  3.  4.]               # Floats are displayed with periods.
>>> x.dtype
dtype('float64')
```

The following functions are for dealing with the diagonal, upper, or lower portion of an array.

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]])

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

# Get the diagonal entries of 'A' as a 1-D array.
>>> np.diag(A)
array([1, 5, 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 using the functions presented in this section (not `np.array()`). 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 uses the slicing syntax `x[start:stop:step]`. If there is no colon, a single entry of that dimension is accessed. With a colon, a range of values is accessed. For multi-dimensional arrays, use a comma to separate slicing syntax for each axis.

```
# 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 elements of the array with slicing syntax.
>>> x[3]                            # The element at index 3.
3
>>> x[:3]                           # Everything up to index 3 (exclusive).
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 1, column 2.
7
>>> A[:, 2:]                        # All of the rows, from column 2 on.
array([[2, 3, 4],
       [7, 8, 9]])
```

> **Note**
>
> Indexing and slicing operations return a *view* of the array. Changing a view of an array also changes the original array. In other words, **arrays are mutable**. To create a copy of an array, use `np.copy()` or the array's `copy()` method. 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.

Fancy Indexing

So-called *fancy indexing* is a second way to access or change the elements of an array. Instead of using slicing syntax, provide either an array of indices or an array of boolean values (called a *mask*) to extract specific elements.

```
>>> x = np.arange(0, 50, 10)        # The integers from 0 to 50 by tens.
>>> 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])     # Get the 3rd, 1st, and 4th elements.
>>> 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]                         # Get the 0th and 3rd entries.
array([0, 30])
```

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

```
>>> y = np.arange(10, 20, 2)        # Every other integer from 10 to 20.
>>> y
array([10, 12, 14, 16, 18])

# Extract the values of 'y' larger than 15.
>>> mask = y > 15                   # Same as np.array([i > 15 for i in y]).
>>> mask
array([False, False, False, True, True], dtype=bool)
>>> y[mask]                         # Same as y[y > 15]
array([16, 18])

# Change the values of 'y' that are larger than 15 to 100.
>>> y[mask] = 100
>>> print(y)
[10 12 14 100 100]
```

While indexing and slicing always return a view, 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 resulting array.

Array Manipulation

Shaping

An array's `shape` attribute describes its dimensions. Use `np.reshape()` or the array's `reshape()` method to give an array a new shape. 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. Using a `-1` in the new shape tuple makes the specified dimension as long as necessary.

```
>>> A = np.arange(12)              # The integers from 0 to 12 (exclusive).
>>> 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 1-D array and `np.transpose()` or the `T` attribute to transpose a 2-D array in the matrix sense.

```
>>> 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. For example, by default an array will 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 to purposefully work well 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]                     # All of the rows, column 1.
>>> x
array([1, 6])                       # Not array([[1],
>>> x.shape                         #             [6]])
(2,)
>>> x.ndim
1
```

However, it is occasionally 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((-1, 1))               # Or x[:,np.newaxis] or np.vstack(x).
array([[0],
       [1],
       [2]])
```

Do not force a 1-D vector to be a column vector unless necessary.

Stacking

NumPy has functions for *stacking* two or more arrays with similar dimensions into a single block matrix. Each of these methods takes in a single tuple of arrays to be stacked in sequence.

Function	Description
`concatenate()`	Join a sequence of arrays along an existing axis
`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(6).reshape((2, 3))
>>> B = np.zeros((4, 3))

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

>>> A = A.T
>>> B = np.ones((3, 4))

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

# column_stack() stacks arrays horizontally, including 1-D arrays.
>>> np.column_stack((A, np.zeros(3), np.ones(3), np.full(3, 2)))
array([[ 0.,  3.,  0.,  1.,  2.],
       [ 1.,  4.,  0.,  1.,  2.],
       [ 2.,  5.,  0.,  1.,  2.]])
```

See `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 3×3 identity matrix and each $\mathbf{0}$ is a matrix of all zeros of appropriate size.

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 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, 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. Universal functions are significantly more efficient than using a loop to operate individually on each element of an 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.

```
>>> x = np.arange(-2, 3)
>>> print(x, np.abs(x))             # Like np.array([abs(i) for i in x]).
[-2 -1  0  1  2] [2 1 0 1 2]

>>> np.sin(x)                       # Like np.array([math.sin(i) for i in x]).
array([-0.90929743, -0.84147098,  0.        ,  0.84147098,  0.90929743])
```

See `http://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs` for a more comprehensive list of universal functions.

Achtung!

The `math` module has many useful functions for numerical computations. However, most of these functions can only act on single numbers, not on arrays. NumPy functions can act on either scalars or entire arrays, but `math` functions tend to be a little faster for acting on scalars.

```
>>> import math

# Math and NumPy functions can both operate on scalars.
>>> print(math.exp(3), np.exp(3))
20.085536923187668 20.0855369232

# However, math functions cannot operate on arrays.
>>> x = np.arange(-2, 3)
>>> np.tan(x)
array([ 2.18503986, -1.55740772,  0.        ,  1.55740772, -2.18503986])
>>> math.tan(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only length-1 arrays can be converted to Python scalars
```

Always use universal NumPy functions, not the `math` module, when working with arrays.

Other Array Methods

The `np.ndarray` class itself has many useful methods for numerical computations.

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.
`clip()`	restrict values in an array to fit within a given range
`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 an equivalent NumPy function. For example, `A.max()` and `np.max(A)` operate the same way. 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 can 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. Refer to the NumPy Visual Guide in the appendix for more visual examples.

```
>>> 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 range(3)↩
    ])
array([0, 1, 2])

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

See `http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html` for a more comprehensive list of array methods.

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

Write a function that accepts a matrix (as a 2-D NumPy array). Divide each row of the matrix by the row sum and return the new row-stochastic matrix. Use array broadcasting and the `axis` argument instead of a loop.

[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 range(20):
...     for j in range(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.)

Achtung!

All of the examples in this lab use NumPy arrays, objects of type `np.ndarray`. NumPy also has a "matrix" data structure called `np.matrix` that was built specifically for MATLAB users who are transitioning to Python and NumPy. It behaves slightly differently than the regular array class, and can cause some unexpected and subtle problems.

For consistency (and your sanity), **never** use a NumPy matrix; **always** use NumPy arrays. If necessary, cast a matrix object as an array with `np.array()`.

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.
`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.

Note that many of these functions have counterparts in the standard library's `random` module. These NumPy functions, however, are much better suited for working with large collections of random samples.

```
# 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 for later use. 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.load("normal.npz")
>>> np.allclose(x, arrays["first"])
True
>>> np.allclose(y, arrays["second"])
True
```

4 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 write clean, efficient, well-designed programs. In this lab we learn how to define and use Python classes. In subsequent labs we will often create customized classes for use in algorithms.*

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:
    """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 list of contents.

        Parameters:
            name (str): the name of the backpack's owner.
        """
        self.name = name                # Initialize some attributes.
        self.contents = []
```

An *attribute* is a variable stored within an object. The `Backpack` class has two attributes: `name` and `contents`. In the body of the class definition, attributes are assigned and accessed via the identifier `self`. The identifier `self` refers to the object internally once it has been created. In the previous example, the line `self.name = name` stores the input argument `name` to the attribute `self.name`.

Instantiation

The `class` code block above only defines a blueprint for backpack objects. To create an actual backpack object, call the class name like a function. This triggers the constructor and returns a new *instance* of the class, an object whose type is the class.

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

# Access the object's attributes with a period and the attribute name.
>>> print(my_backpack.name, my_backpack.contents)
Fred []

# The object's attributes can be modified after instantiation.
>>> my_backpack.name = "George"
>>> print(my_backpack.name, my_backpack.contents)
George []
```

> **Note**
>
> Every object in Python has some built-in attributes. For example, modules have a `__name__` attribute that identifies the scope in which it is being executed. If the module is being run directly, and not imported, then `__name__` is set to `"__main__"`. Therefore, any commands under an `if __name__ == "__main__":` clause are ignored when the module is imported.

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*.

```
class Backpack:
    # ...
    def put(self, item):
        """Add an item to the backpack's list of contents."""
        self.contents.append(item)  # Use 'self.contents', not just 'contents'.

    def take(self, item):
        """Remove an 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 on an instantiation of the class.

```
# Add some items to the backpack object.
>>> my_backpack.put("notebook")         # my_backpack is passed implicitly to
>>> my_backpack.put("pencils")          # Backpack.put() as the first argument.
>>> my_backpack.contents
['notebook', 'pencils']

# Remove an item from the backpack.     # This is equivalent to
>>> my_backpack.take("pencils")         # Backpack.take(my_backpack, "pencils")
>>> my_backpack.contents
['notebook']
```

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

1. Modify the constructor so that it accepts three total arguments: `name`, `color`, and `max_size` (in that order). Make `max_size` a keyword argument that defaults to 5. Store each input as an attribute.
2. Modify the `put()` method to check that the backpack does not go over capacity. If there are already `max_size` items or more, print "No Room!" and do not add the item to the contents list.
3. Write 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`).
4. Documentation is especially important in classes so that the user knows what an object's attributes represent and how to use methods appropriately. Update (or write) the docstrings for the `__init__()`, `put()`, and `dump()` methods, as well as the actual class docstring (under `class` but before `__init__()`) to reflect the changes from parts 1-3 of this problem.

To ensure that your class works properly, write a test function outside of the `Backpack` class that instantiates and analyzes a `Backpack` object.

```
def test_backpack():
    testpack = Backpack("Barry", "black")       # Instantiate the object.
    if testpack.name != "Barry":                # Test an attribute.
        print("Backpack.name assigned incorrectly")
    for item in ["pencil", "pen", "paper", "computer"]:
        testpack.put(item)                      # Test a method.
    print("Contents:", testpack.contents)
    # ...
```

Inheritance

To create a new class that is similar to one that already exists, it is often better to *inherit* the methods and attributes from an existing class rather than create a new class from scratch. 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*. To define a subclass, add the name of the superclass as an argument at the end of the `class` declaration.

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 inside.
        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.

        Parameters:
            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 inside.
        """
        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 from the superclass need to be changed for the subclass, they can be overridden by defining them again in the subclass. New methods can be included normally.

```
class Knapsack(Backpack):
    # ...
    def put(self, item):            # Override the put() method.
        """If the knapsack is untied, use the Backpack.put() method."""
        if self.closed:
```

```
            print("I'm closed!")
        else:                       # Use Backpack's original put().
            Backpack.put(self, item)

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

    def weight(self):               # Define a new method just for knapsacks.
        """Calculate the weight of the knapsack by counting the length of the
        string representations of each item in the contents list.
        """
        return sum(len(str(item)) for item in self.contents)
```

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

The built-in function `issubclass()` shows whether or not one class is derived from another. Similarly, `isinstance()` indicates whether or not an object belongs to a specified class hierarchy. Finally, `hasattr()` shows whether or not a class or object **has** a specified attribute or method.

```
>>> from object_oriented import Knapsack
>>> my_knapsack = Knapsack("Brady", "brown")

# A Knapsack is a Backpack, but a Backpack is not a Knapsack.
>>> print(issubclass(Knapsack, Backpack), issubclass(Backpack, Knapsack))
True False
>>> isinstance(my_knapsack, Knapsack) and isinstance(my_knapsack, 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 Knapsack class has a weight() method, but the Backpack class does not.
>>> print(hasattr(my_knapsack, 'weight'), hasattr(my_backpack, 'weight'))
True False

# The dump method is inherited from the Backpack class.
```

```
>>> my_knapsack.dump()
>>> my_knapsack.contents
[]
```

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

1. Override 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. Override the `dump()` method so that both the contents and the fuel tank are emptied.
4. Write clear, detailed docstrings for the class and each of its methods.

NOTE

All classes are subclasses of the built-in `object` class, even if no parent class is specified in the class definition. In fact, the syntax "`class ClassName(object):`" is not uncommon (or incorrect) for the class declaration, and is equivalent to the simpler "`class ClassName:`".

Magic Methods

A *magic method* is a special method used to make an object behave like a built-in data type. Magic methods begin and end with two underscores, like the constructor `__init__()`. Every Python object is automatically endowed with several magic methods, which can be revealed through IPython.

```
In [1]: %run object_oriented.py

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

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

In [3]: b.__        # Press 'tab' to see magic methods and hidden attributes.
          __add__()           __getattribute__    __new__()
          __class__           __gt__              __reduce__()
          __delattr__         __hash__            __reduce_ex__()
          __dict__            __init__()          __repr__
          __dir__()           __init_subclass__() __setattr__
```

```
__doc__             __le__              __sizeof__()
__eq__              __lt__()            __str__
__format__()        __module__          __subclasshook__()
__ge__              __ne__              __weakref__
```

> **Note**
>
> Many programming languages distinguish between *public* and *private* variables. In Python, all attributes are public, period. However, attributes that start with an underscore are hidden from the user, which is why magic methods do not show up at first in the preceding code box.

The more common magic methods define how an object behaves with respect to addition and other binary operations. For example, 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. To incorporate this idea, we implement the `__add__()` magic method.

```
class Backpack:
    # ...
    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")

# Add the backpacks together.
>>> pack1 + pack2                   # Equivalent to pack1.__add__(pack2).
3
```

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):
        """If 'self' has fewer contents than 'other', return True.
        Otherwise, return False.
        """
        return len(self.contents) < len(other.contents)
```

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, pack2 = Backpack("Maggy", "magenta"), Backpack("Yolanda", "yellow")
>>> pack1 < pack2                    # Equivalent to pack1.__lt__(pack2).
False

>>> pack2.put('pencils')
>>> pack1 < pack2
True
```

Comparison methods should return either `True` or `False`, while methods like `__add__()` might return a numerical value or another kind of object.

Method	Arithmetic Operator
`__add__()`	`+`
`__sub__()`	`-`
`__mul__()`	`*`
`__pow__()`	`**`
`__truediv__()`	`/`
`__floordiv__()`	`//`

Method	Comparison Operator
`__lt__()`	`<`
`__le__()`	`<=`
`__gt__()`	`>`
`__ge__()`	`>=`
`__eq__()`	`==`
`__ne__()`	`!=`

Table 4.1: Common magic methods for arithmetic and comparisons. What each of these operations do is up to the programmer and should be carefully documented. For more methods and details, see `https://docs.python.org/3/reference/datamodel.html#special-method-names`.

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 returns the string representation of an object. This method is invoked by `str()` and used by `print()`. Implement the `__str__()` method in the `Backpack` class so that printing a `Backpack` object yields the following output (that is, construct and return the following string).

   ```
   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 align output nicely.)

Achtung!

Magic methods for comparison are **not** automatically related. For example, even though the `Backpack` class implements the magic methods for `<` and `==`, two `Backpack` objects cannot respond to the `<=` operator unless `__le__()` is explicitly defined. The exception to this rule is the `!=` operator: as long as `__eq__()` is defined, `A!=B` is `False` if and only if `A==B` is `True`.

Problem 4. Write a `ComplexNumber` class from scratch.

1. Complex numbers are denoted $a+bi$ where $a, b \in \mathbb{R}$ and $i = \sqrt{-1}$. Write the constructor so it accepts two numbers. Store the first as `self.real` and the second as `self.imag`.
2. The *complex conjugate* of $a+bi$ is defined as $\overline{a+bi} = a-bi$. Write a `conjugate()` method that returns the object's complex conjugate as a new `ComplexNumber` object.
3. Add the following magic methods:
   (a) Implement `__str__()` so that $a+bi$ is printed out as `(a+bj)` for $b \geq 0$ and `(a-bj)` for $b < 0$.
   (b) The *magnitude* of $a+bi$ is $|a+bi| = \sqrt{a^2+b^2}$. 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.
   (c) Two `ComplexNumber` objects are equal if and only if they have the same real and imaginary parts. Implement `__eq__()` so that it compares the `ComplexNumber` object with another `ComplexNumber` object and returns a bool indicating whether they are equal.
   (d) Implement `__add__()`, `__sub__()`, `__mul__()`, and `__truediv__()` appropriately. Each of these should return a new `ComplexNumber` object.

Write a function to test your class by comparing it to Python's built-in `complex` type.

```
def test_ComplexNumber(a, b):
    py_cnum, my_cnum = complex(a, b), ComplexNumber(a, b)

    # Validate the constructor.
    if my_cnum.real != a or my_cnum.imag != b:
        print("__init__() set self.real and self.imag incorrectly")
```

```
    # Validate conjugate() by checking the new number's imag attribute.
    if py_cnum.conjugate().imag != my_cnum.conjugate().imag:
        print("conjugate() failed for", py_cnum)

    # Validate __str__().
    if str(py_cnum) != str(my_cnum):
        print("__str__() failed for", py_cnum)
    # ...
```

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 methods, and do not use `self`. Since the attribute is not tied to a specific instance of the class, it may be accessed or changed via the class name without even instantiating the class at all.

```
class Backpack:
    # ...
    brand = "Adidas"                # Backpack.brand is a static attribute.
```

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

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

Static Methods

Individual class methods can also be static. A static method cannot be dependent 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 static methods only have access to static attributes and other static methods. Include the tag `@staticmethod` above the function definition to designate a method as static.

```
class Backpack:
    # ...
    @staticmethod
    def origin():                   # Do not use 'self' as a parameter.
        print("Manufactured by " + Backpack.brand + ", inc.")
```

```
# Static methods can be called without instantiating the class.
>>> Backpack.origin()
Manufactured by Nike, inc.

# The method can also be accessed by individual 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 different ID.

More Magic Methods

Consider how the following methods might be implemented for the `Backpack` class. These methods are particularly important for custom data structure classes.

Method	Operation	Trigger Function
`__bool__()`	Truth value	`bool()`
`__len__()`	Object length or size	`len()`
`__repr__()`	Object representation	`repr()`
`__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/3/reference/datamodel.html#special-method-names` for more details and documentation on all magic methods.

Hashing

A *hash function* is a method that maps objects to identifiers which can be numeric or alphanumeric strings. These identifiers are known as *hash values*. The built-in `hash()` function calculates an object's hash value by calling its `__hash__()` magic method. While an object has one unique hash value, it is possible that two distinct objects may share the same hash value. This is called a *collision*. A good hash function is one that minimizes the probability of collisions occurring.

In Python, the built-in `set` and `dict` structures use hash values to store and retrieve objects in memory quickly. If an object is *unhashable*, it cannot be put in a set or be used as a key in a dictionary. See `https://docs.python.org/3/glossary.html#term-hashable` for details. 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. The following simple `__hash__()` method for the `Backpack` class conforms to this rule and returns an integer.

```
class Backpack:
    # ...
    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.

See `https://docs.python.org/3/reference/datamodel.html#object.__hash__` for more on hashing.

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 and 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

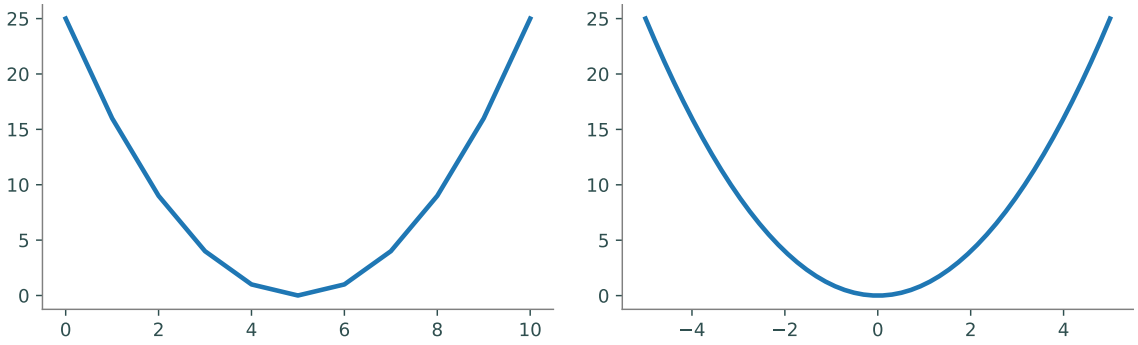
Raw numerical data is rarely helpful unless it can be visualized. The quickest way to visualize a simple 1-dimensional array is with a *line plot*. The following code creates an array of outputs of the function $f(x) = x^2$, then visualizes the array using the `matplotlib` module[1] [Hun07].

```
>>> 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, which in this case is 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. See `https://matplotlib.org/` for the complete Matplotlib documentation.



(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: Plots of $f(x) = x^2$ over the interval $[-5, 5]$.

Problem 1. NumPy's `random` module has tools for sampling from probability distributions. For instance, `np.random.normal()` draws samples from the normal (Gaussian) distribution. The `size` parameter specifies the shape of the resulting array.

```
>>> np.random.normal(size=(2, 3))    # Get a 2x3 array of samples.
array([[ 1.65896515, -0.43236783, -0.99390897],
       [-0.35753688, -0.76738306,  1.29683025]])
```

Write a function that accepts an integer n as input.

1. Use `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 calculate the variance).

Define another 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.

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, define an array `x` for the domain, then use it to calculate the image `y = f(x)`. The command `plt.plot(x,y)` plots `x` against `y` by drawing a line between the consecutive points `(x[i], y[i])`.

Another problem with Figure 5.1a is its poor resolution: the curve is visibly bumpy, especially near the bottom of the curve. NumPy's `linspace()` function makes it easy to get a higher-resolution domain. Recall that `np.arange()` returns an 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.

```
# Get 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 image of f(x) = x**2.
>>> plt.plot(x, y)
>>> plt.show()
```

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

Subsequent calls to `plt.plot()` modify the same figure until `plt.show()` is executed, which displays the current figure and resets the system. This behavior can be altered by specifying separate figures or axes, which we will discuss shortly.

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()
```

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

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.

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.

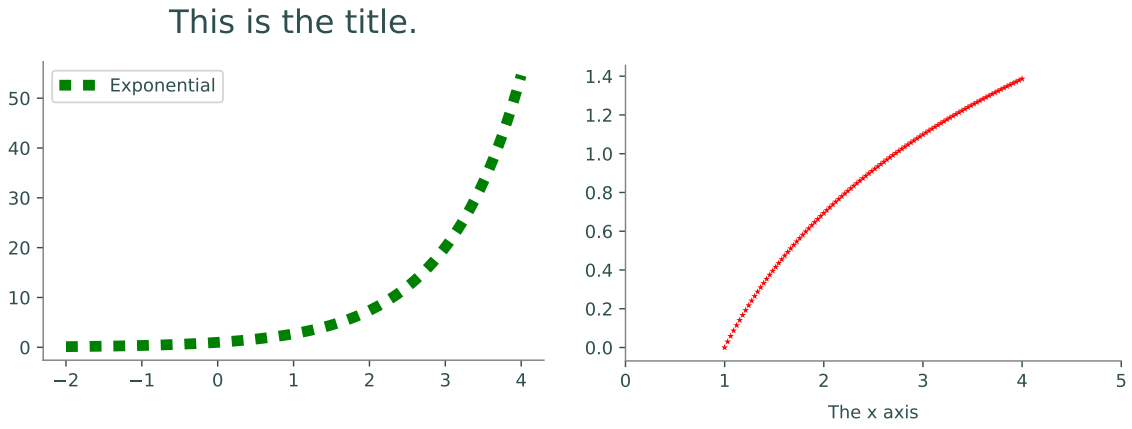
Key	Color
`'b'`	blue
`'g'`	green
`'r'`	red
`'c'`	cyan
`'k'`	black

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

Specify one or both of these string codes as the third 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()` / `ylim()`	Set the limits of the x- or y-axis
`xlabel()` / `ylabel()`	Add a label to the x- or y-axis

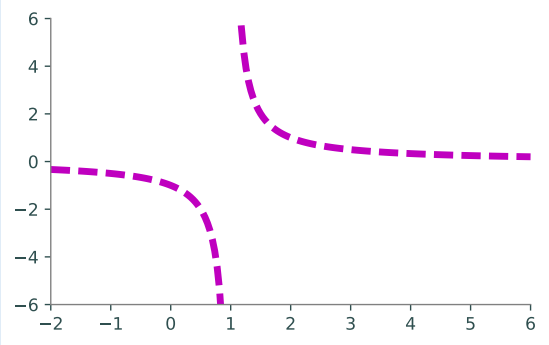
```
>>> x1 = np.linspace(-2, 4, 100)
>>> plt.plot(x1, np.exp(x1), 'g:', linewidth=6, label="Exponential")
>>> plt.title("This is the title.", fontsize=18)
>>> plt.legend(loc="upper left")    # plt.legend() uses the 'label' argument of
>>> plt.show()                      # plt.plot() to create a legend.

>>> 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()
```



Problem 3. Write a function to plot the curve $f(x) = \frac{1}{x-1}$ on the domain $[-2, 6]$.

1. Although $f(x)$ has a discontinuity at $x = 1$, a single call to `plt.plot()` in the usual way will make the curve look continuous. Split up the domain into $[-2, 1)$ and $(1, 6]$. Plot the two sides of the curve separately so that the graph looks discontinuous at $x = 1$.
2. Plot both curves with a dashed magenta line. Set the keyword argument `linewidth` (or `lw`) of `plt.plot()` to 4 to make the line a little thicker than the default setting.
3. Use `plt.xlim()` and `plt.ylim()` to change the range of the x-axis to $[-2, 6]$ and the range of the y-axis to $[-6, 6]$.

The plot should resemble the figure below.



Figures, Axes, and Subplots

The window that `plt.show()` reveals is called a *figure*, stored in Python as a `plt.Figure` object. A space on a figure where a plot is drawn is called an *axes*, a `plt.Axes` object. A figure can have multiple axes, and a single program may create several figures. There are several ways to create or grab figures and axes with `plt` functions.

Function	Description
`axes()`	Add an axes to the current figure
`figure()`	Create a new figure or grab an existing figure
`gca()`	Get the current axes
`gcf()`	Get the current figure
`subplot()`	Add a single subplot to the current figure
`subplots()`	Create a figure and add several subplots to it

Usually when a figure has multiple axes, they are organized into non-overlapping *subplots*. The command `plt.subplot(nrows, ncols, plot_number)` creates an axes in a subplot grid where `nrows` is the number of rows of subplots in the figure, `ncols` is the number of columns, and `plot_number` specifies which subplot to modify. If the inputs for `plt.subplot()` are all integers, the commas between the entries can be omitted. For example, `plt.subplot(3,2,2)` can be shortened to `plt.subplot(322)`.

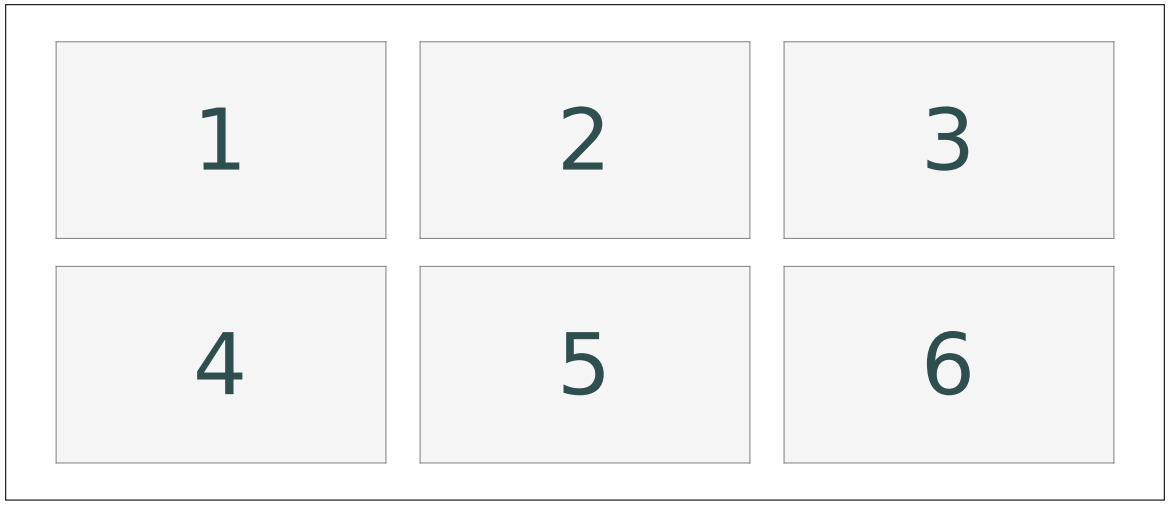
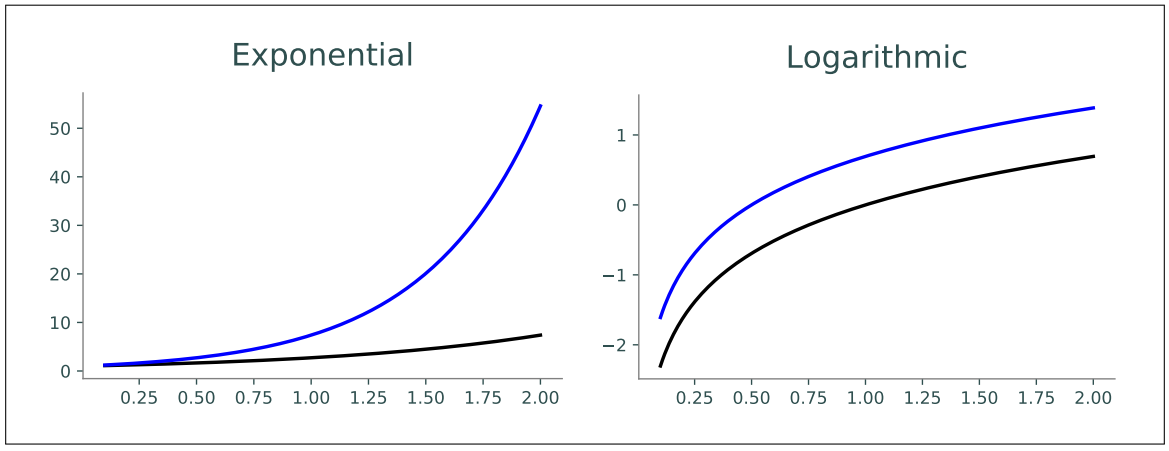


Figure 5.3: The layout of subplots with `plt.subplot(2,3,i)` (2 rows, 3 columns), where `i` is the index pictured above. The outer border is the figure that the axes belong to.

```
>>> x = np.linspace(.1, 2, 200)
# Create a subplot to cover the left half of the figure.
>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, np.exp(x), 'k', lw=2)
>>> ax1.plot(x, np.exp(2*x), 'b', lw=2)
>>> plt.title("Exponential", fontsize=18)

# Create another subplot to cover the right half of the figure.
>>> ax2 = plt.subplot(122)
>>> ax2.plot(x, np.log(x), 'k', lw=2)
>>> ax2.plot(x, np.log(2*x), 'b', lw=2)
>>> ax2.set_title("Logarithmic", fontsize=18)
>>> plt.show()
```



Note

Plotting functions such as `plt.plot()` are shortcuts for accessing the current axes on the current figure and calling a method on that `Axes` object. Calling `plt.subplot()` changes the current axis, and calling `plt.figure()` changes the current figure. Use `plt.gca()` to get the current axes and `plt.gcf()` to get the current figure. Compare the following equivalent strategies for producing a figure with two subplots.

```
>>> x = np.linspace(-5, 5, 100)

# 1. Use plt.subplot() to switch the current.
>>> plt.subplot(121)
>>> plt.plot(x, 2*x)
>>> plt.subplot(122)
>>> plt.plot(x, x**2)

# 2. Use plt.subplot() to explicitly grab the two subplot axes.
>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, 2*x)
>>> ax2 = plt.subplot(122)
>>> ax2.plot(x, x**2)

# 3. Use plt.subplots() to get the figure and all subplots simultaneously.
>>> fig, axes = plt.subplots(1, 2)
>>> axes[0].plot(x, 2*x)
>>> axes[1].plot(x, x**2)
```

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 of a single figure.

1. Arrange the plots in a 2×2 grid of subplots.
2. Set the limits of each subplot to $[0, 2\pi] \times [-2, 2]$.
   (Hint: Consider using `plt.axis([xmin, xmax, ymin, ymax])` instead of `plt.xlim()` and `plt.ylim()` to set all boundaries simultaneously.)
3. Use `plt.title()` or `ax.set_title()` to give each subplot an appropriate title.
4. Use `plt.suptitle()` or `fig.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.

Achtung!

Be careful not to mix up the following functions.

1. `plt.axes()` creates a new place to draw on the figure, while `plt.axis()` (or `ax.axis()`) sets properties of the x- and y-axis in the current axes, such as the x and y limits.
2. `plt.subplot()` (singular) returns a single subplot belonging to the current figure, while `plt.subplots()` (plural) creates a new figure and adds a collection of subplots to it.

Other Kinds of Plots

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

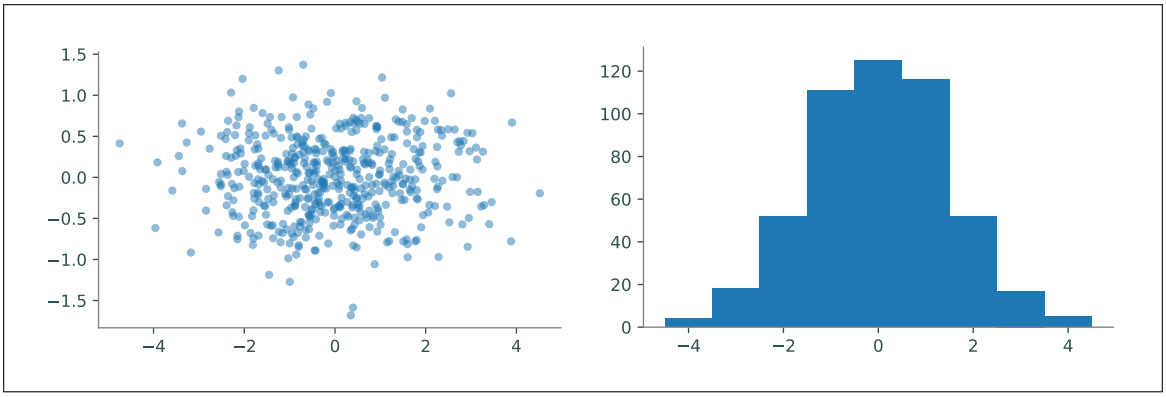
- A *scatter plot* plots two 1-dimensional arrays against each other without drawing lines between the points. Scatter plots are particularly useful for data that is not correlated or ordered.

  To create a scatter plot, use `plt.plot()` and specify a point marker (such as `'o'` or `'*'`) for the line style, or use `plt.scatter()` (or `ax.scatter()`). Beware that `plt.scatter()` has slightly different arguments and syntax than `plt.plot()`.

- 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()` (or `ax.hist()`). 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 transparent circle markers.
>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, y, 'o', markersize=5, alpha=.5)

# Draw a histogram to display the distribution of the data in x.
>>> ax2 = plt.subplot(122)
>>> ax2.hist(x, bins=np.arange(-4.5, 5.5))      # Or, equivalently,
#   ax2.hist(x, bins=9, range=[-4.5, 4.5])

>>> plt.show()
```



Problem 5. The Fatality Analysis Reporting System (FARS) is a nationwide census that provides 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 using `plt.xlabel()` and `plt.ylabel()` (or `ax.set_xlabel()` and `ax.set_ylabel()`). (Hint: Use `plt.axis("equal")` or `ax.set_aspect("equal")` so that the x- and y-axis are scaled the same way.

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. You should be able to clearly see which hours of the day experience more traffic.

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

Matplotlib also has tools for creating other kinds of plots for visualizing 1-dimensional data, including bar plots and box plots. See the Matplotlib Appendix for examples and syntax.

Visualizing 3-D Surfaces

Line plots, histograms, and scatter plots are good for visualizing 1- and 2-dimensional data, including the domain and range of a function $f : \mathbb{R} \to \mathbb{R}$. However, visualizing 3-dimensional data or a function $g : \mathbb{R}^2 \to \mathbb{R}$ (two inputs, one output) requires a different kind of plot. The process is similar to creating a line plot but requires slightly more setup: first construct an appropriate domain, then calculate the image of the function on that domain.

NumPy's `np.meshgrid()` function is the standard tool for creating a 2-dimensional domain in the Cartesian plane. Given two 1-dimensional coordinate arrays, `np.meshgrid()` creates two corresponding coordinate matrices. See Figure 5.6.

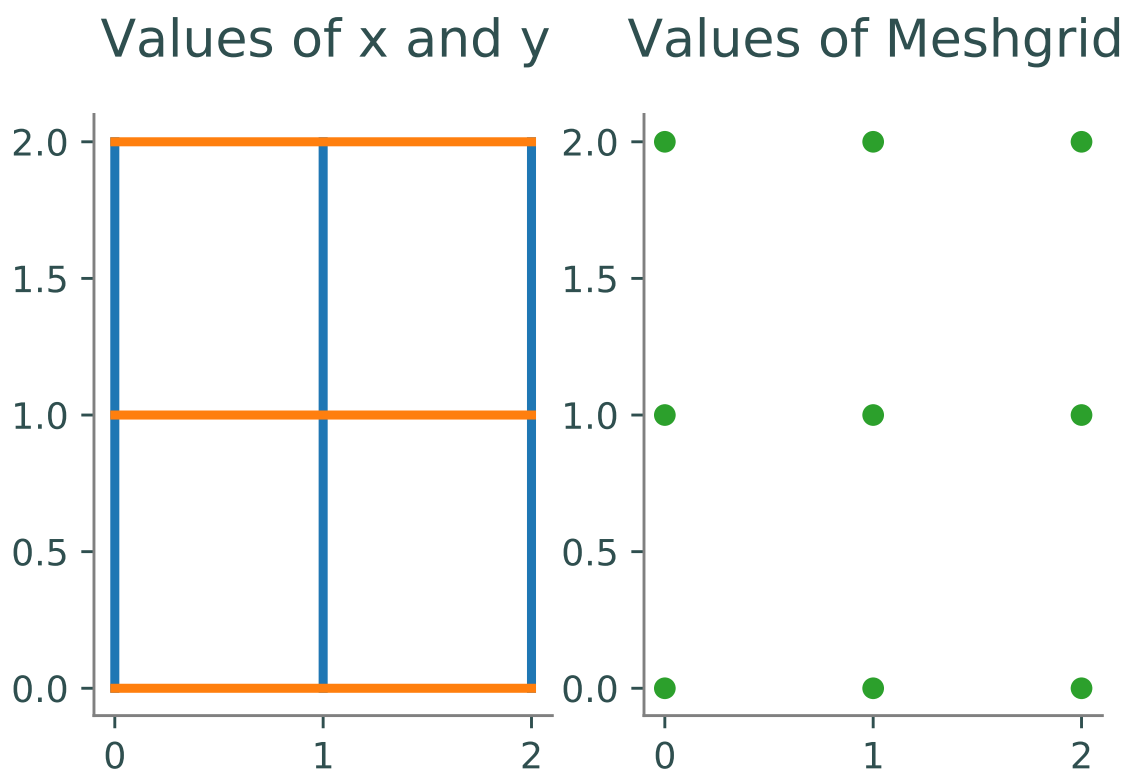


Figure 5.6: In the left plot, we have two arrays where x and y have the values `x = y = [0,1,2]`. The command `np.meshgrid(x, y)` returns the arrays $X = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{bmatrix}$ and $Y = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix}$. These give the x- and y-coordinates of the points in the grid formed by `x` and `y` as seen in the right plot and satisfy the relation `(X[i,j], Y[i,j]) = (x[j],y[i])`.

```
>>> x, y = [0, 1, 2], [3, 4, 5]      # A rough domain over [0,2]x[3,5].
>>> X, Y = np.meshgrid(x, y)         # Combine the 1-D data into 2-D data.
>>> for xrow, yrow in zip(X, Y):
...     print(xrow, yrow, sep='\t')
...
[0 1 2]    [3 3 3]
[0 1 2]    [4 4 4]
[0 1 2]    [5 5 5]
```

With a 2-dimensional domain, $g(x, y)$ is usually visualized with two kinds of plots.

- A *heat map* assigns a color to each point in the domain, producing a 2-dimensional colored 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. You can add an optional argument for the shading type; this determines the layout and fill style of the heat map. This argument defaults to `shading='auto'`, and will automatically choose a fill method suited to the data being graphed.

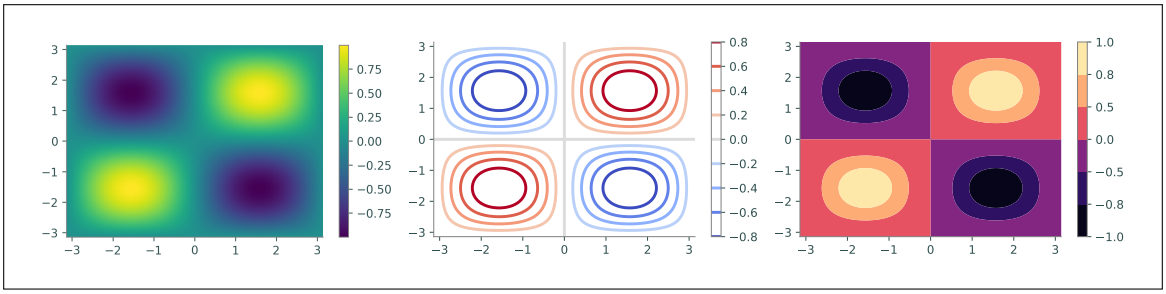
- A *contour map* draws several *level curves* of g on the 2-dimensional domain. A level curve corresponding to the constant c is the collection of points $\{(x, y) \mid c = g(x, y)\}$. Coloring the space between the level curves produces a discretized version of a heat map. Including more and more level 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 functions each receive the keyword argument `cmap` to specify a color scheme (some of the better schemes are `"viridis"`, `"magma"`, and `"coolwarm"`). For the list of all Matplotlib color schemes, see `http://matplotlib.org/examples/color/colormaps_reference.html`.

Finally, `plt.colorbar()` draws the color scale beside the plot to indicate how the colors relate to the values of the function.

```
# 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)
>>> Z = np.sin(X) * np.sin(Y)       # Calculate g(x,y) = sin(x)sin(y).

# Plot the heat map of f over the 2-D domain.
>>> plt.subplot(131)
>>> plt.pcolormesh(X, Y, Z, cmap="viridis", shading="auto")
>>> 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="coolwarm")
>>> 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 $g(x, y) = \frac{\sin(x)\sin(y)}{xy}$ on the domain $[-2\pi, 2\pi] \times [-2\pi, 2\pi]$.

1. Create 2 subplots: one with a heat map of g, and one with a contour map of g. 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 a color scale bar for each subplot.

Additional Material

Further Reading and Tutorials

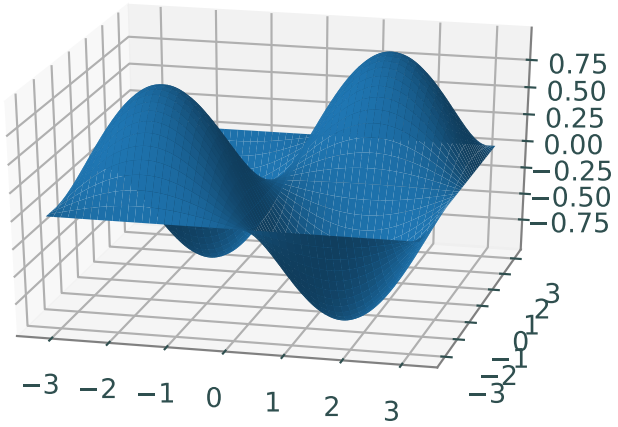
Plotting takes some getting used to. See the following materials for more examples.

- `https://www.labri.fr/perso/nrougier/teaching/matplotlib/`.
- `https://matplotlib.org/stable/tutorials/introductory/pyplot.html`.
- `http://scipy-lectures.org/intro/matplotlib/`.
- The Matplotlib Appendix in this manual.

3-D Plotting

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

```
# 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.
>>> fig = plt.figure()
>>> ax = fig.add_subplot(1, 1, 1, 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 for 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 the window boundaries of the axes.
    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 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.add_subplot(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 Input/Output

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. An understanding of exceptions is essential to safely read data from and write data to external files. Being able to interact with external files is important for analyzing data and communicating results. In this lab we learn exception syntax and file interaction protocols.*

Exceptions

An *exception* formally indicates an error and terminates the program early. Some of the more common exception types are listed below, along with the kinds of problems they typically indicate.

Exception	Indication
`AttributeError`	An attribute reference or assignment failed.
`ImportError`	An `import` statement failed.
`IndexError`	A sequence subscript was out of range.
`NameError`	A local or global name was not found.
`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.
`ZeroDivisionError`	The second argument of a division or modulo operation was zero.

```
>>> 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'
```

Raising Exceptions

Most exceptions are due to coding mistakes and typos. However, 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.

```
>>> if 7 is not 7.0:                # Raise an exception with an error message.
...     raise Exception("ints and floats are different!")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: ints and floats are different!

>>> for x in range(10):
...     if x > 5:                   # Raise a specific kind of exception.
...         raise ValueError("'x' should not exceed 5.")
...     print(x, end=' ')
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ValueError: 'x' should not exceed 5.
0 1 2 3 4 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 (say, 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 = input("Enter a 3-digit number where the first and last "
                                          "digits differ by 2 or more: ")
    step_2 = input("Enter the reverse of the first number, obtained "
                                            "by reading it backwards: ")
    step_3 = input("Enter the positive difference of these numbers: ")
    step_4 = 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:

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

(Hint: `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 with instructions for what to do in the event of an exception.

```
# The 'try' block should hold any lines of code that might raise an exception.
>>> try:
...     print("Entering try block...")
...     raise Exception("for no reason")
...     print("No problem!")        # This line gets skipped.
... # The 'except' block is executed just after the exception is raised.
... except Exception as e:
...     print("There was a problem:", e)
...
Entering try block...
There was a problem: 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. If desired, `e` can be raised again with `raise e` or just `raise`.

The try-except control flow can be expanded with two other blocks, forming a code structure similar to a sequence of `if-elif-else` blocks.

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 `else` block is executed if an exception was **not** raised in the try block.
4. The `finally` block is always executed if it is included.

```
>>> try:
...     print("Entering try block...", end='')
...     house_on_fire = False
...     raise ValueError("The house is on fire!")
... # Check for multiple kinds of exceptions using parentheses.
... except (ValueError, TypeError) as e:
...     print("caught an exception.")
...     house_on_fire = True
... else:                           # Skipped due to the exception.
...     print("no exceptions raised.")
... finally:
...     print("The house is on fire:", house_on_fire)
...
Entering try block...caught an exception.
The house is on fire: True

>>> try:
...     print("Entering try block...", end='')
...     house_on_fire = False
... except ValueError as e:         # Skipped because there was no exception.
...     print("caught a ValueError.")
...     house_on_fire = True
... except TypeError as e:          # Also skipped.
...     print("caught a TypeError.")
...     house_on_fire = True
... else:
...     print("no exceptions raised.")
... finally:
...     print("The house is on fire:", house_on_fire)
...
Entering try block...no exceptions raised.
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.

```
>>> def implode():
...     try:                       # Try to return immediately...
...         return
...     finally:                   # ...but 'finally' goes before 'return'.
...         print("Goodbye, world!")
...
>>> implode()
Goodbye, world!
```

See `https://docs.python.org/3/tutorial/errors.html` for more examples.

Achtung!

An `except` statement with no specified exception type catches **any** exception raised in the corresponding `try` block. This approach can mistakenly mask unexpected errors. Always be specific about the kinds of exceptions you expect to encounter.

```
>>> def divider(x, y):
...     try:
...         return x / yy           # The mispelled yy raises a NameError.
...     except:                     # Catch ANY exception.
...         print("y must not equal zero!")
...
>>> divider(2, 3)
y must not equal zero!

>>> def divider(x, y):
...     try:
...         return x / yy
...     except ZeroDivisionError:   # Specify an exception type.
...         print("y must not equal zero!")
...
>>> divider(2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divider
NameError: name 'yy' is not defined     # Now the mistake is obvious.
```

Problem 2. A *random walk* is a path created by a sequence of random steps. The following function simulates a random walk by repeatedly adding or subtracting 1 to a running total.

```
from random import choice

def random_walk(max_iters=1e12):
    walk = 0
    directions = [1, -1]
    for i in range(int(max_iters)):
        walk += choice(directions)
    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` by pressing `ctrl+c` while the program is running, the function catches 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 exceptions are organized into a class hierarchy. For example, the `ValueError` class inherits from the generic `Exception` class. Thus, a `ValueError` **is** an `Exception`, but an `Exception` is **not** a `ValueError`.

```
>>> try:
...     raise ValueError("caught!")
... except Exception as e:          # A ValueError is an Exception.
...     print(e)
...
caught!                             # The exception was caught.

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

See `https://docs.python.org/3/library/exceptions.html` for the complete list of built-in exceptions and the exception class hierarchy.

Debugging Exceptions

Exceptions are an essential and informative tool for debugging your code. As such, it's important to understand what an exception is saying and how to properly debug (or fix) the issue causing it.

The Stack Trace

The stack trace is the printed-out statement given to the user or programmer whenever an exception is raised. It contains important information about the type of exception and the location where the exception occured.

```
>>> x = 1/0       # Stack trace below
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Exceptions can have more complicated stack traces if the error occurs inside of a function call, among other scenarios. As stated at the top of the stack trace, the most recent function call before the exception occurred is output last. This means that the stack trace could contain several lines indicating where the error occured.

```
>>> def make_error():     # Function to raise an error
```

```
...     return 1/0
...
>>> x = make_error()     # Calling the function
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in make_error
ZeroDivisionError: division by zero   # The exception occurred in the function!
```

Additionally, most modern IDE's display the line of code that caused the error (and may even point to the specific part of the line that is believed to be at fault). If the last function called was a function you did not write, this can mean that unfamiliar code is displayed to you which might not be helpful in debugging your code. As such, it's important to first look at the type of exception raised (listed at the bottom of the stack trace) and then move up the stack trace to identify code that may have caused the issue.

```
# Stack trace as displayed by the terminal
Traceback (most recent call last):
  File "<filepath>", line 4, in <module>
    x = make_error()
  File "<filepath>", line 2, in make_error
    return 1/0
ZeroDivisionError: division by zero
```

Print Debugging

One of the most basic (yet effective) methods for debugging is using print statements. This involves printing out information in the lines leading up to where the exception was raised (which you can find using the stack trace). For example, if a variable being passed into a function is causing an exception, it can be helpful to print out the value of that variable on the line before the function is executed.

```
>>> def divide(a,b):           # Normal function to divide variables
...     return a/b
...
>>> x = 1
>>> y = 0
>>> print(f"{x=} and {y=}")    # Print out the variables that we'll be using
x=1 and y=0
>>> z = divide(x,y)            # Our input values cause an error
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in divide
ZeroDivisionError: division by zero
```

This method is simple to implement and can help identify problems in the code that occured before an exception was raised. It can even help identify problems with function outputs that don't result in an exception (such as unexpected results or values).

Debuggers

While print statements can be a quick solution to small bugs, most IDE's (including VS Code) include a helpful tool called a debugger to aid in finding and fixing errors in code. This is a separate runtime environment to running the code normally and is generally referred to as runing in debug mode. Debuggers have many useful features, including the ability to add special markers called *break points* to your code.

These markers (when running in debug mode) temporarily pause execution just before the line specified and display the values of all variables, as well as other useful information. After execution is paused, it will not resume until the programmer indicates. Additionally, code can be executed line-by-line in this paused state, and variables will display their new values as they update.

There are many more tools available in the debugger, so it's highly encouraged to test out the version specific to your IDE to learn best how it can help you debug your code.

File Input and Output

A file object acts as an interface to a file stream, meaning, it allows a program to read from or write to external files. 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 that the user has to the file. There are four common modes:

`'r'`: **read**. Open an existing file for reading. The file must already exist, or `open()` raises a `FileNotFoundError`. This is the default mode.

`'w'`: **write**. Create a new file or **overwrite an existing file** (careful!) and open it for writing.

`'x'`: **write new**. Create a new file and open it for writing. If the file already exists, `open()` raises a `FileExistsError`. This is a safer form of `'w'` because it never overwrites existing files.

`'a'`: **append**. Open a file for writing and append new data to the end of the file if it already exists.

```
>>> myfile = open("hello_world.txt", 'r')   # Open a file for reading.
>>> print(myfile.read())                    # Print the contents of the file.
Hello,                                      # (it's a really small file.)
World!

>>> myfile.close()                          # Close the file connection.
```

The With Statement

An `IOError` indicates that some input or output operation has failed. 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 and safely closed, even in the event of an exception. This is the preferred file-reading method when a file only needs to be accessed briefly.

```
>>> myfile = open("hello_world.txt", 'r')   # Open a file for reading.
>>> 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("hello_world.txt", 'r') as myfile:
...    contents = myfile.readlines()
...                                        # The file is closed automatically.
```

In both cases, if the file `hello_world.txt` does not exist in the current directory, `open()` raises a `FileNotFoundError`. However, errors in the `try` or `with` blocks do 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 the 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 rest of the 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 automatically).
`writelines()`	Write a list of strings to the file (newline characters are **not** added automatically).

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 range(10):
...         outfile.write(str(i**2)+' ')    # Write some strings (and spaces).
...
>>> outfile.closed                          # The file is closed automatically.
True
```

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 file name is invalid in any way, prompt the user for another filename using `input()`. Continue prompting the user until they provide a valid filename.

```
>>> cf1 = ContentFilter("hello_world.txt")  # File exists.
>>> cf2 = ContentFilter("not-a-file.txt")   # File doesn't exist.
Please enter a valid file name: still-not-a-file.txt
Please enter a valid file name: hello_world.txt
>>> cf3 = ContentFilter([1, 2, 3])          # Not even a string.
Please enter a valid file name: hello_world.txt
```

   (Hint: `with open()` might raise a `FileNotFoundError`, a `TypeError`, or an `OSError`.)

2. Read the file and store its name and contents as attributes (store the contents as a single string). Make sure the file is securely closed.

String Formatting

The `str` 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 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'
```

```
# 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 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("hello_world.txt", 'r') as myfile:
...     contents = myfile.readlines()
...
>>> with open("hello_world.txt", 'r') as myfile:
...     contents = myfile.read().split('\n')
```

Variables within Strings

In addition to formatting strings using the methods above there may also be times when you wish to insert generalized variable values into the middle of strings. There are two ways to do this: using .format() or using f-strings. Of the two, f-strings are generally more readable, concise, faster, and less prone to error. As a result, an explanation of how to use .format() has been banished to the "Additional Materials" section, which we know is rarely read, and a short explanation of f-strings will be included here.

We define an f-string with an 'f' preceding the string declaration (i.e f'your words here', or f"your words here"). Within the f-string, curly brackets can be used along with variable names to insert the variable value into the string. Consider the following code which puts an argument defaulting to 'bread and butter pickles' into the sentence: "I think that (blank) are an abomination." Note that f-strings can also be used with multiple variables at once by using multiple sets of curly brackets.

```
# defining function
>>> def abomination(userOpinion="Bread and Butter Pickles"):
...         # notice the use of the f-string
...         return f"I think that {userOpinion} are an abomination"

# calling the function
>>> abomination()
'I think that Bread and Butter Pickles are an abomination.'
```

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 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 `'w'`, `'x'`, or `'a'`, raise a `ValueError` with an informative message.

1. `uniform()`: write the data to the outfile with uniform case. Include a 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 a 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.

4. `__str__()`: Also implement the `__str__()` magic method so that printing a `ContentFilter` object yields the following output. You may want to calculate these statistics in the constructor. (Note: Using f-strings will also make this implementation much simpler).

```
Source file:              <filename>
Total characters:         <The total number of characters in 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>
```

(Hint: list comprehensions are **very** useful for some of these functions. For example, what does `[line[::-1] for line in lines]` do? What about `sum([s.isspace() for s in data])`?)

Compare your class to the following example.

```
# cf_example1.txt
A b C
d E f
```

```
>>> cf = ContentFilter("cf_example1.txt")
>>> cf.uniform("uniform.txt", mode='w', case="upper")
>>> cf.uniform("uniform.txt", mode='a', case="lower")
>>> cf.reverse("reverse.txt", mode='w', unit="word")
>>> cf.reverse("reverse.txt", mode='a', unit="line")
>>> cf.transpose("transpose.txt", mode='w')
```

```
# uniform.txt
A B C
D E F
a b c
d e f
```

```
# reverse.txt
C b A
f E d
d E f
A b C
```

```
# transpose.txt
A d
b E
C f
```

Achtung!

When parsing though the datafiles be sure to account for trailing whitespaces at the end of lines. Each file may or may not contain them, which can lead to unexpected behavior. Consider using `strip()` to avoid bugs in your code.

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.

Chaining Exceptions

Sometimes, especially in large programs, it is useful to raise one kind of exception just after catching another. The two exceptions can be linked together using the `from` statement. This syntax makes it possible to see where the error originated from and to "pass it up" to another part of the program.

```
>>> try:
...     raise TooHardError("This lab is impossible!")
... except TooHardError as e:
...     raise NotImplementedError("Lab is incomplete") from e
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
__main__.TooHardError: This lab is impossible!

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NotImplementedError: Lab is incomplete
```

More String Formatting Tools

Concatenating string values with non-string values can be cumbersome and tedious. 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.

```
# Join the data using string concatenation.
>>> day, month, year = 10, "June", 2017
```

```
>>> print("Is today", day, str(month) + ',', str(year) + "?")
Is today 10 June, 2017?

# Join the data using str.format().
>>> print("Is today {} {}, {}?".format(day, month, year))
Is today 10 June, 2017?

# Join the data more easily using an f-string.
>>> print(f"Is today {day} {month}, {year}?")
Is today 10 June, 2017?
```

This method is extremely flexible and provides many convenient ways to format string output nicely. Consider the following code for printing out a simple progress bar from within a loop.

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

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 `end` parameter can change the default newline added to the output to another ending. Setting to an empty string, `end=''`, the output ends without any whitespace. The `flush` parameter defaults to `False` and does not need to be changed if the `end` parameter is not changed. As shown in the example above, setting the parameter equal to `True` forces the output to be printed on the terminal before it is complete. If it is `False` with the `end=''` the `print()` function will first build the output with the set ending before printing each iteration rather than printing incrementally.

Printing at each iteration dramatically slows down the progression through the loop. How does the following code solve that problem?

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

See `https://docs.python.org/3/library/string.html#format-string-syntax` for more examples and specific syntax for using `str.format()`. For a more robust progress bar printer, research the `tqdm` module.

Standard Library Modules for I/O

The standard library has other tools for input and output operations. For details on each module, see `https://docs.python.org/3/library`.

Module	Description
`csv`	CSV (comma separated value) file writing and parsing.
`io`	Support for file objects and `open()`.
`os`	Communication with the operating system.
`os.path`	Common path operations such as checking for file existence.
`pickle`	Create portable serialized representations of Python objects.

7 Unit 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 help make sure that every unit of code functions as intended, thereby decreasing the chances for errors in the program. 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 `pytest` in the current directory. Without any test files, the output should be similar to the following.

```
$ pytest
============================= test session starts ==============================
platform linux -- Python 3.10.6, pytest-7.4.0, py-1.11.0, pluggy-1.2.0
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 `pytest` identifies failed tests and provides details on why they failed.

[1] Pytest is not part of the standard libray. Install pytest with `[basicstyle=]pip install pytest if needed. The standard library's [basicstyle=]unittest module also provides a testing framework, but is less popular and straightforward than PyTest.`

```
$ pytest
============================= test session starts ==============================
platform linux -- Python 3.10.6, pytest-7.4.0, py-1.11.0, pluggy-1.2.0
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 to 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. To install, run the following code in a terminal.

```
$ pip install pytest-cov
```

Add the flag `--cov` to the `pytest` command to print out code coverage information. Running `pytest --cov` in the same directory as `specs.py` and `test_specs.py` yields the following output.

```
$ pytest --cov
============================= test session starts ==============================
platform linux -- Python 3.10.6, pytest-7.4.0, py-1.11.0, pluggy-1.2.0
rootdir: /Users/Student/Testing, inifile:
plugins: cov-2.3.1
collected 7 items

test_specs.py .......

---------- coverage: platform darwin, python 3.6.6-final-0 ----------
Name            Stmts   Miss  Cover
-----------------------------------
specs.py           73     38    48%
test_specs.py      42      0   100%
-----------------------------------
TOTAL             115     38    67%


========================== 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.

```
$ pytest --cov-report html --cov
============================= test session starts ==============================
# ...
---------- coverage: platform linux, python 3.10.6-final-0 ----------
Coverage HTML written to dir htmlcov
```

Instead of printing coverage statistics, this command creates various files with coverage details in a new directory called `htmlcov/`. The file `htmlcov/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. For example, the following code should raise a `ZeroDivisionError` if the divisor is 0.

```
# 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 / b
```

The corresponding unit test checks that the function raises the `ZeroDivisionError` 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 / 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 f"{self.numer} / {self.denom}"
        else:
            return str(self.numer)

    def __float__(self):
        return self.numer / 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 __truediv__(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 `__truediv__()`. 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`.

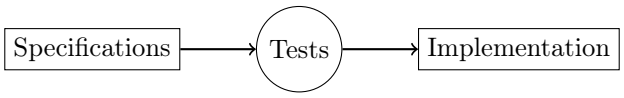
Test-driven Development

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.

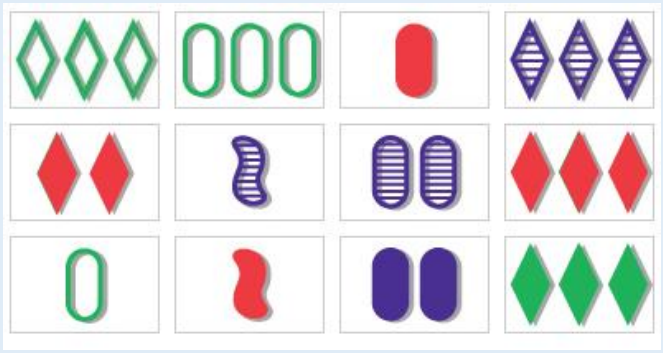


If the test cases are sufficiently thorough, then when the tests all pass, the program can be considered complete. Remember, however, that it is not sufficient to just have tests, but to have tests that accurately and rigorously test the code. To check that the test cases are sufficient, examine the 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", "0200", "2110", "1020"]
```

The following function definitions provide a framework for partially 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 on *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()                                    # Set a break point.
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/3/library/pdb.html` for documentation and examples for the Python debugger.

Other Testing Suites

There are several frameworks other than `pytest` for writing unit tests. Each shares the same basic structure, but the setup, syntax, and particular features vary. For more unit testing practice, try out the standard library's `unittest` (`https://docs.python.org/3/library/unittest.html`) or `doctest` (`https://docs.python.org/3/library/doctest.html`), or the third-party `nose` module (`https://nose.readthedocs.io/en/latest/`). For a much larger list of unit testing tools, see `https://wiki.python.org/moin/PythonTestingToolsTaxonomy`.

The Fractions Module

The standard library's `fractions` module (`https://docs.python.org/3/library/fractions.html`) has a `Fraction` class that is similar to the `Fraction` class presented in this lab. Its structure and syntax is a little different from this lab's class, but it is a little more robust in that it can take in floats, decimals, integers, and strings to its constructor. See also the `decimals` module (`https://docs.python.org/3/library/decimal.html`) for tools relating to decimal arithmetic.

8 Data Visualization

Lab Objective: *This lab demonstrates how to communicate information through clean, concise, and honest data visualization. We recommend completing the exercises in a Jupyter Notebook.*

The Importance of Visualizations

Visualizations of data can reveal insights that are not immediately obvious from simple statistics. The data set in the following exercise is known as *Anscombe's quartet*. It is famous for demonstrating the importance of data visualization.

Problem 1. The file `anscombe.npy` contains the quartet of data points shown in the table below. For each section of the quartet,

- Plot the data as a scatter plot on the box $[0, 20] \times [0, 13]$.
- Use `scipy.stats.linregress()` to calculate the slope and intercept of the least squares regression line for the data and its correlation coefficient (the first three return values).
- Plot the least squares regression line over the scatter plot on the domain $x \in [0, 20]$.
- Report (print) the mean and variance in x and y, the slope and intercept of the regression line, and the correlation coefficient. Compare these statistics to those of the other sections.
- Describe how the section is similar to the others and how it is different.

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

Improving Specific Types of Visualizations

Effective data visualizations show specific comparisons and relationships in the data. Before designing a visualization, decide what to look for or what needs to be communicated. Then choose the visual scheme that makes sense for the data. The following sections demonstrate how to improve commonly used plots to communicate information visually.

Line Plots

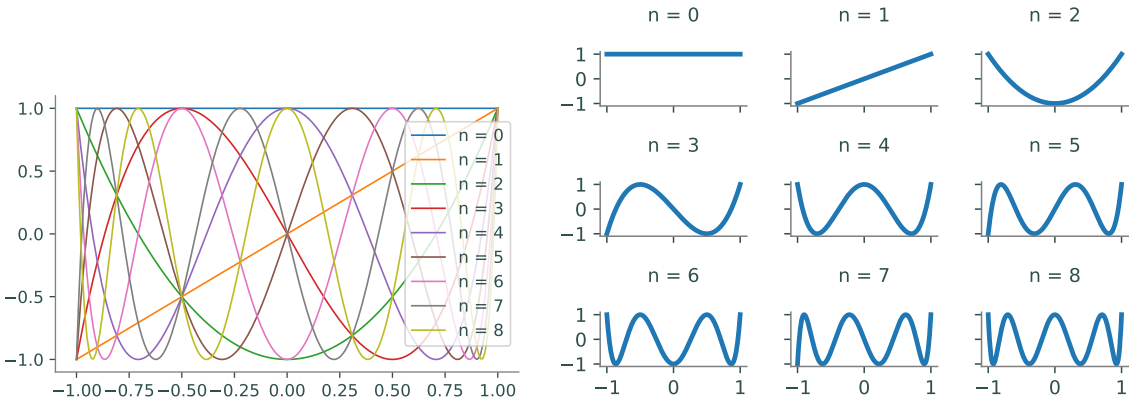


Figure 8.1: Line plots can be used to visualize and compare mathematical functions. For example, this figure shows the first nine Chebyshev polynomials in one plot (left) and small multiples (right). Using small multiples makes comparison easy and shows how each polynomial changes as n increases.

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

# 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(loc="right")
>>> plt.show()
```

A line plot connects ordered (x, y) points with straight lines, and is best for visualizing one or two ordered arrays, such as functional outputs over an ordered domain or a sequence of values over time. Sometimes, plotting multiple lines on the same plot helps the viewer compare two different data sets. However, plotting several lines on top of each other makes the visualization difficult to read, even with a legend. For example, Figure 8.1 shows the first nine *Chebyshev polynomials*, a family of orthogonal polynomials that satisfies the recursive relation

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

The plot on the right makes comparison easier by using *small multiples*. Instead of using a legend, the figure makes a separate subplot with a title for each polynomial. Adjusting the figure size and the line thickness also makes the information easier to read.

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, add the following line of code to the loop from the previous example.

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

The f-string 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 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.comb()`.

Bar Charts

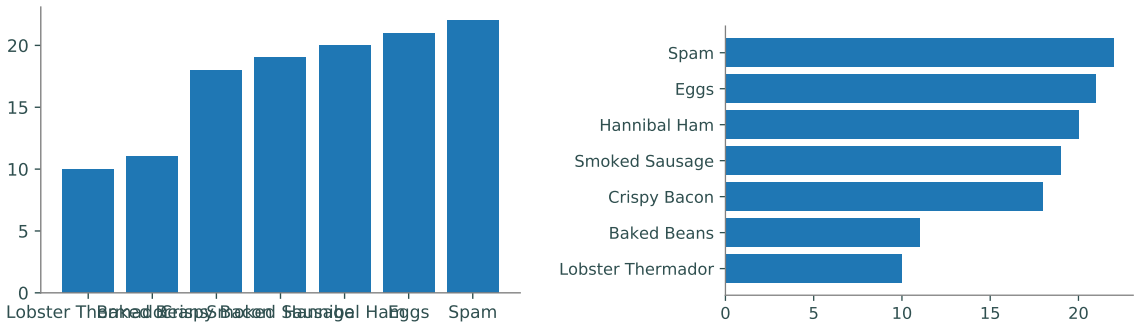


Figure 8.2: Bar charts are used to compare quantities between categorical variables. The labels on the vertical bar chart (left) are more difficult to read than the labels on the horizontal bar chart (right). Although the labels can be rotated, horizontal text is much easier to read than vertical text.

```
>>> 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.bar(positions, values, align="center")  # Vertical bar chart.
>>> plt.xticks(positions, labels)
>>> plt.show()

>>> plt.barh(positions, values, align="center") # Horizontal bar char (better).
>>> plt.yticks(positions, labels)
>>> plt.tight_layout()
>>> plt.show()
```

A bar chart plots categorical data in a sequence of bars. They are best for small, discrete, one-dimensional data sets. In Matplotlib, `plt.bar()` creates a vertical bar chart or `plt.barh()` creates a horizontal bar chart. These functions receive the locations of each bar followed by the height of each bar (as lists or arrays). In most situations, horizontal bar charts are preferable to vertical bar charts because horizontal labels are easier to read than vertical labels. Data in a bar chart should also be sorted in a logical way, such as alphabetically, by size, or by importance.

Histograms

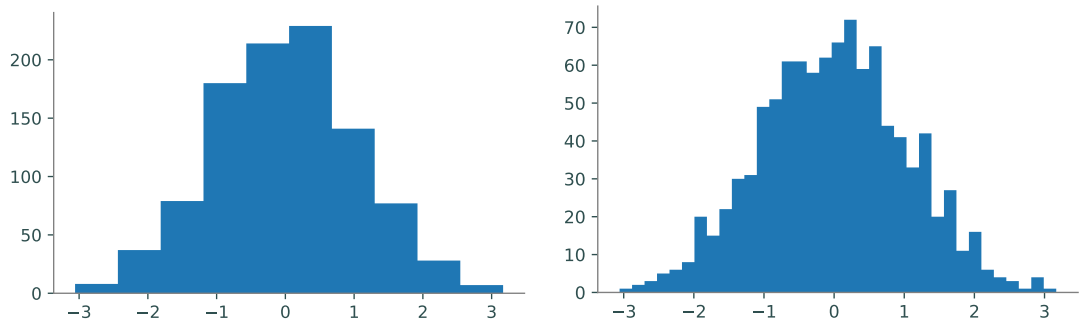


Figure 8.3: Histograms are used to show the distribution of one-dimensional data. Experimenting with different values for the bin size is important when plotting a histogram. Using only 10 bins (left) doesn't give a good sense for how the randomly generated data is distributed. However, using 35 bins (right) reveals the shape of a normal distribution.

```
>>> data = np.random.normal(size=10000)
>>> fig, ax = plt.subplots(1, 2)
>>> ax[0].hist(data, bins=10)
>>> ax[1].hist(data, bins=35)
>>> plt.show()
```

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. For example, if data are drawn from a probability distribution, a histogram approximates the distribution's probability density function. Use `plt.hist()` to create a histogram. The arguments `bins` and `range` specify the number of bins to draw and over what domain. A histogram with too few or too many bins will not give a clear view of the distribution.

Scatter Plots

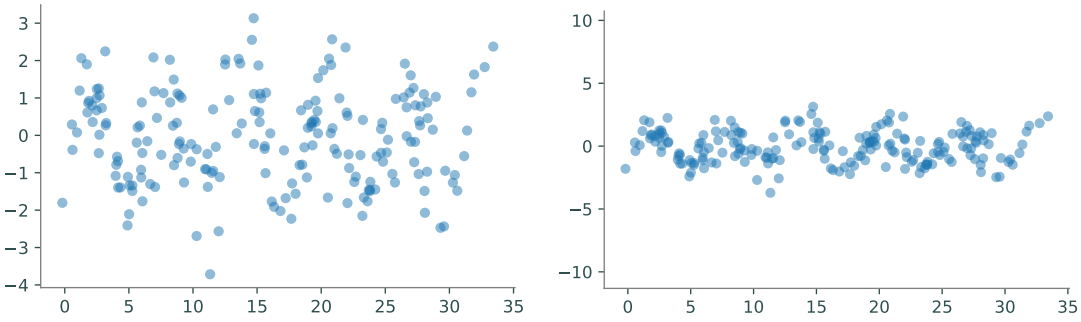


Figure 8.4: Scatter plots show correlations between variables by plotting markers at coordinate points. The figure above displays randomly perturbed data that is visualized using two scatter plots with `alpha=.5` and `edgecolor='none'`. The default (left) makes it harder to see correlation and pattern whereas making the axes equal better reveals the oscillatory behavior in the perturbed sine wave.

```
>>> np.random.seed(0)
>>> x = np.linspace(0, 10*np.pi, 200) + np.random.normal(size=200)
>>> y = np.sin(x) + np.random.normal(size=200)

>>> plt.scatter(x, y, alpha=.5, edgecolor='none')
>>> plt.show()

>>> plt.scatter(x, y, alpha=.5, edgecolor='none')
>>> plt.axis('equal')
>>> plt.show()
```

A scatter plot draws (x, y) points without connecting them. Scatter plots are best for displaying data sets without a natural order, or where each point is a distinct, individual instance. They are frequently used to show correlation between variables in a data set. Use `plt.scatter()` to create a scatter plot.[1]

[1]Scatter plots can also be drawn with `plt.plot()` by specifying a point marker such as `'.'`, `','`, `'o'`, or `'+'`. The keywords `markersize` and `color` can be used to change the marker size and marker color, respectively.

Similar data points in a scatter plot may overlap, as in Figure 8.4. Specifying an *alpha value* reveals overlapping data by making the markers transparent (see Figure 8.5 for an example). The keyword `alpha` accepts values between 0 (completely transparent) and 1 (completely opaque). When plotting lots of overlapping points, the outlines on the markers can make the visualization look cluttered. Setting the edgecolor keyword to zero removes the outline and improves the visualization.

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 player; the columns are the player's height (in inches), weight (in pounds), and age (in years), in that order.

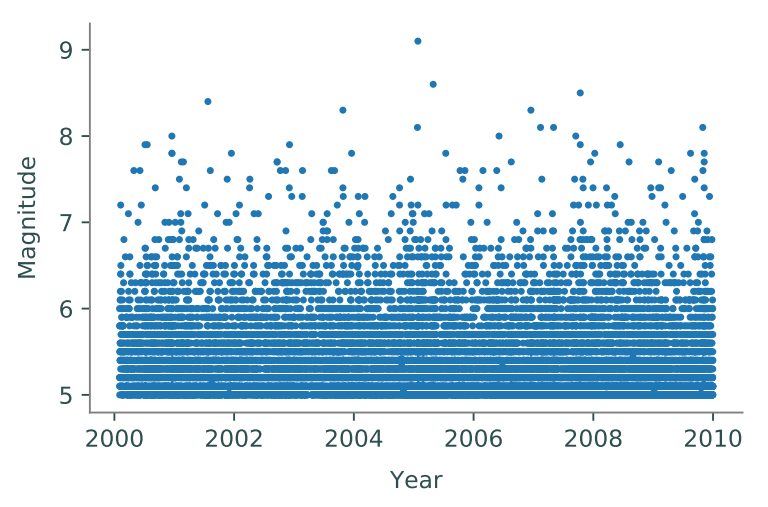
Create several visualizations to show the correlations between height, weight, and age in the MLB data set. Use at least one scatter plot. Adjust the marker size, plot a regression line, change the window limits, and use small multiples where appropriate.

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

Problem 4. The file `earthquakes.npy` contains data from over 17,000 earthquakes from the beginning of 2000 to the end of 2009 that were at least a 5 on the Richter scale.[a] Each row in the array represents an earthquake; the columns are the earthquake's date (as a fraction of the year, so March 14 2001 would be 2001.2), 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 at least two better visualizations. Include line plots, scatter plots, and histograms as appropriate. Your plots should 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")` or `ax.set_aspect("equal")` to fix the aspect ratio, which may improve comparisons between longitude and latitude.)

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

Hexbins

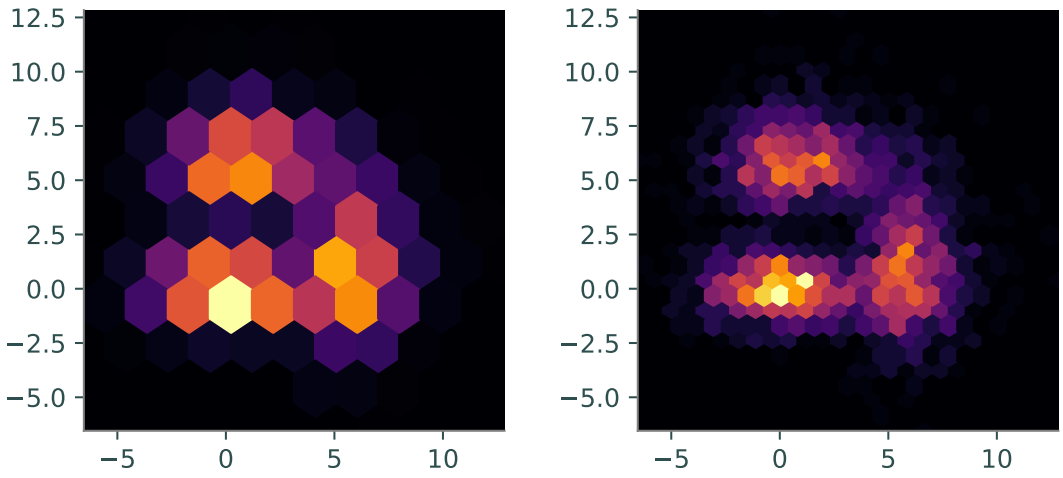


Figure 8.5: Hexbins can be used instead of using a three-dimensional histogram to show the distribution of two-dimensional data. Choosing the right gridsize will give a better picture of the distribution. The figure above shows random data plotted as hexbins with a gridsize of 10 (left) and 25 (right). Hexbins use color to show height via a colormap and both histograms above use the `'inferno'` colormap.

```
# Add random draws from various distributions in two dimensions.
>>> a = np.random.exponential(size=1000) + np.random.normal(size=1000) + 5
>>> b = np.random.exponential(size=1000) + 2*np.random.normal(size=1000)
>>> x = np.hstack((a, b, 2*np.random.normal(size=1000)))
>>> y = np.hstack((b, a,   np.random.normal(size=1000)))

# Plot the samples with hexbins of gridsize 10 and 25.
>>> fig, axes = plt.subplots(1, 2)
>>> window = [x.min(), x.max(), y.min(), y.max()]
>>> for ax, size in zip(axes, [10, 25]):
```

```
...     ax.hexbin(x, y, gridsize=size, cmap='inferno')
...     ax.axis(window)
...     ax.set_aspect("equal")
...
>>> plt.show()
```

A *hexbin* is a way of representing the frequency of occurrences in a two-dimensional plane. Similar to a histogram, which sorts one-dimensional data into bins, a hexbin sorts two-dimensional data into hexagonal bins arranged in a grid and uses color instead of height to show frequency. Creating an effective hexbin relies on choosing an appropriate `gridsize` and colormap. The *colormap* is a function that assigns data points to an ordering of colors. Use `plt.hexbin()` to create a hexbin and use the `cmap` keyword to specify the colormap.

Heat Maps and Contour Plots

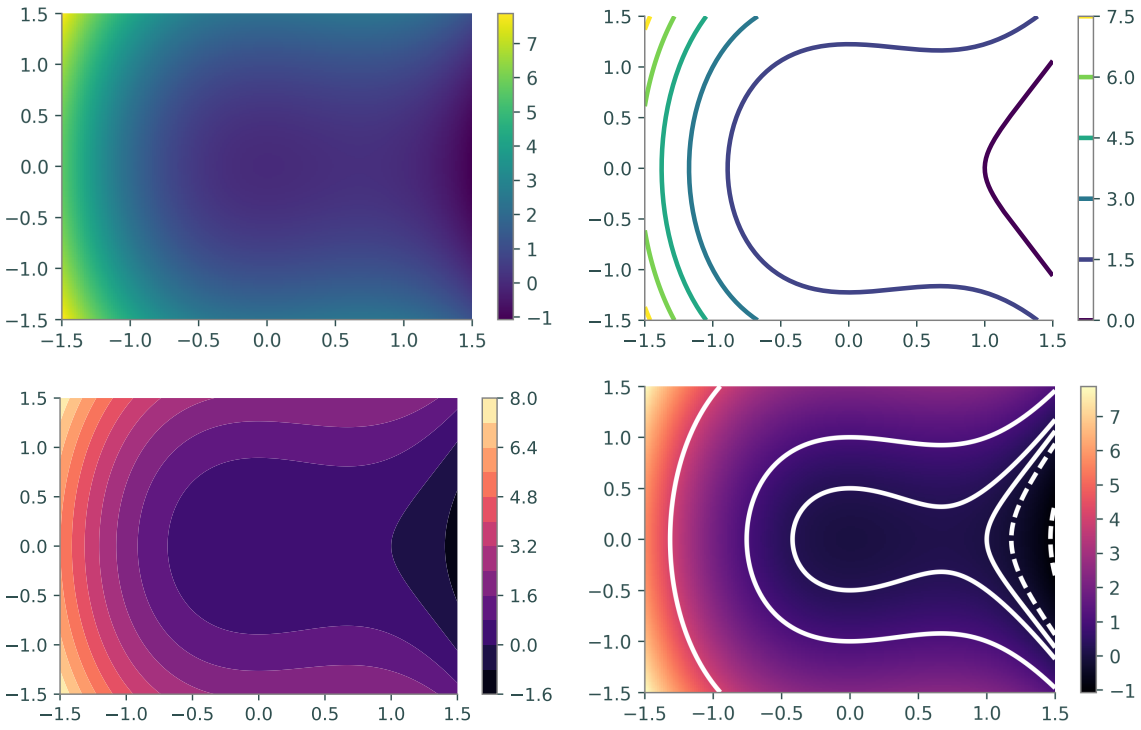


Figure 8.6: Heat maps visualize three-dimensional functions or surfaces by using color to represent the value in one dimension. With continuous data, it can be hard to identify regions of interest. Contour plots solve this problem by visualizing the level curves of the surface. Top left: heat map. Top right: contour plot. Bottom left: filled contour map. Bottom right: contours plotted on a heat map.

```
# 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)
```

```
>>> Z = Y**2 - X**3 + X**2

# Plot f using a heat map, a contour map, and a filled contour map.
>>> fig, ax = plt.subplots(2, 2)
>>> ax[0, 0].pcolormesh(X, Y, Z, cmap="viridis")     # Heat map.
>>> ax[0, 1].contour(X, Y, Z, 6, cmap="viridis")     # Contour map.
>>> ax[1, 0].contourf(X, Y, Z, 12, cmap="magma")   # Filled contour map.


# Plot specific level curves and a heat map with a colorbar.
>>> ax[1, 1].contour(X, Y, Z, [-1, -.25, 0, .25, 1, 4], colors="white")
>>> cax = ax[1, 1].pcolormesh(X, Y, Z, cmap="magma")
>>> fig.colorbar(cax, ax=ax[1, 1])

>>> plt.show()
```

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 colors in the sections between the level curves and is a discretized version of a heat map. 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. Since $f(0, 0) = 0$, choosing several level curves close to 0 more closely describes the topography of the basin. The fourth subplot in 8.6 uses the curves with $c = -1,\ -\frac{1}{4},\ 0,\ \frac{1}{4},\ 1$, and 4.

When plotting hexbins, heat maps, and contour plots, be sure to choose a colormap that best represents the data. Avoid using spectral or rainbow colormaps like `"jet"` because they are not *perceptually uniform*, meaning that the rate of change in color is not constant. Because of this, data points may appear to be closer together or farther apart than they actually are. This creates visual false positives or false negatives in the visualization and can affect the interpretation of the data. As a default, we recommend using the sequential colormaps `"viridis"` or `"inferno"` because they are designed to be perceptually uniform and colorblind friendly. For the complete list of Matplotlib color maps, see `http://matplotlib.org/examples/color/colormaps_reference.html`.

Problem 5. The *Rosenbrock function* is defined as

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

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 a heat map and a contour plot to visualize the Rosenbrock function. Also plot the minimizer $(1, 1)$. Use a different sequential colormap for each visualization.

Best Practices

Good scientific visualizations make comparison easy and clear. The eye is very good at detecting variation in one dimension and poor in two or more dimensions. For example, consider Figure 8.7. Despite the difficulty, most people can probably guess which slice of a pie chart is the largest or smallest. However, it's almost impossible to confidently answer the question *by how much?* The bar charts may not be as aesthetically pleasing but they make it much easier to precisely compare the data. Avoid using pie charts as well as other visualizations that make accurate comparison difficult, such as radar charts, bubble charts, and stacked bar charts.

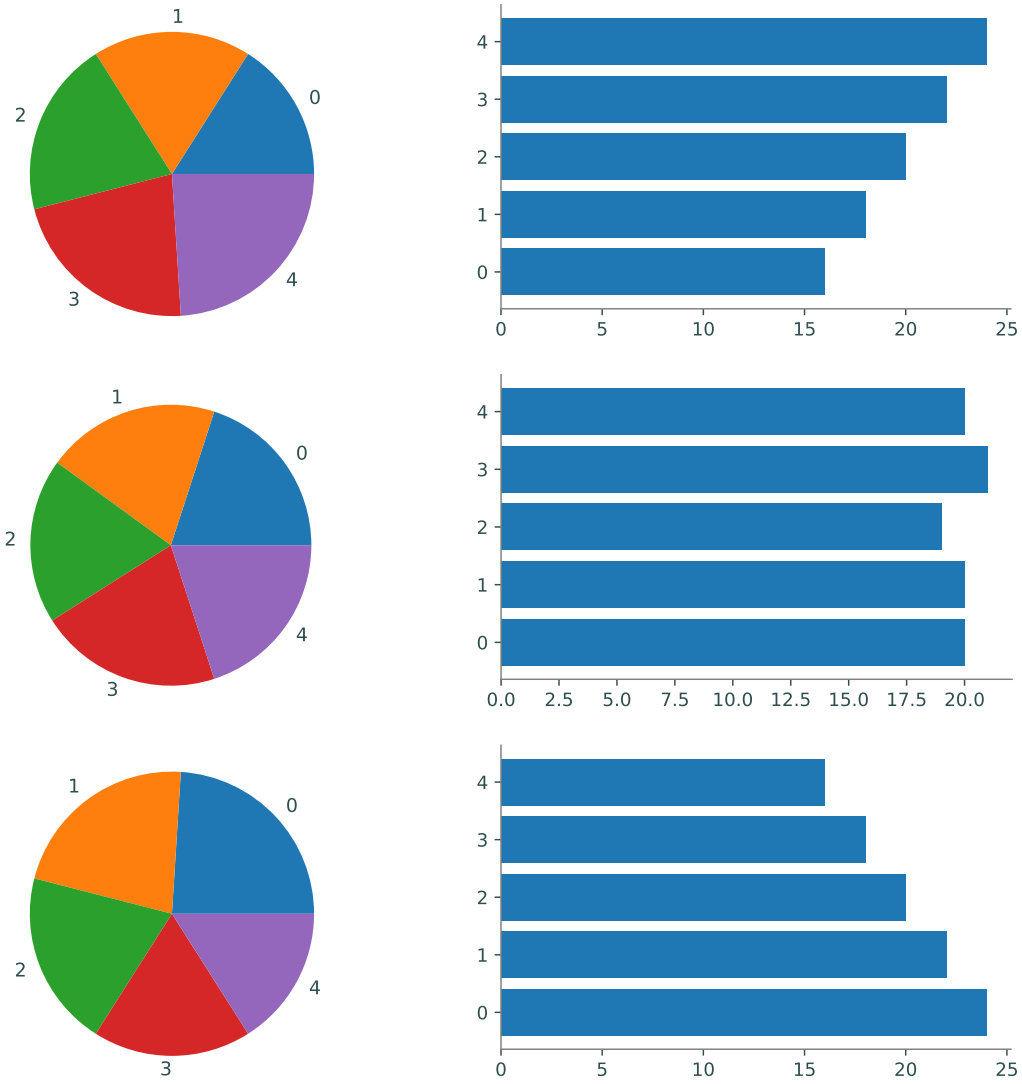


Figure 8.7: The pie charts on the left may be more colorful but it's extremely difficult to quantify the difference between each slice. Instead, the horizontal bar charts on the right make it very easy to see the difference between each variable.

No visualization perfectly represents data, but some are better than others. Finding the best visualization for a data set is an iterative process. Experiment with different visualizations by adjusting their parameters: color, scale, size, shape, position, and length. It may be necessary to use a data transformation or visualize various subsets of the data. As you iterate, keep in mind the saying attributed to George Box: "All models are wrong, but some are useful." Do whatever is needed to make the visualization useful and effective.

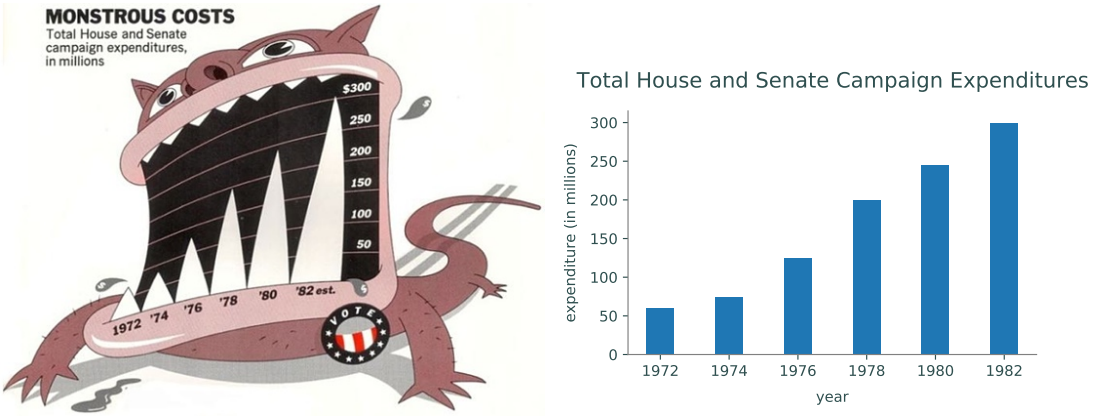


Figure 8.8: Chartjunk refers to anything that does not communicate data. In the image on the left, the cartoon monster distorts the bar chart and manipulates the feelings of the viewer to think negatively about the results. The image on the right shows the same data without chartjunk, making it simple and very easy to interpret the data objectively.

Good visualizations are as simple as possible and no simpler. Edward Tufte coined the term *chartjunk* to mean anything (pictures, icons, colors, and text) that does not represent data or is distracting. Though chartjunk might appear to make data graphics more memorable than plain visualizations, **it is more important to be clear and precise in order to prevent misinterpretation**. The physicist Richard Feynman said, "For a successful technology, reality must take precedence over public relations, for Nature cannot be fooled." Remove chartjunk and anything that prevents the viewer from objectively interpreting the data.

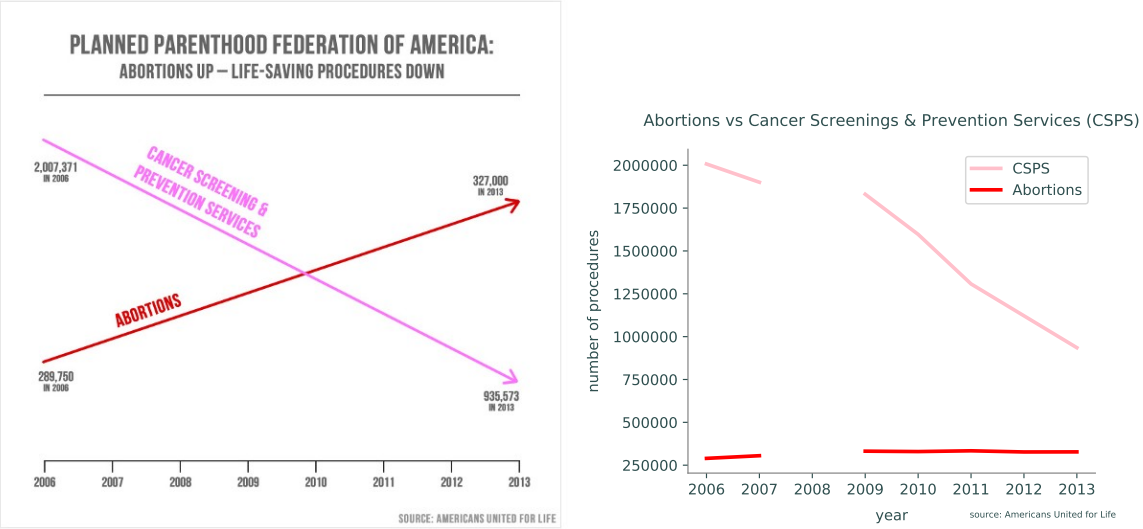


Figure 8.9: The chart on the left is an example of a dishonest graphic shown at a United States congressional hearing in 2015. The chart on the right shows a more accurate representation of the data by showing the y-axis and revealing the missing data from 2008. Source: PolitiFact.

Visualizations should be honest. Figure 8.9 shows how visualizations can be dishonest. The misleading graphic on the left was used as evidence in a United States congressional hearing in 2015. With the y-axis completely removed, it is easy to miss that each line is shown on a different y-axis even though they are measured in the same units. Furthermore, the chart fails to indicate that data is missing from the year 2008. The graphic on the right shows a more accurate representation of the data.[2]

Never use data visualizations to deceive or manipulate. Always present information on who created it, where the data came from, how it was collected, whether it was cleaned or transformed, and whether there are conflicts of interest or possible biases present. Use specific titles and axis labels, and include units of measure. Choose an appropriate window size and use a legend or other annotations where appropriate.

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"]
```

[2]For more information about this graphic, visit `http://www.politifact.com/truth-o-meter/statements/2015/oct/01/jason-chaffetz/chart-shown-planned-parenthood-hearing-misleading-/`.

Visualize this data set with at least four plots, using at least one scatter plot, one histogram, and one bar chart. List 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)`, `https://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_population`, and `http://www.averageheight.co/`.

For more about data visualization, we recommend the following books and websites.

- *How to Lie with Statistics* by Darrell Huff (1954).
- *The Visual Display of Quantitative Information* by Edward Tufte (2nd edition).
- *Visual Explanations* by Edward Tufte.
- *Envisioning Information* by Edward Tufte.
- *Beautiful Evidence* by Edward Tufte.
- *The Functional Art* by Alberto Cairo.
- *Visualization Analysis and Design* by Tamara Munzner.
- *Designing New Default Colormaps*: `https://bids.github.io/colormap/`.

9 Profiling

Lab Objective: *Efficiency is essential to algorithmic programming. Profiling is the process of measuring the complexity and efficiency of a program, allowing the programmer to see what parts of the code need to be optimized. In this lab we present common techniques for speeding up Python code, including the built-in profiler and the Numba module.*

Magic Commands in IPython

IPython has tools for quickly timing and profiling code. These "magic commands" start with one or two `%` characters—one for testing a single line of code, and two for testing a block of code.

- `%time`: Execute some code and print out its execution time.
- `%timeit`: Execute some code several times and print out the average execution time.
- `%prun`: Run a statement through the Python code profiler,[1] printing the number of function calls and the time each takes. We will demonstrate this tool a little later.

```
# Time the construction of a list using list comprehension.
In [1]: %time x = [i**2 for i in range(int(1e5))]
CPU times: user 36.3 ms, sys: 3.28 ms, total: 39.6 ms
Wall time: 40.9 ms

# Time the same list construction, but with a regular for loop.
In [2]: %%time                      # Use a double %% to time a block of code.
   ...: x = []
   ...: for i in range(int(1e5)):
   ...:     x.append(i**2)
   ...:
CPU times: user 50 ms, sys: 2.79 ms, total: 52.8 ms
Wall time: 55.2 ms                  # The list comprehension is faster!
```

[1] `%prun` is a shortcut for `cProfile.run()`; see `https://docs.python.org/3/library/profile.html` for details.

Choosing Faster Algorithms

The best way to speed up a program is to use an efficient algorithm. A bad algorithm, even when implemented well, is never an adequate substitute for a good algorithm.

Problem 1. This problem comes from `https://projecteuler.net` (problems 18 and 67).

By starting at the top of the triangle below and moving to adjacent numbers on the row below, the maximum total from top to bottom is 23.

$$\begin{array}{c} \mathbf{3} \\ \mathbf{7}\ 4 \\ 2\ \mathbf{4}\ 6 \\ 8\ 5\ \mathbf{9}\ 3 \end{array}$$

That is, $3 + 7 + 4 + 9 = 23$.

The following function finds the maximum path sum of the triangle in `triangle.txt` by recursively computing the sum of every possible path—the "brute force" approach.

```
def max_path(filename="triangle.txt"):
    """Find the maximum vertical path in a triangle of values."""
    with open(filename, 'r') as infile:
        data = [[int(n) for n in line.split()]
                        for line in infile.readlines()]
    def path_sum(r, c, total):
        """Recursively compute the max sum of the path starting in row r
        and column c, given the current total.
        """
        total += data[r][c]
        if r == len(data) - 1:      # Base case.
            return total
        else:                       # Recursive case.
            return max(path_sum(r+1, c,   total), # Next row, same column.
                       path_sum(r+1, c+1, total)) # Next row, next column.

    return path_sum(0, 0, 0)        # Start the recursion from the top.
```

The data in `triangle.txt` contains 15 rows and hence 16384 paths, so it is possible to solve this problem by trying every route. However, for a triangle with 100 rows, there are 2^{99} paths to check, which would take billions of years to compute even for a program that could check one trillion routes per second. No amount of improvement to `max_path()` can make it run in an acceptable amount of time on such a triangle—we need a different algorithm.

Write a function that accepts a filename containing a triangle of integers. Compute the largest path sum with the following strategy: starting from the next to last row of the triangle, replace each entry with the sum of the current entry and the greater of the two "child entries." Continue this replacement up through the entire triangle. The top entry in the triangle will be the maximum path sum. In other words, work from the bottom instead of from the top.

$$\begin{array}{c} 3 \\ 7\ 4 \\ 2\ 4\ 6 \\ 8\ 5\ 9\ 3 \end{array} \longrightarrow \begin{array}{c} 3 \\ 7\ 4 \\ 10\ 13\ 15 \\ 8\ 5\ 9\ 3 \end{array} \longrightarrow \begin{array}{c} 3 \\ 20\ 19 \\ 10\ 13\ 15 \\ 8\ 5\ 9\ 3 \end{array} \longrightarrow \begin{array}{c} \mathbf{23} \\ 20\ 19 \\ 10\ 13\ 15 \\ 8\ 5\ 9\ 3 \end{array}$$

Use your function to find the maximum path sum of the 100-row triangle stored in `triangle_large.txt`. Make sure that your new function still gets the correct answer for the smaller `triangle.txt`. Finally, use `%time` or `%timeit` to time both functions on `triangle.txt`. Your new function should be about 100 times faster than the original.

The Profiler

The profiling command `%prun` lists the functions that are called during the execution of a piece of code, along with the following information.

Heading	Description
`primitive calls`	The number of calls that were not caused by recursion.
`ncalls`	The number of calls to the function. If recursion occurs, the output is `<total number of calls>/<number of primitive calls>`.
`tottime`	The amount of time spent in the function, not including calls to other functions.
`percall`	The amount of time spent in each call of the function.
`cumtime`	The amount of time spent in the function, including calls to other functions.

```
# Profile the original function from Problem 1.
In[3]: %prun max_path("triangle.txt")
```

```
         81947 function calls (49181 primitive calls) in 0.036 seconds
   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  32767/1    0.025    0.000    0.034    0.034 profiling.py:18(path_sum)
    16383    0.005    0.000    0.005    0.000 {built-in method builtins.max}
    32767    0.003    0.000    0.003    0.000 {built-in method builtins.len}
        1    0.002    0.002    0.002    0.002 {method 'readlines' of '_io._IOBase' objects}
        1    0.000    0.000    0.000    0.000 {built-in method io.open}
        1    0.000    0.000    0.036    0.036 profiling.py:12(max_path)
        1    0.000    0.000    0.000    0.000 profiling.py:15(<listcomp>)
        1    0.000    0.000    0.036    0.036 {built-in method builtins.exec}
        2    0.000    0.000    0.000    0.000 codecs.py:318(decode)
        1    0.000    0.000    0.036    0.036 <string>:1(<module>)
       15    0.000    0.000    0.000    0.000 {method 'split' of 'str' objects}
        1    0.000    0.000    0.000    0.000 _bootlocale.py:23(getpreferredencoding)
        2    0.000    0.000    0.000    0.000 {built-in method _codecs.utf_8_decode}
        1    0.000    0.000    0.000    0.000 {built-in method _locale.nl_langinfo}
        1    0.000    0.000    0.000    0.000 codecs.py:259(__init__)
        1    0.000    0.000    0.000    0.000 codecs.py:308(__init__)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Optimizing Python Code

A poor implementation of a good algorithm is better than a good implementation of a bad algorithm, but clumsy implementation can still cripple a program's efficiency. The following are a few important practices for speeding up a Python program. Remember, however, that such improvements are futile if the algorithm is poorly suited for the problem.

Avoid Repetition

A clean program does no more work than is necessary. The `ncalls` column of the profiler output is especially useful for identifying parts of a program that might be repetitive. For example, the profile of `max_path()` indicates that `len()` was called 32,767 times—exactly as many times as `path_sum()`. This is an easy fix: save `len(data)` as a variable somewhere outside of `path_sum()`.

```
In [4]: def max_path_clean(filename="triangle.txt"):
   ...:     with open(filename, 'r') as infile:
   ...:         data = [[int(n) for n in line.split()]
   ...:                         for line in infile.readlines()]
   ...:     N = len(data)       # Calculate len(data) outside of path_sum().
   ...:     def path_sum(r, c, total):
   ...:         total += data[r][c]
   ...:         if r == N - 1:  # Use N instead of len(data).
   ...:             return total
   ...:         else:
   ...:             return max(path_sum(r+1, c,   total),
   ...:                        path_sum(r+1, c+1, total))
   ...:     return path_sum(0, 0, 0)
   ...:
In [5]: %prun max_path_clean("triangle.txt")
```

```
         49181 function calls (16415 primitive calls) in 0.026 seconds
   Ordered by: internal time

    ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  32767/1    0.020    0.000    0.025    0.025 <ipython-input-5-9e8c48bb1aba>:6(path_sum)
    16383    0.005    0.000    0.005    0.000 {built-in method builtins.max}
        1    0.002    0.002    0.002    0.002 {method 'readlines' of '_io._IOBase' objects}
        1    0.000    0.000    0.000    0.000 {built-in method io.open}
        1    0.000    0.000    0.026    0.026 <ipython-input-5-9e8c48bb1aba>:1(max_path_clean)
        1    0.000    0.000    0.000    0.000 <ipython-input-5-9e8c48bb1aba>:3(<listcomp>)
        1    0.000    0.000    0.027    0.027 {built-in method builtins.exec}
       15    0.000    0.000    0.000    0.000 {method 'split' of 'str' objects}
        1    0.000    0.000    0.027    0.027 <string>:1(<module>)
        2    0.000    0.000    0.000    0.000 codecs.py:318(decode)
        1    0.000    0.000    0.000    0.000 _bootlocale.py:23(getpreferredencoding)
        2    0.000    0.000    0.000    0.000 {built-in method _codecs.utf_8_decode}
        1    0.000    0.000    0.000    0.000 {built-in method _locale.nl_langinfo}
        1    0.000    0.000    0.000    0.000 codecs.py:308(__init__)
        1    0.000    0.000    0.000    0.000 codecs.py:259(__init__)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Note that the total number of primitive function calls decreased from 49,181 to 16,415. Using `%timeit` also shows that the run time decreased by about 15%. Moving code outside of a loop or an often-used function usually results in a similar speedup.

Another important way of reducing repetition is carefully controlling loop conditions to avoid unnecessary iterations. Consider the problem of identifying Pythagorean triples, sets of three distinct integers $a < b < c$ such that $a^2 + b^2 = c^2$. The following function identifies all such triples where each term is less than a parameter N by checking all possible triples.

```
>>> def pythagorean_triples_slow(N):
...     """Compute all pythagorean triples with entries less than N."""
...     triples = []
...     for a in range(1, N):               # Try values of a from 1 to N-1.
...         for b in range(1, N):           # Try values of b from 1 to N-1.
...             for c in range(1, N):       # Try values of c from 1 to N-1.
...                 if a**2 + b**2 == c**2 and a < b < c:
...                     triples.append((a, b, c))
...     return triples
...
```

Since $a < b < c$ by definition, any computations where $b \leq a$ or $c \leq b$ are unnecessary. Additionally, once a and b are chosen, c can be no greater than $\sqrt{a^2 + b^2}$. The following function changes the loop conditions to avoid these cases and takes care to only compute $a^2 + b^2$ once for each unique pairing (a, b).

```
>>> from math import sqrt
>>> def pythagorean_triples_fast(N):
...     """Compute all pythagorean triples with entries less than N."""
...     triples = []
...     for a in range(1, N):               # Try values of a from 1 to N-1.
...         for b in range(a+1, N):         # Try values of b from a+1 to N-1.
...             _sum = a**2 + b**2
...             for c in range(b+1, min(int(sqrt(_sum))+1, N)):
...                 if _sum == c**2:
...                     triples.append((a, b, c))
...     return triples
...
```

These improvements have a drastic impact on run time, even though the main approach—checking by brute force—is the same.

```
In [6]: %time triples = pythagorean_triples_slow(500)
CPU times: user 1min 51s, sys: 389 ms, total: 1min 51s
Wall time: 1min 52s         # 112 seconds.

In [7]: %time triples = pythagorean_triples_fast(500)
CPU times: user 1.56 s, sys: 5.38 ms, total: 1.57 s
Wall time: 1.57 s           # 98.6% faster!
```

Problem 2. The following function computes the first N prime numbers.

```
def primes(N):
    """Compute the first N primes."""
    primes_list = []
    current = 2
    while len(primes_list) < N:
        isprime = True
        for i in range(2, current):     # Check for nontrivial divisors.
            if current % i == 0:
                isprime = False
        if isprime:
            primes_list.append(current)
        current += 1
    return primes_list
```

This function takes about 6 minutes to find the first 10,000 primes on a fast computer.

Without significantly modifying the approach, rewrite `primes()` so that it can compute the first 10,000 primes in under 0.1 seconds. Use the following facts to reduce unnecessary iterations.

- A number is not prime if it has one or more divisors other than 1 and itself. (Hint: recall the `break` statement.)
- If $p \nmid n$, then $ap \nmid n$ for any integer a. Also, if $p \mid n$ and $0 < p < n$, then $p \leq \sqrt{n}$.
- Except for 2, primes are always odd.

Your new function should be helpful for solving problem 7 on `https://projecteuler.net`.

Avoid Loops

NumPy routines and built-in functions are often useful for eliminating loops altogether. Consider the simple problem of summing the rows of a matrix, implemented in three ways.

```
>>> def row_sum_awful(A):
...     """Sum the rows of A by iterating through rows and columns."""
...     m, n = A.shape
...     row_totals = np.empty(m)        # Allocate space for the output.
...     for i in range(m):              # For each row...
...         total = 0
...         for j in range(n):          # ...iterate through the columns.
...             total += A[i, j]
...         row_totals[i] = total       # Record the total.
...     return row_totals
...
>>> def row_sum_bad(A):
...     """Sum the rows of A by iterating through rows."""
```

```
...     return np.array([sum(A[i, :]) for i in range(A.shape[0])])
...
>>> def row_sum_fast(A):
...     """Sum the rows of A with NumPy."""
...     return np.sum(A, axis=1)    # Or A.sum(axis=1).
...
```

None of the functions are fundamentally different, but their run times differ dramatically.

```
In [8]: import numpy as np
In [9]: A = np.random.random((10000, 10000))

In [10]: %time rows = row_sum_awful(A)
CPU times: user 22.7 s, sys: 137 ms, total: 22.8 s
Wall time: 23.2 s         # SLOW!

In [11]: %time rows = row_sum_bad(A)
CPU times: user 8.85 s, sys: 15.6 ms, total: 8.87 s
Wall time: 8.89 s         # Slow!

In [12]: %time rows = row_sum_fast(A)
CPU times: user 61.2 ms, sys: 1.3 ms, total: 62.5 ms
Wall time: 64 ms          # Fast!
```

In this experiment, `row_sum_fast()` runs several hundred times faster than `row_sum_awful()`. This is primarily because looping is expensive in Python, but NumPy handles loops in C, which is much quicker. Other NumPy functions like `np.sum()` with an `axis` argument can often be used to eliminate loops in a similar way.

Problem 3. Let A be an $m \times n$ matrix with columns $\mathbf{a}_0, \dots, \mathbf{a}_{n-1}$, and let $\mathbf{x}$ be a vector of length m. The *nearest neighbor problem*[a] is to determine which of the columns of A is "closest" to $\mathbf{x}$ with respect to some norm. That is, we compute

$$\operatorname*{argmin}_j \|\mathbf{a}_j - \mathbf{x}\|.$$

The following function solves this problem naïvely for the usual Euclidean norm.

```
def nearest_column(A, x):
    """Find the index of the column of A that is closest to x."""
    distances = []
    for j in range(A.shape[1]):
        distances.append(np.linalg.norm(A[:, j] - x))
    return np.argmin(distances)
```

Write a new version of this function without any loops or list comprehensions, using array broadcasting and the `axis` keyword in `np.linalg.norm()` to eliminate the existing loop. Try to implement the entire function in a single line.
(Hint: See the NumPy Visual Guide in the Appendix for a refresher on array broadcasting.)

Profile the old and new versions with `%prun` and compare the output. Finally, use `%time` or `%timeit` to verify that your new version runs faster than the original.

[a]The nearest neighbor problem is a common problem in many fields of artificial intelligence. The problem can be solved more efficiently with a k-d tree, a specialized data structure for storing high-dimensional data.

Use Data Structures Correctly

Every data structure has strengths and weaknesses, and choosing the wrong data structure can be costly. Here we consider three ways to avoid problems and use sets, dictionaries, and lists correctly.

- **Membership testing**. The question "is `<value>` a member of `<container>`" is common in numerical algorithms. Sets and dictionaries are implemented in a way that makes this a trivial problem, but lists are not. In other words, the `in` operator is near instantaneous with sets and dictionaries, but not with lists.

```
In [13]: a_list = list(range(int(1e7)))

In [14]: a_set = set(a_list)

In [15]: %timeit 12.5 in a_list
413 ms +- 48.2 ms per loop (mean+-std.dev. of 7 runs, 1 loop each)

In [16]: %timeit 12.5 in a_set
170 ns +- 3.8 ns per loop (mean+-std.dev. of 7 runs, 10000000 loops each)
```

  Looking up dictionary values is also almost immediate. Use dictionaries for storing calculations to be reused, such as mappings between letters and numbers or common function outputs.

- **Construction with comprehension**. Lists, sets, and dictionaries can all be constructed with comprehension syntax. This is slightly faster than building the collection in a loop, and the code is highly readable.

```
# Map the integers to their squares.
In [17]: %%time
   ...: a_dict = {}
   ...: for i in range(1000000):
   ...:     a_dict[i] = i**2
   ...:
CPU times: user 432 ms, sys: 54.4 ms, total: 486 ms
Wall time: 491 ms

In [18]: %time a_dict = {i: i**2 for i in range(1000000)}
CPU times: user 377 ms, sys: 58.9 ms, total: 436 ms
Wall time: 440 ms
```

- **Intelligent iteration**. Unlike looking up dictionary values, indexing into lists takes time. Instead of looping over the indices of a list, loop over the entries themselves. When indices and entries are both needed, use `enumerate()` to get the index and the item simultaneously.

```
In [19]: a_list = list(range(1000000))

In [20]: %%time          # Loop over the indices of the list.
    ...: for i in range(len(a_list)):
    ...:     item = a_list[i]
    ...:
CPU times: user 103 ms, sys: 1.78 ms, total: 105 ms
Wall time: 107 ms

In [21]: %%time          # Loop over the items in the list.
    ...: for item in a_list:
    ...:     _ = item
    ...:
CPU times: user 61.2 ms, sys: 1.31 ms, total: 62.5 ms
Wall time: 62.5 ms      # Almost twice as fast as indexing!
```

Problem 4. This is problem 22 from `https://projecteuler.net`.

Using the rule $A \mapsto 1, B \mapsto 2, \dots, Z \mapsto 26$, the *alphabetical value* of a name is the sum of the digits that correspond to the letters in the name. For example, the alphabetic value of "COLIN" is $3 + 15 + 12 + 9 + 14 = 53$.

The following function reads the file `names.txt`, containing over five-thousand first names, and sorts them in alphabetical order. The *name score* of each name in the resulting list is the alphabetic value of the name multiplied by the name's position in the list, starting at 1. "COLIN" is the 938th name alphabetically, so its name score is $938 \times 53 = 49714$. The function returns the total of all the name scores in the file.

```
def name_scores(filename="names.txt"):
    """Find the total of the name scores in the given file."""
    with open(filename, 'r') as infile:
        names = sorted(infile.read().replace('"', '').split(','))
    total = 0
    for i in range(len(names)):
        name_value = 0
        for j in range(len(names[i])):
            alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
            for k in range(len(alphabet)):
                if names[i][j] == alphabet[k]:
                    letter_value = k + 1
            name_value += letter_value
        total += (names.index(names[i]) + 1) * name_value
    return total
```

Rewrite this function—removing repetition, eliminating loops, and using data structures correctly—so that it runs in less than 10 milliseconds on average.

Use Generators

A *generator* is an iterator that yields multiple values, one at a time, as opposed to returning a single value. For example, `range()` is a generator. Using generators appropriately can reduce both the run time and the spatial complexity of a routine. Consider the following function, which constructs a list containing the entries of the sequence $\{x_n\}_{n=1}^{N}$ where $x_n = x_{n-1} + n$ with $x_1 = 1$.

```
>>> def sequence_function(N):
...     """Return the first N entries of the sequence x_n = x_{n-1} + n."""
...     sequence = []
...     x = 0
...     for n in range(1, N+1):
...         x += n
...         sequence.append(x)
...     return sequence
...
>>> sequence_function(10)
[1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
```

A potential problem with this function is that all of the values in the list are computed before anything is returned. This can be a big issue if the parameter N is large. A generator, on the other hand, *yields* one value at a time, indicated by the keyword `yield` (instead of `return`). When the generator is asked for the next entry, the code resumes right where it left off.

```
>>> def sequence_generator(N):
...     """Yield the first N entries of the sequence x_n = x_{n-1} + n."""
...     x = 0
...     for n in range(1, N+1):
...         x += n
...         yield x         # "return" a single value.
...
# Get the entries of the generator one at a time with next().
>>> generated = sequence_generator(10)
>>> next(generated)
1
>>> next(generated)
3
>>> next(generated)
6

# Put each of the generated items in a list, as in sequence_function().
>>> list(sequence_generator(10))    # Or [i for i in sequence_generator(10)].
[1, 3, 6, 10, 15, 21, 28, 36, 45, 55]

# Use the generator in a for loop, like range().
```

```
>>> for entry in sequence_generator(10):
...     print(entry, end=' ')
...
1 3 6 10 15 21 28 36 45 55
```

Many generators, like `range()` and `sequence_generator()`, only yield a finite number of values. However, generators can also continue yielding indefinitely. For example, the following generator yields the terms of $\{x_n\}_{n=1}^{\infty}$ forever. In this case, using `enumerate()` with the generator is helpful for tracking the index n as well as the entry x_n.

```
>>> def sequence_generator_forever():
...     """Yield the sequence x_n = x_{n-1} + n forever."""
...     x = 0
...     n = 1
...     while True:
...         x += n
...         n += 1
...         yield x         # "return" a single value.
...

# Sum the entries of the sequence until the sum exceeds 1000.
>>> total = 0
>>> for i, x in enumerate(sequence_generator_forever()):
...     total += x
...     if total > 1000:
...         print(i)        # Print the index where the total exceeds.
...         break           # Break out of the for loop to stop iterating.
...
17

# Check that 18 terms are required (since i starts at 0 but n starts at 1).
>>> print(sum(sequence_generator(17)), sum(sequence_generator(18)))
969 1140
```

Problem 5. This is problem 25 from `https://projecteuler.net`.

The *Fibonacci sequence* is defined by the recurrence relation $F_n = F_{n-1} + F_{n-2}$, where $F_1 = F_2 = 1$. The 12th term, $F_{12} = 144$, is the first term to contain three digits.

Write a generator that yields the terms of the Fibonacci sequence indefinitely. Next, write a function that accepts an integer N. Use your generator to find the first term in the Fibonacci sequence that contains N digits. Return the index of this term.
(Hint: a generator can have more than one `yield` statement.)

Problem 6. The function in Problem 2 could be turned into a prime number generator that yields primes indefinitely, but it is not the only strategy for yielding primes. The *Sieve of Eratosthenes*[a] is a faster technique for finding all of the primes below a certain number.

1. Given a cap N, start with all of the integers from 2 to N.
2. Remove all integers that are divisible by the first entry in the list.
3. Yield the first entry in the list and remove it from the list.
4. Return to step 2 until the list is empty.

Write a generator that accepts an integer N and that yields all primes (in order, one at a time) that are less than N using the Sieve of Eratosthenes. Your generator should be able to find all primes less than 100,000 in under 5 seconds.

Your generator and your fast function from Problem 2 may be helpful in solving problems 10, 35, 37, 41, 49, and 50 (for starters) of `https://projecteuler.net`.

[a] See `https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`.

Numba

Python code is simpler and more readable than many languages, but Python is also generally much slower than compiled languages like C. The `numba` module bridges the gap by using *just-in-time* (JIT) compilation to optimize code, meaning that the code is actually compiled right before execution.

```
>>> from numba import jit

>>> @jit                # Decorate a function with @jit to use Numba.
... def row_sum_numba(A):
...     """Sum the rows of A by iterating through rows and columns,
...     optimized by Numba.
...     """
...     m, n = A.shape
...     row_totals = np.empty(m)
...     for i in range(m):
...         total = 0
...         for j in range(n):
...             total += A[i, j]
...         row_totals[i] = total
...     return row_totals
```

Python is a *dynamically typed* language, meaning variables are not defined explicitly with a datatype (`x = 6` as opposed to `int x = 6`). This particular aspect of Python makes it flexible, easy to use, and slow. Numba speeds up Python code primarily by assigning datatypes to all the variables. Rather than requiring explicit definitions for datatypes, Numba attempts to infer the correct datatypes based on the datatypes of the input. In `row_sum_numba()`, if `A` is an array of integers, Numba will infer that `total` should also be an integer. On the other hand, if `A` is an array of floats, Numba will infer that `total` should be a *double* (a similar datatype to float in C).

Once all datatypes have been inferred and assigned, the original Python code is translated to machine code. Numba caches this compiled version of code for later use. The first function call takes the time to compile and then execute the code, but subsequent calls use the already-compiled code.

```
In [22]: A = np.random.random((10000, 10000))

# The first function call takes a little extra time to compile first.
In [23]: %time rows = row_sum_numba(A)
CPU times: user 408 ms, sys: 11.5 ms, total: 420 ms
Wall time: 425 ms

# Subsequent calls are consistently faster that the first call.
In [24]: %timeit row_sum_numba(A)
138 ms +- 1.96 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

Note that the only difference between `row_sum_numba()` and `row_sum_awful()` from a few pages ago is the `@jit` decorator, and yet the Numba version is about 99% faster than the original!

The inference engine within Numba does a good job, but it's not always perfect. Adding the keyword argument `nopython=True` to the `@jit` decorator raises an error if Numba is unable to convert each variable to explicit datatypes. The `inspect_types()` method can also be used to check if Numba is using the desired types.

```
# Run the function once first so that it compiles.
>>> rows = row_sum_numba(np.random.random((10, 10)))
>>> row_sum_numba.inspect_types()
# The output is very long and detailed.
```

Alternatively, datatypes can be specified explicitly in the `@jit` decorator as a dictionary via the `locals` keyword argument. Each of the desired datatypes must also be imported from Numba.

```
>>> from numba import int64, double

>>> @jit(nopython=True, locals=dict(A=double[:, :], m=int64, n=int64,
...                                 row_totals=double[:], total=double))
... def row_sum_numba(A):           # 'A' is a 2-D array of doubles.
...     m, n = A.shape               # 'm' and 'n' are both integers.
...     row_totals = np.empty(m)    # 'row_totals' is a 1-D array of doubles.
...     for i in range(m):
...         total = 0               # 'total' is a double.
...         for j in range(n):
...             total += A[i, j]
...         row_totals[i] = total
...     return row_totals
...
```

While it sometimes results in a speed boost, there is a caveat to specifying the datatypes: `row_sum_numba()` no longer accepts arrays that contain anything other than floats. When datatypes are not specified, Numba compiles a new version of the function each time the function is called with a different kind of input. Each compiled version is saved, so the function can still be used flexibly.

Problem 7. The following function calculates the nth power of an $m \times m$ matrix A.

```
def matrix_power(A, n):
    """Compute A^n, the n-th power of the matrix A."""
    product = A.copy()
    temporary_array = np.empty_like(A[0])
    m = A.shape[0]
    for power in range(1, n):
        for i in range(m):
            for j in range(m):
                total = 0
                for k in range(m):
                    total += product[i, k] * A[k, j]
                temporary_array[j] = total
            product[i] = temporary_array
    return product
```

1. Write a Numba-enhanced version of `matrix_power()` called `matrix_power_numba()`.
2. Write a function that accepts an integer n. Run `matrix_power_numba()` once with a small random input so it compiles. Then, for $m = 2^2, 2^3, \dots, 2^7$,
   (a) Generate a random $m \times m$ matrix A with `np.random.random()`.
   (b) Time (separately) `matrix_power()`, `matrix_power_numba()`, and NumPy's `np.linalg.matrix_power()` on A with the specified value of n. (If you are unfamiliar with timing code inside of a function, see the Additional Material section on timing code.)

   Plot the times against the size m on a log-log plot with a base 2 scale (use `plt.loglog()`).

With $n = 10$, the plot should show that the Numba and NumPy versions far outperform the pure Python implementation, with NumPy eventually becoming faster than Numba.

Achtung!

Optimizing code is an important skill, but it is also important to know when to refrain from optimization. The best approach to coding is to write unit tests, implement a solution that works, test and time that solution, **then** (and only then) optimize the solution with profiling techniques. As always, the most important part of the process is choosing the correct algorithm to solve the problem. Don't waste time optimizing a poor algorithm.

Additional Material

Other Timing Techniques

Though %time and %timeit are convenient and work well, some problems require more control for measuring execution time. The usual way of timing a code snippet by hand is via the `time` module (which %time uses). The function `time.time()` returns the number of seconds since the Epoch[2]; to time code, measure the number of seconds before the code runs, the number of seconds after the code runs, and take the difference.

```
>>> import time

>>> start = time.time()              # Record the current time.
>>> for i in range(int(1e8)):        # Execute some code.
...     pass
... end = time.time()                # Record the time again.
... print(end - start)               # Take the difference.
...
4.20402193069458 # (seconds)
```

The `timeit` module (which %timeit uses) has tools for running code snippets several times. The code is passed in as a string, as well as any setup code to be run before starting the clock.

```
>>> import timeit

>>> timeit.timeit("for i in range(N): pass", setup="N = int(1e6)", number=200)
4.884839255013503       # Total time in seconds to run the code 200 times.
>>> _ / 200
0.024424196275067516    # Average time in seconds.
```

The primary advantages of these techniques are the ability automate timing code and being able save the results. For more documentation, see `https://docs.python.org/3.6/library/time.html` and `https://docs.python.org/3.6/library/timeit.html`.

Customizing the Profiler

The output from %prun is generally long, but it can be customized with the following options.

Option	Description
`-l <limit>`	Include a limited number of lines in the output.
`-s <key>`	Sort the output by call count, cumulative time, function name, etc.
`-T <filename>`	Save profile results to a file (results are still printed).

For example, %prun `-l 3 -s ncalls -T path_profile.txt max_path()` generates a profile of `max_path()` that lists the 3 functions with the most calls, then write the results to `path_profile.txt`. See `http://ipython.readthedocs.io/en/stable/interactive/magics.html#magic-prun` for more details.

[2]See `https://en.wikipedia.org/wiki/Epoch_(reference_date)#Computing`.

10 Introduction to SymPy

Lab Objective: *Most implementations of numerical algorithms focus on crunching, relating, or visualizing numerical data. However, it is sometimes convenient or necessary to represent parts of an algorithm symbolically. The SymPy module provides a way to do symbolic mathematics in Python, including algebra, differentiation, integration, and more. In this lab we introduce SymPy syntax and emphasize how to use symbolic algebra for numerical computing.*

Symbolic Variables and Expressions

Most variables in Python refer to a number, string, or data structure. Doing computations on such variables results in more numbers, strings, or data structures. A *symbolic variable* is a variable that represents a mathematical symbol, such as x or θ, not a number or another kind of data. Operating on symbolic variables results in an *expression*, representative of an actual mathematical expression. For example, if a symbolic variable `Y` refers to a mathematical variable y, the multiplication `3*Y` refers to the expression $3y$. This is all done without assigning an actual numerical value to `Y`.

SymPy [MSP+17] is Python's library for doing symbolic algebra and calculus. It is typically imported with `import sympy as sy`, and symbolic variables are usually defined using `sy.symbols()`.

```
>>> import sympy as sy
>>> x0 = sy.symbols('x0')                    # Define a single variable.

# Define multiple symbolic variables simultaneously.
>>> x2, x3 = sy.symbols('x2, x3')            # Separate symbols by commas,
>>> m, a = sy.symbols('mass acceleration')   # by spaces,
>>> x, y, z = sy.symbols('x:z')              # or by colons.
>>> x4, x5, x6 = sy.symbols('x4:7')

# Combine symbolic variables to form expressions.
>>> expr = x**2 + x*y + 3*x*y + 4*y**3
>>> force = m * a
>>> print(expr, force, sep='\n')
x**2 + 4*x*y + 4*y**3
acceleration*mass
```

SymPy has its own version for each of the standard mathematical functions like $\sin(x)$, $\log(x)$, and $\sqrt{x}$, and includes predefined variables for special numbers such as π. The naming conventions for most functions match NumPy, but some of the built-in constants are named slightly differently.

Functions	$\sin(x)$	$\arcsin(x)$	$\sinh(x)$	e^x	$\log(x)$	$\sqrt{x}$
	`sy.sin()`	`sy.asin()`	`sy.sinh()`	`sy.exp()`	`sy.log()`	`sy.sqrt()`

Constants	π	e	$i = \sqrt{-1}$	∞
	`sy.pi`	`sy.E`	`sy.I`	`sy.oo`

Other trigonometric functions like $\cos(x)$ follow the same naming conventions. For a complete list of SymPy functions, see `http://docs.sympy.org/latest/modules/functions/index.html`.

Achtung!

Always use SymPy functions and constants when creating expressions instead of using NumPy's functions and constants. Later we will show how to make NumPy and SymPy cooperate.

```
>>> import numpy as np

>>> x = sy.symbols('x')
>>> np.exp(x)                          # Try to use NumPy to represent e**x.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Symbol' object has no attribute 'exp'

>>> sy.exp(x)                          # Use SymPy's version instead.
exp(x)
```

Note

SymPy defines its own numeric types for integers, floats, and rational numbers. For example, the `sy.Rational` class is similar to the standard library's `fractions.Fraction` class, and should be used to represent fractions in SymPy expressions.

```
>>> x = sy.symbols('x')
>>> (2/3) * sy.sin(x)                  # 2/3 returns a float, not a rational.
0.666666666666667*sin(x)

>>> sy.Rational(2, 3) * sy.sin(x)      # Keep 2/3 symbolic.
2*sin(x)/3
```

Always be aware of which numeric types are being used in an expression. Using rationals and integers where possible is important for simplifying expressions.

Problem 1. Write a function that returns the expression $\frac{2}{5}e^{x^2-y}\cosh(x+y) + \frac{3}{7}\log(xy+1)$ symbolically. Make sure that the fractions remain symbolic.

Sums and Products

Expressions that can be written as a sum or a product can be constructed with `sy.summation()` or `sy.product()`, respectively. Each of these functions accepts an expression that represents one term of the sum or product, then a tuple indicating the indexing variable and which values it should take on. For example, the following code constructs the sum and product given below.

$$\sum_{i=1}^{4} x + iy \qquad\qquad \prod_{i=0}^{5} x + iy$$

```
>>> x, y, i = sy.symbols('x y i')

>>> sy.summation(x + i*y, (i, 1, 4))    # Sum over i=1,2,3,4.
4*x + 10*y

>>> sy.product(x + i*y, (i, 0, 5))      # Multiply over i=0,1,2,3,4,5.
x*(x + y)*(x + 2*y)*(x + 3*y)*(x + 4*y)*(x + 5*y)
```

Simplifying Expressions

The expressions for the summation and product in the previous example are automatically simplified. More complicated expressions can be simplified with one or more of the following functions.

Function	Description
`sy.cancel()`	Cancel common factors in the numerator and denominator.
`sy.expand()`	Expand a factored expression.
`sy.factor()`	Factor an expanded expression.
`sy.radsimp()`	Rationalize the denominator of an expression.
`sy.simplify()`	Simplify an expression.
`sy.trigsimp()`	Simplify only the trigonometric parts of the expression.

```
>>> x = sy.symbols('x')
>>> expr = (x**2 + 2*x + 1) / ((x+1)*((sy.sin(x)/sy.cos(x))**2 + 1))
>>> print(expr)
(x**2 + 2*x + 1)/((x + 1)*(sin(x)**2/cos(x)**2 + 1))

>>> sy.simplify(expr)
(x + 1)*cos(x)**2
```

The generic `sy.simplify()` tries to simplify an expression in any possible way. This is often computationally expensive; using more specific simplifiers when possible reduces the cost.

```
>>> expr = sy.product(x + i*y, (i, 0, 3))
>>> print(expr)
x*(x + y)*(x + 2*y)*(x + 3*y)

>>> expr_long = sy.expand(expr)          # Expand the product terms.
>>> print(expr_long)
x**4 + 6*x**3*y + 11*x**2*y**2 + 6*x*y**3

>>> expr_long /= (x + 3*y)
>>> print(expr_long)
(x**4 + 6*x**3*y + 11*x**2*y**2 + 6*x*y**3)/(x + 3*y)

>>> expr_short = sy.cancel(expr_long)    # Cancel out the denominator.
>>> print(expr_short)
x**3 + 3*x**2*y + 2*x*y**2

>>> sy.factor(expr_short)                # Factor the result.
x*(x + y)*(x + 2*y)

# Simplify the trigonometric parts of an expression.
>>> sy.trigsimp(2*sy.sin(x)*sy.cos(x))
sin(2*x)
```

See `http://docs.sympy.org/latest/tutorial/simplification.html` for more examples.

Achtung!

1. Simplifications return new expressions; they do not modify existing expressions in place.
2. The == operator compares two expressions for exact structural equality, not algebraic equivalence. Simplify or expand expressions before comparing them with ==.
3. Expressions containing floats may not simplify as expected. Always use integers and SymPy rationals in expressions when appropriate.

```
>>> expr = 2*sy.sin(x)*sy.cos(x)
>>> sy.trigsimp(expr)
sin(2*x)
>> print(expr)
2*sin(x)*cos(x)                 # The original expression is unchanged.

>>> 2*sy.sin(x)*sy.cos(x) == sy.sin(2*x)
False                           # The two expression structures differ.

>>> sy.factor(x**2.0 - 1)
x**2.0 - 1                      # Factorization fails due to the 2.0.
```

Problem 2. Write a function that symbolically computes and simplifies the following expression.

$$\prod_{i=1}^{5}\sum_{j=i}^{5} j(\sin(x)+\cos(x))$$

Evaluating Expressions

Every SymPy expression has a `subs()` method that substitutes one variable for another. The result is usually still a symbolic expression, even if a numerical value is used in the substitution. The `evalf()` method actually evaluates the expression numerically after all symbolic variables have been assigned a value. Both of these methods can accept a dictionary to reassign multiple symbols simultaneously.

```
>>> x,y = sy.symbols('x y')
>>> expr = sy.expand((x + y)**3)
>>> print(expr)
x**3 + 3*x**2*y + 3*x*y**2 + y**3

# Replace the symbolic variable y with the expression 2x.
>>> expr.subs(y, 2*x)
27*x**3

# Replace x with pi and y with 1.
>>> new_expr = expr.subs({x: sy.pi, y: 1})
>>> print(new_expr)
1 + 3*pi + 3*pi**2 + pi**3
>>> new_expr.evalf()                # Numerically evaluate the expression.
71.0398678443373

# Evaluate the expression by providing values for each variable.
>>> expr.evalf(subs={x: 1, y: 2})
27.0000000000000
```

These operations are good for evaluating an expression at a single point, but it is typically more useful to turn the expression into a reusable numerical function. To this end, `sy.lambdify()` takes in a symbolic variable (or list of variables) and an expression, then returns a callable function that corresponds to the expression.

```
# Turn the expression sin(x)^2 into a function with x as the variable.
>>> f = sy.lambdify(x, sy.sin(x)**2)
>>> print(f(0), f(np.pi/2), f(np.pi), sep='  ')
0.0  1.0  1.4997597826618576e-32

# Lambdify a function of several variables.
>>> f = sy.lambdify((x, y), sy.sin(x)**2 + sy.cos(y)**2)
>>> print(f(0, 1), f(1, 0), f(np.pi, np.pi), sep='  ')
0.2919265817264289  1.708073418273571  1.0
```

By default, `sy.lambdify()` uses the `math` module to convert an expression to a function. For example, `sy.sin()` is converted to `math.sin()`. By providing `"numpy"` as an additional argument, `sy.lambdify()` replaces symbolic functions with their NumPy equivalents instead, so `sy.sin()` is converted to `np.sin()`. This allows the resulting function to act element-wise on NumPy arrays, not just on single data points.

```
>>> f = sy.lambdify(x, 2*sy.sin(2*x), "numpy")
>>> f(np.linspace(0, 2*np.pi, 9))   # Evaluate f() at many points.
array([  0.00000000e+00,   2.00000000e+00,   2.44929360e-16,
        -2.00000000e+00,  -4.89858720e-16,   2.00000000e+00,
         7.34788079e-16,  -2.00000000e+00,  -9.79717439e-16])
```

Note

It is almost always computationally cheaper to lambdify a function than to use substitutions. According to the SymPy documentation, using `sy.lambdify()` to do numerical evaluations "takes on the order of hundreds of nanoseconds, roughly two orders of magnitude faster than the `subs()` method."

```
In [1]: import sympy as sy
In [2]: import numpy as np

# Define a symbol, an expression, and points to plug into the expression.
In [3]: x = sy.symbols('x')
In [4]: expr = sy.tanh(x)
In [5]: points = np.random.random(10000)

# Time using evalf() on each of the random points.
In [6]: %time _ = [expr.subs(x, pt).evalf() for pt in points]
CPU times: user 5.29 s, sys: 40.3 ms, total: 5.33 s
Wall time: 5.36 s

# Lambdify the expression and time using the resulting function.
In [7]: f = sy.lambdify(x, expr)
In [8]: %time _ = [f(pt) for pt in points]
CPU times: user 5.39 ms, sys: 648 micros, total: 6.04 ms
Wall time: 7.75 ms      # About 1000 times faster than evalf().

# Lambdify the expression with NumPy and repeat the experiment.
In [9]: f = sy.lambdify(x, expr, "numpy")
In [10]: %time _ = f(points)
CPU times: user 381 micros, sys: 63 micros, total: 444 micros
Wall time: 282 micros   # About 10 times faster than regular lambdify.
```

Problem 3. The Maclaurin series up to order N for e^x is defined as

$$e^x \approx \sum_{n=0}^{N} \frac{x^n}{n!}. \tag{10.1}$$

Write a function that accepts an integer N. Define an expression for (10.1), then substitute in $-y^2$ for x to get a truncated Maclaurin series of e^{-y^2}. Lambdify the resulting expression and plot the series on the domain $y \in [-2, 2]$. Plot e^{-y^2} over the same domain for comparison.
(Hint: use `sy.factorial()` to compute the factorial.)

Call your function with increasing values of N to check that the series converges correctly.

Solving Symbolic Equations

A SymPy expression by itself is not an equation. However, `sy.solve()` equates an expression with zero and solves for a specified variable. In this way, SymPy can be used to solve equations.

```
>>> x,y = sy.symbols('x y')

# Solve x^2 - 2x + 1 = 0 for x.
>>> sy.solve(x**2 - 2*x + 1, x)
[1]                                 # The result is a list of solutions.

# Solve x^2 - 1 = 0 for x.
>>> sy.solve(x**2 - 1, x)
[-1, 1]                             # This equation has two solutions.

# Solutions can also be expressions involving other variables.
>>> sy.solve(x/(y-x) + (x-y)/y, x)
[y*(-sqrt(5) + 3)/2, y*(sqrt(5) + 3)/2]
```

Problem 4. The following equation represents a rose curve in cartesian coordinates:

$$0 = 1 - \frac{(x^2+y^2)^{7/2} + 18x^5y - 60x^3y^3 + 18xy^5}{(x^2+y^2)^3}. \tag{10.2}$$

The curve is not the image of a single function (such a function would fail the vertical line test), so the best way to plot it is to convert (10.2) to a pair of parametric equations that depend on the angle parameter θ.

Construct an expression for the nonzero side of (10.2) and convert it to polar coordinates with the substitutions $x = r\cos(\theta)$ and $y = r\sin(\theta)$. Simplify the result, then solve it for r. There are two solutions due to the presence of an r^2 term; pick one and lambdify it to get a function $r(\theta)$. Use this function to plot $x(\theta) = r(\theta)\cos(\theta)$ against $y(\theta) = r(\theta)\sin(\theta)$ for $\theta \in [0, 2\pi]$.
(Hint: use `sy.Rational()` for the fractional exponent.)

Linear Algebra

Sympy can also solve systems of equations. A system of linear equations $A\mathbf{x} = \mathbf{b}$ is solved in a slightly different way than in NumPy and SciPy: instead of defining the matrix A and the vector $\mathbf{b}$ separately, define the augmented matrix $M = [A \mid \mathbf{b}]$ and call `sy.solve_linear_system()` on M.

SymPy matrices are defined with `sy.Matrix()`, with the same syntax as 2-dimensional NumPy arrays. For example, the following code solves the system given below.

$$\begin{aligned} x &+ y &+ z &= 5 \\ 2x &+ 4y &+ 3z &= 2 \\ 5x &+ 10y &+ 2z &= 4 \end{aligned}$$

```
>>> x, y, z = sy.symbols('x y z')

# Define the augmented matrix M = [A|b].
>>> M = sy.Matrix([[1,  1, 1, 5],
                   [2,  4, 3, 2],
                   [5, 10, 2, 4]])

# Solve the system, providing symbolic variables to solve for.
>>> sy.solve_linear_system(M, x, y, z)
{x: 98/11, y: -45/11, z: 2/11}
```

SymPy matrices support the standard matrix operations of addition `+`, subtraction `-`, and multiplication `@`. Additionally, SymPy matrices are equipped with many useful methods, some of which are listed below. See `http://docs.sympy.org/latest/modules/matrices/matrices.html` for more methods and examples.

Method	Returns
`det()`	The determinant.
`eigenvals()`	The eigenvalues and their multiplicities.
`eigenvects()`	The eigenvectors and their corresponding eigenvalues.
`inv()`	The matrix inverse.
`is_nilpotent()`	`True` if the matrix is nilpotent.
`norm()`	The Frobenius, ∞, 1, or 2 norm.
`nullspace()`	The nullspace as a list of vectors.
`rref()`	The reduced row-echelon form.
`singular_values()`	The singular values.

Achtung!

The `*` operator performs matrix multiplication on SymPy matrices. To perform element-wise multiplication, use the `multiply_elementwise()` method instead.

Problem 5. Find the eigenvalues of the following matrix by solving for λ in the characteristic equation $\det(A - \lambda I) = 0$.

$$A = \begin{bmatrix} x-y & x & 0 \\ x & x-y & x \\ 0 & x & x-y \end{bmatrix}$$

Also compute the eigenvectors by solving the linear system $A - \lambda I = \mathbf{0}$ for each eigenvalue λ. Return a dictionary mapping the eigenvalues to their eigenvectors.
(Hint: the `nullspace()` method may be useful.)

Check that $A\mathbf{v} = \lambda\mathbf{v}$ for each eigenvalue-eigenvector pair $(\lambda, \mathbf{v})$. Compare your results to the `eigenvals()` and `eigenvects()` methods for SymPy matrices.

Calculus

SymPy is also equipped to perform standard calculus operations, including derivatives, integrals, and taking limits. Like other elements of SymPy, calculus operations can be temporally expensive, but they give exact solutions whenever solutions exist.

Differentiation

The command `sy.Derivative()` creates a closed form, unevaluated derivative of an expression. This is like putting $\frac{d}{dx}$ in front of an expression without actually calculating the derivative symbolically. The resulting expression has a `doit()` method that can be used to evaluate the actual derivative. Equivalently, `sy.diff()` immediately takes the derivative of an expression.

Both `sy.Derivative()` and `sy.diff()` accept a single expression, then the variable or variables that the derivative is being taken with respect to.

```
>>> x, y = sy.symbols('x y')
>>> f = sy.sin(y)*sy.cos(x)**2

# Make an expression for the derivative of f with respect to x.
>>> df = sy.Derivative(f, x)
>>> print(df)
Derivative(sin(y)*cos(x)**2, x)

>>> df.doit()                          # Perform the actual differentiation.
-2*sin(x)*sin(y)*cos(x)

# Alternatively, calculate the derivative of f in a single step.
>>> sy.diff(f, x)
-2*sin(x)*sin(y)*cos(x)

# Calculate the derivative with respect to x, then y, then x again.
>>> sy.diff(f, x, y, x)
2*(sin(x)**2 - cos(x)**2)*cos(y)    # Note this expression could be simplified.
```

Problem 6. Let $f : \mathbb{R} \to \mathbb{R}$ be a smooth function. A *critical point* of f is a number $x_0 \in \mathbb{R}$ satisfying $f'(x_0) = 0$. The second derivative test states that a critical point x_0 is a local minimum of f if $f''(x_0) > 0$, or a local maximum of f if $f''(x_0) < 0$ (if $f''(x_0) = 0$, the test is inconclusive).

Now consider the polynomial

$$p(x) = 2x^6 - 51x^4 + 48x^3 + 312x^2 - 576x - 100.$$

Use SymPy to find all critical points of p and classify each as a local minimum or a local maximum. Plot $p(x)$ over $x \in [-5, 5]$ and mark each of the minima in one color and the maxima in another color. Return the collections of the x-values corresponding to the local minima and local maxima as two separate sets.

The *Jacobian matrix* of a multivariable function $f : \mathbb{R}^n \to \mathbb{R}^m$ at a point $\mathbf{x}_0 \in \mathbb{R}^n$ is the $m \times n$ matrix J whose entries are given by

$$J_{ij} = \frac{\partial f_i}{\partial x_j}(\mathbf{x}_0).$$

For example, the Jacobian for a function $f : \mathbb{R}^3 \to \mathbb{R}^2$ is defined by

$$J = \left[\begin{array}{c|c|c} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \frac{\partial f}{\partial x_3} \end{array}\right] = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \end{bmatrix}, \quad \text{where} \quad f(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

To calculate the Jacobian matrix of a multivariate function with SymPy, define that function as a symbolic matrix (`sy.Matrix()`) and use its `jacobian()` method. The method requires a list of variables that prescribes the ordering of the differentiation.

```
# Create a matrix of symbolic variables.
>>> r, t = sy.symbols('r theta')
>>> f = sy.Matrix([r*sy.cos(t), r*sy.sin(t)])

# Find the Jacobian matrix of f with respect to r and theta.
>>> J = f.jacobian([r, t])
>>> J
Matrix([
[cos(theta), -r*sin(theta)],
[sin(theta),  r*cos(theta)]])

# Evaluate the Jacobian matrix at the point (1, pi/2).
>>> J.subs({r:1, t:sy.pi/2})
Matrix([
[0, -1],
[1,  0]])

# Calculate the (symbolic) determinant of the Jacobian matrix.
>>> sy.simplify(J.det())
r
```

Integration

The function `sy.Integral()` creates an unevaluated integral expression. This is like putting an integral sign in front of an expression without actually evaluating the integral symbolically or numerically. The resulting expression has a `doit()` method that can be used to evaluate the actual integral. Equivalently, `sy.integrate()` immediately integrates an expression.

Both `sy.Integral()` and `sy.integrate()` accept a single expression, then a tuple or tuples containing the variable of integration and, optionally, the bounds of integration.

```
# Calculate the indefinite integral of sec(x).
>>> sy.integrate(sy.sec(x), x)
-log(sin(x) - 1)/2 + log(sin(x) + 1)/2

# Integrate cos(x)^2 from 0 to pi/2.
>>> sy.integrate(sy.cos(x)**2, (x, 0, sy.pi/2))
pi/4

# Compute the integral of (y^2)(x^2) dx dy with x from 0 to 2, y from -1 to 1.
>>> sy.integrate(y**2 * x**2, (x, 0, 2), (y, -1, 1))
16/9
```

Problem 7. Let $f : \mathbb{R}^3 \to \mathbb{R}$ be a smooth function. The volume integral of f over the sphere S of radius r can written in spherical coordinates as

$$\iiint_S f(x,y,z)dV = \int_0^{\pi}\int_0^{2\pi}\int_0^{r} f(h_1(\rho,\theta,\phi), h_2(\rho,\theta,\phi), h_3(\rho,\theta,\phi))|\det(J)|\, d\rho\, d\theta\, d\phi,$$

where J is the Jacobian of the function $h : \mathbb{R}^3 \to \mathbb{R}^3$ given by

$$h(\rho,\theta,\phi) = \begin{bmatrix} h_1(\rho,\theta,\phi) \\ h_2(\rho,\theta,\phi) \\ h_3(\rho,\theta,\phi) \end{bmatrix} = \begin{bmatrix} \rho\sin(\phi)\cos(\theta) \\ \rho\sin(\phi)\sin(\theta) \\ \rho\cos(\phi) \end{bmatrix}.$$

Calculate the volume integral of $f(x,y,z) = (x^2+y^2+z^2)^2$ over the sphere of radius r. Lambdify the resulting expression (with r as the independent variable) and plot the integral value for $r \in [0,3]$. In addition, return the value of the integral when $r = 2$.
(Hint: simplify the integrand before computing the integral. In this case, $|\det(J)| = -\det(J)$.)

To check your answer, when $r = 3$, the value of the integral is $\frac{8748}{7}\pi$.

Achtung!

SymPy isn't perfect. It solves some integrals incorrectly, simplifies some expressions poorly, and is significantly slower than numerical computations. However, it is generally very useful for simplifying parts of an algorithm, getting exact answers, and handling tedious algebra quickly.

Additional Material

Pretty Printing

SymPy expressions, especially complicated ones, can be hard to read. Calling `sy.init_printing()` changes the way that certain expressions are displayed to be more readable; in a Jupyter Notebook, the rendering is done with LaTeX, as displayed below. Furthermore, the function `sy.latex()` converts an expression into actual LaTeX code for use in other settings.

In [1]:
```
import sympy as sy

sy.init_printing()
```

In [2]:
```
x, y, z, theta = sy.symbols('x y z \\theta')
expr = sy.sin(theta) * sy.exp(y) * sy.log(z) * (x + y*theta)**4
I = sy.Integral(expr, (x,0,2), (y,-1,1), (z,1,sy.pi))
dI = sy.Derivative(I, theta)

dI
```

Out[2]:
$$\frac{d}{d\theta}\int_{1}^{\pi}\int_{-1}^{1}\int_{0}^{2}(\theta y + x)^4 e^y \log(z)\sin(\theta)\,dx\,dy\,dz$$

Limits

Limits can be expressed, similar to derivatives or integrals, with `sy.Limit()`. Alternatively, `sy.limit()` (lowercase) evaluates a limit directly.

```
# Define the limit of a^(1/x) as x approaches infinity.
>>> a, x = sy.symbols('a x')
>>> sy.Limit(a**(1/x), x, sy.oo)
Limit(a**(1/x), x, oo, dir='-')

# Use the doit() method or sy.limit() to evaluate a limit.
>>> sy.limit((1+x)**(1/x), x, 0)
E

# Evaluate a limit as x approaches 0 from the negative direction.
>>> sy.limit(1/x, x, 0, '-')
-oo
```

Use limits instead of the `subs()` method when the value to be substituted is ∞ or is a singularity.

```
>>> expr = x / 2**x
>>> expr.subs(x, sy.oo)
nan
>>> sy.limit(expr, x, sy.oo)
0
```

Refer to `http://docs.sympy.org/latest/tutorial/calculus.html` for SymPy's official documentation on calculus operations.

Numerical Integration

Many integrals cannot be solved analytically. As an alternative to the `doit()` method, the `as_sum()` method approximates the integral with a summation. This method accepts the number of terms to use and a string indicating which approximation rule to use (`"left"`, `"right"`, `"midpoint"`, or `"trapezoid"`).

```
>>> x = sy.symbols('x')

# Try integrating e^(x^2) from 0 to pi.
>>> I = sy.Integral(sy.exp(x**2), (x, 0, sy.pi))
>>> I.doit()
sqrt(pi)*erfi(pi)/2                # The result is not very helpful.

# Instead, approximate the integral with a sum.
>>> I.as_sum(10, 'left').evalf()
1162.85031639195
```

See `http://docs.sympy.org/latest/modules/integrals/integrals.html` for more documentation on integration with SymPy.

Differential Equations

SymPy can be used to solve both ordinary and partial differential equations. The documentation for working with PDE functions is at `http://docs.sympy.org/dev/modules/solvers/pde.html`

The general form of a first-order differential equation is $\frac{dx}{dt} = f(x(t), t)$. To represent the unknown function $x(t)$, use `sy.Function()`. Just as `sy.solve()` is used to solve an expression for a given **variable**, `sy.dsolve()` solves an ODE for a particular **function**. When there are multiple solutions, `sy.dsolve()` returns a list; when arbitrary constants are involved they are given as `C1`, `C2`, and so on. Use `sy.checkodesol()` to check that a function is a solution to a differential equation.

```
>>> t = sy.symbols('t')
>>> x = sy.Function('x')

# Solve the equation x''(t) - 2x'(t) + x(t) = sin(t).
>>> ode = x(t).diff(t, t) - 2*x(t).diff(t) + x(t) - sy.sin(t)
>>> sy.dsolve(ode, x(t))
Eq(x(t), (C1 + C2*t)*exp(t) + cos(t)/2) # C1 and C2 are arbitrary constants.
```

Since there are many types of ODEs, `sy.dsolve()` may also take a hint indicating what solving strategy to use. See `sy.ode.allhints` for a list of possible hints, or use `sy.classify_ode()` to see the list of hints that may apply to a particular equation.

11 Advanced Numpy

Lab Objective: *NumPy is a vast library with many useful functions that can be easily forgotten if not used and reviewed. This lab will help you remember some of its functionality that you may have forgotten and give you a few new functions to master.*

> **Note**
>
> Some of this lab is review, but there is new material near the end and additional materials beyond that.

Data Access

Array Slicing

Indexing for a 1-D NumPy array uses the slicing syntax `x[start:stop:step]`. If there is no colon, a single entry of that dimension is accessed. With a colon, a range of values is accessed. For multi-dimensional arrays, use a comma to separate slicing syntax for each axis.

```
# 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 elements of the array with slicing syntax.
>>> x[3]                            # The element at index 3.
3
>>> x[:3]                           # Everything up to index 3 (exclusive).
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 1, column 2.
7
>>> A[:, 2:]                        # All of the rows, from column 2 on.
array([[2, 3, 4],
       [7, 8, 9]])
```

> **Note**
>
> Indexing and slicing operations return a *view* of the array. Changing a view of an array also changes the original array. In other words, **arrays are mutable**. To create a copy of an array, use `np.copy()` or the array's `copy()` method. 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.

Fancy Indexing

So-called *fancy indexing* is a second way to access or change the elements of an array. Instead of using slicing syntax, provide either an array of indices or an array of boolean values (called a *mask*) to extract specific elements.

```
>>> x = np.arange(0, 50, 10)        # The integers from 0 to 50 by tens.
>>> 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])     # Get the 3rd, 1st, and 4th elements.
>>> 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]                         # Get the 0th and 3rd entries.
array([0, 30])
```

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

```
>>> y = np.arange(10, 20, 2)        # Every other integers from 10 to 20.
>>> y
array([10, 12, 14, 16, 18])
```

```
# Extract the values of 'y' larger than 15.
>>> mask = y > 15                    # Same as np.array([i > 15 for i in y]).
>>> mask
array([False, False, False, True, True], dtype=bool)
>>> y[mask]                          # Same as y[y > 15]
array([16, 18])

# Change the values of 'y' that are larger than 15 to 100.
>>> y[mask] = 100
>>> print(y)
[10 12 14 100 100]
```

While indexing and slicing always return a view, fancy indexing always returns a copy.

Problem 1. 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 resulting array.

Array Manipulation

Shaping

An array's `shape` attribute describes its dimensions. Use `np.reshape()` or the array's `reshape()` method to give an array a new shape. 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. Using a `-1` in the new shape tuple makes the specified dimension as long as necessary.

```
>>> A = np.arange(12)               # The integers from 0 to 12 (exclusive).
>>> 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 1-D array and `np.transpose()` or the `T` attribute to transpose a 2-D array in the matrix sense.

```
>>> 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. For example, by default an array will 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 to purposefully work well 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]                      # All of the rows, column 1.
>>> x
array([1, 6])                       # Not array([[1],
>>> x.shape                         #            [6]])
(2,)
>>> x.ndim
1
```

However, it is occasionally 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((-1, 1))              # Or x[:,np.newaxis] or np.vstack(x).
array([[0],
```

```
       [1],
       [2]])
```

Do not force a 1-D vector to be a column vector unless necessary.

Stacking

NumPy has functions for *stacking* two or more arrays with similar dimensions into a single block matrix. Each of these methods takes in a single tuple of arrays to be stacked in sequence.

Function	Description
`concatenate()`	Join a sequence of arrays along an existing axis
`vstack()`	Stack arrays in sequence vertically (row wise).
`hstack()`	Stack arrays in sequence horizontally (column wise).
`column_stack()`	Stack 1-D or 2-D arrays horizontally (column wise) into a 2-D array.

```
>>> 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.],            # A
       [3.,  4.,  5.],
       [0.,  0.,  0.],            # B
       [0.,  0.,  0.],
       [0.,  0.,  0.],
       [0.,  0.,  0.],
       [0.,  1.,  2.],            # A
       [3.,  4.,  5.]])

>>> A = A.T
>>> B = B.T

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

# column_stack() stacks arrays horizontally, including 1-D arrays.
>>> np.column_stack((A, np.zeros(3), np.ones(3), np.full(3, 2)))
array([[0.,  3.,  0.,  1.,  2.],
       [1.,  4.,  0.,  1.,  2.],
       [2.,  5.,  0.,  1.,  2.]])
```

See `http://docs.scipy.org/doc/numpy-1.10.1/reference/routines.array-manipulation.html` for more array manipulation routines and documentation.

Working with Dimensions

In many scientific disciplines, arrays of more than 2 dimensions are readily utilized. Numpy's function are designed to work on these larger arrays but sometimes it's necessary to convert arrays to a different dimension. To do this we'll use `np.squeeze()` and `np.dstack`

`np.squeeze()` eliminates any superfluous dimensions in an array. These will be any dimension of value 1 when the shape attribute is called. It does not matter in which position the dimension is located. As a result of this, using `np.squeeze()` on several arrays of different shape can result in arrays of the same shape.

```
# Define 3 arrays of different dimensions
>>> test0 = np.arange(9).reshape(1, 3, 3)
>>> test1 = np.arange(9).reshape(3, 1, 3)
>>> test2 = np.arange(9).reshape(3, 3, 1)

# but all arrays become the same after being squeezed
>>> np.squeeze(test0)
array([[0.,  1.,  2.],
       [3.,  4.,  5.],
       [6.,  7.,  8.]])

>>> np.squeeze(test1)
array([[0.,  1.,  2.],
       [3.,  4.,  5.],
       [6.,  7.,  8.]])

>>> np.squeeze(test2)
array([[0.,  1.,  2.],
       [3.,  4.,  5.],
       [6.,  7.,  8.]])

# even arrays with many extra dimensions reduce to the same matrix
>>> ridiculous = np.arange(9).reshape(1, 1, 1, 1, 3, 3, 1, 1, 1, 1)
>>> np.squeeze(ridiculous)
array([[0.,  1.,  2.],
       [3.,  4.,  5.],
       [6.,  7.,  8.]])

# however arrays that have no 1 value in their shape will remain the same
>>> stoic = np.arange(20).reshape(2, 2, 5)
>>> print(stoic)
array([[[0,  1,  2,  3,  4],
        [5,  6,  7,  8,  9]],

       [[10, 11, 12, 13, 14],
```

```
        [15, 16, 17, 18, 19]]])

>>> print(np.squeeze(stoic))
array([[[0,  1,  2,  3,  4],
        [5,  6,  7,  8,  9]],

       [[10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]]])
```

`np.dstack()`, on the other hand, performs like the other two stacking function except that it stacks on the third dimensions rather than the first or second. For example, using `np.dstack()` on two matrices of shape $(3, 3)$ would make a matrix of shape $(3, 3, 2)$. If a matrix already has three dimensions or more, `np.dstack()` will only affect the third one (i.e, shapes $(3, 3, 2, 2)$ with $(3, 3, 2, 2)$ will create $(3, 3, 4, 2)$)

Problem 2. Write a function that accepts a list of arrays, squeezes them, pads them with 0's on the "right" and "bottom" so that they're all the same dimensions, and then stacks them along the 3rd dimension. Thus, the arrays in the list can be, individually, any size and dimension. However, you may assume the arrays will all be 2-dimensional once the extra dimensions are squeezed out.

Hint: Use the various stacking commands to pad the inputted arrays appropriately with 0's on the "right" and "bottom" so that they can easily be stacked into the three dimensional array in the end. Again, you may assume all arrays in the list, once squeezed, will be two dimensional arrays.

Array Broadcasting

As we have seen up to this point, when two arrays have the same shape, performing a binary operation (such as addition or multiplication) between these arrays automatically performs that operation elementwise.

```
>>> x = np.array([1, 2, 3])
>>> y = np.array([4, 5, 6])
>>> x + y
array([5, 7, 9])
```

While this is useful, there are many times in which we would like to perform a binary operation on two arrays with different shapes.

Suppose, for example, that we would like to multiply a scalar value `b` (in other words, a zero-dimensional array) by every element in a 1-dimensional array `a`. We could accomplish this naively using a loop:

```
>>> a = np.array([1, 2, 3])
>>> b = 2.0
>>> z = np.zeros(3)
>>> for i in range(3):
...     z[i] = a[i] * b
```

```
...
>>> z
array([2., 4., 6.])
```

While this approach is intuitive, we can use *array broadcasting* to accomplish this task more efficiently:

```
>>> a = np.array([1, 2, 3])
>>> b = 2.0
>>> z = a * b
>>> z
array([2., 4., 6.])
```

We can think of what happened in this operation as a two step process:

1) First, this operation duplicated the value of `b=2` into the array `[2, 2, 2]`, which matches the dimensions of `a`.

2) Second, the multiplication `a * [2, 2, 2]` was performed elementwise, giving the same result as the looping procedure we used above.

This process is visualized in the following figure: [1]

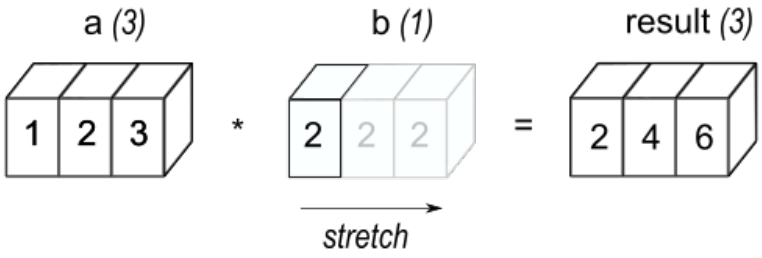


Figure 11.1: Adding a scalar value to a 1-dimensional array using array broadcasting

The beauty of array broadcasting is that this duplication doesn't actually happen in memory, but the results are the same as if it had.

The exact same idea can be extended to arrays of higher dimension by using the following set of rules:

1) If the two arrays have different numbers of dimensions, add size-1 dimensions *to the left* of the current dimensions of the smaller array until the number of dimensions matches.

2) Stretch any singleton (size-one) dimensions in either of the two arrays to match the size of the corresponding dimension in the other array.

3) Add the resulting arrays together elementwise.

[1] This and all other diagrams in this section are taken from Numpy's documentation for array broadcasting, which can be found at `https://numpy.org/doc/stable/user/basics.broadcasting.html`

If, at any point, there are corresponding non-singleton dimensions in each array that do not match, broadcasting will raise a `ValueError: operands could not be broadcast together` exception.

To illustrate this concept more concretely, consider the following examples.

- **Example 1**: Adding a vector to the rows of a matrix

  Suppose, for example, that we have a $m \times n$ array `a`, and we would like to add the n-dimensional vector `b` to each row of `a` individually.

  We can list the shapes of these arrays as follows:

```
a   (2d array):  m x n
b   (1d array):      n
```

  Step (1) of broadcasting will pad the shape of `b` with a size-1 dimension on the left:

```
a   (2d array):  m x n
b   (1d array):  1 x n
```

  Notice that the leftmost dimension is a mismatch, with a 1 in `b`'s slot and an m in `a`'s slot. Thus, step (2) of the broadcasting operation will copy the inner portion of `b` m times to match the dimension of `a`:

```
a      (2d array):  m x n
b      (1d array):  1 x n -> m x n
```

  We can think of this operation as stacking the n-dimensional row vector `b` vertically with itself m times.

  Finally, step (3) of the broadcasting operation is that these two resulting arrays are added elementwise. The result is the following:

```
>>> a = np.array([[ 0.0,  0.0,  0.0],
...               [10.0, 10.0, 10.0],
...               [20.0, 20.0, 20.0],
...               [30.0, 30.0, 30.0]])
>>> b = np.array([1.0, 2.0, 3.0])
>>> a + b
array([[ 1.,  2.,  3.],
       [11., 12., 13.],
       [21., 22., 23.],
       [31., 32., 33.]])
```

  Thus, each row of `a` is added to `b`, as desired.

  A visual representation of this process is given in Figure 11.2:

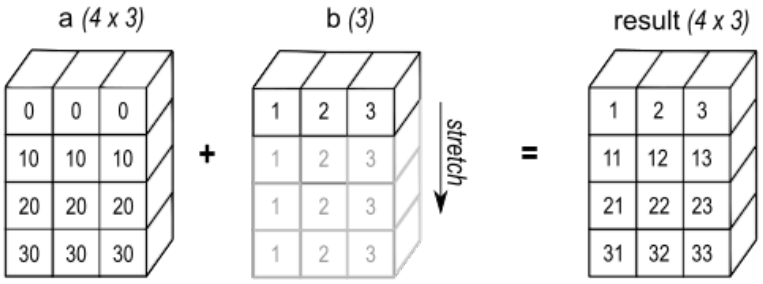


Figure 11.2: Adding entries of vector to the columns of a 2-dimensional array using array broadcasting

- **Example 2**: Adding a vector to the columns of a matrix

  As another example, suppose we have the same $m \times n$ array `a`, and we would like to add the m-dimensional vector `b` to each of its columns.

  Notice that doing this in the same way as the previous example would throw an error:

```
>>> a
array([[ 0.,  0.,  0.],
       [10., 10., 10.],
       [20., 20., 20.],
       [30., 30., 30.]])
>>> b = np.array([1., 2., 3., 4.])
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (4, 3) ↩
    (4,)
```

  To understand why this error is being thrown, notice the shapes of these two arrays after step (1) of broadcasting:

```
a      (2d array):  m x n
b      (1d array):  1 x m
```

  The rightmost dimension is a mismatch, but neither dimension is 1. This is the reason the error is thown. A visual representation of this error is given in Figure 11.3.

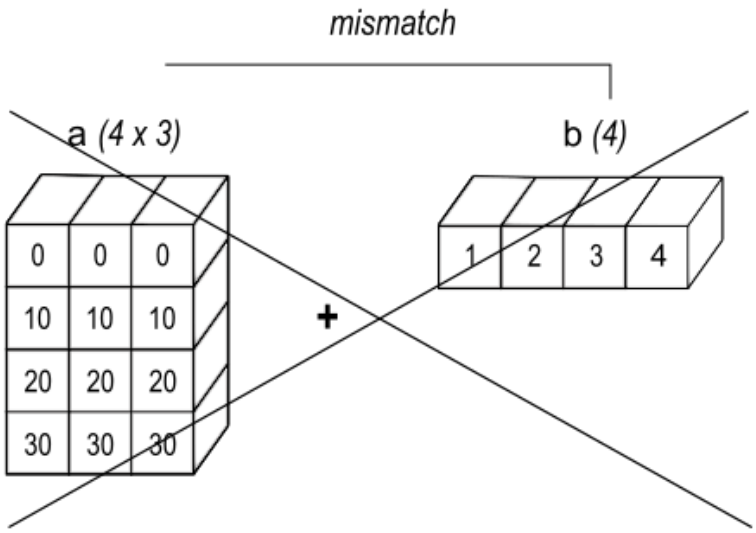


Figure 11.3: The trailing dimensions are a mismatch, so these arrays cannot be broadcast together

In reality, we want to add `b` as a *column vector*; that is, we want to make sure that the m's match up in the *leftmost* column. However, as we saw above, we can't rely on array broadcasting to do this automatically for us since it only pads with ones to the *left* (thus, implicitly treating `b` as a row vector).

To accomplish what we want, we must explicitly reshape `b` to be a column vector using a `np.newaxis` argument (or its equivalent). This will pad the shape of `b` with a size-one dimension to the *right* of its normal dimension, which will allow the dimensions to line up appropriately:

```
a                  (2d array):  m x n
b[:, np.newaxis]   (1d array):  m x 1
```

The resulting operation is exactly what we want:

```
>>> a
array([[ 0.,  0.,  0.],
       [10., 10., 10.],
       [20., 20., 20.],
       [30., 30., 30.]])
>>> b = np.array([1., 2., 3., 4.])
>>> a + b[:, np.newaxis]
array([[ 1.,  1.,  1.],
       [12., 12., 12.],
       [23., 23., 23.],
       [34., 34., 34.]])
```

- **Example 3**: Outer Sum of Two Vectors

The outer sum of two vectors $\mathbf{a} \in \mathbb{R}^n, \mathbf{b} \in \mathbb{R}^m$ is defined be the matrix $A \in M_{n \times m}(\mathbb{R})$ such that

$$[A]_{ij} = a_i + b_j, \quad 1 \leq i \leq n, 1 \leq j \leq m$$

We can define this operation by thinking of wanting to make m copies of $\mathbf{a}$ (horizontally stacked as column vectors), and add it to n copies of $\mathbf{b}$ (horizontally stacked as row vectors).

As it stands, if `a` and `b` are each defined to be one-dimensional arrays with n and m elements, respectively, lining up their shapes looks like the following:

```
a      (1d array):  n
b      (1d array):  m
```

We want the result to be an $n \times m$ matrix. To do this, we want to add the elements of `a` along the leftmost dimension (columns), and the elements of `b` along the rightmost dimension (rows). So we use a `np.newaxis` on `a` to make it a column vector:

```
a[:, np.newaxis]       (2d array):  n x 1
b                      (1d array):      m
```

At this point, broadcasting rules would pad the dimensions of `b` to match the number of dimensions in this `a[:, np.newaxis]`:

```
a[:, np.newaxis]       (2d array):  n x 1
b                      (2d array):  1 x m
```

Next, the column vector `a[:, np.newaxis]` would repeat itself in m horizontally-stacked copies, and the row vector `b` would repeat itself in n vertically-stacked copies, to get matching sizes along each dimension:

```
a[:, np.newaxis]       (2d array):  n x 1 -> n x m
b                      (2d array):  1 x m -> n x m
```

Finally, the results would be summed elementwise, producing the $n \times m$ outer-sum matrix A we desired:

```
>>> a = np.array([0.0, 10.0, 20.0, 30.0])
>>> b = np.array([1.0, 2.0, 3.0])
>>> a[:, np.newaxis] + b
array([[ 1.,   2.,   3.],
       [11.,  12.,  13.],
       [21.,  22.,  23.],
       [31.,  32.,  33.]])
```

A visual representation of what is occurring here can be seen in Figure 11.4.

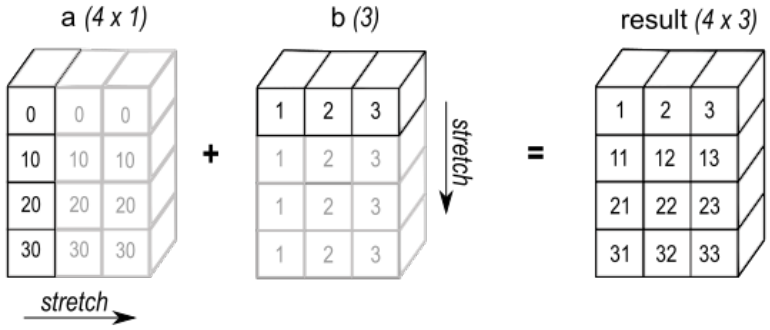


Figure 11.4: Outer sum of two one-dimensional arrays using array broadcasting

Another way we can think about array broadcasting is that we start at the outermost dimension and work our way in, either multiplying intermediate arrays elementwise or broadcasting a single element to match the size of a multi-dimensional element. To see this, let's consider the following two arrays:

$$x = [[1],\\ [2]]$$

$$y = [[3, 4]]$$

To multiply these two arrays together using array broadcasting, we start at the outermost dimension of the arrays and then work our way inward. In this case, the outermost dimension (axis=0) of our arrays is colored in blue, and the inner dimension (axis=1) is colored in orange.

Starting with x, there are two objects (the two orange arrays) inside of the blue brackets. In y, there is only one object in the blue brackets (the one orange array). Thus, we will distribute the one orange array in y to the two orange arrays in x. For now, we will outline these two arrays with blank arrays as place holders.

$$[[],\\ []]$$

Performing the first multiplication of orange arrays, there is one element in the orange array from x (the number 1) and two elements in the orange array from y (the numbers 3 and 4). We will thus distribute the single element from x to the two elements of y by multiplying them elementwise. The result of this is shown below:

$$[[3, 4],\\ []]$$

Performing the second multiplication of orange arrays is similar. The one element of the array from x (the number 2) is distributed to the two elements from the array in y (the numbers 3 and 4) and multiplied by each of them individually. The result of this multiplication is shown below:

$$A = \begin{matrix}[[3, 4],\\ [6, 8]]\end{matrix}$$

These are some simple examples of array broadcasting. Much more complicated cases come up all the time in scientific computing. However, by lining up dimensions appropriately, many loops can be turned into elementwise broadcasting operations. This, in turn, saves a significant amount of time compared to using a for loop to accomplish the same task.

Numerical Computing with NumPy

Universal Functions

A *universal function* is one that operates on an entire array element-wise. Universal functions are significantly more efficient than using a loop to operate individually on each element of an 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.

```
>>> x = np.arange(-2, 3)
>>> print(x, np.abs(x))             # Like np.array([abs(i) for i in x]).
[-2 -1  0  1  2] [2 1 0 1 2]

>>> np.sin(x)                       # Like np.array([math.sin(i) for i in x]).
array([-0.90929743, -0.84147098,  0.        ,  0.84147098,  0.90929743])
```

Problem 3. Write a function that accepts a universal function and an $n \times n$ NumPy array, and returns how many times as fast it is to operate on the entire array element-wise, rather than by using a nested for loop to operate on each element individually. Run each way of operating on the matrix 10 times, and return the ratio of the averages of the two methods. Vow that you will avoid unnecessary nested for loops, especially when operating on large arrays.

See `http://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs` for a more comprehensive list of universal functions.

ACHTUNG!

The `math` module has many useful functions for numerical computations. However, most of these functions can only act on single numbers, not on arrays. NumPy functions can act on either scalars or entire arrays, but `math` functions tend to be a little faster for acting on scalars.

```
>>> import math

# Math and NumPy functions can both operate on scalars.
>>> print(math.exp(3), np.exp(3))
20.085536923187668 20.0855369232

# However, math functions cannot operate on arrays.
>>> x = np.arange(-2, 3)
>>> np.tan(x)
array([ 2.18503986, -1.55740772,  0.        ,  1.55740772, -2.18503986])
>>> math.tan(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only length-1 arrays can be converted to Python scalars
```

Always use universal NumPy functions, not the `math` module, when working with arrays.

Other Array Methods

The `np.ndarray` class itself has many useful methods for numerical computations.

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.
`clip()`	restrict values in an array to fit within a given range
`max()`	The maximum element of the array.
`mean()`	The average value of the array.
`min()`	The minimum element of the array.
`roll()`	shuffles the elements of the array according to specified amount.
`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 an equivalent NumPy function. For example, `A.max()` and `np.max(A)` operate the same way. 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 can 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 range(3)])
array([0, 1, 2])

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

Similar to the above discussion about array broadcasting, we can color the arrays to gain more visual intuition for how operations can be performed across axes.

Consider again the array

$$A = [[3, 4],\\ [6, 8]]$$

Suppose that we wanted to evaluate the call `B = np.sum(A, axis = 1)`. Axis dimensions are numbered from outmost to inmost so in our case axis 0 denotes the blue brackets and axis 1 denotes the orange brackets. Summing over axis 1 means that we sum everything in the orange brackets, but leave the result where it lies in the larger array. So in this case the result would be:

$$B = [7, 14]$$

If we instead wanted to compute the sum `B = np.sum(A, axis = 0)`, we would sum the things inside of the blue brackets. In this case, there are two things inside of the blue brackets (the two orange arrays). These are summed elementwise, resulting in the following array:

$$B = [9, 12]$$

Refer to the NumPy Visual Guide in the appendix for more visual examples.

Also see `http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html` for a more comprehensive list of array methods.

Problem 4. A matrix is called *row-stochastic*[a] if its rows each sum to 1. 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 NumPy array). Divide each row of the matrix by the row sum and return the new row-stochastic matrix. Use array broadcasting and the `axis` argument instead of a loop.

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

Vectorizing functions

Whenever possible making your functions 'numpy aware' can greatly reduce complexity, increase readability and simplicity of code and make functions more versatile. Designing functions to be able to work with and utilize numpy arrays and numpy functions is one of the best ways to optimize code. However, sometimes the functions we need to use are very difficult to vectorize. In this case it can be useful to employ `np.vectorize()`

`np.vectorize()` accepts as an argument a function whose input and output is a scalar. It returns a new function that is 'numpy aware', meaning that it will accept a numpy array of values and output an array where each entry had the operation defined by the original function performed on it.

```
# Define a function to double a number
>>> def Double(x):
...     return 2*x

>>> test = np.array([1, 2, 3, 4, 5])
# Using a for loop we can get the doubled array
>>> for i,val in enumerate(test):
...     test[i] *= 2
>>> test
array([2, 4, 6, 8, 10])

# Vectorizing our Double function
>>> DoubleVectorized = np.vectorize(Double)

>>> test = np.array([1, 2, 3, 4, 5])
# with the function vectorized the implementation is simple
>>> DoubleVectorized(test)
array([2, 4, 6, 8, 10])
```

Note

While the above example can easily be done with array broadcasting, `np.vectorize()` can be implemented with very complex scalar functions for which no array broadcasting method exists. However, it should be noted that this function is used only for convenience and readability since it does not improve temporal complexity like normal array broadcasting would. Even though it doesn't improve the complexity, it is often simpler than trying to formulate the for loop.

Achtung!

Note that `np.vectorize()` will infer the type of its output based on the first element of the input. This behavior can cause confusing results if multiple return types are used within one function. Make sure to either use the same return type in all branches of the function to be vectorized or specify a return type with the `otypes` keyword argument. The following example illustrates the importance of carefully handling datatypes:

```
def half_heavyside(x):
    if (x > 0):
        return 0.5
    else:
        return 0

>>> f = np.vectorize(half_heavyside)

>>> f([-1., -0.5, 0., 0.5, 1.])
array([0, 0, 0, 0, 0])
```

At first glance, it looks like `np.vectorize()` is simply broken. However, what is actually happening is `np.vectorize()` is infering the datatype from our first input. Since the first input, `-1`, returned `0` (an integer), `np.vectorize()` assumes that all subsequent outputs will be of type `int` as well. When `half_heavyside` does return a `0.5`, the `float` is cast to an `int`, thus returning a `0`.

Two example fixes for this problem are shown below. The first fix explicitly ensures both return statements are of type `float`. The second fix uses the `otypes` keyword argument of `np.vectorize()` to specify the correct return type. Only one type per return value of the wrapped function may be specified with `otypes`.

```
def half_heavyside(x):
    if (x > 0):
        return 0.5
    else:
        return 0.

>>> f = np.vectorize(half_heavyside)

>>> f([-1., -0.5, 0., 0.5, 1.])
array([0. , 0. , 0. , 0.5, 0.5])
```

```
def half_heavyside(x):
    if (x > 0):
        return 0.5
    else:
        return 0

>>> f = np.vectorize(half_heavyside, otypes = [float])

>>> f([-1., -0.5, 0., 0.5, 1.])
array([0. , 0. , 0. , 0.5, 0.5])
```

Problem 5. Given to you is the code that finds the prime factorization of a number and returns the largest prime in the factorization. Vectorize the function using `np.vectorize()` and program a function that either uses the vectorized function or the naive for loop depending on the argument 'naive' being passed in as True or False.

Make sure you function returns a numpy array of the same size for both cases.

Hint: Make sure the naive approach returns the array with a dtype of 'int32'

Einsum

While numpy has many functions to help multiply arrays, multiplying the elements of arrays in unorthodox ways usually requires the conglomeration of quite a few of these functions. `np.einsum()` is designed to eliminate this problem by making a general framework for multiplication and addition in arrays using their shapes and allowing the coder to tell the function which elements exactly are to be multiplied or summed and how those operations are to be returned.

Einsteinian summation notation

The function name `einsum()` comes from the term *Einsteinian summation*, which is a standard notational technique in physics and engineering for matrix and tensor operations.

To understand how this works, consider the simple operation of the dot product $\mathbf{x} \cdot \mathbf{y}$, where $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$. As we know, this is defined as

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i$$

In the equation above, the i is what we refer to as a *dummy index*; that is, i is only used inside the sum, and never appears outside the sum.

As another example, consider the operation of matrix multiplication. Given $A \in M_{n\times m}(\mathbb{R})$ and $B \in M_{m\times \ell}(\mathbb{R})$, we know that the ij-element of the resulting matrix product $AB \in M_{n\times \ell}(\mathbb{R})$ is given by

$$[AB]_{ij} = \sum_{k=1}^{m} a_{ik}b_{kj}$$

Again, since k only appears inside the summation, it is a dummy index. However, i and j appear on both the left and right hand sides of this equation, so they are called *free indexes* (and they are not summed over).

Notice that, in both of these examples, we have written vector-valued and matrix-valued operations, respectively, by denoting what an element of the output array should look like in terms of indexes. In the dot product example, the output is a scalar, so we have denoted what the scalar should be in terms of the elements of the input vectors to that operation. In the matrix multiplication example, the output is a matrix, so we have denoted what each ij element of that matrix should be in terms of the elements of the input matrices to that operation.

In essence, Einsteinian summation notation does exactly this: it denotes what a given element of the output array should be in terms of the elements of the input array, just as we have done above. However, for brevity, all summation symbols are dropped in this notation, and all dummy indexes are implicitly summed over.

In this notation, the dot product would be written

$$\mathbf{x} \cdot \mathbf{y} = x_i y_i$$

Similarly, the matrix product would be written

$$[AB]_{ij} = a_{ik}b_{kj}$$

For this reason, Einsteinian summation notation is often called *index notation*.

Notice that clever, unorthodox ways of taking "products" of matrices and vectors can be defined in this way. For example, given two matrices $A \in M_{n\times m}(\mathbb{R})$ and $B \in M_{m\times n}(\mathbb{R})$, we can define a clever product $A \odot B$ that (1) takes the normal matrix product and (2) takes the trace of that resulting matrix as

$$A \odot B = \sum_{i=1}^{n} [AB]_{ii} = \sum_{i=1}^{n}\sum_{k=1}^{m} a_{ik}b_{ki}$$

By noting that i and k are now dummy variables, and the output is a scalar, we can write this unusual product in index notation as

$$A \odot B = a_{ik}b_{ki}$$

Note

In traditional index notation, the only time an index is summed over is when it appears twice in an expression, such as the index i in the dot product $\mathbf{u} \cdot \mathbf{v} = u_i v_i$. Thus, the only things considered "dummy indexes" are indexes which appear twice in an expression.

However, there are many more expressions that we can represent by summing over indexes which are not repeated. For example, we can take a row sum of a $m \times n$ matrix A, resulting in the m-dimensional vector $\mathbf{v}$, using the following rule:

$$[v]_i = \sum_{j=1}^{n} a_{ij}$$

Note that we are summing over j here, but it is not a repeated index. So this expression would not be representable in traditional index notation without a much more convoluted expression.

Despite this limitataion, such an expression *is* representable using `np.einsum()`, as we will describe in the next section. So, while the notation of `np.einsum()` is similar to that of traditional index notation, it is more flexible.

So, through the rest of this manual, we will write mathematical development of these expressions with summation notation, rather than index notation, to make clear exactly what variables are being summed over (that is, to distinguish between free variables and dummy variables).

Numpy's Einsum

The function `np.einsum()` is a function which is designed to be used with operations defined in terms of Einsteinian summation notation, defined in the previous section. It can be used on arrays of higher dimensions, but we'll keep the scope of this section to working with input arrays of 1 or 2 dimensions.

The numbers in the following syntax only represent positions, each position will be explained below:

```
np.einsum("12,34 -> 56", 7, 8)
```

The meaning of the positions are as follows:

1 and 2) Symbols representing the index of an element in the first input array. In the case of 1 dimensional vectors, there will only be one variable and the second will be omitted.

3 and 4) Similarly, symbols representing the index of an element in the second input array.

5 and 6) Symbols representing the index of an item in the output array. There can zero (for scalars), one (for vectors), two (for matrices), or even three (for higher-order tensors) variables here.

7) The first input array. Make sure it is outside the quotation mark ending the variables section and preceded by a comma.

8) The second input array.

Positions 7 and 8 are obviously arrays, but position 1-6 are filled with textual symbols, usually i, j, and k, which represent the indexes of elements in the input and output arrays.

The key here is that we can translate anything in summation notation to an einsum expression. To do this, the subscripts of anything *inside* the summation are the letters on the left-hand side of the arrow, with commas separating different quantities. The subscripts of anything on the *result* of a summation notation expression are the letters on the right-hand side of the arrow. This idea will be demonstrated more fully in the following section.

Einsum Rules

The way `einsum()` interprets its inputted variables are as follows:

1) If the variables contained in positions 1 or 2 share a variable in 3 or 4, the values along the axes specified by the repeated variables positions will be multiplied together

2) On the other side of the arrow (positions 5 and 6), specify the dimensions of the output. Ommiting a variable from these positions causes the products to be summed.

   (That is, any variables that appear on both sides of the arrow are free variables, and any variables only on the left-hand side are dummy variables that are summed over).

As an example, we will define normal matrix multiplication using einsum. Recall that the summation notation for this (rewritten with a different dummy index) is

$$[AB]_{ik} = \sum_{j=1}^{m} a_{ij} b_{jk}$$

This means that the following `np.einsum()` command will perform normal matrix multiplication:

```
>>> import numpy as np

>>> A = np.eye(3)
>>> A[0,:] += A[2,:]
>>> A
array([[1, 0, 1],
       [0, 1, 0],
       [0, 0, 1]])

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

# We use einsum to define normal matrix multiplication
# note the 'j' is repeated in axis 1 and axis 0 respectively
>>> np.einsum("ij,jk -> ik", A, B)
array([[6, 8, 10],
       [3, 4, 5],
       [6, 7, 8]])
```

Since the `j` was repeated in the previous example, the elements on those 2 axes are paired and multiplied together elementwise. The first pair of indicies has `j` in the second position, which corresponds to the column position (axis=1; moving from column to column down a single row). The second pair of indices has `j` in the first position, which corresponds to the row position (axis=0; moving row to row down a single column). Thus, Einsum takes the elementwise products of the elements along each respective row of `A` and column of `B`. This corresponds to the $a_{ij}b_{jk}$ in the summation notation, above.

Additionally, the output indexes were specified to be `ik` instead of `ijk`. Therefore, since j was present on the left-hand-side of the Einsum expression, and not the right-hand side, the elementwise products $a_{ij}b_{jk}$ were summed over all of the j indexes (exactly as defined in the summation notation above).

It's important to note here that Einsum adds and multiplies in a very similar manner to the `axis` argument in `np.sum()` and other numpy functions.

Common operations using Einsum

The below table describes some classical operations on vectors and matrices, and how they may be defined in terms of (1) summation notation and (2) the corresponding `np.einsum()` command.

Operation	Summation Notation	Einsum Command
Transpose of a matrix A	$[A^\intercal]_{ij} = a_{ji}$	`np.einsum("ji -> ij", A)`
Row sum vector r of a matrix A	$[r]_i = \sum_{j=1}^{m} a_{ij}$	`np.einsum("ij -> i", A)`
Column sum vector c of a matrix A	$[c]_j = \sum_{i=1}^{n} a_{ij}$	`np.einsum("ij -> j", A)`
Sum s of all elements of a matrix A	$s = \sum_{i=1}^{n}\sum_{j=1}^{m} a_{ij}$	`np.einsum("ij->",A)`
Trace of a matrix A	$\mathrm{tr}(A) = \sum_{i=1}^{n} a_{ii}$	`np.einsum("ii->", A)`
Dot product of u and v	$u \cdot v = \sum_{i=1}^{n} u_i v_i$	`np.einsum("i, i ->", x, y)`
Outer product of u and v	$[u \otimes v]_{ij} = u_i v_j$	`np.einsum("i, j -> ij", u, v)`
Matrix product of A and B	$[AB]_{ik} = \sum_{j=1}^{n} a_{ij}b_{jk}$	`np.einsum("ij, jk -> ik", A, B)`
Elementwise product B of a vector u by each *row* of a matrix A	$[B]_{ij} = a_{ij}u_j$	`np.einsum("ij, j -> ij", A, u)`
Elementwise product B of a vector v by each *column* of a matrix A	$[B]_{ij} = a_{ij}v_i$	`np.einsum("ij, i -> ij", A, v)`

> **Note**
>
> Notice again that excluding a letter after the `->` symbol will cause summation of the specified scalar products along the axis whose letter was excluded. On the other hand, including a letter after the `->` symbol will *disallow* summation along that index, and will instead create an output dimension of the same size as the input dimension corresponding to that letter.
>
> For example, see the row sum entry in the above table. Since the row index `i` is included after the `->` symbol, the resulting vector will have the same dimension as the number of rows. However, since the column index `j` is *not* included after the `->` symbol, the scalar values a_{ij} must be summed over all `j` values corrresponding to each `i` value before they are output. This corresponds to summing over all of the columns in the `i`th row before storing the output in the `i`th entry of the output vector.

A longer example

Now, to demonstrate Einsum's true power we'll show a more complicated example.

Imagine you needed to take an outer product of each column of a matrix $A \in M_{n,m}(\mathbb{R})$ with the corresponding column of another matrix $B \in M_{\ell,m}(\mathbb{R})$, storing the result in an array called `outer_products`, where `outer_products[k]` is defined to be the $n \times \ell$ outer product array of the kth column of A with the kth column of B for every $1 \leq k \leq m$.

You could accomplish this in a for loop, but it would be slow and not very effective. Conversely, this can be done in one line with `np.einsum`, with the correct mathematical setup.

Let $\mathbf{u}_k$ being the kth column of A, and $\mathbf{v}_k$ be the corresponding kth column of B. We look to find the outer product

$$[\mathbf{u}_k \otimes \mathbf{v}_k]_{ij} = [u_k]_i [v_k]_j, \quad 1 \leq i \leq n, 1 \leq j \leq \ell$$

for $1 \leq k \leq m$.

We know that the ith element of $\mathbf{u}_k$ (the kth column of A) is just a_{ik}. Similarly, the jth element of $\mathbf{v}_k$ (the kth column of B) is just b_{jk}. Substituting these values into the above expression

$$[\mathbf{u}_k \otimes \mathbf{v}_k]_{ij} = a_{ik} b_{jk}, \quad 1 \leq i \leq n, 1 \leq j \leq \ell$$

Finally, letting C denote the three-dimensional array `outer_products`, we have found that

$$[C]_{kij} = [\mathbf{u}_k \otimes \mathbf{v}_k]_{ij} = a_{ik} b_{jk}$$

This is already in index notation, which allows us to write the corresponding Einsum expression in one line:

```
>>> A = np.arange(9).reshape((3, 3))
>>> B = np.arange(12).reshape((4, 3))
# Notice these are just the indexes from the final index notation expression
>>> outer_products = np.einsum("ik, jk -> kij", A, B)
>>> outer_products
array([[[0,  0,  0,  0],
        [0,  9, 18, 27],
        [0, 18, 36, 54]],

       [[1,  4,  7, 10],
        [4, 16, 28, 40],
        [7, 28, 49, 70]],

       [[ 4, 10, 16, 22],
        [10, 25, 40, 55],
        [16, 40, 64, 88]]])
```

The output dimension having three variables creates a 3-dimensional matrix where each element in the first axis is a 3 by 4 outer product matrix.

If we then wanted to sum the rows of each of those outer-product matrices, resulting in a $k \times n$ array of row sums, notice that we would want an output array C such that

$$[C]_{ki} = \sum_{j=1}^{n} [\mathbf{u}_k \otimes \mathbf{v}_k]_{ij} = \sum_{j=1}^{n} a_{ik} b_{jk}$$

Notice now that j has become a dummy variable, and thus, we must eliminate it from the right-hand side of the arrow in the Einsum expression:

```
>>> A = np.arange(9).reshape((3, 3))
>>> B = np.arange(12).reshape((4, 3))
>>> outer_products_sum = np.einsum("ik, jk -> ki", A, B)
>>> outer_products_sum
array([[ 0,  54, 108],
       [22,  88, 154],
       [52, 130, 208]])
```

Finally, perhaps we want to multiply a different vector $\mathbf{v}$ elementwise to the rows of this resulting `outer_products_sum` array. That is (again letting $[C]_{ki}$ denote the output elements of this output array), we want the ith element of v to be multiplied by our earlier expression for $[C]_{ki}$:

$$[C]_{ki} = \left(\sum_{j=1}^{n} a_{ik} b_{jk} \right) v_i = \sum_{j=1}^{n} a_{ik} b_{jk} v_i$$

The resulting code displays the results:

```
>>> A = np.arange(9).reshape((3, 3))
>>> B = np.arange(12).reshape((4, 3))
>>> v = np.array([0, 1, -1])
>>> outer_products_with_broadcast = np.einsum("ik, jk, i -> ki", A, B, v)
>>> outer_products_with_broadcast
array([[0,  54, -108],
       [0,  88, -154],
       [0, 130, -208]])
```

Thus, three difficult operations have been reduced to a few letters by the power of `np.einsum()`.

Einsum Optimize

`np.einsum()`, in most cases, performs faster than built in numpy functions. However, the way Einsum organizes its operations creates redundancy when trying to perfrom multiple operations at once, such as multiplying two matrices, broadcasting a vector to its rows and then summing the resulting matrix columns. One should be cautious when using Einsum to perform multiple operations since you may actually be making your complexity worse rather than improving it.

There are two ways around this problem. First, performing each operation individually will preserve the integrity of Einsum's performance, although brevity of code will suffer. Second, multiple operations can be performed efficiently using kwarg `optimize=True`.

Setting `optimize=True` creates an extra step of operational analysis before any calculations are made to ensure efficient order of operations. In addition, this mode uses a more spatially complex method of computation in exchange for ensured temporal gains. This is why `optimize` defaults to `False`: to allow the programmer to know whether or not there is sufficient memory or need for the optimize functionality.

Problem 6. Write a function that accepts 3 vectors and a matrix of appropriate sizes and returns a matrix that is the result of an outer product of the first 2 vectors, the 3rd vector array broadcast multiplied onto the columns of that matrix and then the multiplication via normal matrix multiplication of that result to the inputed matrix. Your function should take a keyword argument `split` that defaults to `False`. If `split=True`, then the einsum operations should be performed one at a time. Otherwise, the operations should be performed all at once while using `optimize=True`.

Write an additional function that performs the same operations only using numpy operations.

Hint: Your result should return the equivalent of `np.outer(x,y)*z.reshape(-1,1)@A` where `x`, `y`, and `z` are vectors and `A` is a matrix.

Problem 7. Time your einsum function from Problem 6 versus its numpy function equivalent for vectors of size 3 through 500 and arrays of size (3,3) through (500,500). You should time your einsum function both with and without `split=True`. Plot the results on a neatly formatted and labeled graph. Compare your results to Figure 11.5.

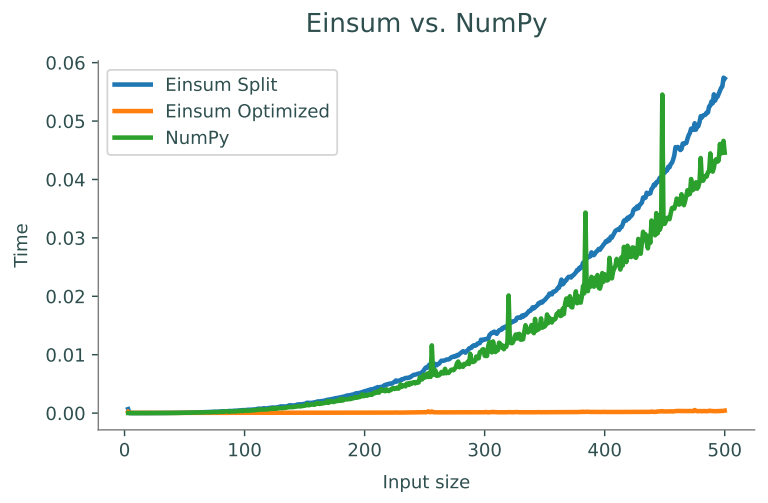


Figure 11.5

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.

Note that many of these functions have counterparts in the standard library's `random` module. These NumPy functions, however, are much better suited for working with large collections of random samples.

```
# 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 for later use. 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.load("normal.npz")
>>> np.allclose(x, arrays["first"])
True
>>> np.allclose(y, arrays["second"])
True
```

Polynomials

The `np.poly1d` object represents a polynomial in NumPy. The constructor is called with the coefficients of the desired polynomial.

```
>>> poly = np.poly1d([3, 5, 1, 2, 0, 1])
>>> print(poly)
   5     4     3     2
3 x + 5 x + 1 x + 2 x + 1
```

The object `poly` represents the polynomial $3x^5+5x^4+x^3+2x^2+1$. NumPy provides many functions to operate on `poly1d` objects (see `http://docs.scipy.org/doc/numpy/reference/routines.polynomials.polynomial.html`).

Recall that

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

The following function evaluates the Nth partial sum of this series at the value a.

```
>>> from scipy.special import factorial
>>> def exp(a, N=25):
...     """Construct an array in reverse order from n to 0."""
...     n = np.arange(N, -1, -1)
...     # Use broadcasting to compute coefficients
...     coeffs = 1. / factorial(n)
...     poly = np.poly1d(coeffs)        # Make a polynomial object.
...     return poly(a)
...
```

The last two lines can be condensed by using the following command:

```
np.polyval(p, a)
```

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.

Expressing Einsum Expressions in Terms of Other Functions

Conceptually, we can think of `np.einsum()` as projecting tensors into higher dimensional space, elementwise multiplying the matrices together, and then summing along the specified dimensions. We can thus write many `np.einsum()` expressions as a combination of transposes, elementwise multiplications, sums, and expanding dimensions. Note that writing `np.einsum()` expressions in this way is much less efficient and convenient than just using a call to `np.einsum()`. This is rather used as an alternate way of understanding what `np.einsum()` is doing. We will work through an example of how this is done to illustrate the concepts.

Consider the following `np.einsum()` expression.

```
np.einsum("ijk, jil -> kil", A, B)
```

Each of the different letters in the Einsum expression will label a different axis. Since there are four letters, we project into four dimensions. It doesn't necessarily matter how this axis-to-letter pairing is done, as long as we are consistent, but its often convenient to do it alphabetically. We will let `i` denote axis 0, `j` denote axis 1, `k` denote axis 2, and `l` denote axis 3.

We can increase the dimensions of the matrices with a call to `np.expand_dims()`. In this case we would call,

```
A = np.expand_dims(A, 4)

B = np.expand_dims(B, 4)
```

We now need to align all the axes of all matrices we are working with. In our example, with our choice of axis-to-letter pairing, the axes of matrix A are already in the correct alignment. We can order the dimensions of B with the following call to `np.transpose()`, where the axes passed to `np.transpose` correspond to the axis-to-letter pairing that we denoted earlier:

```
B = np.transpose(B, (1, 0, 3, 2))
```

After correctly ordering the dimensions, we elementwise multiply the matrices together.

```
C = A * B
```

With our final matrix, we sum along all dimensions that are missing from the output specifier. In our case, since `j` is missing, we sum along axis 1.

```
C = np.sum(C, axis = 1)
```

Finally, we transpose the result so that the axis labels are consistent with the output specifier. Note that `k` has now become axis 1 and `l` has become axis 2.

```
C = np.transpose(C, (1, 0, 2))
```

An Advanced Example of Broadcasting and Einsum

The techniques of array broadcasting and `np.einsum()`, while quite daunting at first, constitute an arsenal of extremely powerful techniques that allow you to design algorithms that run quickly on large quantities of data.

Consider the following real-world example, built off of the machine-learning idea of Gaussian Mixture Models (GMMs), which demonstrates the power of using these techniques in tandem.

Assume you have have n data points $\{z_0, \dots, z_{n-1}\}$, where each data point is d-dimensional (that is, $z_i \in \mathbb{R}^d$ for $i = 0, \dots, n-1$).

Underlying the phenomenon we are studying, assume there are K multivariate normal distributions $\{\mathcal{N}(\mu_0, \Sigma_0), \dots, \mathcal{N}(\mu_{K-1}, \Sigma_{K-1})\}$, each of which contribute with some corresponding weight $\{w_0, \dots, w_{K-1}\}$ to the full probability distribution:

$$P(z|\theta) = \sum_{k=0}^{K-1} w_k \mathcal{N}(z|\mu_k, \Sigma_k)$$

(where $\mathcal{N}(z|\mu_k, \Sigma_k)$ denotes the standard p.d.f. for the normal distribution with mean μ_k and covariance Σ_k).

Our goal, given the data points z_i, is to figure out which parameters w_k, μ_k, and Σ_k (for $k = 0, \dots, K-1$) best fit the data.

Skipping some details, the GMM algorithm iteratively updates guesses for these $3K$ parameters at each timestep t. Specifically, given (1) an updated mean μ_k^{t+1} and (2) some probabilities $q_i^t(k)$ dependent on (a) the data point z_i, (b) the distribution index $k \in [0, \ldots, K-1]$, and (c) the timestep t, the update step for the covariance matrices is given by:

$$\Sigma_k^{t+1} = \frac{\sum_{i=1}^n q_i^t(k)(z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top}{\sum_{i=1}^n q_i^t(k)}$$

The goal of this section will be to describe how to implement this in Python code.

Assume that you have the following arrays available to you, each having to do with one of the three symbols in the right-hand side of the above expression:

- `q_values` : `(K,n)` array with the elements being `q_values[k,i]` $= q_i^t(k)$.
- `Z`: `(n,d)` array with each row being `Z[i,:]` $= z_i$
- `new_means`: `(K,d)` array with each row being `new_means[k,:]` $= \mu_k^{t+1}$

We want to store the resulting covariance matrices in an array `new_covars`, which will satisfy `new_covars[k,:,:]` $= \Sigma_k^{t+1}$. Noting that each covariance matrix will be $d \times d$, we can see that `new_covars` will have shape `(K, d, d)`.

One naive way to implement this would be to use a nested for loop:

```
>>> K, d = new_means.shape
>>> _, n = q_values.shape
>>> new_covars = np.zeros((K, d, d))
>>> for k in range(K):
...     for i in range(n):
...         centered = Z[i, :] - new_means[k, :]
...         outer_prod = np.outer(centered, centered)
...         new_covars[k, :, :] += q_values[k, i] * outer_prod
...     new_covars[k, :, :] /= np.sum(q_values[k, :])
```

However, this is extremely inefficient, and can take hours to run as the number of data points becomes large. As a result, this operation calls for some more advanced techniques.

Our goal will be to do this entire operation in a series of array broadcasting steps, using only a single Einsum command. This will make the code much more efficient.

To do so, we will start by taking bite-size chunks of the expression, figuring out how to write them in code using broadcasting and Einsum. Then, through a series of steps, we will gradually come up with a formula that will perform the calculation for this expression without a need for any looping whatsoever.

- **Step 1:** (Computing the differences between the means μ_i and the data points z_k)

  Notice that the numerator contains the expression

  $$(z_i - \mu_k^{t+1})$$

  more than once.

  As a first step, we look to create an array called `obs_centered` with shape `(K,n,d)`, such that `obs_centered[k,i,:]` $= z_i - \mu_k^{t+1}$.

That is, we want a 3-dimensional array, where each inner element is a 2-dimensional matrix. Each 2-dimensional matrix will be produced by subtracting one row of the `new_means` matrix from all n rows of `Z`.

Recall: if we have two 1-dimensional arrays `a` and `b`, and we want to take the "outer difference" between them (that is, we want to create a 2-dimensional array containing the differences of every element of `a` with every element of `b`), we can do something akin to what is shown in Figure 11.4 and its preceding example:

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([4, 5])
>>> a - b[:, np.newaxis]
array([[-3, -2, -1],
       [-4, -3, -2]])
```

Notice that the shapes of the arrays in this broadcast are `(3,)` and `(2,)`, respectively, and the output shape is `(2,3)`.

Digging deeper into what's going on here, the broadcasting operation of subtraction (and padding of dimensions) executed the following rearranging of shapes before performing the elementwise subtraction:

```
a   :  (3,) -> (1, 3) -> (2, 3)
b   :  (2,) -> (2, 1) -> (2, 3)
```

Now, what we seek to do in our current problem is to take a very similar "outer difference" of the *rows* of `Z` and `new_means`. This is almost identical to the above case, but instead of subtracting 0-dimensional scalars, we are looking to take the difference between 1-dimensional vectors, each with d elements.

Therefore, we look to outer-subtract the n rows of `Z` with the K rows of `new_means` in all possible combinations, in the exact way we looked to outer-subtract the scalars of the input arrays `a` and `b` in our example above.

The ultimate output we desire is a `(K,n,d)` array (similar to how we ended up with a `(2,3)` array in the scalar example).

The code in this case is, therefore, nearly identical to the scalar case. All we need to figure out is how to pad the dimensions of these array in the correct way to get this outer difference of vectors. This is a bit trickier since having a d in the rightmost slot means we can't just use `np.newaxis` as easily as before.

Despite this setback, however, looking to the two leftmost dimensions, and leaving the rightmost dimension (the vectors we wish to subtract) alone, comparison to the scalar example shows that we want to achieve the following shapes:

```
Z           : (n, d) -> (1, n, d)
new_means   : (K, d) -> (K, 1, d)
```

So, what commands to we need to use to achieve these paddings?

The command to add a singleton dimension in the middle of the original shape of `new_means` is `np.expand_dims(new_means, 1)`.

Once that is done, the padding to the left of `Z` is automatic by the rules of array broadcasting.

So the desired command is:

```
>>> obs_centered = Z - np.expand_dims(new_means, 1)
```

- **Step 2:** (Computing the outer products of these differences with themselves)

  Notice that there is a term in the numerator of our desired expression that looks like:

  $$(z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top$$

  This is exactly an outer product of one of the vectors found in Step (1) with itself.

  Our goal in this step is to produce an intermediate array `outer_products` of shape with shape `(K, n, d, d)` such that `outer_products[k,i,:,:]` $= (z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top$.

  Given that `obs_centered[k,i,:]` $= (z_i - \mu_k^{t+1})$ by our design, we therefore see that `outer_products[k,i,:,:]` is just the outer product of `obs_centered[k,i,:]` with itself.

  In other words,
  `outer_products[k,i,:, :]` = `obs_centered[k,i,:]` $\otimes$ `obs_centered[k,i,:]`.

  It follows that the pq-element of `outer_products[k,i,:, :]` is equal to the pq-element of `obs_centered[k,i,:]` $\otimes$ `obs_centered[k,i,:]`.

  Writing this in summation/index notation, letting u denote `obs_centered` and v denote `outer_products`, we have shown that

  $$v_{kipq} = [v_{ki::}]_{pq} = [u_{ki:} \otimes u_{ki:}]_{pq} = u_{kip}u_{kiq}$$

  Therefore, we can write the full einsum expression (with all indexes *inside* the expression) as

  ```
  >>> outer_products = np.einsum("kip, kiq -> kipq", obs_centered, ↩
      obs_centered)
  ```

- **Step 3:** (Summing the outer products over the dimension representing the number of data points)

  Notice that the numerator has a summation (excluding some information) that looks like

  $$\sum_{i=1}^{n}(z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top$$

  In this step, we look to create an array `outer_product_sums` with shape `(K, d, d)` such that `outer_product_sums[k,:,:]` $= \sum_{i=1}^{n}(z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top$.

  Notice that $(z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top$ = `outer_products[k,i,:,:]` by our construction.

  So, all we are doing is summing along the `i`-axis of `outer_products`.

  Therefore, letting w be a variable representing `outer_product_sums`, with u represening `obs_centered` and v representing `outer_products`, we have that

  $$w_{kpq} = \sum_{i=1}^{n} v_{kipq} = \sum_{i=1}^{n} u_{kip}u_{kiq}$$

  Thus, we can write this directly with the following Einsum (which looks nearly identical to Step 2, but excludes the index `i` from the right-hand-side of the arrow to denote a summation):

```
>>> outer_product_sums = np.einsum("kip, kiq -> kpq", obs_centered, ↩
    obs_centered)
```

- **Step 4:** (Finishing the numerator)

  Finally, we look to tackle the entire numerator in a single Einsum expression.

  The numerator looks exactly like the following weighted sum of inner products:

$$\sum_{i=1}^{n} q_i^t(k)(z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top$$

  In this step, we look to create an array `weighted_sum_numerator` with shape `(K, d, d)` such that
  `weighted_sum_numerator[k,:,:]` $= \sum_{i=1}^{n} q_i^t(k)(z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top$.
  Recall that $q_i^t(k)$ = `q_values[k,i]` and $(z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top$ = `outer_products[k,i,:,:]` by our construction.

  Letting q represent `q_values` and z represent `weighted_sum_numerator`, with u represening `obs_centered` and v representing `outer_products` yet again, we have that

$$z_{kpq} = \sum_{i=1}^{n} q_{ki} v_{kipq} = \sum_{i=1}^{n} q_{ki} u_{kip} u_{kiq}$$

  Therefore, we end up with the following Einsum expression for the numerator:

```
>>> weighted_sum_numerator = np.einsum("ki, kip, kiq -> kpq", q_values, ↩
    obs_centered, obs_centered)
```

- **Step 5:** (Full expression)

  Finally, we look to compute the full expression

$$\Sigma_k^{t+1} = \frac{\sum_{i=1}^{n} q_i^t(k)(z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top}{\sum_{i=1}^{n} q_i^t(k)}$$

  We have found the numerator to be `weighted_sum_numerator` from Step (4). The denominator is exactly just the sum $\sum_{i=1}^{n} q_i^t(k)$. Notice that this sum depends on k, as does the result Σ_k^{t+1}. Thus, we must make sure that the sum is taken over all data points, leaving a K-dimensional vector of sums that we must then broadcast to the corresponding dimension of the numerator.

  Recall that since the shape of `q_values` is `(K,n)`, we will take the sum over the second axis (axis=1) to get:

```
>>> q_sum = q_values.sum(axis=1)
```

  Now, we have that the shape of `q_sum` is `(K,)`. We need to broadcast this to the corresponding dimension of `weighted_sum_numerator`.

  Recall that the shape of `weighted_sum_numerator` is `(K, d, d)`.

  Thus, we want to change the shape of `q_sum` to match this to make broadcasting work properly:

```
weighted_sum_numerator      :  (K, d, d)
q_sum                       :        (K,) -> (K, 1, 1)
```

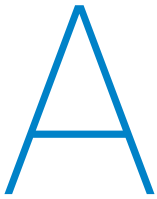
Thus, we need to add 2 singleton dimensions to the *right* of the shape of `q_sum`. This can be accomplished by the `reshape` command just fine.

Hence, we get the following sequence of Python commands for the final expression, with no loops:

```
>>> obs_centered = Z - np.expand_dims(new_means, 1)
>>> weighted_sum_numerator = np.einsum("ki, kip, kiq -> kpq", q_values, ↩
    obs_centered, obs_centered)
>>> q_sum = q_values.sum(axis=1)
>>> new_covars = weighted_sum_numerator / q_sum.reshape(-1, 1, 1)
```

Part II

Appendices

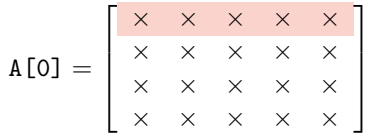


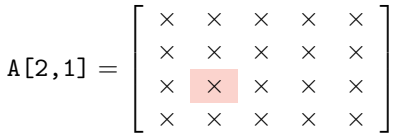
NumPy Visual Guide

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

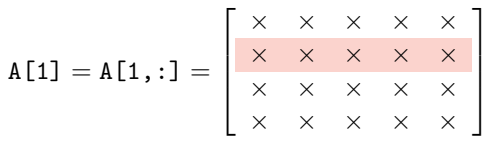
The entries of a 2-D array are the rows of the matrix (as 1-D arrays). To access a single entry, enter the row index, a comma, and the column index. Remember that indexing begins with 0.

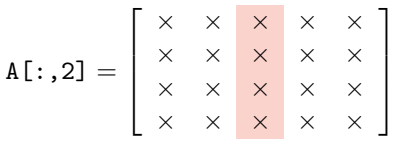


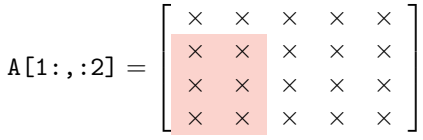


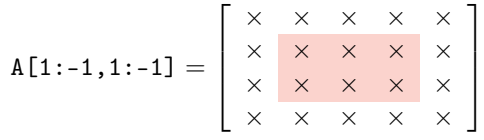
Slicing

A lone colon extracts an entire row or column from a 2-D array. The syntax `[a:b]` can be read as "the ath entry up to (but not including) the bth entry." Similarly, `[a:]` means "the ath entry to the end" and `[:b]` means "everything up to (but not including) the bth entry."



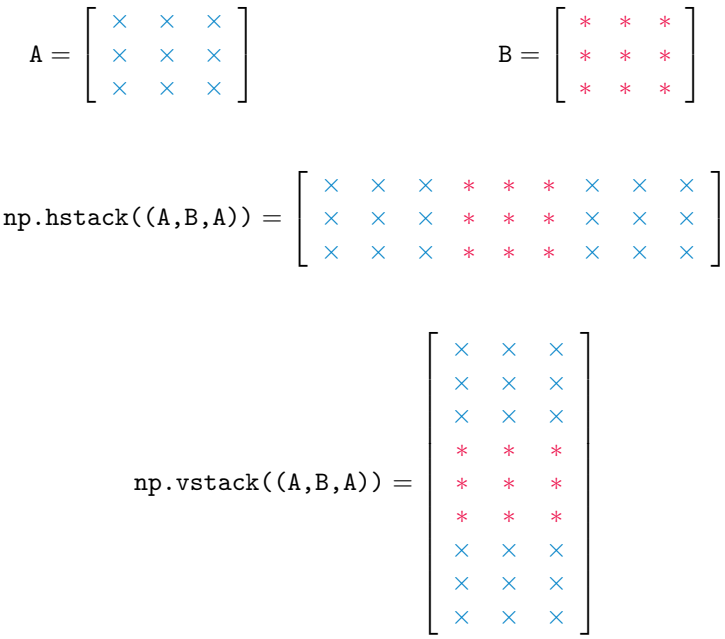




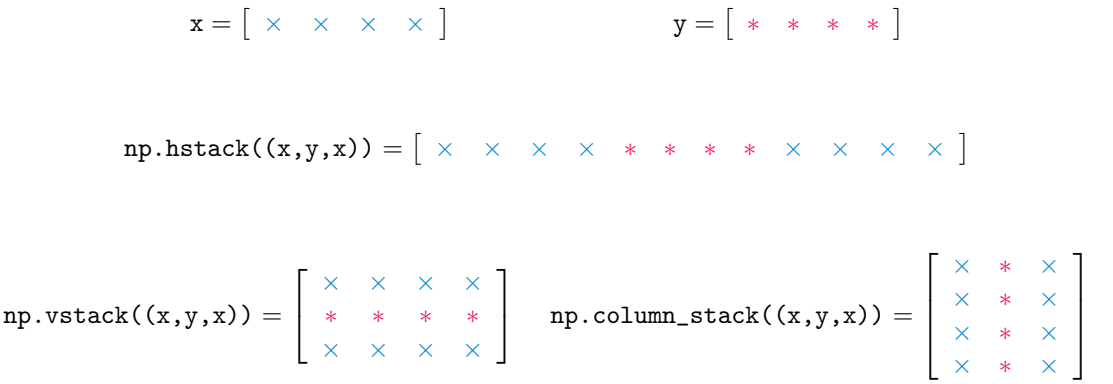


Stacking

`np.hstack()` stacks sequence of arrays horizontally and `np.vstack()` stacks a sequence of arrays vertically.



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()`.



The functions `np.concatenate()` and `np.stack()` are more general versions of `np.hstack()` and `np.vstack()`, and `np.row_stack()` is an alias for `np.vstack()`.

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.

$$\texttt{A} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \qquad \texttt{x} = \begin{bmatrix} 10 & 20 & 30 \end{bmatrix}$$

$$\texttt{A + x} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 & 20 & 30 \end{bmatrix} = \begin{bmatrix} 11 & 22 & 33 \\ 11 & 22 & 33 \\ 11 & 22 & 33 \end{bmatrix}$$

$$\texttt{A + x.reshape((1,-1))} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$$

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`.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$\texttt{A.sum(axis=0)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 8 & 12 & 16 \end{bmatrix}$$

$$\texttt{A.sum(axis=1)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 10 & 10 \end{bmatrix}$$

B Matplotlib Syntax and Customization 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. It is not intended to be read all at once, but rather to be used as a reference when needed. For an interative introduction to Matplotlib, see the Introduction to Matplotlib lab in Python Essentials. For more details on any specific function, refer to the Matplotlib documentation at* `https://matplotlib.org/`.

Matplotlib Interface

Matplotlib plots are made in a `Figure` object that contains one or more `Axes`, which themselves contain the graphical plotting data. Matplotlib provides two ways to create plots:

1. Call plotting functions directly from the module, such as `plt.plot()`. This will create the plot on whichever `Axes` is currently active.

2. Call plotting functions from an `Axes` object, such as `ax.plot()`. This is particularly useful for complicated plots and for animations.

Table B.1 contains a summary of functions that are used for managing `Figure` and `Axes` objects.

Function	Description
`add_subplot()`	Add a single subplot to the current figure
`axes()`	Add an axes to the current figure
`clf()`	Clear the current figure
`figure()`	Create a new figure or grab an existing figure
`gca()`	Get the current axes
`gcf()`	Get the current figure
`subplot()`	Add a single subplot to the current figure
`subplots()`	Create a figure and add several subplots to it

Table B.1: Basic functions for managing plots.

`Axes` objects are usually managed through the functions `plt.subplot()` and `plt.subplots()`. The function `subplot()` is used as `plt.subplot(nrows, ncols, plot_number)`. Note that if the inputs for `plt.subplot()` are all integers, the commas between the entries can be omitted. For example, `plt.subplot(3,2,2)` can be shortened to `plt.subplot(322)`.

The function `subplots()` is used as `plt.subplots(nrows, ncols)`, and returns a `Figure` object and an array of `Axes`. This array has the shape `(nrows, ncols)`, and can be accessed as any other array. Figure B.1 demonstrates the layout and indexing of subplots.

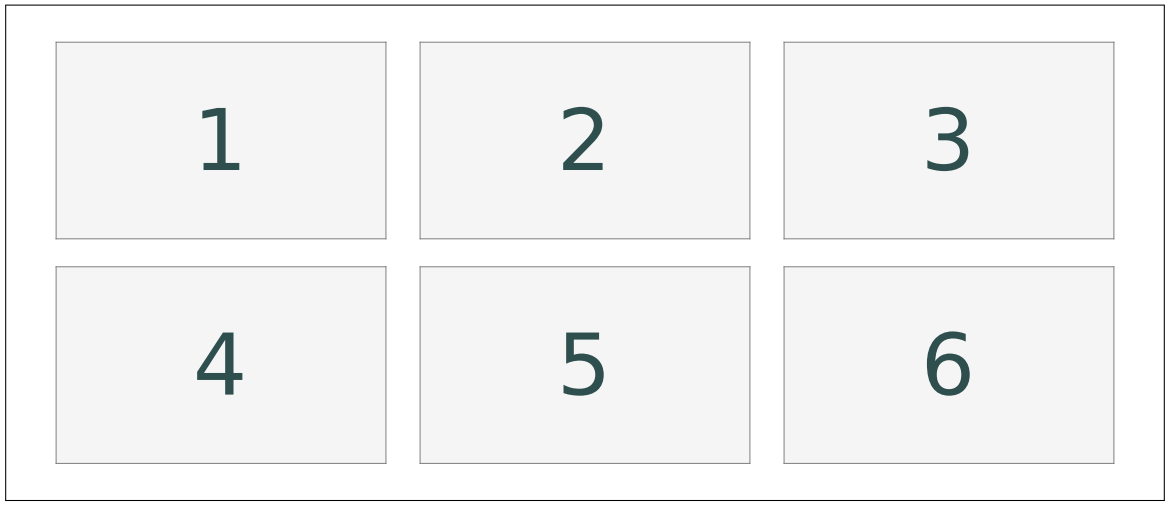


Figure B.1: The layout of subplots with `plt.subplot(2,3,i)` (2 rows, 3 columns), where `i` is the index pictured above. The outer border is the figure that the axes belong to.

The following example demonstrates three equivalent ways of producing a figure with two subplots, arranged next to each other in one row:

```
>>> x = np.linspace(-5, 5, 100)

# 1. Use plt.subplot() to switch the current axes.
>>> plt.subplot(121)
>>> plt.plot(x, 2*x)
>>> plt.subplot(122)
>>> plt.plot(x, x**2)

# 2. Use plt.subplot() to explicitly grab the two subplot axes.
>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, 2*x)
>>> ax2 = plt.subplot(122)
>>> ax2.plot(x, x**2)

# 3. Use plt.subplots() to get the figure and all subplots simultaneously.
>>> fig, axes = plt.subplots(1, 2)
>>> axes[0].plot(x, 2*x)
>>> axes[1].plot(x, x**2)
```

Achtung!

Be careful not to mix up the following similarly-named functions:

1. `plt.axes()` creates a new place to draw on the figure, while `plt.axis()` or `ax.axis()` sets properties of the x- and y-axis in the current axes, such as the x and y limits.
2. `plt.subplot()` (singular) returns a single subplot belonging to the current figure, while `plt.subplots()` (plural) creates a new figure and adds a collection of subplots to it.

Plot Customization

Styles

Matplotlib has a number of built-in styles that can be used to set the default appearance of plots. These can be used via the function `plt.style.use()`; for instance, `plt.style.use("seaborn")` will have Matplotlib use the "seaborn" style for all plots created afterwards. A list of built-in styles can be found at `https://matplotlib.org/stable/gallery/style_sheets/style_sheets_reference.html`.

The style can also be changed only temporarily using `plt.style.context()` along with a `with` block:

```
with plt.style.context('dark_background'):
    # Any plots created here use the new style
    plt.subplot(1,2,1)
    plt.plot(x, y)
    # ...
# Plots created here are unaffected
plt.subplot(1,2,2)
plt.plot(x, y)
```

Plot layout

Axis properties

Table B.2 gives an overview of some of the functions that may be used to configure the axes of a plot.

The functions `xlim()`, `ylim()`, and `axis()` are used to set one or both of the x and y ranges of the plot. `xlim()` and `ylim()` each accept two arguments, the lower and upper bounds, or a single list of those two numbers. `axis()` accepts a single list consisting, in order, of `xmin, xmax, ymin, ymax`. Passing `None` instead of one of the numbers to any of these functions will make it not change the corresponding value from what it was. Each of these functions can also be called without any arguments, in which case it will return the current bounds. Note that `axis()` can also be called directly on an `Axes` object, while `xlim()` and `ylim()` cannot.

`axis()` also can be called with a string as its argument, which has several options. The most common is `axis('equal')`, which makes the scale of the x- and y-scales equal (i.e. makes circles circular).

Function	Description
`axis()`	set the x- and y-limits of the plot
`grid()`	add gridlines
`xlim()`	set the limits of the x-axis
`ylim()`	set the limits of the y-axis
`xticks()`	set the location of the tick marks on the x-axis
`yticks()`	set the location of the tick marks on the y-axis
`xscale()`	set the scale type to use on the x-axis
`yscale()`	set the scale type to use on the y-axis
`ax.spines[side].set_position()`	set the location of the given spine
`ax.spines[side].set_color()`	set the color of the given spine
`ax.spines[side].set_visible()`	set whether a spine is visible

Table B.2: Some functions for changing axis properties. `ax` is an `Axes` object.

To use a logarithmic scale on an axis, the functions `xscale("log")` and `yscale("log")` can be used.

The functions `xticks()` and `yticks()` accept a list of tick positions, which the ticks on the corresponding axis are set to. Generally, this works the best when used with `np.linspace()`. This function also optionally accepts a second argument of a list of labels for the ticks. If called with no arguments, the function returns a list of the current tick positions and labels instead.

The spines of a Matplotlib plot are the black border lines around the plot, with the left and bottom ones also being used as the axis lines. To access the spines of a plot, call `ax.spines[side]`, where `ax` is an `Axes` object and `side` is `'top'`, `'bottom'`, `'left'`, or `'right'`. Then, functions can be called on the `Spine` object to configure it.

The function `spine.set_position()` has several ways to specify the position. The two simplest are with the arguments `'center'` and `'zero'`, which place the spine in the center of the subplot or at an x- or y-coordinate of zero, respectively. The others are a passed as a tuple `(position_type, amount)`:

- `'data'`: place the spine at an x- or y-coordinate equal to `amount`.
- `'axes'`: place the spine at the specified `Axes` coordinate, where 0 corresponds to the bottom or left of the subplot, and 1 corresponds to the top or right edge of the subplot.
- `'outward'`: places the spine `amount` pixels outward from the edge of the plot area. A negative value can be used to move it inwards instead.

`spine.set_color()` accepts any of the color formats Matplotlib supports. Alternately, using `set_color('none')` will make the spine not be visible. `spine.set_visible()` can also be used for this purpose.

The following example adjusts the ticks and spine positions to improve the readability of a plot of $\sin(x)$. The result is shown in Figure B.2.

```
>>> x = np.linspace(0,2*np.pi,150)
>>> plt.plot(x, np.sin(x))
>>> plt.title(r"$y=\sin(x)$")

#Set the ticks to multiples of pi/2, make nice labels
>>> ticks = np.pi / 2 * np.array([0,1,2,3,4])
```

```
>>> tick_labels = ["$0$", r"$\frac{\pi}{2}$", r"$\pi$", r"$\frac{3\pi}{2}$",
...                 r"$2\pi$"]
>>> plt.xticks(ticks, tick_labels)

#Move the bottom spine to zero, remove the top and right ones
>>> ax = plt.gca()
>>> ax.spines['bottom'].set_position('zero')
>>> ax.spines['right'].set_color('none')
>>> ax.spines['top'].set_color('none')

>>> plt.show()
```

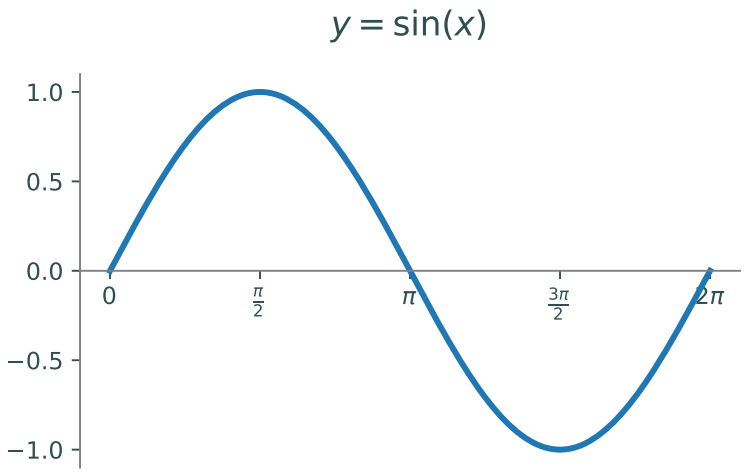


Figure B.2: Plot of $y = \sin(x)$ with axes modified for clarity

Plot Layout

The position and spacing of all subplots within a figure can be modified using the function `plt.subplots_adjust()`. This function accepts up to six keyword arguments that change different aspects of the spacing. `left`, `right`, `top`, and `bottom` are used to adjust the rectangle around all of the subplots. In the coordinates used, 0 corresponds to the bottom or left edge of the figure, and 1 corresponds to the top or right edge of the figure. `hspace` and `wspace` set the vertical and horizontal spacing, respectively, between subplots. The units for these are in fractions of the average height and width of all subplots in the figure. If more fine control is desired, the position of individual `Axes` objects can also be changed using `ax.get_position()` and `ax.set_position()`.

The size of the figure can be configured using the `figsize` argument when creating a figure:

```
>>> plt.figure(figsize=(12,8))
```

Note that many environments will scale the figure to fill the available space. Even so, changing the figure size can still be used to change the aspect ratio as well as the relative size of plot elements.

The following example uses `subplots_adjust()` to create space for a legend outside of the plotting space. The result is shown in Figure B.3.

```
#Generate data
>>> x1 = np.random.normal(-1, 1.0, size=60)
>>> y1 = np.random.normal(-1, 1.5, size=60)
>>> x2 = np.random.normal(2.0, 1.0, size=60)
>>> y2 = np.random.normal(-1.5, 1.5, size=60)
>>> x3 = np.random.normal(0.5, 1.5, size=60)
>>> y3 = np.random.normal(2.5, 1.5, size=60)

#Make the figure wider
>>> fig = plt.figure(figsize=(5,3))

#Plot the data
>>> plt.plot(x1, y1, 'r.', label="Dataset 1")
>>> plt.plot(x2, y2, 'g.', label="Dataset 2")
>>> plt.plot(x3, y3, 'b.', label="Dataset 3")

#Create a legend to the left of the plot
>>> lspace = 0.35
>>> plt.subplots_adjust(left=lspace)
#Put the legend at the left edge of the figure
>>> plt.legend(loc=(-lspace/(1-lspace),0.6))
>>> plt.show()
```

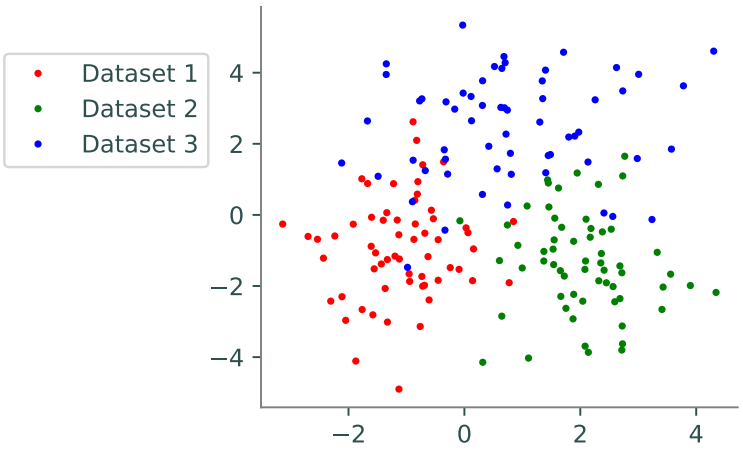


Figure B.3: Example of repositioning axes.

Colors

The color that a plotting function uses is specified by either the `c` or `color` keyword arguments; for most functions, these can be used interchangeably. There are many ways to specific colors. The most simple is to use one of the basic colors, listed in Table B.3. Colors can also be specified using an RGB tuple such as `(0.0, 0.4, 1.0)`, a hex string such as `"0000FF"`, or a CSS color name like `"DarkOliveGreen"` or `"FireBrick"`. A full list of named colors that Matplotlib supports can be found at `https://matplotlib.org/stable/gallery/color/named_colors.html`. If no color is specified for a plot, Matplotlib automatically assigns it one from the default color cycle.

Code	Color
`'b'`	blue
`'g'`	green
`'r'`	red
`'c'`	cyan
`'m'`	magenta

Code	Color
`'y'`	yellow
`'k'`	black
`'w'`	white
`'C0'` - `'C9'`	Default colors

Table B.3: Basic colors available in Matplotlib

Plotting functions also accept an `alpha` keyword argument, which can be used to set the transparency. A value of `1.0` corresponds to fully opaque, and `0.0` corresponds to fully transparent.

The following example demonstrates different ways of specifying colors:

```
#Using a basic color
>>> plt.plot(x, y, 'r')
#Using a hexadecimal string
>>> plt.plot(x, y, color='FF0080')
#Using an RGB tuple
>>> plt.plot(x, y, color=(1, 0.5, 0))
#Using a named color
>>> plt.plot(x, y, color='navy')
```

Colormaps

Certain plotting functions, such as heatmaps and contour plots, accept a colormap rather than a single color. A full list of colormaps available in Matplotlib can be found at `https://matplotlib.org/stable/gallery/color/colormap_reference.html`. Some of the more commonly used ones are `"viridis"`, `"magma"`, and `"coolwarm"`. A colorbar can be added by calling `plt.colorbar()` after creating the plot.

Sometimes, using a logarithmic scale for the coloring is more informative. To do this, pass a `matplotlib.colors.LogNorm` object as the `norm` keyword argument:

```
# Create a heatmap with logarithmic color scaling
>>> from matplotlib.colors import LogNorm
>>> plt.pcolormesh(X, Y, Z, cmap='viridis', norm=LogNorm())
```

Function	Description	Usage
`annotate()`	adds a commentary at a given point on the plot	annotate('text',(x,y))
`arrow()`	draws an arrow from a given point on the plot	arrow(x,y,dx,dy)
`colorbar()`	Create a colorbar	colorbar()
`legend()`	Place a legend in the plot	legend(loc='best')
`text()`	Add text at a given position on the plot	text(x,y,'text')
`title()`	Add a title to the plot	title('text')
`suptitle()`	Add a title to the figure	suptitle('text')
`xlabel()`	Add a label to the x-axis	xlabel('text')
`ylabel()`	Add a label to the y-axis	ylabel('text')

Table B.4: Text and annotation functions in Matplotlib

Text and Annotations

Matplotlib has several ways to add text and other annotations to a plot, some of which are listed in Table B.4. The color and size of the text in most of these functions can be adjusted with the `color` and `fontsize` keyword arguments.

Matplotlib also supports formatting text with LaTeX, a system for creating technical documents.[1] To do so, use an `r` before the string quotation mark and surround the text with dollar signs. This is particularly useful when the text contains a mathematical expression. For example, the following line of code will make the title of the plot be $\frac{1}{2}\sin(x^2)$:

```
>>> plt.title(r"$\frac{1}{2}\sin(x^2)$")
```

The function `legend()` can be used to add a legend to a plot. Its optional `loc` keyword argument specifies where to place the legend within the subplot. It defaults to `'best'`, which will cause Matplotlib to place it in whichever location overlaps with the fewest drawn objects. The other locations this function accepts are `'upper right'`, `'upper left'`, `'lower left'`, `'lower right'`, `'center left'`, `'center right'`, `'lower center'`, `'upper center'`, and `'center'`. Alternately, a tuple of `(x,y)` can be passed as this argument, and the bottom-left corner of the legend will be placed at that location. The point (0,0) corresponds to the bottom-left of the current subplot, and (1,1) corresponds to the top-right. This can be used to place the legend outside of the subplot, although care should be taken that it does not go outside the figure, which may require manually repositioning the subplots.

The labels the legend uses for each curve or scatterplot are specified with the `label` keyword argument when plotting the object. Note that `legend()` can also be called with non-keyword arguments to set the labels, although it is less confusing to set them when plotting.

The following example demonstrates creating a legend:

```
>>> x = np.linspace(0,2*np.pi,250)

# Plot sin(x), cos(x), and -sin(x)
# The label argument will be used as its label in the legend.
>>> plt.plot(x, np.sin(x), 'r', label=r'$\sin(x)$')
>>> plt.plot(x, np.cos(x), 'g', label=r'$\cos(x)$')
>>> plt.plot(x, -np.sin(x), 'b', label=r'$-\sin(x)$')
```

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

```
# Create the legend
>>> plt.legend()
```

Line and marker styles

Matplotlib supports a large number of line and marker styles for line and scatter plots, which are listed in Table B.5.

character	description	character	description
-	solid line style	3	tri_left marker
--	dashed line style	4	tri_right marker
-.	dash-dot line style	s	square marker
:	dotted line style	p	pentagon marker
.	point marker	*	star marker
,	pixel marker	h	hexagon1 marker
o	circle marker	H	hexagon2 marker
v	triangle_down marker	+	plus marker
^	triangle_up marker	x	x marker
<	triangle_left marker	D	diamond marker
>	triangle_right marker	d	thin_diamond marker
1	tri_down marker	|	vline marker
2	tri_up marker	_	hline marker

Table B.5: Available line and marker styles in Maplotlib.

The function `plot()` has several ways to specify this argument; the simplest is to pass it as the third positional argument. The `marker` and `linestyle` keyword arguments can also be used. The size of these can be modified using `markersize` and `linewidth`. Note that by specifying a marker style but no line style, `plot()` can be used to make a scatter plot. It is also possible to use both a marker style and a line style. To set the marker using `scatter()`, use the `marker` keyword argument, with `s` being used to change the size.

The following code demonstrates specifying marker and line styles. The results are shown in Figure B.4.

```
#Use dashed lines:
>>> plt.plot(x, y, '--')
#Use only dots:
>>> plt.plot(x, y, '.')
#Use dots with a normal line:
>>> plt.plot(x, y, '.-')
#scatter() uses the marker keyword:
>>> plt.scatter(x, y, marker='+')

#With plot(), the color to use can also be specified in the same string.
#Order usually doesn't matter.
#Use red dots:
>>> plt.plot(x, y, '.r')
```

```
#Equivalent:
>>> plt.plot(x, y, 'r.')

#To change the size:
>>> plt.plot(x, y, 'v-', linewidth=1, markersize=15)
>>> plt.scatter(x, y, marker='+', s=12)
```

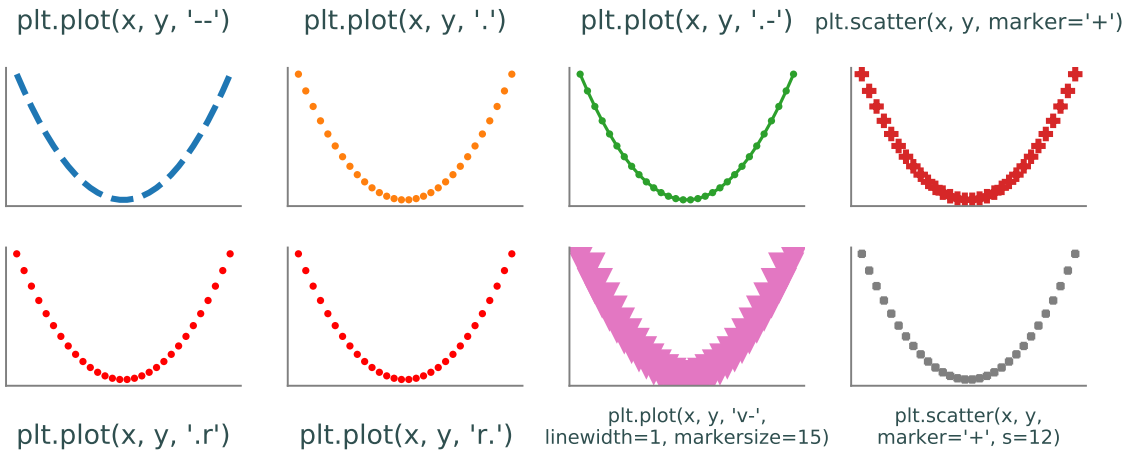


Figure B.4: Examples of setting line and marker styles.

Plot Types

Matplotlib has functions for creaing many different types of plots, many of which are listed in Table B.6. This section gives details on using certain groups of these functions.

Function	Description	Usage
`bar`	makes a bar graph	bar(x,height)
`barh`	makes a horizontal bar graph	barh(y,width)
`boxplots`	makes one or more boxplots	boxplots(data)
`contour`	makes a contour plot	contour(X,Y,Z)
`contourf`	makes a filled contour plot	contourf(X,Y,Z)
`imshow`	shows an image	imshow(image)
`fill`	plots lines with shading under the curve	fill(x,y)
`fill_between`	plots lines with shading between two given y values	fill_between(x,y1, y2=0)
`hexbin`	creates a hexbin plot	hexbin(x,y)
`hist`	plots a histogram from data	hist(data)
`pcolormesh`	makes a heatmap	pcolormesh(X,Y,Z)
`pie`	makes a pie chart	pie(x)
`plot`	plots lines and data on standard axes	plot(x,y)
`plot_surface`	plot a surface in 3-D space	plot_surface(X,Y,Z)
`polar`	plots lines and data on polar axes	polar(theta,r)
`loglog`	plots lines and data on logarithmic x and y axes	loglog(x,y)
`scatter`	plots data in a scatterplot	scatter(x,y)
`semilogx`	plots lines and data with a log scaled x axis	semilogx(x,y)
`semilogy`	plots lines and data with a log scaled y axis	semilogy(x,y)
`specgram`	makes a spectogram from data	specgram(x)
`spy`	plots the sparsity pattern of a 2D array	spy(Z)
`triplot`	plots triangulation between given points	triplot(x,y)

Table B.6: Some basic plotting functions in Matplotlib.

Line plots

Line plots, the most basic type of plot, are created with the `plot()` function. It accepts two lists of x- and y-values to plot, and optionally a third argument of a string of any combination of the color, line style, and marker style. Note that this method only works with the single-character color codes; to use other colors, use the `color` argument. By specifying only a marker style, this function can also be used to create scatterplots.

There are a number of functions that do essentially the same thing as `plot()` but also change the axis scaling, including `loglog()`, `semilogx()`, `semilogy()`, and `polar`. Each of these functions is used in the same manner as `plot()`, and has identical syntax.

Bar Plots

Bar plots are a way to graph categorical data in an effective way. They are made using the `bar()` function. The most important arguments are the first two that provide the data, `x` and `height`. The first argument is a list of values for each bar, either categorical or numerical; the second argument is a list of numerical values corresponding to the height of each bar. There are other parameters that may be included as well. The `width` argument adjusts the bar widths; this can be done by choosing a single value for all of the bars, or an array to give each bar a unique width. Further, the argument `bottom` allows one to specify where each bar begins on the y-axis. Lastly, the `align` argument can be set to 'center' or 'edge' to align as desired on the x-axis. As with all plots, you can use the `color` keyword to specify any color of your choice. If you desire to make a horizontal bar graph, the syntax follows similarly using the function `barh()`, but with argument names `y`, `width`, `height` and `align`.

Box Plots

A box plot is a way to visualize some simple statistics of a dataset. It plots the minimum, maximum, and median along with the first and third quartiles of the data. This is done by using `boxplot()` with an array of data as the argument. Matplotlib allows you to enter either a one dimensional array for a single box plot, or a 2-dimensional array where it will plot a box plot for each column of the data in the array. Box plots default to having a vertical orientation but can be easily laid out horizontally by setting `vert=False`.

Scatter and hexbin plots

Scatterplots can be created using either `plot()` or `scatter()`. Generally, it is simpler to use `plot()`, although there are some cases where `scatter()` is better. In particular, `scatter()` allows changing the color and size of individual points within a single call to the function. This is done by passing a list of colors or sizes to the `c` or `s` arguments, respectively.

Hexbin plots are an alternative to scatterplots that show the concentration of data in regions rather than the individual points. They can be created with the function `hexbin()`. Like `plot()` and `scatter()`, this function accepts two lists of x- and y-coordinates.

Heatmaps and contour plots

Heatmaps and contour plots are used to visualize 3-D surfaces and complex-valued functions on a flat space. Heatmaps are created using the `pcolormesh()` function. Contour plots are created using `contour()` or `contourf()`, with the latter creating a filled contour plot.

Each of these functions accepts the x-, y-, and z-coordinates as a mesh grid, or 2-D array. To create these, use the function `np.meshgrid()`:

```
>>> x = np.linspace(0,1,100)
>>> y = np.linspace(0,1,80)
>>> X, Y = np.meshgrid(x, y)
```

The z-coordinate can then be computed using the x and y mesh grids.

Note that each of these functions can accept a colormap, using the `cmap` parameter. These plots are sometimes more informative with a logarithmic color scale, which can be used by passing a `matplotlib.colors.LogNorm` object in the `norm` parameter of these functions.

With `pcolormesh()`, it is also necessary to pass `shading='auto'` or `shading='nearest'` to avoid a deprecation error.

The following example demonstrates creating heatmaps and contour plots, using a graph of $z = (x^2 + y)\sin(y)$. The results is shown in Figure B.5

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

>>> x = np.linspace(-3,3,100)
>>> y = np.linspace(-3,3,100)
>>> X, Y = np.meshgrid(x, y)
>>> Z = (X**2+Y)*np.sin(Y)

#Heatmap
>>> plt.subplot(1,3,1)
```

```
>>> plt.pcolormesh(X, Y, Z, cmap='viridis', shading='nearest')
>>> plt.title("Heatmap")

#Contour
>>> plt.subplot(1,3,2)
>>> plt.contour(X, Y, Z, cmap='magma')
>>> plt.title("Contour plot")

#Filled contour
>>> plt.subplot(1,3,3)
>>> plt.contourf(X, Y, Z, cmap='coolwarm')
>>> plt.title("Filled contour plot")
>>> plt.colorbar()

>>> plt.show()
```

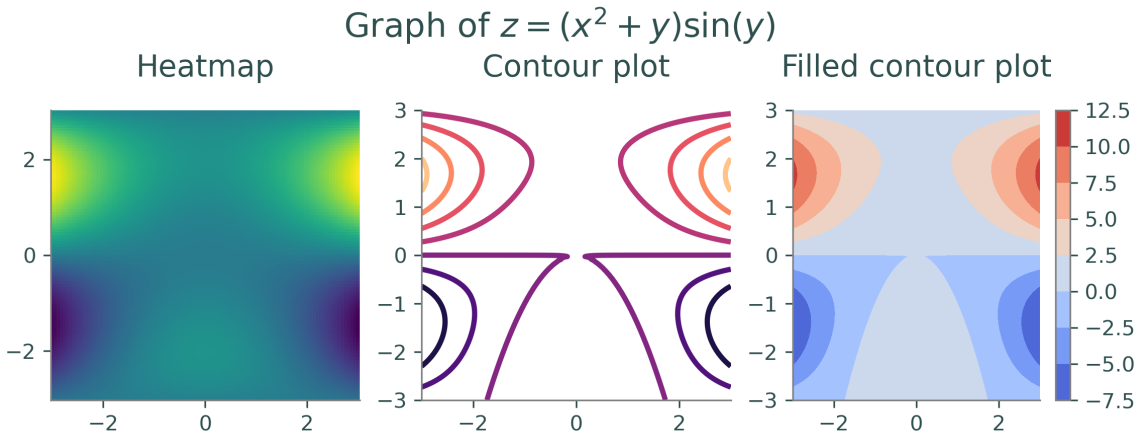


Figure B.5: Example of heatmaps and contour plots.

Showing images

The function `imshow()` is used for showing an image in a plot, and can be used on either grayscale or color images. This function accepts a 2-D $n \times m$ array for a grayscale image, or a 3-D $n \times m \times 3$ array for a color image. If using a grayscale image, you also need to specify `cmap='gray'`, or it will be colored incorrectly.

It is best to also use `axis('equal')` alongside `imshow()`, or the image will most likely be stretched. This function also works best if the images values are in the range $[0, 1]$. Some ways to load images will format their values as integers from 0 to 255, in which case the values in the image array should be scaled before using `imshow()`.

3-D Plotting

Matplotlib can be used to plot curves and surfaces in 3-D space. In order to use 3-D plotting, you need to run the following line:

```
>>> from mpl_toolkits.plot3d import Axes3D
```

The argument `projection='3d'` also must be specified when creating the subplot for the 3-D object:

```
>>> plt.subplot(1,1,1, projection='3d')
```

Curves can be plotted in 3-D space using `plot()`, by passing in three lists of x-, y-, and z-coordinates. Surfaces can be plotted using `ax.plot_surface()`. This function can be used similar to creating contour plots and heatmaps, by obtaining meshes of x- and y- coordinates from `np.meshgrid()` and using those to produce the z-axis. More generally, any three 2-D arrays of meshes corresponding to x-, y-, and z-coordinates can be used. Note that it is necessary to call this function from an Axes object.

The following example demonstrates creating 3-D plots. The results are shown in Figure B.6.

```
#Create a plot of a parametric curve
ax = plt.subplot(1,3,1, projection='3d')
t = np.linspace(0, 4*np.pi, 160)
x = np.cos(t)
y = np.sin(t)
z = t / np.pi
plt.plot(x, y, z, color='b')
plt.title("Helix curve")

#Create a surface plot from np.meshgrid
ax = plt.subplot(1,3,2, projection='3d')
x = np.linspace(-1,1,80)
y = np.linspace(-1,1,80)
X, Y = np.meshgrid(x, y)
Z = X**2 - Y**2
ax.plot_surface(X, Y, Z, color='g')
plt.title(r"Hyperboloid")

#Create a surface plot less directly
ax = plt.subplot(1,3,3, projection='3d')
theta = np.linspace(-np.pi,np.pi,80)
rho = np.linspace(-np.pi/2,np.pi/2,40)
Theta, Rho = np.meshgrid(theta, rho)
X = np.cos(Theta) * np.cos(Rho)
Y = np.sin(Theta) * np.cos(Rho)
Z = np.sin(Rho)
ax.plot_surface(X, Y, Z, color='r')
plt.title(r"Sphere")

plt.show()
```

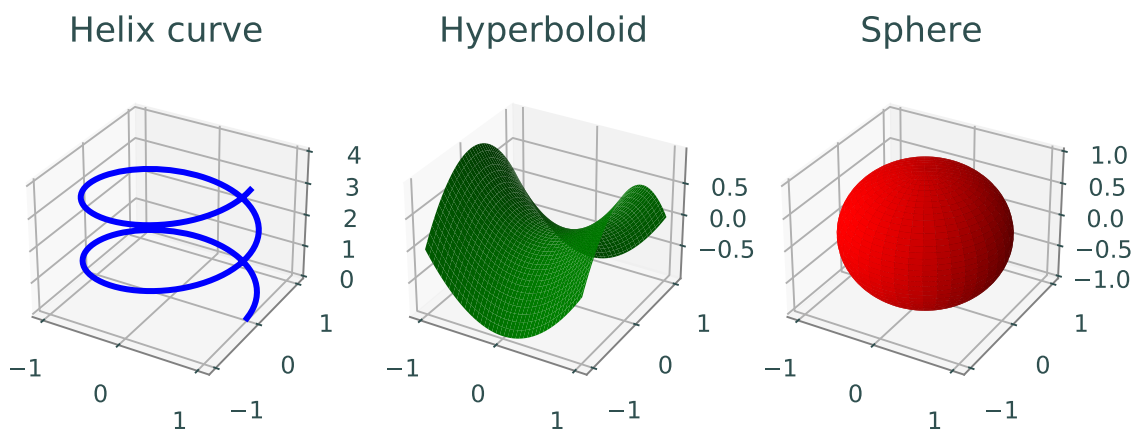


Figure B.6: Examples of 3-D plotting.

Additional Resources

rcParams

The default plotting parameters of Matplotlib can be set individually and with more fine control than styles by using `rcParams`. `rcParams` is a dictionary that can be accessed as either `plt.rcParams` or `matplotlib.rcParams`.

For instance, the resolution of plots can be changed via the `"figure.dpi"` parameter:

```
>>> plt.rcParams["figure.dpi"] = 600
```

A list of parameters that can set via `rcParams` can be found at `https://matplotlib.org/stable/api/matplotlib_configuration_api.html#matplotlib.RcParams`.

Animations

Matplotlib has capabilities for creating animated plots. The Animations lab in Volume 4 has detailed instructions on how to do so.

Matplotlib gallery and tutorials

The Matplotlib documentation has a number of tutorials, found at `https://matplotlib.org/stable/tutorials/index.html`. It also has a large gallery of examples, found at `https://matplotlib.org/stable/gallery/index.html`. Both of these are excellent sources of additional information about ways to use and customize Matplotlib.

Bibliography

[ADH+01] David Ascher, Paul F Dubois, Konrad Hinsen, Jim Hugunin, Travis Oliphant, et al. Numerical python, 2001.

[Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[jup] Jupyter notebooks—a publishing format for reproducible computational workflows. pages 87–90.

[MSP+17] Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondrej Certík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, 2017.

[Oli06] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[Oli07] Travis E Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3), 2007.

[PG07] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, may 2007.

[VD10] Guido VanRossum and Fred L Drake. *The python language reference*. Python software foundation Amsterdam, Netherlands, 2010.