IAHR & IWRA UIUC Student Chapters

Python Workshop (Basic and Advanced)

Sushobhan Sen

Doctoral Candidate

October 5, 2019

Urbana, IL

Contents

1 Introduction

1.1 Why Python?

The Python programming language was conceived in the 1980s and the first implementation was deployed in 1991 by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands. Python is designed to be a high-level, interpreted, general-purpose computer programming language with exception-handling and the ability to be extended by users. Crucially, Python is an open-source language, meaning anybody can see the code behind it. Python today is a very popular language is a broad variety of disciplines: machine learning, web development, Geographical Information Systems (GIS), engineering, database management, high performance computing, etc.

Python 2.0, which truly launched the popularity of the language, was released in 2000, with the last version in the series, Python 2.7, set to be phased out in 2020. The current recommended standard as of when I wrote this is Python 3.7.4, which is crucially **not backwards compatible** with Python 2.x. All subversions of Python 3.x are backwards compatible though, so feel free to update regularly.

(The above information was taken from Wikipedia)

The beauty of Python is its ease - the syntax is extremely simple as compared to many other popular programming languages. It is often said, not entirely without truth, that the difference between Python code and pseudocode is merely the indentation. Thus, the language is very easy to learn for programming novices. However, its ease should not be misconstrued to mean that it is only for easy or trivial applications: Python is an extremely powerful language with a wide variety of applications, it has extensive documentation, and its open-source nature ensures that new features are being developed continuously while bugs are also being fixed.

1.2 Learning Objectives

This workshop is broken up into two sessions:

1. A Basic Python session for beginners with zero knowledge of programming in any language

2. An Advanced Python session for those with prior knowledge of Python programming

At the end of the **Basic Python** session, participants will be able to:

1. List the types of Python variables and define them
2. Add control statements (`if-then-else`) and `for` loops to their program
3. Define and use functions
4. Define and use object oriented programming

At the end of the **Advanced Python** session, participants will be able to:

1. Use the `numpy` library to define and use matrices, and read and process data from files
2. Use the `pandas` library to read and analyze data
3. Use the `scipy` library to process an image, analyze a sound wave, and fit data to a curve
4. Use the `matplotlib` library to create publication-quality plots for 1D, 2D, and 3D data

The lists above are what I will try to accomplish in our two two-hour long sessions. Depending on how quickly the workshop goes, I may or may not be able to meet all the learning objectives. However, I will get you along far enough so that you can complete any remaining items on your own.

1.3 Installing Python

The easiest and **recommended** way to get all popular Python libraries and IDEs is by downloading and installing the latest Anaconda distribution for your computer. Make sure to download the latest version corresponding to Python 3.x.

Alternatively, you can install and use Python using the terminal:

- On Windows 10, activate Windows Subsystem for Linux
- On Mac OS or any UNIX-like OS (such as Linux), just run your favorite Terminal application

Then install Jupyter with `pip`. With any other version of Windows (which you should not be using for too long on your personal computers anyway for security reasons), Anaconda is your best option.

Once installed, you can now create a new notebook. If you installed Anaconda, open the Anaconda Prompt, navigate to your directory, and use the `jupyter notebook` command. If you choose to use the terminal, follow the same steps on the terminal. Both methods will launch Jupyter Notebooks in your browser. From there, you can create a new Notebook.

If you prefer not to install anything on your computer but would rather run Python remotely from your browser, you can use the Online IDE from repl.it. This doesn't always work very well though. Any other online IDE that you find should be OK too.

2 Variables in Python

2.1 Data Types

Python defines the following data types for variables:

Data Type	**Syntax**	**Description**	**Comments**
Integer	`x = 5`	Signed integer	Use `int` to typecast, if valid
Float	`x = 5.`	IEEE floating point number	Use `float` to typecast, if valid
Complex	`x = 3.1+4.6j`	Complex number	Use `complex` to typecast, if valid
String	`x = 'Hello World!'`	Strings are always enclosed within quotation marks	Use `str` to typecast, slicing operator valid
List	`x = [1, 2.2, 'otter']`	List of variables of any type	Use `list()` to typecast, if valid
Tuple	`x = (1, 2.2, 'otter')`	Tuple of variables of any tupe	Use `tuple()` to typecast, if valid
Dictionary	`x = {'one':1, 'two':2}`	Key-value pairs	Use `get()` to get `value` from `key`, use `dict()` to typecast, if valid
Bool	`x = True`	Boolean value	Python uses keywords `True` and `False`, not 1 and 0

2.2 Creating a Variable

Variables are usually given a name, which is any string of characters you use to call it. Any string of characters is a valid name, although there are a couple of rules to follow:

1. The name **cannot** contain spaces - consider using an underscore character or camelCase instead for readability
2. The name cannot start with a number, but can contain a number anywhere
3. The name is case-sensitive, so `temp` and `Temp` are different variables
4. Python has a number of reserved keywords (see documentation, of just follow along and you'll get the hang of it), which cannot be used as names

Creating a variable is as simple as giving it a name, and assigning a value to it. For example, run this piece of code:

```
1 x = 1.0
2 y = 'Hello, World!'
3 z = (x==y)
4 print(type(x),type(y),type(z),z)
```

Here, we defined three variables: `x` of type `int`, `y` of type `str`, and `z` of type `bool`. Note that the `==` is a comparison operator that compares two values, while `=` is an assignment operator that assigns a value to a variable. The `print()` function, as the name suggests, prints the comma-separated inputs as a string , while the `type()` function returns the data type of the input. Note that every new line of code in Python starts on a new line, and **there is no end of line character** (such as a semi-colon in many languages).

2.3 Mutability

Some Python data types are **immutable**. This means that, once defined, ibjects of those types cannot be changed without creating a new object entirely. Most objects are immutable: `int, float, complex, str, bool, tuple` are immutable. If you try to change their values, you will either get an error or a new object containing the new value will be created. For example, run the following code, where `id()` is a function that returns the memory location of the object passed to it:

```
1 x = 1
2 print(id(x))
3 x = 2.0
4 print(id(x))
```

You'll see that the memory location of `x` has changed entirely, instead of just the value being changed. This is because `x` was defined as type `int`, which is immutable. Similarly, try changing the value inside a tuple and see what happens (note that `x[0]` is a way to reference the first element of `x` - **Python is zero-indexed**):

```
1 x = (1,2,3)
2 x[0] = 10
3 print(x)
```

Lists and dictionaries are mutable, which means their values can be changed at any time. Try running the following code and compare it to the last one:

```
1 x = [1,2,3]
2 print(id(x))
3 x[0] = 10
4 print(id(x))
5 print(x)
```

2.4 Operations on Variables

Python provides a number of operations that can be performed between variables:

- **Arithmetic operators:** + (add), - (subtract), * (multiply), / (divide), % (modulus or remainder), // (floor division), ** (exponent)
- **Comparison operators:** > (grater than), < (less than), == (equal to), != (not equal to), >= (greater than or equal to), <= (less than or equal to)
- **Logical operators:** `and`, `or`, and `not`

- **Bitwise operators:** & (bitwise AND), | (bitwise OR), ∼ (bitwise NOT), ˆ (bitwise XOR), >> (bitwise right shift), << (bitwise left shift)
- **Assignment operators:** A combination of the basic assignment operator (=) and an optional arithmetic or bitwise operator. For example, `x **= 5` is the same as `x = x**5`
- **Special operators:** `is` or `is not` check if two variables have the same memory address, `in` or `not in` check if a variable is in a sequences of variables

Operators are mostly self-explanatory, but their behavior can be different based on the data type of the input variables. Consider adding two integers, adding an integer to a float, adding two strings, and adding an integer to a string, all of which use the same + operator:

```
1 x = 2
2 print(x+2)
3 print(x+2.5)
4 print('Goodbye'+'Hello')
5 print(x+'Hello')
```

The first operation is as expected and returns an integer. In the second operation, `x` is first typecast to `float` *implicitly* (the program does it by itself) and then added to another float to return a float. In the third operation, two strings are simply concatenated together to return another string.

However, the last operation of adding an integer to a string returns an error, because Python is unable to figure out which variable to cast to which type. Of course, casting a string to an integer makes no sense, but Python doesn't even try, because it's unsure about what to do. You can help it by *explicitly* casting `x` from an integer to string, and then see what happens:

```
1 x = 2
2 print(str(x)+'Hello')
```

3 Control Statements

Control statements are blocks of code that are used to control the flow of the program - that could mean skipping some lines, or repeating some lines a certain number of times. There are two main types of control statements:

1. **Conditional statements**: Better known as if-else blocks, these test a condition before executing a line
2. **Loops**: These repeatedly execute a line for a certain number of times or till a break condition is met

We'll look at each of these below.

3.1 Conditional Statements

The most basic control statement is an `if` statement, which checks a condition and executes the code after it only if the condition evaluates `True`. Consider the following block of code:

```
1 x = 6
2 if x<5:
3     print('x is less than 5')
4     print('Finished evaluating if block')
5 print('Finished this code block')
```

First, notice the syntax: the `if` statement starts with the word itself, followed by a space, and followed by the condition (`x<5`). This line then ends with a colon (:), signifying that things after the colon are to be executed if the condition returns `True`. But how do you tell Python what lines of code are part of the `if`-statement to be executed, and what are not? In other words, how do you signify the scope of the `if`-statement? In this case, we want the two `print()` statements after the `if`-statement to be executed only if the condition return `True`, while the third `print()` statement to be executed regardless. How do we tell it to do that?

That's where indentation (a tab character, usually four spaces long) comes into play. In many computer languages, a whitespace like a tab character is simply ignored. However, in Python, it is an *integral part* of the code - **this**

is a major different between Python and other languages! Mistakes made with indentation can and will return an error, so be careful.

Now, coming back to the code. We want the first two `print()` statements to be executed only if the condition is True, so we indent them by one level (one tab character) with respect to the indentation level of the `if`-statement. The additional indentation level makes these lines associated with the parent line. Whereas, we don't want to last `print()` statement to be associated with the `if`-statement, so we keep it at the same indentation level as the statement, signifying that there is no association or dependency between them. This can seem confusing at first, but it is a very elegant way of writing code and is indeed, almost the same as writing pseudo-code.

To drive home the point, let's put an `if`-statement within another `if`-statement:

```
1 x = 6
2 if x<5:
3     print('x is less than 5')
4     print('Finished evaluating outer if block')
5 
6     if x>2:
7         print('but x is greater than 2')
8         print('Finished evaluating inner if block')
9 print('Finished this code block')
```

Now see the indentation here. The inner `if`-statement is one indentation level after the outer one, indicating that it will be executed only if the outer condition is true. And the `print()` statements associated with the inner statement is at yet another indentation level (two tab characters) or one character after the inner `if`-statement, indicating that they will be executed only if the inner statement is true. This indentation, unlike a lot of languages, is **not optional**. Python has no other way of knowing which lines of code are associated with which control statement without proper indentation.

The code above is simple enough, but can actually be combined by using a logical operator to combine both `if`-statements:

```
1 x = 6
2 if x<5 and x>2:
3     print('x is less than 5 and greater than 2')
```

```
4       print('Finished evaluating  if block')
5   print('Finished this code block')
```

Thus, logical operators can be used to write more concise conditional statements, and they can even be nested together into increasingly complex conditions.

It is however common for conditional evaluations to be binary in nature - if something is true, do this thing, or else do this other thing. In principle, this could be achieved by using two `if`-statements one after the other:

```
1   x = 6
2   if x<5:
3       print('x is less than 5')
4   if x>5:
5       print('x is greater than 5')
6   print('Finished this code block')
```

However, this is an unnecessary evaluation: if `x<5` is `False`, it automatically follows that `x>5` is `True` (except for one case, which we'll see in a bit), so there's no need to check again. Thus, Python has an `if-else` statement that implements just that idea:

```
1   x = 6
2   if x<5:
3       print('x is less than 5')
4   else:
5       print('x is greater than 5')
6   print('Finished this code block')
```

Note again how the line after `else` is indented after the colon, signifying an association between the two.

If you're alert, you'll nice that `x==5` is a third possibility. We could add another `if` statement for it, or we could group the three conditions together into an `if-elif-else` block:

```
1   x = 6
2   if x<5:
3       print('x is less than 5')
4   elif x>5:
```

```
5       print('x is greater than 5')
6   else:
7       print('x is equal to 5')
8   print('Finished this code block')
```

Here, "elif" is short for "else if". It is not necessary to use `elif` statements, but it is recommended because it makes the code more readable. Once again, notice the indentation and which lines are associated with which.

Note that the last else statement could've been replaced with an `elif x==5:` statement. However, it is usually good practice to end the `if-elif-else` block with an `else` statement as a fail-safe case to catch all other possible outcomes, both expected and unexpected.

3.2 Loops

Loops are blocks of code that are executed repeatedly either for a fixed number of times, or till a break condition is satisfied. Loop statements offer a convenient way to execute code repeatedly without having to write very long, unwieldy programs. Python offers two types of loops: `while` and `for` loops. Let's look at these individually.

`While` loops execute a block of statements (grouped by indentation) as long as a condition is true. Consider the following code, which prints a number and increments it until it reaches a threshold:

```
1   x = 3
2   while(x<6):
3       print('The value of x is ',x)
4       x+=1
```

Here, `x` is set to 3. When the `while` loop begins, the condition `x<6` is evaluated and if it is true, the associated lines of code (which are indented by one level more than the `while` statement) are executed. In this case, the condition is true, so first the value of `x` is printed, and then it is incremented by 1. At this point, control returns (or 'loops back') to the `while` statement, and the condition is evaluated again. This is why it's called a loop. The loop continues to iterate until the condition is `False`, at which point any code after the loop and its associated lines are executed till the program ends.

`While` loops are used when the number of times the code has to run is unknown. However, a more common case is that the lines of code are

executed for a fixed number of times. The `While` loop could be 'hacked' by coming up a with a condition that runs for a fixed number of times (like the example above), but there's a better way: `for` loops. These types of loops are extremely common, and most programming languages offer them. Compared to many other popular languages though, Python's `for` loop works a little differently: it is an iterator-based loop, similar to the `foreach` loop provided in C# and VBA, or a version of the `for` loop in Java. It is not the same as the `for` loop used in C/C++.

An iterator is a specific type of Python object that allows the program to traverse through all the elements in a sequence (tuples, lists, or dictionaries), regardless of how that sequence is implemented. And since sequences have a fixed length at any point of time in a code, iterating through them is equivalent to running a loop for a fixed number of times. Consider the following block of code:

```
1 x = range(3)
2 print(list(x))
3 for i in x:
4     print('The value of i is ',i)
```

Here, `range()` is a function that returns a fixed number of *integers* (*not floats*), *starting from 0 by default*, although the default behavior can be modified (see documentation). The return type is an iterable `range` object, which can be cast into a list type to see its contents - in this case, it's a list `[0,1,2]`.

Next, the `for` loop defines a variable `i` which is in `x`. This means that the variable `i` traverses through each the value in `x` in each iteration, until it runs out of values to traverse. Execute the function to print `i` and see this for yourself.

The beauty of this approach in Python and perhaps the most startling difference between it and the `for` loop in C/C++ is that the iterator can have any implementation - it can contain anything, not just numbers! Consider this example:

```
1 for i in ['Harry',(0,1,2),1]:
2     print(i)
```

Here, `i` iterates over a list, which itself is composed of three items of three different data types: a string, a tuple (yes, a list can hold a tuple, or

even another list), and an integer. And it does this naturally, without any additional code or typecasting necessary. This is a really big deal!

And it gets better - a single `for` loop can have more than one loop variable. Consider the following example:

```
1 x = [0,1,2]
2 y = [10,11,12]
3 z = [21,22,23]
4 print(list(zip(x,y,z)))
5 for i,j,k in zip(x,y,z):
6     print(i,'+',j,'+',k,'=',i+j)
```

Here, the `zip()` function (see documentation) takes three lists (or technically, iterable objects) *of the same length* (or it truncates to the shortest length) and combines them into an iterable object of tuples, with each tuple having three members. Then, the `for` loop **unpacks** the tuples into the constituent members, which can then be iterated over *simultaneously*. Python also provides several functions to efficiently create complex iterable objects through the itertools package (see documentation). We'll see how to import a package later.

Finally, because the `for` loop is based on an iterator, it can actually be used anywhere in a line of code, and not just in a formal loop. Consider the following piece of code, which creates a list of the squares of the first five whole numbers in just one line:

```
1 x = [i**2 for i in range(5)]
2 print(x)
```

This feature makes writing Python loops particularly elegant and simple.

3.3 Break, Continue, and Pass

Python provides three more useful features for loops:

- **Break**: When executed, the code breaks out of the loop without performing any more iterations
- **Continue**: When executed, the code skips everything after that line and returns to the top of the loop to perform the next iteration

- **Pass**: When executed, the just ignores that line, but continues to execute to the rest of the code for that iteration (good for use as a placeholder)

The following code succinctly demonstrates all three statements:

```
1 for x in range(10):
2     if x==3:
3         continue
4     if x==5:
5         pass
6     if x==8:
7         break
8     print(x)
```

Here, integers from 0 to 9 are iterated over the variable `x` and in general, the value of the integer is printed. However, when `x==3`, the loop reaches a `continue` statement, which means that the rest of the code is not executed and so 3 is not printed, while the loop continues with the next value. When `x==5`, a `pass` statement is executed, so the rest of the code is executed and 5 is printed and the loop continues. Finally, when `x==8`, a `break` statement is encountered, which forces to loop to end without being executed for the last value, 9. Try it yourself to see.

4 Functions

In just a few pages, we've already learned all the basic building blocks of any Python program. Now, we can move to higher abstractions. These abstractions are useful in making code more efficient, readable, and modifiable, and while they are strictly speaking not necessary write a program, they are used universally to write a *good* program.

The first abstraction is functions. Functions are bits of code that are separated from the main function, but can be called by the main function to perform a specific task. Functions have their own scope, which means that they cannot access variables in the main program, unless those variables explicitly passed to the function by the program. Here's a simple function that simply prints whatever input is passed to it:

```
1  def print_input(x):
2      print(x)
3
4  k = 'Hello'
5  print_input(k)
6
7  j = 'How are you'
8  print_input(j)
9
10 x = 'I am good'
11 print_input(x)
```

Let's break this down. A function is defined with a `def` statement, followed by the name of the function (which generally follows the same rules as variable names), followed by any inputs within parentheses, and finally a colon. The colon, like with loops, implies that items indented below it are part of the function. In this case, the print statement is the only piece of code in the function.

The main program comes after the function definition, without an indentation. This is not necessary - functions can be defined anywhere in the code, but should be defined before their first call. However, it is good practice to define all functions at the beginning of the program. In the main program, a string variable is defined and passed to the function (variables are passed by object reference, which means the value of its memory address is passed, but changing it involves the same principles as mutability).

4.1 Passing Variables

Note that the name of the variable passed to the function (`k, j, m` here), and the name the function uses for it internally (`x` here) could be the same or different. It doesn't matter: the function has its own scope, which means within the function, `x` is the name of the variable, irrespective of whether another variable outside the code has the same name or not. Think of `x` as a local alias for whatever variable is passed to the function. Thus, when the function is called, it receives a copy of variables passed to it, executes its code, and then returns control to the main program. The function itself only has to be defined once and can be called repeatedly. Furthermore, if the function is changes, it only has to be changed once, instead of having to change every instance of the same code.

However, the data type of the variable passed to the function may be important. Consider the following piece of code:

```
1 def add5(x):
2     y = []
3     for i in x:
4         y.append(i+5)
5     return y
6
7 a = ['one','two']
8 b = add5(a)
9 print(b)
```

Here, a function `add5(x)` is defined, which takes in a collection called `x`. Within the function, an empty list `y` is initialized. Then, every value in `x` is iterated over, with a value of 5 added to it, and the new value appended to `y`. Once all values are exhausted, the function returns `y` to the main program with a `return` statement. In the main program, a list of stings `a` is created and passed to the `add5()` function, and the value returned from it is stored in the variable `b`. This won't work however: a string cannot be added to an integer, and so the function will return an error. Thus, in this case, it is important to check what type of data is passed to the function. Change the main program to define `a = range(3)` and see what it returns.

4.2 Multiple Inputs and Outputs

A function can take in any number of variables and also return any number. Consider a function `solve_eqs()` to solve the system of linear equations $ax + by = e$ and $cx + dy = f$, whose solution is $x = (de - bf)/(ad - bc)$ and $y = (af - ce)/(ad - bc)$. The function takes in the constants `a,b,c,d,e,f` and returns the value of `x,y`:

```
1 def solve_eqs(a,b,c,d,e,f):
2     if (a*d-b*c)==0:
3         return 'Error'
4     x = (d*e-b*f)/(a*d-b*c)
5     y = (a*f-c*e)/(a*d-b*c)
6     return (x,y)
7
8 a = solve_eqs(3.2,3.2,7.8,7.8,1.3,2.4)
9 print(a)
```

The function first checks whether a unique solution exists, and if it doesn't, it returns 'Error' to both the expected variables `x,y` (there are better ways to handle this, but it's good enough to get the point). After the first `return` statement is executed, control returns to the main program, irrespective of whether any more lines could have been executed in the function. If a unique solution does exist, the `if` statement's condition is false and the first return statement is not executed. Then, the values of `x,y` are evaluated and returned as a tuple. In the main program, the output from the function is stored and printed in `a`. Try changing the inputs to the function call and see the results.

4.3 Keyword and Default Arguments

There are two more things to keep in mind: keyword arguments and default arguments. Consider the `solve_eqs()` function above: it has a lot of inputs, and the programmer has to input them in exactly the right order. This can get confusing. Fortunately, Python allows the function call to include the the name of the variable (also called an **argument**) being passed. In the code above, change line 8 to the following and see this in action:

```
a = solve_eqs(a=3.2,f=3.2,d=7.8,b=7.8,c=1.3,e=2.4)
```

This method of passing variables is called keyword arguments (where the name of the variable inside the function becomes its keyword in the function call). With this, arguments can be passed in any order without confusion.

Python can also assume some default values for arguments, which will be used if the user does not pass that particular argument. Consider this code:

```
def addnumber(x,n=5):
    y = []
    for i in x:
        y.append(i+n)
    return y

a = range(3)
print(addnumber(a))
print(addnumber(a,n=2))
```

This code defines a function `addnumber()`, which is similar to the `add5()` function defined previously, but takes an extra argument: the number to add to the first variable `n`, which is set to 5 in the function definition. Consider the first function call in the main program: only one variable is passed, so the function will use the default value of `n`. In the second call, both variables are passed, and the function uses the value of `n` passed in the call. Note that variables with default values are usually defined at the end of a function definition, and it is good practice to pass them using keywords to avoid confusion, as shown above.

5 Classes and Objects

The last abstraction that is useful but not necessary is classes, and objects created from those classes. Think of a class as a set of related variables and functions that are logically grouped together, and **an object as an instance of a class**. Let's take a concrete example: a person. What are the characteristics of a person? Their name, age, and height. Let's say the height is in centimeters, and we'd like a function to tell us what it is in feet and inches. These variables and functions can be grouped together into a logical class, and then we can create objects from those classes for every person we have data for. Let's put this into practice below.

5.1 Defining a Class

The `Person` class described above is constructed in the following code, after which an object is created from it:

```
1  class Person:
2      def __init__(self,name,age,height_cm):
3          self.name = name
4          self.age = age
5          self.height_cm = height_cm #in centimeters
6
7      def print_name(self):
8          print(self.name)
9
10     def print_age(self):
11         print(self.age)
12
13     def print_height_cm(self):
14         print(self.height_cm)
15
16     def height_ftin(self):
17         inches = self.height_cm/2.54
18         feet = inches//12
19         inches = inches%12
20         return feet,inches
21
22 adam = Person('Adam Levine',38,190)
```

```
23 adam.print_name()
24 adam.print_age()
25 adam.print_height_cm()
26
27 feet,inches = adam.height_ftin()
28 print('Height is ',feet,' ft and ',
29       "{:.1f}".format(inches),' in')
```

Like a function, a class is defined by prefixing it with a keyword `class` followed by the name of the class and colon. All indented lines after the colon represent the member variables and functions of that class. Within the class, member variables like `gender` and member functions can be defined. A class definition usually contains a special function `__init__()`, which is called a **constructor**. A constructor, as the name suggests, can be used to create an object of a class by creating variables and setting their values (unlike other object-oriented languages, a Python class can have only one constructor). Note that any member function, including a constructor, is defined just like any other Python function, with appropriate indentation for lines associated with that function. However, unlike ordinary Python functions, member functions of a class **must** be passed a `self` argument at the very beginning, as this is necessary for the function to be associated with the object.

In the `Person` class above, the constructor has four arguments: `self, name, age, height_cm`. The last three are self-explanatory, while `self` is a reference to the class itself (or strictly speaking, the object, similar to a `this` pointer in C++). The `self` argument is usually provided to all functions in a class so that they can access the variables in the associated object using the dot (.) operator. Consider the constructor above: it takes the value of the `name` variable passed to it and stores it in a **local** variable, also called `name` (it could've been called something else too). It knows that this is a local variable because it is defined using `self.name`, which implies that it is a member of the object itself. Thus, the dot operator is used to access member variables and functions. A similar operation is performed for the other member variables, `age, height_cm`. Then, four more member functions are defined, which are also self-explanatory. Remember, all members functions must be passed a `self` argument in addition to any other optional arguments, *even if it never uses any local variables*. Similar to regular Python functions, member functions can also return values, like the `height_ftin()` function defined above.

Now that we've defined the class, we can create an object, which is an instance of the class. This means that the object is a specific implementation of a class. In the code above, this is defined similar to how a new variable is defined. An object called `adam` is created and the `Person` class is assigned to it. In this assignment, the constructor of the `Person` class is called, which needs the `name, age, height_cm` variables, which are thus provided. Below that, members functions are used to print the objects's name, age, and height in cm, and to obtain the height in feet and inches. These member functions are accessed using the dot operator on the object, similar to how we used the dot operator on `self` to access members in the definition of the class.

5.2 Inheritance

Classes can seem unwieldy at first, but they are actually very useful in writing well-organized and efficient code. One of the most useful features of classes is called inheritance: the ability of a class to inherit all the members of a parent or *base* class while adding more to itself. This allows for a more granular level of abstraction. Let's see this with a concrete example: consider a student. A student is also a person (and thus has all the features of a person), but they have more features unique to them: GPA and level of study. We can define a class `Student` that inherits all the members of the `Parent` class, but also has these extra members:

```
1  class Student(Person):
2      def __init__(self,name,age,height_cm,GPA,level):
3          Person.__init__(self,name,age,height_cm)
4          self.GPA = GPA
5          self.level = level
6
7      def print_name(self,status='regular'):
8          print(self.name,' STUDENT: ',status)
9
10     def print_GPA(self):
11         print(self.GPA)
12
13     def print_level(self):
14         print(self.level)
15
```

```
16 nicole = Student('Nicole Kidman',19,170,3.6,'freshman')
17
18 nicole.print_name('in absentia')
19 nicole.print_age()
20 nicole.print_height_cm()
21 nicole.print_GPA()
22 nicole.print_level()
23
24 feet,inches = nicole.height_ftin()
25 print('Height is ',feet,' ft and ',
26       "{:.1f}".format(inches),' in')
```

Once again, a class `Student` is defined with the `class` keyword, but this time, the class is also passed the `Person` class so that it inherits from it. The inherited class has all the members of the base class, and can change them if necessary. The constructor is defined as usual, with the necessary variables passed to it and stored in local member variables. However, within the constructor, variables that were inherited from the base class can be initialized using the constructor of the base class, which can be accessed with the dot operator. This avoids having to rewrite lines of code that were already written for the base class' constructor, with code being written only for the variables unique to the child class.

After the constructor, the `Student` class has access to all the member functions of its base class, and thus, they do not have to be rewritten. Three new member functions are created: `print_GPA()` and `print_level()`, which are self-explanatory, and a new `print_name()` that overwrites the function of the same name in the base class by appending the name of the person with the word STUDENT, and accepting a new argument `status` with a specified default value.

Thus, the child class not only has access to the members functions of the base class, but can *also* change them by changing the implementation of the function and also the number of arguments. This ability of functions with the same name to do different things in different situations is called **polymorphism**. Indeed, it would've also been possible to define the `print_name()` function in the base `Person` class *without any implementation at all*, and then allow all child classes to have their own specific implementations of it. Such a class, which only has the name of a function but no implementation, is called an **Abstract Base Class (ABC)**. Polymorphism and ABCs form the basis of modern object-oriented programming.

The definition of the child class is followed by defining an object in the main program, and then all its member functions are called. Try to do this yourself to see the result.

6 Challenge: Basic Python

That's all you needed to know about Python! Well, that's all the theory anyway. Like a human language, programming languages cannot be learned by listening to someone in a classroom or reading a book. They can only be learned through practice, and Python is no different. You can find a good list of practice exercises here. In this section though, we'll solve one problem using all the tools we've learned so far (except classes since they're a bit more advanced, but you should practice that yourself).

6.1 Problem Statement

Using Python, define a function `prime_check()` that takes in an integer, and prints "This is a prime number" if it is a prime number, and "This is not a prime number" otherwise. The algorithm is as follows:

1. Negative integers and zero are invalid inputs
2. If the number is 1 or 2, it's a prime number
3. For other numbers, divide it sequentially by each number between 2 and *half* the number
4. If, during the above process, the remainder is ever zero, it is not a prime number, otherwise it is

Remember to use integer or floor division. Good luck! Remember, there are many possible solutions. You can see two of my solutions below (but try it yourself first).

6.2 A Simple Solution

This is a simple solution that I came up with:

```
1 def prime_check(x):
2     '''
3     Prints a message indicating if x is a prime number
4     or not
5     If x is invalid (negative integer or zero),
```

```
6       prints a message
7       x is of type int
8       '''
9       if x<=0:
10          print("Invalid")
11      elif x==1 or x==2:
12          print("This is a prime number")
13      else:
14          if is_prime(x):
15              print("This is a prime number")
16          else:
17              print("This is not a prime number")
18
19  def is_prime(x):
20      '''
21      Checks if a valid input x is a prime number
22      Returns True is x is prime
23      Returns False is x is not prime
24      x is of type int
25      '''
26      prime = True
27
28      for y in range(2,(x//2)+1):
29          if x%y == 0:
30              prime = False
31              break
32
33      return prime
34
35  prime_check(1373)
```

I defined two functions. My first function is the codeprime_check() function that the problem requires. It first checks for an invalid number or if the number is 1 or 2. Then, it calls another function `is_prime()`, which returns a boolean value `True` if the number is prime and `False` otherwise. Accordingly, it prints the appropriate message. That's it - the actual work of checking if a valid number is prime or not is in the `is_prime()`. Breaking up a code like this called refactoring, and it is good practice as it makes the code readable and easy to modify. Also note the explanatory statements at the beginning of the function, contained between three ' characters. This

is called a docstring, and provides information about the function and is another piece of good programming practice.

Now consider the codeis_prime() function itself. It starts out by defining a boolean variable `prime` to be `True`. Following the algorithm, it creates a range of numbers from 2 to half the input number (note that it uses floor division `//2` to ensure that we get an integer for odd numbers). The '`+1`' in the second argument of the `range()` function is because the range is exclusive of the second argument (see documentation). Then, the function iterates over each value in that range, checking to see if the remainder with the input number is ever zero. If it is, then `prime` is set to `False` and the loop is broken, because there is no need to perform any more checks (the code will work without that, but will take longer). If the number really is prime, the `if` condition is never true and `prime` stays `True`. Finally, `prime` is returned.

I tried this code with the number 1373, which is big enough that I can't say immediately whether it is prime or not. Is it?

6.3 A Complicated Solution

Here is a more complicated - and much shorter - solution that I came up with, which truly illustrates how powerful yet simple Python is:

```
1  def prime_check(x):
2      '''
3      Prints a message indicating if x is a prime number
4       or not
5      If x is invalid (negative integer or zero),
6      prints a message
7      x is of type int
8      '''
9      if x<=0:
10         print("Invalid")
11     elif x==1 or x==2:
12         print("This is a prime number")
13     else:
14         if is_prime(x):
15             print("This is a prime number")
16         else:
```

```
17              print("This is not a prime number")
18
19  def is_prime(x):
20      '''
21      Checks if a valid input x is a prime number
22      Evaluates in one line
23      Returns True is x is prime
24      Returns False is x is not prime
25      x is of type int
26      '''
27      return not bool(len([i for i
28          in range(2,(x//2)+1) if x%i==0]))
29
30  prime_check(1373)
```

This code seems... abrupt. So let's break it down. This has the same overall structure as the last one: the `prime_check()` function only prints the message, while the much-shorter `is_prime()` function implements the algorithm to check if the number is prime or not. It may not seem obvious at first, but it really is the same algorithm written in just one line of code. How does it do that?

First, the function creates a list using an iterator over a variable `i`, which iterates numbers between 2 and half the input number, just as in the `for` loop before. But then, at the end of that loop, an additional condition is added using an in-line `if` statement: the loop returns a value to `i` only if its remainder with the input number is zero. We know from the algorithm that if the remainder is ever zero, then it's not a prime number. So now, we've built a list of every instance when that remainder was zero for the given number, and we know that the length of this list must be zero for a prime number, and non-zero (actually 1 if you think about it) otherwise. So we find the length of this list by plugging it into the `len()` function.

Finally, we use Python's `bool()` function (see documentation) to convert that length into a `True` or `False` value. The `bool()` function returns `False` if the input is 0 and `True` otherwise (see truth testing in Python). But if the length is 0, we need this to return `True` as that means it's a prime number and `False` if it's not - so we need to reverse the result of the `bool()` function. We do this by sticking a `not` operator in front of it. And just like that, we have a statement that tells us if the number is prime or not - and we just return whatever value it gives.

Python beautifully let us create a list with a loop *and* an `if` statement in just one line! This made the code much shorter and (with practice) easier to read. That's the beauty of Python - it lets you get stuff done without having to write a lot of code.

7 Python Libraries

7.1 Why Libraries?

Python is great and can do a lot of things. But it would probably have remained an also-ran language, if not for two things: it's open-source code, which allows anybody in the world to shape and improve the language; and the wide variety of libraries that those very users have developed to extend the use of Python to just about every major application. Some Python libraries - like numpy and matplotlib - are so ubiquitous that many developers consider them to be an indispensable part of Python itself.

However, this decentralized approach to extending the capabilities of the language does come with some downsides. For one, there are so many libraries that it's impossible to learn all of then, and incredibly hard to figure out which one is best-suited for your application (except some libraries - like numpy - that have near-universal approval). Also, not all libraries are good: some of them are out of development and do not work properly, while others are slow in development. In addition, many libraries have scant documentation, which make them very difficult to use. And some have such voluminous documentation as to scare away new users! The bottom line is that while Python libraries are an indispensible tool and a victory of open-source programming, they can feel like absolute anarchy to use.

7.2 Some Libraries

There are so many Python libraries that it's impossible to teach all of them. It's also unnecessary - just a small number of libraries can take care of most of your needs. In the table below, I present a list of some popular libraries and their application areas. By no means is this list exhaustive, but all the libraries listed are, in general, very good. Note that these applications are the most popular ones - many of these libraries have applications that span several areas. Several of these libraries - especially numpy and matplotlib - are actually used by other libraries too.

Libraries	**Popular Applications**
numpy, scipy	Numerical and scientific analysis (comparable to MATLAB)
matplotlib, seaborn, vtk	Scientific visualization
pandas	Data science
scikit-learn, tensorflow, pytorch	Machine learning and deep learning
scrapy, requests, selenium, PyPDF2	Data mining
geopandas, rasterio, rasterstats, descartes, pySAL, arcpy	Geospatial data analysis and GIS
wxPython, tkinter, pyQt, pyGUI	Graphical User Interface (GUI) development
SQLAlchemy, postgreSQL	Database management
django, TurboGears, flask, web2py	Web development
OpenCV, scikit-image	Image and signal processing
MetPy, netCDF4, SatPy	Meteorology
mpi4py, cython, jython	High performance computing
biopython	Biology
astropy	Astronomy
scikit-chem	Chemistry
SymPy	Symbolic math (comparable to Mathematica)
FiPy, OpenSeesPy	Finite element analysis
fluidity, fluidsim, fluiddyn, cfdpython	Computational fluid dynamics
Py2B, pySerial, instrumentino	Arduoino programming and IoT
pyfin, QuantPy, pynance, analyzer, pyfolio	Finance and trading
nltk, spacy, polyglot	Natural language processing
pygame, arcade, cocos2d, PyODE	Game development and animation

In addition to these libraries, Python itself has several modules (which are like libraries, but distributed with Python itself) that helps extend its ap-

plications. See documentation for an exhaustive list. Libraries and modules are used very similarly, so I may use them interchangeably.

7.3 Using Libraries

Almost all libraries follow the same syntax as Python itself, so using them is as simple as knowing what functions they have and how to call them. However, libraries are not included in Python by default (for good reason - there are too many of them!). To use a library, you have to follow two steps.

First, make sure the library is installed on your computer. If you're using Anaconda, many popular libraries come pre-installed. However, on a terminal, you may have to install any libraries that you need, together with any dependencies. See the library's documentation to install it.

Second, once the library is installed, it has to be imported into the Python script. This can be done easily using the `import` command followed by the name of the library. Libraries can be imported at any point in the script (but before they have to be used, of course), but are usually imported at the very beginning, for clarity. In addition, since libraries tend to have long names (which, as we'll see, need to be typed every time), they are usually aliased using an `as` statement right after being imported. For example, this is how the numpy library is usually imported:

```
1 import numpy as np
```

The alias could be anything, but since numpy is so commonly-used, it has become an unofficial standard of sorts to use the np alias for it. Now, whenever we use a numpy function in this script, we have to prefix it with np followed by a dot (.) operator and then the name of the function, for example a sine function in numpy can be called using:

```
1 np.sin(0)
```

Now, numpy has a lot of useful functions, but there are many libraries from which we might be interested in only a subset of functions, or only a single function. We can selectively import functions using a `from` statement, which is good practice to speed up execution. For example, matplotlib contains a sub-library of functions under pyplot for MATLAB-like plotting in Python, which is usually imported as follows:

```
1 from matplotlib import pyplot as plt
```

Or as follows:

```
1 import matplotlib.pyplot as plt
```

Here again, the alias can be anything, but since pyplot is so commonly used, plt has become somewhat of a standard for it. In the following chapters, we'll go through four of the most popular Python libraries - numpy, pandas, scipy, and matplotlib - with some examples to see how to use them for a variety of applications.

8 Numpy

Numpy is a Python library for fast matrix manipulation. By fast, I mean that the library is optimized for vectorizing data and implements the popular and efficient linear algebra package, BLAS. In fact, numpy is actually a Python wrapper - if you look inside its functions, you'll see a lot of pre-compiled C and FORTRAN code. This is what makes it much faster than regular Python, but its syntax is a lot easier than those compiled languages.

In the following example, we'll see how to use numpy for some popular applications. They key point to remember is that numpy is for matrix manipulation - so don't think about manipulating individual elements of a matrix (say in a loop), but about manipulating all elements simultaneously.

8.1 Numpy Arrays

The backbone of numpy is numpy arrays, a data type that numpy uses to define matrices containing data of the same type. Numpy arrays look similar to Python lists (or a list of lists for multidimensional arrays), but they store data in contiguous memory blocks for fast manipulation. A numpy array is created using the `array()` member function of numpy - since it's a member function, we need to dot operator to call it:

```
1 import numpy as np
2 a = np.array([[1,2,3],[4,5,6],[7,8,9]])
3 print(a)
4 print(type(a))
5 print(a.shape)
```

Let's unpack this (pun intended!). The first line imports numpy and gives it the alias np - you only have to do this once in a script. The next line defines a numpy array `a` by calling the `array()` member function and passing a list of lists. Each list inside this list defines a row on the matrix. The shape property of this array returns a tuple of the number of rows and the number of columns in the array. Try this code to see how it works.

Numpy also has some useful built-in functions to quickly define an array with some specific properties:

```
1 b = np.zeros((4,4),dtype=np.float)
```

```
2 print(b)
3 c = np.ones((4,3),dtype=np.int)
4 print(c)
5 d = np.eye(3,dtype=np.float)
6 print(d)
7 e = np.reshape(np.arange(10),(2,5))
8 print(e)
```

The `zeros()` method defines an array of zeros, with the `dtype` attribute used to specify whether those zeros are integers, floats, or complex numbers. Similarly, `ones` defines an array of ones, and `eye` defines an identity matrix. Numpy can also take a matrix and reshape it into a different matrix. The `arange()` member function is similar to the `range()` function in Python, except that it returns a numpy array. The `reshape()` function then takes this array and reshapes it based on the dimensions provided in the tuple in the second argument - of course, the user has to ensure that the total number of elements remains the same.

8.2 Array Manipulation

The true power of numpy is its ability to manipulate arrays fast. Typically, we want to manipulate all the members of an array at one time, and not individual members. Nonetheless, it is possible to extract a smaller part of an array using the slicing operator `[:]`. In the following example, an array consisting of ten members in one row and ten columns (arrays with either one row or one column are called vectors) is sliced to extract the third through sixth members as another array:

```
1 x = np.arange(10)
2 print(x)
3 x1 = x[2:6]
4 print(x1)
```

Note that numpy arrays, like Python lists, are indexed from zero, so the index of the third item is 2 and that of the sixth item is 5. Indexing is also exclusive of the second number, so to get the sixth item, we need to slice to index 6. Leaving the second number blank represents the last index, and leaving the first number blank represents the first index. Try it yourself to

see. For multidimensional arrays, slicing in each dimension is separate by a comma, for example `[1:3,3:]` for a 2D array.

Numpy arrays can manipulate their elements simultaneously by bringing all (or large chunks of) array values into memory at once, as opposed to Python's approach of handling them one by one. This approach to speeding up computation is called **vectorization**, and together with algorithm complexity, forms the foundation of modern scientific computing. Consider this short piece of code:

```
1 a = np.array([12,13,14])
2 print(a+3)
3 print(a.mean(),a.sum())
```

Here, a numpy array was initialized and a scalar was added to it. From a purely mathematical perspective, this makes no sense, and indeed with a Python list, this operation would throw an error. But numpy knows what you mean: you want to add a scalar to all the elements of the vector simultaneously, and it does just that. This is called **broadcasting**, works for all other operators too. In fact, broadcasting works between arrays too, not just between an array and a scalar. In addition, every numpy array includes a number of member functions that can quickly manipulate all the elements as well, such as the `mean()` and `sum()` functions shown above.

But numpy goes one step further: it defines hundreds of *vectorized functions* too, which can apply the function to all elements in the array fast, and return another array of the result. For example, consider this short program:

```
1 a = np.arange(5)**3
2 print(np.sqrt(a))
```

An array of five numbers is defined, and each member is cubed as before. In the second line, the `sqrt()` member function of numpy is used to calculate the square root of each member and return an array of the results. Thus, there is no need to manipulate each element - all elements can be manipulated simultaneously.

Another useful feature of numpy arrays is masking, which allows Boolean operations to be vectorized. A masked array is an array of Boolean values (`True` or `False`). When this is passed to another array of the same dimensions, it returns a 1D array of those values corresponding to `True`. As an

example, consider the following code:

```
1 a = np.random.rand(10).reshape((2,5))
2 b = np.logical_and((a>0.1),(a<0.5))
3 print(b)
4 print(a[b])
```

An array `a` is defined with 10 random numbers between 0 and 1, using the `random.rand()` numpy function, and reshaped into a 2×5 matrix. We want to extract those values of `a` that are between 0.1 and 0.5. We use a comparison operator `a>0.1` for values greater than 0.1, which returns a Boolean matrix applying the operator to each element of `a`. Similarly, a Boolean matrix for values less than 0.5 is created. Since we want values that are true for both cases, we perform a logical element-wise `and` operation on these Boolean matrices with the `logical_and()` numpy function, and save it to a matrix `b`. This matrix is our masked array - it's an array of Boolean values of the same shape as `a`. We pass this masked array to `a` and get an array of values that correspond to `True`. Thus, without ever writing a loop, we used a vectorized (and hence, fast) approach to solve the problem.

8.3 Challenge: Linear Algebra

Possibly the most important operation in scientific computing is solving a system of linear equations. Almost all numerical methods as well as machine learning algorithms are constrained by how fast they are able to solve the simple equation $Ax = b$, where the unknown vector x can have millions of elements. In this challenge section, we'll see how to use numpy's linear algebra package linalg to analyze and solve a system of linear equations.

Here are the equations that we will solve:

$$\begin{aligned} x + y + z &= 6 \\ 2y + 5z &= -4 \\ 2x + 5y - z &= 27 \end{aligned}$$

Or in matrix form:

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 2 & 5 \\ 2 & 5 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 6 \\ -4 \\ 27 \end{pmatrix}$$

Which we can write vectorially as $Ax = b$. The challenge is to first check whether a unique solution exists (by checking if the determinant is zero or not) and if so, find it, using numpy functions. Try it yourself before seeing my solution below.

```
1 A = np.array([[1,1,1],[0,2,5],[2,5,-1]])
2 b = np.array([6,-4,27])
3 if np.linalg.det(A) == 0.0:
4     print("No unique solution exists")
5 else:
6     x = np.linalg.solve(A,b)
7     print("Solution is: ",x)
```

The first step is to define the matrices `A,b` using numpy's `array()` function as before. Then, we check if the determinant of `A` is zero using an `if` statement, in which case we print a message that there is no unique solution. The determinant is calculated using numpy's `linalg.det()` function. If the determinant is non-zero, we solve for `x` using numpy's `linalg.solve()`, function, which uses a fast algorithm to solve the system (specifically, it uses LAPACK's gesv algorithm, which itself is an efficient implementation of LU decomposition). Finally, we print the solution. Try it yourself.

This example was simple but useful to illustrate how numpy can be used to write simple scientific programs without having to worry too much about specific implementation while ensuring that the program runs fast. Numpy forms the basis of almost all the major scientific programming libraries in Python, so we'll come across it again in the following chapters.

9 Matplotlib

They say that a picture is worth a thousand words, and this rings as true for scientific computing as it does for anything else. There are several packages - either standalone or wrapped together with some other application - to quickly create publication-quality visualizations of their scientific data. Python also has several libraries, of which matplotlib is quite popular. Specifically, matplotlib's pyplot library contains Python functions that can interact with matplotlib to generate scientific plots.

9.1 Plotting 1D Data

The most basic type of data you can get is 1D data. The most basic way of plotting such data is using a bar chart. Consider the example below:

```
1  %matplotlib inline
2  import numpy as np
3  from matplotlib import pyplot as plt
4
5  barnumber = np.arange(4)+1
6  barheight= np.random.uniform(5.0,10.0,(4,))
7
8  plt.bar(barnumber,barheight)
9  plt.xlabel('Bar Number')
10 plt.ylabel('Bar Height')
11 plt.title('Random Bar Heights')
12 plt.ylim([0,10])
13 plt.xticks(barnumber)
14 plt.show()
```

Note that the `%matplotlib inline` command in the beginning is to that figures are shown in-line in Jupyter Notebooks. It is not necessary if you're using a script. The next two lines import numpy and pyplot and assign the somewhat standard aliases to them - this only needs to be done once for each script. Numpy arrays are used to pass data to pyplot. Then, some data is generated - four labels are generated using numpy's `arange()` function and stored in the `barnumber` variable. Four random numbers between 5 and 10 are generated using numpy's `random.uniform()` function and stored in `barheight`. We're now ready the plot the height of each bar.

Barplots are generated using pyplot's `bar()` function (see documentation). This function takes at least two arguments: the coordinates of the bars and the height of each bar. A number of optional parameters to define things like color, width, error bars, etc. are also available. However, the function on its own just generates a plot without any labels - not exactly publication quality yet. We can add labels to the axes using the `xlabel()`, `ylabel()` member functions of pyplot, and a title with the `title()` function. Since these are member functions, they are accessed through the dot operator. **Pro-tip:** the axis labels and title functions accept LaTeXinputs.

There are two more problems with this graph. On the x-axis, we would like to show only the bar number as a categorical value, instead of as a coordinate, since this is 1D data. We can do this by setting `xticks()` to the bar number. The y-axis scales automatically to the range of the heights, which is between 5 and 10. However, it's usually good practice to start the y-axis from zero instead. We can adjust this by setting the `ylim()` function to the appropriate range. Finally, we use the `show()` function to show the graph. And viola! In a few simple lines, we have a professional graph. If you've used MATLAB before, all the commands will seem very familiar, with just the extra `plt.` prefix. Play around with other options from the documentation to make your graph even better.

9.2 Plotting 2D Data

Arguably the most common type of plot involves plotting the value of a dependent variable as a function of an independent one. This could be experimental data or data obtained from an analytic function. For example, let's say we want to plot the function $f(x) = (\sin(x) + \cos(x))/2$ in the range $[0^o, 360^o]$. Here's the code to do that:

```
1 x = np.linspace(0,360,1000)
2 y = (np.sin(x*np.pi/180.) + np.cos(x*np.pi/180.))/2.0
3 plt.plot(x,y)
4 plt.grid()
5 plt.xlabel('x (degrees)')
6 plt.ylabel('f(x)')
7 plt.show()
```

First, we use numpy's `linspace()` function to get 1000 uniformly-spaced

points in the required range of x. Then, we use numpy's `sin(), cos()` functions to define the function and store it in `y`. Note that numpy's trigonometric functions expect an input in radians, so we multiplies `x` with $\pi/180$ for conversion (using numpy's `pi` variable). Finally, we plot the data using pyplot's `plot()` function, which takes the independent variable as the first argument and the dependent variable as the second, with several optional variables available (see documentation). Once again, I added axis labels for clarity. I also used the `grid()` function to show a grid, which makes it easier to read the graph.

Multiple data can also be plotted on the same graph, with a legend to differentiate between them. Suppose we want to plot $f(x) = e^{-x}$ and $f(x) = e^{-2x}$ on the same semi-logarithmic graph for $x \in [0, 2]$. Here's the code to do that:

```
1  x = np.linspace(0.,2.,1000)
2  y1 = np.exp(-x)
3  y2 = np.exp(-2.0*x)
4
5  plt.semilogy(x,y1,label='$e^{-x}$')
6  plt.semilogy(x,y2,label='$e^{-2x}$')
7  plt.xlabel('x')
8  plt.ylabel('f(x)')
9  plt.legend()
10 plt.grid(which='both')
11 plt.show()
```

The first three lines define the independent and dependent variables as usual. Then, a semi-logarithmic (y-axis logarithmic) graph is plotted with the `semilogy()` function for the first function, with a label provided. The expression in the label (which uses LaTeXhere!) will be used in the legend. On the next line, another `semilogy()` function is used for the second function. Pyplot knows that these are meant to be used in the same graph, and will take care of it. Labels and grids are provided as usual, and the `legend()` function is used to display the legend. Run the code above and have a look at the results.

Finally, let's look at a case where we have different types data stored in a file. In your data folder, I have created a Comma-Separated Values (CSV) file weather.csv, which contains the hourly forecast temperature (in ^{o}F) and

probability of precipitation (in %) for some location. Open the file and have a look. The first row contains headers, which we don't need for plotting. Because the two data series have different units, we'd like to plot them on separate subplots of the same plot. Here's the code that will do that:

```
1  hour,temperature,precip_prob =
2          np.loadtxt('../data/weather.csv', dtype=np.float,
3          delimiter=',',skiprows=1,unpack=True)
4
5  plt.subplot(1,2,1)
6  plt.plot(hour,temperature)
7  plt.xlabel('Hour')
8  plt.ylabel('Temperature ($^o$F)')
9  plt.title('Temperature')
10
11 plt.subplot(1,2,2)
12 plt.bar(hour,precip_prob)
13 plt.xlabel('Hour')
14 plt.ylabel('Probability (%)')
15 plt.title('Probability of Precipitation')
16 plt.show()
```

The data is loaded using numpy's `loadtxt()` function (see documentation). The first argument to this function is the path to the CSV file, while the second argument specifies the data type. The `delimiter` argument is used to specify a delimiter, since it could be any character (it's a comma in a CSV file). The next argument is used to skip a certain *number* of rows from the top - in this case, we want to skip the first row because it's a header, so we set this to 1. Finally, we set `unpack` to `True` because we want to extract hour, temperature and probability of precipitation as three separate numpy arrays, or in other words we want to unpack the data.

We want two subplots to plot each of the data series, so we use pyplot's `subplot()` function to create them. The first argument is the number of rows and the second argument is the number of columns - we want to plot it as one row with two columns. The third argument is used to plot a specific subplot, whose numbering starts from 1 from the top-left and continues from left to right along a row for each row. Thus, 1 represents the first subplot and 2 the second. Once we access the first subplot, we plot the temperature as usual. We then access the second subplot and plot the probability data, and

we finally use the `show()` function to show the entire plot. Try this yourself to see the results.

9.3 Plotting 3D Data

Finally, we look at plotting 3D data. The most common application of this is for plotting values of field functions. There are usually two ways of presenting 3D data:

- As an actual 3D graph, shown using an isometric drawing. This is great for showing data through an interactive graph on a browser, but may not be easy to interpret when printed on paper.
- A 2D graph with a third "axis" in the form of contours or colorbars (or both). This is the more traditional approach and is easy to interpret.

Let's first plot a graph of the first type. Let's say we want to plot $f(x, y) = \sin(\sqrt{x^2 + y^2})$ in the region $[-6, 6] \times [-6, 6]$. Plotting this is much like plotting a 2D function, in that we have a pyplot function which takes in the independent and dependent variables. However, there is one key difference: the independent variables are no longer defined on the 1D axes, but in a 2D mesh, and the dependent variable must then be defined on that mesh. Thus, as a first step, we need to define a 2D mesh from the underlying 1D points from each axis. The full code is shown below:

```
1  from mpl_toolkits import mplot3d
2
3  x = np.linspace(-6,6,100)
4  y = np.linspace(-6,6,100)
5
6  X,Y = np.meshgrid(x,y)
7  Z = np.sin(np.sqrt(X**2+Y**2))
8
9  fig = plt.figure()
10 ax = plt.axes(projection='3d')
11
12 ax.contour3D(X, Y, Z, 50, cmap='spring')
13 ax.set_xlabel('x')
14 ax.set_ylabel('y')
```

```
15 ax.set_zlabel('f(x,y)')
16 plt.show()
```

The first thing to notice is that we imported a new library, `mplot3d`. This is because Pyplot does not provide native support for 3D graphs, but MPlot3D provides the necessary extensions for it to. In the next two lines, 100 points along each axis are sampled. These are 1D points on each axis, which have to be meshed together. This is done using numpy's `meshgrid` command, which returns the 2D meshed coordinates `X, Y` of all the points in he domain (there will be 100×100 points). Since `X,Y` are 2D matrices, when the function is evaluated with them, it will also return a 2D matrix, with one value for each point on the mesh.

The last step is to plot the mesh. Because MPlot3D needs to work with Pyplot, this is a little more involved. First, we create a blank figure `fig`. Any plot can be put in this figure. We create a blank 3D plot `ax`, which will go into the figure. Finally, the mesh is plotted using the `contour3D()` function, which takes in the independent and dependent variables, the number of contours to plot (50), and a colormap to use (see here for a list of available colormaps). Since this graph has a z-axis to directly read off the values, the colormap is optional.

This is good, but a 3D plot can 'hide' a lot of data points 'behind' some features. A contour plot avoids that pitfall. For the same data as above, a contour plot can be created using the code below, which is very similar to the full 3D plot's code:

```
1 plt.contourf(X,Y,Z,50,cmap='spring')
2 plt.xlabel('x')
3 plt.ylabel('y')
4 plt.colorbar(label='f(x,y)')
5 plt.show()
```

The `colorbar()` function adds a colorbar to the plot, and the `label` argument to it adds a colorbar title so you can show what the colors represent. Unlike the full 3D plot, a colorbar is necessary here for the reader to interpret the graph.

Matplotlib has many more options for 1D, 2D, and 3D plots than described here. However, they all follow the same general procedure to create the plot. Try these examples, and the try it on your own data. When in

doubt, always go back to the documentation for whatever plot you're making.

10 Scipy

Numpy and matplotlib form the basis for most other scientific computing libraries in Python. One of the most popular libraries to build on them is Scipy, a Scientific Python library with implementations of specific functions that are often used in science and engineering. Specifically, Scipy has packages that can be used for the following applications:

Package	**Application**
scipy.cluster	Clustering (K-means) and vector quantization
scipy.constants	Mathematical constants
scipy.fftpack	Fast Fourier Transform
scipy.integrate	Integration
scipy.interpolate	Interpolation
scipy.io	Input/Output
scipy.linalg	Linear Algebra
scipy.ndimage	Image analysis
scipy.odr	Orthogonal regression
scipy.optimize	Optimization
scipy.signal	Signal processing
scipy.sparse	Sparse matrix manipulation
scipy.spatial	Spatial data and algorithms
scipy.special	Special functions
scipy.stats	Statistics

Scipy is written using numpy, and the two libraries share much in common (including contributors). In fact, some packages can be found in both libraries, such as the `linalg` package. However, in general, scipy tends to have a wider suite of functions than numpy. In the following pages, we'll see how to use scipy for three common applications: curve fitting, image analysis, and fourier transforms.

10.1 Challenge: Curve Fitting

Scipy's functions for optimization and fitting are found under `scipy.optimize`. Specifically, the curve fitting function is `curve_fit()`, which fits data to a function with non-linear least squares optimization (see documentation. In this challenge, we'll see how to use this function to fit rainfall data to an

empirical intensity-duration-frequency (IDF) curve equation. The specific curve we'll be looking to fit the data to is the empirical function develop by Chow et al:

$$I(t) = \frac{a}{t^n + c}$$

Where $I(t)$ is the intensity of precipitation in inches per hour, t is the duration of rainfall in minutes, and a, n, c are empirical coefficients that are a function of the frequency (return period) that have to be determined using curve fitting. Data for this challenge was obtained from NOAA for the Champaign hydrological station and can be found in the idf.csv file in the data folder.

Here is my solution:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def idf(t,a,n,c):
    '''
    Chow et al IDF relationship
    '''
    return a/((t**n)+c)

#Get data
data = np.loadtxt('../data/idf.csv',
        skiprows=1,dtype=np.float,delimiter=',')
period = data[0,1:] #return period in years
duration = data[1:,0] #duration in hr
intensity = data[1:,1:] #intensity in in/hr

#Fit curves
popt = [] #optimized parameters
pcov = [] #covariance matrices

for i in range(len(period)):
    p = curve_fit(idf,duration,intensity[:,i])
    popt.append(p[0])
    pcov.append(p[1])

```

```
27      plt.loglog(duration,
28          idf(duration,p[0][0],p[0][1],p[0][2]),
29          label=str(int(period[i]))+' yr')
30
31  plt.xlabel('Duration (hours)')
32  plt.ylabel('Intensity (in/hr)')
33  plt.title('IDF curves for Champaign, IL')
34  plt.grid(which='both',linestyle='--')
35  plt.legend()
36  plt.show()
```

That's quite a lot of code! Let's break it down. The first three lines import the usual libraries and functions. Then, we define a function `idf()` that represents the Chow curve, which returns the intensity for a given set of inputs. Now, the next part of the code is a little messy, but it's just getting the data from the idf.csv file using numpy's `loadtxt()` as single numpy array, and then slicing the array to get the return period (a vector), duration (another vector), and intensity (a 2D matrix). Look at the form of the data in the CSV file and the slicing should make sense. Note that each row of the intensity matrix represents the precipitation intensity for a given duration over all return periods (i.e., return period is in columns and duration is in rows).

Now we finally get to the curve fitting from line 18 onward. If you see the documentation of the `curve_fit()` function, you'll see that it returns two numpy arrays: an array of the optimized parameters and another one with a covariance matrix of those parameters. These vary for each return period, so we'll store them in a list (the list will hold numpy arrays). The lists are initialized, and then we enter a loop, where the curve fitting is performed for each return period, and a line is added to a log-log plot for the fitted IDF curve. The curve fitting function takes in three inputs: the `idf()` function, duration, and intensity. The function assumes that `duration` is the independent data, `intensity` is the dependent data, and the rest of the inputs to `idf()` are the parameters to be fitted. The function then returns the optimized parameters and covariance matrix, which are then appended to the appropriate list and also used to plot the fitted curve.

After the loop, we add some paraphernalia to the curve and plot it. Thus, we have a series of IDF curves from the data.

10.2 Challenge: Image Processing

Image processing is an important scientific skill, and several commercial and open-source programs are available for it. However, at their heart, all image processing programs implement matrix manipulation algorithms. After all, an image is nothing but a matrix, with each element representing the intensity of a pixel (in one or more channels). Scipy's ndimage package offers useful image analysis functions, which we'll use in this challenge.

The challenge is to take the otter.png grayscale image provided in the data folder (courtesy of Wikimedia Commons), and perform the following manipulations on it:

- Rotate it by 10^o anti-clockwise
- Sharpen it
- Detect its edges

Here is my solution:

```
from scipy.ndimage import gaussian_filter, rotate, sobel

img = plt.imread('../data/otter.png')
plt.subplot(2,2,1)
plt.imshow(img,cmap='gray')
plt.axis('off')

img_rotate = rotate(img,angle=10)
plt.subplot(2,2,2)
plt.imshow(img_rotate,cmap='gray')
plt.axis('off')

img_filter = gaussian_filter(img_rotate,sigma=3)
img_sharp = img_rotate + 30.*(img_rotate-img_filter)
plt.subplot(2,2,3)
plt.imshow(img_sharp,cmap='gray')
plt.axis('off')

sx = sobel(img_sharp, axis=0)
sy = sobel(img_sharp, axis=1)
```

```
21 img_sobel = np.hypot(sx,sy)
22 plt.subplot(2,2,4)
23 plt.imshow(img_sobel,cmap='gray')
24 plt.axis('off')
25
26 plt.show()
```

This program is straightforward: it applies each of the required manipulations and shows the final image. First, the necessary functions are imported. Then, the file is read using pyplot's `imread()` function. While ndimage also has a similar function, it has been deprecated, and pyplot's function is recommended. This returns a numpy array of the intensity values of each pixel (since the image is grayscale, there is only one channel). This is then plotted in a subplot using pyplot's `imshow()` function.

Now, the first manipulation is to rotate the image, which is done using ndimage's `rotate()` function, and the resulting image is plotted. Under the hood, the function simply multiplies the numpy array of the image by a rotation operator (another matrix) and returns the result.

The second manipulation is to sharpen the image. Sharpening involves enhancing sharp edges in an image and suppressing other parts of it. So first, we need to extract the sharp edges of the image, which can be done by applying a low-pass gaussian filter with the `gaussian_filter()` function, and then turning it into a high-pass filter by subtracting it from the rotated image. These sharp edges are then enhanced by a factor (30) and added to the rotated image, which gives a sharpened image. This is then plotted.

Finally, the last manipulation is edge detection, for which there are many algorithms available. I chose to use a Sobel filter using ndimage's `sobel()` function. The final image of edges is then plotted. Note that for each of the plots, pyplot by default creates axes, which doesn't make sense for an image. There are thus turned `off`.

And... that's it! It was that simple - just a series of matrix manipulations, with the convolution kernels conveniently created by ndimage's functions. On it's own, this is a powerful utility, but ndimage isn't even the best image processing library around. That title arguably goes to scikit-image, which you should explore further if you are looking to use Python for image analysis.

10.3 Challenge: Fourier Transform

Fourier transforms are commonly used in signal processing to convert data from a time domain to a frequency domain. This, in turn, has a wide variety of applications, from file compression to spectroscopy. Fourier transforms are actually very cumbersome to evaluate, and was a major computing challenge until the invention/discovery of the Fast Fourier Transform (FFT) algorithm, also called the Cooley-Tukey Algorithm. Strictly speaking, this is a discrete FFT algorithm, as it works on discrete data. Together with its inverse, it is probably one of the most important algorithms of the last century, and is implemented in every major scientific analysis program, including Scipy's codescipy.fftpack package.

In this challenge, we'll take signal data and determine the main frequencies in it by converting it from the time domain to the frequency domain. The data is provided in the cat.csv file, which was created from the sound of cats meowing in the cat.wav file. Here's my solution:

```
1  from scipy.fftpack import fft
2  t,sig = np.loadtxt('../data/cat.csv',
3          unpack=True,skiprows=1,
4          dtype=np.float,delimiter=',')
5
6  plt.subplot(2,1,1)
7  plt.plot(t,sig)
8  plt.xlabel('Time (s)')
9  plt.ylabel('Amplitude')
10
11 spec = fft(sig)
12 T = t[1]-t[0]
13 N = sig.size
14 freq = np.linspace(0,1.0/T,N)
15 plt.subplot(2,1,2)
16 plt.plot(freq[:N//2]/1000.,
17         np.abs(spec)[:N//2]*1.0/np.max(np.abs(spec)))
18 plt.xlabel('Frequency (kHz)')
19 plt.ylabel('Power')
20 plt.grid()
21
22 plt.show()
```

The first line imports the `fft()` function, while the second line imports the data and unpacks it. Then, the data is plotted in the first subplot. We're now set to convert the data into the frequency domain, which is done simply by passing the signal to the `fft()` function. Then, we convert the time into frequency. If you know a little bit of Fourier Transform, you'll know that you only really need half of this data, as the other half is simply the complex conjugate. That's why only half the data is plotted, while the power in the frequency domain is also normalized. If we plot this, we can see a clear frequency that dominates over all others, even though the time-domain plot had no clear pattern. Thus, in a few short lines of code, scipy's fftpack was able to perform a frequency-domain analysis for us, and also create some nice plots!

11 Pandas

Data science is a fast-growing field that is spreading its tentacles to virtually every traditional field: from railroads to agriculture to the social sciences, data is the currency of the digital millennium. But while we may have a lot of data, it can quickly become difficult to analyze it with enough confidence to make decisions. That's a gap that data science is trying to fill, and one of the most popular tools in it is the pandas Python library.

Pandas is built on top of numpy, but with further abstractions that makes it easier to read and analyze data. What's more, pandas is highly interactive, giving the programmer almost instant-feedback - indeed, it feels a lot like Excel in that respect. And perhaps the most useful feature is pandas suite of data-cleaning tools: after all, while there's a lot of data, there's still a shortage of *good* data. In this chapter, we'll use pandas do analyze drives from the 2013 college football season (data courtesy CFB Stats), which can be found in the drive.csv file in the data folder.

11.1 Series and DataFrames

Pandas defines two important data types: series and dataframes. A pandas series is essentially a 1D numpy array containing data of the same type, while a dataframe is a 2D array consisting of individual series. A dataframe can ingest data from a CSV file in a 'smart' way i.e., it can parse the data and handle missing data very well. Here's how to create a dataframe out of the drive.csv file:

```
1 import numpy as np
2 import pandas as pd
3
4 df = pd.read_csv('../data/drive.csv')
5 df.head()
```

The first two lines import numpy and pandas (pandas requires numpy), with pd being ht usual alias. Then, a dataframe `df` was created and data from the data.csv file was passed to it using the `read_csv()` function. Pandas is great at instantly visualizing data - the `head()` member function of any dataframe shows the first five entries. Try it and see. The first row of the dataframe contains headers, which as we'll see, are very useful to access data.

Let's get some information about the dataframe:

```
1 print(df.shape, df.columns, df.dtypes)
2 df['Game Code'][10:25]
3 df.describe()
```

Three properties of the dataframe were analyzed:

- `shape` is similar to numpy's `shape`, which returns the dimensions of the dataframe
- `columns` returns a list of column headers
- `dtypes` returns the data type of each columns, as interpreted by `csv_reader()`

Individual columns can be accessed using the column header, as shown in the second line of the code above. This is a particularly useful feature, because it allows the programmer to avoid having to remember the index of a column, and also makes the code more readable. Multiple columns can be accessed at the same time by separating their headers with a comma. Once a column(s) has been accessed, individual rows can be sliced as usual. Pandas furthermore has a `describe()` function that gives a quick statistical summary of the data in each column.

11.2 Cleaning Data

As is often the case, the dataset has some missing values. The `isna()` function returns a Boolean value, which is `True` if an element is missing (thus it returns as many values as there are elements). This isn't particularly useful for a big dataset since we can't really check each and every value, but we can then sum up the total number of `True`'s (which are converted to a integer 1, while `False` becomes 0 for each columns to see how many missing values we have in each, by applying the `sum()` function. We can do this with just one line of code:

```
1 df.isna().sum()
```

We see that there are some missing values in the Start Clock, End Clock, and Time of Possession columns, though the number is small as compared

to the size of the dataset. There are many ways to handle missing data, but in this case, since only a small number of values are missing, we can simply drop the rows that have any missing values using the `dropna()` function:

```
1 df = df.dropna()
```

Note that `dropna()` returns a cleaned-up dataframe, which we then use to overwrite the existing one. We can now analyze this cleaned up dataframe.

11.3 Challenge: Analyzing Data

The `describe()` function is good for providing a quick summary of all the data, but we often want to ask specific questions from it. In this challenge, we'll look to analyze the data to answer three questions:

1. What was the distribution of the ways that drives ended?
2. Of the drives that ended in a touchdown, what was the distribution of the time of possession?
3. Which team made the most yards per play on average?

The first question is easily answered. Here's my short solution:

```
1 N = df['End Reason'].count()
2 df['End Reason'].value_counts()*100./N
```

Essentially, the problem boils down to figuring out the number of unique values in the 'End Reason' column. This column contains categorical information, so this operation is valid (it would not be valid on a column with continuous data like floats). The `value_counts()` function does just that, and then we normalize it as a percentage of the total. We see that over half of all drives ended in punts and touchdowns, while only a minuscule fraction ended in a safety.

The second question involves two steps: extracting only those rows where the 'End Reason' is a touchdown, and then finding descriptive statistics of the time of possession (this is a float data type, so we can apply descriptive statistics to it, not just value counts). Here is my solution:

```
td = df[df['End Reason']=='TOUCHDOWN']
td['Time Of Possession'].describe()
```

First, we create a mask for all those rows where 'End Reason' is a touchdown. This is exactly the same as creating masked arrays in numpy (indeed, it is built on top of it). Then, we pass this mask to the dataframe itself, which returns only those rows where the make is `True`. We save these rows to a new dataframe called `td`. Now, we just take the 'Time of Possession' column of this new dataframe and use the `describe()` function get get its distribution. We see that on average, each drive saw the offense in possession for about 159 sec (about 2.6 minutes).

The third question is more complicated, partly because it involves a new variable (yards per play) that isn't already in the dataframe, and partly because we need to group the data by team before we can analyze it. Here's my solution:

```
df['Yards Per Play'] = df['Yards']/df['Plays']
teams_grouped = df.groupby(['Team Code'])
teams_grouped_mean =
        teams_grouped['Yards Per Play'].mean()
teams_grouped_mean.sort_values(ascending=False).head()
```

The first step is to calculate the yards per play of each drive, for which we define a new series, 'Yards Per Play', just as we would do in a numpy array. The difference is that in this case, the operation on the first line both creates the new series *and* adds it to the dataframe (whereas in numpy, we'd have to append it separately). So that takes care of the first difficulty. The second step requires us to group the data by team (represented by the 'Team Code' column). This can be fairly complicated to do on a spreadsheet program, but pandas has a `groupby()` function, which creates a special grouped dataframe `teams_grouped` for one or more columns passed to it. This grouped dataframe doesn't actually have any data yet (it does have access to all the data in the original dataframe from which it was created though) - but when any function is applied to it, it returns values of the function grouped by those columns.

So, armed with this grouped data frame, we tell it to take the mean of the 'Yards Per Play' column - and it returns a series `teams_grouped_mean` that has the mean of those columns corresponding to each team. However, the

question was which team had the most yards per play on average, so we need to sort this data. That's what we do in the last line, sorting in descending order, and then using `head()` to see just the top five results. From the results, we see that Team 2 has the best performance, making about 10.6 yards per play - pretty impressive!

Pandas can do much more. Analysis that would take a lot of time and effort by hand or even on a spreadsheet can be completed in just a few lines of code. And the cherry on top is that Pandas can use matplotlib to visualize the data too. That's why it is one of the most popular libraries used by data scientists.

12 Conclusion

12.1 Reminiscence

Teaching is an extremely challenging but enormously rewarding profession. I once met a sprightly young freshman that was struggling with a homework problem - a Python code, no less - and asked for help. I sat down with them, talked about a plausible algorithm, which they dutifully typed in. And right on queue, it threw up an error! We spent the next few minutes (mostly on Google) figuring out what the syntactical error was, and once we fixed it (turns out I was using the wrong function to append data to a list), it worked.

My young friend's reaction after the code worked is what caught my attention though, something on the lines of "so do I really need to keep searching for commands on Google all the time?" Unfortunately, their memory of that homework problem remained the struggle of finding the right syntax, and not the discussion we had about the algorithm to solve the problem. Not my finest hour of tutoring.

12.2 Next Steps

Why did I share that anecdote? Because I want to emphasize that *programming* and *coding* are not the same thing. Initially, you'll find that you're just struggling to find the right function and trying to second-guess the people that created the program, instead of focusing on the actual problem itself. Even though Python probably has the simplest syntax of all mainstream programming languages, it too can take some time to get used to.

But eventually, with enough practice, you'll find yourself being able to recollect functions with ease. This is *not* because you've somehow memorized all of them, although that might be a secondary effect. It's because you've learned to *think* like a programmer, like the people who created the language. At that point, the code will not only seem natural, it will seem obvious - 'of course this is the best way to do it!' It's at that point that you go from writing code to writing a program.

How do you get there? Like Dan LaRusso did in *The Karate Kid*. By trying, failing, and trying again, until you can do it on just one leg (although you may need to keep your eyes open). Enjoy the ride!