Introduction to Scientific Python

Sam Morrell

November 9, 2017
v1.0

Contents

1 Introduction

When approaching data analysis, there are so many different tools that it's often hard to know what to choose. There are many different packages, both free and commercial, designed to help you process your data or generate models. All of these come with their strengths and weaknesses. Within astrophysics research, the software that is currently dominating in the field of research software is Python. In this document, I want to just give you some reasons why you might want to pick up Python, and also get you started with the Python language and some of the most popular packages for dealing with scientific data.

1.1 Why Should You Try Python?

So, the big question is, why is Python the choice for you? As with many things, your choice of tool should depend on the job you wish to undertake with it. As is often the case with popular language, Python is being used for all manner of different things, many of them to which the language is rather poorly suited. So, before we get started, let's have a quick look at what Python is good at, and what you might want to think again for. As a language, Python is very high-level, making it quick and easy to write very complicated scripts to carry out all manner of tasks. This makes it the ideal language for writing code when you're very much experimenting with data. It's ideal for code that you want to quickly try out new ideas and rapidly evolve over time. It's also really good for those quick little scripts where you want to process some data you collected and do a really nice looking plot. Another advantage of this language is the whole plethora of packages written by other people that you can leverage to quickly get experiment with an idea.

What is it not good for? Python very rapidly starts to fall down when performance is concerned, because it's an **interpreted** language, meaning that code you write doesn't run straight on your computer. It first gets run through the Python interpreter, which translates it to bytecode and then gets run on a virtual machine; so there are many steps between clicking run and getting an output. Compiled languages, such as C, C++ and Fortran, will run much faster. These should definitely be used for high-performance code. To get around this, many Python packages are actually written in C or Fortran, and bridge to Python in order to improve performance. Another thing that Python falls down on is long term maintainability of code. On a daily basis I compile Fortran code that was written 30 years ago and it compiles and runs without a hitch on modern systems. The same cannot be said of Python. The reason this works so well is that Fortran code defines an interface that you have to stick to if you want to use the code. In Python, there is no means for formal definitions of interfaces. Additionally, Python has a very, very large segmentation problem. For years there are have been two competing dialects of Python; Python2 and Python 3. These two dialects are ever so subtly different, but different enough that one will not run on the interpreter of the other. This is fine until it comes to maintaining old code which has to be painstakingly converted to Python3, or even worse a Python package that is written in the wrong version. Because of this issue, I strongly suggest that you learn and write with Python 3. Python 2 is in the process of being phased out and due to the issues that have been caused by this upheaval, future versions are much more likely to support Python 3 code.

With that out of the way, let's go and write some Python code.

I've tried to only use concepts that I've introduced in the examples in each section, but the **print()** and **input()** functions are so useful I've used them throughout. They are given a proper introduction in section 5, but all you need to get the gist of the code is that **print()** and **input()** output to and accept user input from the console respectively.

2 Basic Syntax

Compared to many python is a very simple language. One of the main advantages of it is that if you understand english and some basic mathematical operations, you will easily be able to read most basic Python programs with minimal effort. In this section, I'll take you through the basic language features and syntax of Python to show you what to expect when reading and writing your very first programs.

2.1 Assignments

At their core, programming languages are used for performing tasks. Many tasks, such as a mathematical equation, will return a value as an output. If you're doing a single calculation, you can just output the answer and be done with it, but in reality a single solution is very rarely useful. An assignment operation allow you to store the output from an operation within a variable:

```
a = 1
b = 2.3
c = a + b
```

2.2 Arithmetic Operators

The snippet above is a very basic example of an assignment using an addition, a type arithmetic operation. The Python language contains operators for all the commond arithmetic operations. An operator is placed between two operands to form a statement that can be used or assigned. The type of the result of the operation is determined by the operands that go into it. The arithmetic operators within Python are:

+ The addition operator is used to add two operands together.

- The subtraction operator is used to subtract the operand on the right from the operator on the left.

* The multiplication operator is used to multiply the two operands.

/ The division operator is used to divide the left hand operand by the right hand operand.

% The modulus operator divides the left hand operand by right hand operand and returns the remainder.

** The exponent operator is used to raise the left hand operand to the power of the right hand operand.

It's important to remember that operator precedence in Python is the same as in mathematics, so exponent operations will be performed before divisions and multiplications, which in turn will be performed before additions and subtractions. Remember also that parts of your equation contained within brackets will be calculated first, so parathesising a portion of your equation will raise its precedence and cause it to be calculated first. Here are some common examples of arithmetic operations in Python:

```
# Examples of common arithmetic operations in Python
a = 2 + 4 # 6
b = 3.4 - 7.2 # -3.8
c = 5.8 * 3.6 # 20.88
d = -4.5 / 7.3 # -0.616
e = 59 % 4 # 3
f = 3.6 ** 1.3 # 5.29

# Remember that the precedence of an operation can be controlled using brackets
g = 10.0 * (4.7 - 3.2) # 15.0
h = ((1.3 + 7.2) * (9.4 ** 2)) / (9.2 + 1.4) # 70.85
```

2.3 Comparison Operators

As well as being great at doing a load of operations quickly, you can also make deicions in your programs. Comparison operators are the way that you make these decisions. In Python, the comparison operators are:

`==` This is the equivalence operator, and it returns True if both the left and right hand operators are equal, else it returns False.

`!=` This is the non-equivalence operator, and it is the negated version of the equivalence operator. If the left and right hand operators are equal, it will return False, and vice versa.

`>` The greater than operator returns True if the left hand operand has a larger value than the right hand operand, else it returns False.

`<` The less than operator returns True if the right hand operand has a smaller value than the right hand operand, else it returns False.

`>=` The greater than or equal operator works the same as the greater than operator, but it also returns True if the left and right hand operands are equal.

`<=` The less than or equal operator also works the same as the less than operator, but it also return True if the left and right hand operands are equal.

```
# Below is a demonstration of our comparison operators.
# In here I've compared integers, but it works identically for floats.
1 == 2 # False
1 != 2 # True

1 > 2 # False
1 < 2 # True

3 > 2 # True
2 > 2 # False
2 >= 2 # True

1 < 2 # True
2 < 2 # False
2 <= 2 #True
```

2.4 Negation

Negating an expression simply inverts the truth value of it, so *True* becomes *False* and vice versa. This is done using the **not** keyword. Any expression this keyword is applied to will become its mirror opposite. So, for example:

```
not 1 == 2 # True
not 1 != 2 # False

not 1 > 2 # True
not 1 < 2 # False

# Another place this is useful is for checking
# to see if an element is in a list
list1 = [1, 3, 5, 7, 11]
2 in list1 # False
3 in list1 # True
9 not in list1 # True
```

2.5 Combining Expressions

When constructing comparison expressions, it's possible to glue multiple expressions together with the **and** and **or** operators. *and* requires that both expressions on either side be *True* for the expression as a whole to be *True*. The *or* operator merely requires one of the two expressions to be *True* for the expression as a whole to be *True*.

```
1 == 2 and 3 == 2 # False
1 != 2 and 3 == 2 # False
1 != 2 and 3 != 2 # True
1 == 1 and 2 == 2 # True

1 == 2 or 3 == 2 # False
1 != 2 or 3 == 2 # True
1 == 2 or 3 != 2 # True
1 != 2 or 3 != 2 # True
```

3 Variables, Types and Collections

3.1 Types and Variables

As with many programming languages, Python contains variables. Variables are storage locations for values. You can think of them as boxes that are used to contain values. The kind of information it stores is determined by the **type**. In some languages, you have to worry about the type that a specific variable is, however Python is dynamically typed, which means it deals with this for you.

Python is capable for working with many different types of data. There are a slew of useful primitive types. These are:

Boolean

A boolean data type is a simple True or False value (note the upper case T and F). This can be use for switching parts of your code on and off depending on a value. It's also worth noting that all comparisons that you will use in your code will output a True or False value depending on the input.

Integer

An integer is simply a whole number. This number can be negative or positive and has a maximum value of $2^{31} - 1$ or $2^{63} - 1$ depending on whether you're using a 32-bit of 64-bit system. Don't worry if you exceed the maximum length though. Because python is dynamically typed, it will automatically change your int into a long number, which can hold **much** larger numbers.

Floats

These, once again are numbers and they can be both positive and negative in the same way that ints can, but the crucial difference is that floats have a fractional part, meaning that values stored in floats don't have to be whole numbers. If you're writing scientific code, I would imagine the vast majority of your numbers will be floats. Something that will likely come in useful when assigning using numeric literals is that Python embraces

Complex Numbers

I won't be going into complex numbers here, but it's worth noting that they exist in Python in case you ever need them. As I'm sure you are aware complex numbers have both a real and imaginary part and are used in the solutions to a variety of physics and mathematical problems. Should you ever need them, the documentation for complex numbers is most useful and can be found here.

```
# This file shows all of the different types that Python understands

# Boolean variables can store true or false values
space_exe_is_awesome = True

# An integer variable is a whole number
meaning_of_universe = 42
negative_integer = -1234

# A floating point variable is a number that contains a fractional part
pi = 3.14159265
negative = -1.68394
c = 3.0E8 # Scientific notation

# A string is a variable that contains some text
greeting = 'Hello, world.'
```

It's worth noting that sometimes Python may have issues deciding what type of information it's trying to store within a variable, and odd behaviour may creep up in your code. If in doubt, you can tell Python what type it is manually. Some of the essential conversions are listed below.

```
bool_from_string = bool("true")
string_from_numbers = str(12.3)

# You can convert a string to a floating point number of
float_from_string = float("3.14159265")
integer_from_string = int("42")
```

3.2 Collections

It's often useful to combine more than one value together into an object. This can be done in multiple different ways, depending on what you require. First off, we shall start with the simplest, the List.

Lists

Lists are really simple Python objects whose purpose is to store other Python objects, whatever they may be, in order. The key thing about lists is that they are lightweight, the guarantee that your objects will remain in the same order that you set them in and they are capable of storing multiple different types at once; a property known as type heterogeneity.

```
# Lists are capable of storing a load of different types.
# For example, integers, floats, strings and a combination thereof.
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
b = [13.2, 95.35, 2634.27, -0.251, 0.00152]
c = ['Hello', 'world']
d = [7, 42.4, -9, 36.75, 'g', 'Hello', 'World']

# Elements within a list can be accessed and assigned simply using square brackets.
a[5] = 23
b[3] # Outputs: -0.251

# Something that can be really useful is the ability to append things to your lists
a.append(10) # Now a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# At any point you can find the number of elements inside the list using the len() function.
len([0, 1, 2, 3, 4]) # In this case, len() gives you 5.
```

Tuples

Pronounced *two-pulls*, tuples share many features with lists. Their purpose is also to store a selection of Python objects of varying type in a sequence. However, they have one crucial difference. Whereas lists are mutable data structures, meaning you can change them after you make them, tuples are immutable, so if you try and alter when they have been instantiated, Python will throw an error. They are used like so:

```
# Like lists, tuples can store a sequence of heterogeneously typed python objects.
# Note that you use round brackets instead of square brackets to instantiate a tuple.
a = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
b = (13.2, 95.35, 2634.27, -0.251, 0.00152)
c = ('Hello', 'world')
d = (7, 42.4, -9, 36.75, 'g', 'Hello', 'World')

# In the same way as lists, elements of tuples can be accessed
b[1] # Gives 95.35

# We can also use len() to get the length of them
len(b) # Gives 5

# However, it's important to note that assignments and appends will result in an error
b[1] = 42 # Causes an error!
b.append(42) # Causes an error!
```

Slicing

Lists and tuples appear on the surface to be very similar. Because of this, they share many high level features, including slicing. Slicing in Python allows you to quickly and easy take a subset of a sequence of data. To slice a list or tuple, you simply use the square brackets as you would with an access operation, but you include two numbers separated by a colon between. The two numbers are the first and last index of the subset that you want to extract. The easiest way to show how it works is with an example.

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
b = (13.2, 95.35, 2634.27, -0.251, 0.00152)

# Now we can slice both of these collections to extract a subset of their elements.
a[1:6] # Gives [1, 2, 3, 4, 5, 6]
b[1:3] # Gives (95.35, 2634.27, -0.251)
```

Dictionaries

If your data is a little less sequential and lend itself to a little more structure, then dictionaries are probably a little more useful to you. Whereas you've been able to think of lists and tuples as sequences of objects, dictionaries are an object that contains other objects in the form of key-value pairs. A key value pair somply means that each piece of value within a dictionary has a string of text assodicated with it. If you give the object this text, it will give you back the value associated with it. Simple as that. They are very easy to initialise, you write out a string for the key, a colon to seperate the key and value, and then the value. If you want to include more than one, you simple separate them with a comma and when you're done enclose the whole thing in curly braces. To access or modify elements of the dictionary, the syntax is remarkably similar to arrays and tuples, but instead of using a number in the square bracket, we just simply put the key in the form of the string inside insteam. This may seem complicated, but the code snippet below will show you how it works:

```
# This is to show you how to use dictionaries.
# They contain key value pairs of Python objects.
dict_1 = { 'number_property' : 1953.452, 'text_property' : 'Value' }
dict_2 = {
        'name' : 'V* LM Tau',
        'ra' : 55.5417,
        'dec' : 24.0856,
        'spectral-type' : 'M',
        'variable' : True
}

# Indiviual elements within dictionaries can be access like so
star_name = dict_2['name']
variable_star = dict_2['variable']

# And they can be set like so
dict_2['variable'] = False
dict_2['name'] = 'LM Tau'
```

Combining Collections

The really neat thing about all of these collections, and one of the main reasons Python is so popular, is because we actually really easily combine these containers together however we want. So, for example we can have an array of arrays:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = [7, 8, 9]

combined_list = [list1, list2, list2]

print(combined_list[1]) # Will print [4, 5, 6]
print(combined_list[1][1]) # Will print 5
```

or maybe you want to include tuples inside a dictionary:

```
tuple2 = (4, 5, 6)
tuple3 = (7, 8, 9)
tuple1 = (1, 2, 3)

dictionary = { 'tuple1' : tuple1, : 'tuple2' : tuple2, 'tuple3' : tuple3 }

print(dictionary['tuple2']) # Gives (4, 5, 6)
print(dictionary['tuple2'][1]) # Gives 5
```

I'm sure you can start to see how many combinations there are and how incredibly useful the ability to combine these together. I won't bore you with any more examples, but get combining.

4 Control Flow

So far, our code has been a very linear affair, we have been typing code that just gets run in sequence from start to end. What if we could execute different bits of code if certain conditions were met, and a different one if not. That's what the **control flow** of a program is for, for controlling which bit of code gets run under which circumstances. In this section, I'm going to be showing you the control flow statements that are contained within python. Before we delve in head first, I should brifly tell you about the zany way that Python denotes this different chunks of code. Whereas most other languages will denote the start and end of a block of code outside the flow of normal execution using a character, like the curly brace in the case of C, Java, Javascript and many other languages, or **end** statements in the case of Fortran, Python choses to perform this task using purely indentation. The logic behind this is that any programmer worth their salt will indent correctly anyway, so why not make a language feature out of it.

> Warning: Due to its importance in the syntax of the language, Python is very picky about the kind of indentation that is used. It doesn't matter which indentation you use, I recommend a single tab or 4 spaces, but stick to it throghout your code as it doesn't appreciate you changing your choice of indentation part way through a file. This can be a real pain if you're copying in a code snippet from the internet.

In the Python language there are three different flow control statements: **if**, **for** and **while**. They are all incredibly useful for different purposes and we'll be going through them all in this section.

4.1 *if* Statements

The **if** statement is an incredibly powerful construct in an programming language. It's the feature that takes your program from just straight-forwardly executing commands in order to making decisions and executing a piece of code that makes sense given the current state of your program. This is where the comparison operators that we introduced in subsection 2.3 come into their own. We can build these up with variables in your code to make very powerful, and very flexible, programs. *if* statments are structures like so:

```
if <expression>:
        <code block>
elif <expression>: # Optional
        <code block>
else: # Optional
        <code block>
```

Here's a general example of an *if* statement:

```
# In this example we have an unknown integer and the user to guess what it is.
# We're going to make it easier and tell them whether it's higher or lower.
number = 42

guess = int(input('Guess the integer: '))

if number == guess:
        # Remember to indent, because this is a block controlled by the if statement.
        print("Well done, you got the number!")
elif number > guess:
        print('Sorry, your guess was incorrect. The number is higher.')
else:
        print('Sorry, your guess was incorrect. The number is lower.')
```

This example shows all of the components that make up your typical *if* statment, this includes *if*, *elif* and *else*. Here's how it works. The *if* statement presents a logical argument, in this case: does the variable *guess* contain the same value as *guess*? If this is false, the code will just skip over the entireity if the indented statement and go on to the next statement at the same indentation; in this case the accompanying *elif*. However, if the expression is true, the code that is contained within the indented if block is executed. The *elif* statement performs a very similar function, but it's purpose is to catch another specific case that isn't addressed by the original *if* statement; in this case that the *guess* is less than the *number* we're after. In the case that it is, the code inside the indented statement is executed, in the same way that the *if* statement works. We've now caught all eventualities apart from one, the *guess* is higher than the number. It would be totally valid to do another *elif* here, but since we know that all other cases have already been caught by the statements above it, we can just use the *else* statement, which executes the indented code only if none of the *if* and *elif* statements earlier in the flow have been true.
Some things worth noting at this point is that it's only possible for one of these indented pieces of code to run, as they are mutually exclusive. This is what *elif* statements are useful for. If there are branches that could be true, only the first in the flow of the program is executed before skipping the remaining ones. It's also worth noting that only the *if* part of this is the only mandatory part, so if you only want to run a piece of code if something is *True* or *False*, you needn't include the *elif* or *if* parts.
It's also worth remembering that **all** of the code in the indented block is executed, including another other if statements and blocks of code that it may contain, so you can simply nest as many if statements inside other blocks as you want.

> Warning: Just because you can nest infinitely, it doesn't mean you should. The principle that should be driving all of your programming is to minimise the complexity of your code. This makes it more readable and easier to understand, but most of all it reduces the chances of something going wrong!

4.2 *for* Statements

We've come across lists and tuples in the preceeding sections. We can do small manipulations, but the chances are you'll likely want to do some kind of calculation using every single data point in your list. This is where the **for** statement comes in.

```
for <variable> in <collection>:
        <code block>
```

The *for* statement essentially itertates over every element in some kind of iterable collection and runs a piece of code, passing the value of that list element as a variable. So for example, if we wanted to calculate the square of every elemenmt of a list, we could use:

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
square = []

for number in list:
        square.append(number ** 2.0)

print(square) # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

which will print out the array of the squares of these numbers. In this case the *(*list.append()) can be useful to use with an empty array to generate the result. I should stress that it doesn't matter what the variable is called as long as it's a valid name in Python. As long as the name is the same in the *for* statement and in the block everything will work just fine. Just as *if* statements, *for* statements can be nested, which can be particularly useful for iterating over lists within a list.

4.3 *while* Statements

The *for* loop is great if we want to iterate over some kind of iterable data structure, but what if we just want to keep looping until some expression is false, like a combination of the *if* and *for* statements? The **while** statement serves just this purpose. It works like so:

```
while <expression>:
        <code block>
```

In this case, the code block will be repeatedly run for as long as the expression that is being tested in the *while* statement evaluates to *True*. When the expression evaluates to *False* the code will skip past the indented block and continue on to the proceeding statement. Let's make some changes to our guessing game from earlier in the section and show you how it works.

```
# In this example we have an unknown integer and the user to guess what it is.
# We're going to make it easier and tell them whether it's higher or lower.
number = 42
guess = 0

while number != guess:
        guess = int(input('Guess the integer: '))

        if number == guess:
                print("Well done, you got the number!")
        elif number > guess:
                print('Sorry, your guess was incorrect. The number is higher. Try again.')
        else:
                print('Sorry, your guess was incorrect. The number is lower. Try again.')
```

We are essentially telling this piece of code to keep on asking the user to enter an integer until they get the correct answer. You can use any number of expressions to perform iterations in your code; as long is it evaluates to *True* or *False* it will work.

5 Basic I/O

We've covered a fair amount about how to write code in the Python language, but the chances are if you're doing scientific Python you'll want to give get an output and maybe give values to your code when you run it. You'll likely also have a load of data sat on your hard drive in files that you want to read in, process and then output again. The good news is that Python 3 can read and write from the console and has the ability to read and write files built right in, so let's dive into how it all works.

5.1 Basic Output

One of the key things you'll want to do is to output things that you calculate from your code to the console. If you're familiar with C, you'll likely instantly think of the *printf* function as you goto function for this. The good news is that Python has it's own version of function called **print**, and it's actually a lot more forgiving then its C counterpart. The basic usage of *print* is like so:

```
print("<text>")
```

Basically, you call *print* as a function and give it a string of text to output. The cool thing is that you can actually output a large variety of Python types and statements, as long as it can be easily converted to text. Here are some examples:

```


# Here are some examples of things that
# can be easily output using the print function in Python.
print("Just a string of text") # Just a string of text
print(42) # 42
print(24.5) # 24.5

boolean_value = True
print(boolean_value) # True

# You can even perform a calculation inside a print
print(17.4 + 34.1 - 10) # 41.5
```

This is really useful, but you probably want to give a little more information than just the value. It's likely that you want to describe what it is you're outputting, and maybe add multtiple strings together, a process known as **concatenation**, or even substitute values into a string of text. The good news is that this is also very easy. You can concatenate as many strings was you want together just by using the + operator between them, so for example this is possible:

```
string = 'Spa' + 'ce ' + 'Exe ' + 'is awesome!'
print(string) # Space Exe is awesome!

# You can even do this inside a print call
print('Hello, ' + 'Space Exe' + ' awesome people!') # Hello, Space Exe awesome people!
```

What if you want to put values that you've calculated into your strings. Python 3 strings are actually objects that hide away a load of the details of how they work from you so you don't need to worry about them. This means that they are very easy to use and astoundingly flexible, but it also means that you can tap into the power when you need it. One such useful thing is the *string.format()* method, which allows you to inject values directly into a string of text. The basic syntax goes like so:

```
'{} {}...{}'.format(<value1>, <value2>,..., <valueN>)
```

It's as simple as this, if you put a pair of curly braces into your Python string and call *.format()* on it, you'll be able to substitute values into it. For each of your pairs of curly braces, you should provide a value as an argument into the *.format()*. The easiest way to understand how it works is to just show you some examples:

```
string = '{} {}'.format('one', 'two')
print(string) # one two

print('Acceleration due to gravity: {} m/s'.format(9.81))

meaning_of_universe = 42
print('Meaning of the Universe: {}'.format(meaning_of_universe))
```

format is incredibly flexible! You can include various format specifiers within the curly braces to get it to output the values you feed in more or less however you want; much in the same way *printf* works in C. There's far too much to cover here, and there's really no point repeating it when others have done such a fantastic job, so if you want to learn about the power of *format* visit this site maintained by Ulrich Petri that goes into nauseating details about the workings of it: https://pyformat.info/

5.2 Basic User Input

In the same way Python can output to the terminal with *print()*, it can read from the terminal with **input()**. You can use it like so:

```
input(<message>)
```

So, you call *input()*, giving a message to prompt the user what value you're looking for, and it will return the value the user types. In same way Python dynamically types variables, it will automatically decide the best way to represent the value the user types and return it, so you can assign it to a variable. If you want the input in a specific type, you can wrap the *input()* call in a conversion function, as you would any other variable.

```
# You can get a value from the console like this
name = input('What is your name?: ') # What is your name?: Sam

# You can chain this together with .format() and type
# conversions to make your program really interactive
age = int(input('Hello, {}. How old are you?: '.format(name)))

# Inputting a true or false here will convert it to a bool
member = bool(input('Are you a member of Space Exe, {}?: '.format(name))) # True
```

5.3 Reading from a File

Okay, so let's say I have a file sat on my drive called *read-me.txt* that contains a load of vitally important data. It's at this point that dealing with data can get fairly difficult, as Python needs to be able to **parse** the format that your data is in. Put simply, Python needs to be able to load the file and give it to your code in useful form. Generally, this is a problem that you can hand off to other code. However, for the sake of arguments, let's see how we can read in a text file ourself. If we want to open a text file, we need to use:

```
f = open(<filename>, <mode>)
<code to read the file>
f.close()
```

So, for example, what if we had a file with x and y coordinates on each line with a space separating them. The code to read this would be:

```
# Code to read x and y coordinates

filename = 'xydata.dat'
x = []
y = []

# First we open the file
f = open(filename, 'r')

# We can read the entire contents of the file into
# a string, like so
entire_file = f.read()

# Or we can read each line into an element in a list.
# Before that, we seek the start of the file to read it in again.
f.seek(0, 0)
rows_of_file = f.readlines()

# We can loop over the lines in a file like so
for row in rows_of_file:
        tmp = row.replace('\n', '')
        x_tmp, y_tmp = tmp.split(" ")
        x.append(float(x_tmp))
        y.append(float(y_tmp))

# Finally we close the file
f.close()

print(x) # [12.0, 23.0, 13.0, 12.0, 16.0]
print(y) # [562.0, 762.0, 87.0, 97.0, 212.0]
```

We first read in the lines of a file into the code, then loop over those lines. For each of those lines we remove the newline character at the end of the line and then call the *string.split()* method, which converts the string into a list by splitting the string each time it finds an instance of the string you give it. This gives us the x and the y values which we can then append to the final lists.

Warning: This is the code to read one particular type of file. Generally your files will be more complicated to read. This alos doesn't do any error checking. When writing actual code, I highly suggest using third-party modules specially designed for the format you want to read, as it will likely be much more reliable.

5.4 Writing a File

We can also write to a file in much the same way. The only difference is that instead of using **file.read()**, we can use **file.write()** to just write to the text file. Here's an example of it in action:

```
# Code to write x and y coordinates.
x = [12, 23, 13, 12, 16]
y = [562, 762, 87, 97, 212]

# Opens the file for writing, creating it
# if it doesn't exist.
f = open('xydata.dat', 'w')

# Check that the file is open before we try to write
if f:
        # Iterate from 0 to length of the arrays minus 1
        # because the arrays are zero indexed.
        for i in range(0, len(x) - 1):
                f.write("{}\t{}\n".format(x[i], y[i]))

        # When we're done writing, we can close the file
        f.close()
```

It's as easy as this. We open the file with the correct mode, in this case we are using *w* so that we can write to the file, which will be created if it doesn't exist. Because we are saving an array, we want to iterate over all of the elements of the list. Because we have two Lists, using a normal for loop won't work, however we can use the **range()** function which we can iterate from 1 to the length of the list minus 1, because Python Lists are zero indexed. Since we have just opened a text file, we can write strings to it. We can use the **string.format()** to easily format the string of text just as we want to in a really convenient way. We can just write the element of the list at the current index into the string, remembering to include a tab
t character to separate the x and y coordinates and a newline
n character to end the current line. When we've finished writing to the file, we simply call $file.close()$ to close the file. You can write whatever you want to the file, in whatever format you want in this way. I'm just showing you a very basic example to get you started.

6 Functions and Modules

So far we've been through how to write code and you can put together a pretty decent Python program, but that's only half the battle. Keeping your code organised into seperate chunks solves a variety of different problems. In Python, organising is done with modules and functions. Modules are integral to Python, in fact much of the core Python library is contained in modules. I've found trying to keep as much of your code as possible away from the main script of your program can be very, very useful. Also, putting that little bit of extra thought into how your structure your programs can really help you out in the future.

Just as example, I wrote a module for importing our data into a script. I could have just written this in the main flow of the program and it would have worked just fine, but because I kept it contained in a module well away from the rest of the code, it meant that when it came to writing another program that needed to access the main data I could just completely reuse that code with no copy and pasting and zero modifications. I am a firm believer in writing good, decoupled, reusable code, so I'll take you through the basics. Writing code like this is definitely more of an art than a science, so don't be put off if your first attempts at doing this cause hassle. It's definitely a skill worth developing in the long run, as it can save you days of work.

6.1 Functions

If you have a single piece of code that you think you will end up using a fair amount and lends itself to being segregated from the rest of the flow of your program, you can tuck it away into a function. They are very easy to put together, like so:

```
def function_name(arg1, arg2):
        # contents of function go here
        print(arg1, ', ', arg2)

# You can call your function like this
function_name('Hello', 'World') # Hello, World
```

The **def** keyword marks the start of a function definition, followed by the name of the function. Normal Pthon naming rules apply, so it cannot start with a number or contain any spaces or other characters that make up the syntax of the language. Remember that blocks of code in Python are denoted with an indentation, and functions are no exception. When you have finished the code for your function, simply being the indentation back in. In the same way is other functions you may have used already, you can feed values into your block of code using arguments contained within the brackets. You can make use of these anywhere within the function.

Just as with other functions, your functions can return values to the code that calls them. This us done using the **return** statement. You can return any value from a function, in either variable or literal form.

```
a = 0
def meaniing_of_universe():
        return 42

def area_of_cricle(radius):
        area = 3.14169265 * ( radius ** 2.0 )
        return area

print(meaning_of_universe()) # 42
a = area_of_cricle(2.5)
print(a) # 19.63
```

Any code that you could write normally can be written inside the function. There are a few things that you should remember though. The scope of a piece of code essentially describes the environment its being run in

and which variables are available to it. For example, in the *area_of_circle* function, the area variable is contained within the scope of that function, making it a variable with local scope. The variable *a* outside of the function is in the global scope of this function, meaning that it's available to all the functions that call it. The crucial thing to remember is that variables in the local scope of a block of code cannot be accessed outside of that block, but variables in a higher scope can be accessed lower down. A word of warning at this point is that variables within the global scope should be avoided where possible. Global variables in Python are more far more acceptable in Python than they are in a language like C or C++, however relying on access to global variables, especially within functions, is the sign of a badly designed piece of code, so keep that in mind when thinking about your variables.

6.2 Modules

We know that we can enclose bits of code within functions to make them reusable, but what if we wanted to make a whole reposity of useful function that we can just pull in to any programs that we want. That is preciesly what modules are. The core Python library itself is packaged into modules that you can easily import into your own code. Python can simultaneously offer a large number of standard libraries without the overhead of loading them every time by allowing you to import only the libraries that you need. If you've been following me through this, you've probably already seen me import a module from standard library when working with files. When working with files, I imported the os module like so:

```
import os.path

file = 'test.dat'
if os.path.isfile(file):
        print(file, 'exists!') # test.dat exists!
```

What we are doing here is loading in the **os.path** module and then using the *isfile* function contained within this module to check and see if a file exists before opening it for reading. That's good but how do we make our own module? Let's say I want to make a circle module that has a load of useful functions in it for dealing with circles. First, I make a Python file called *circle.py*:

```

# Circle module - circle.py

# Calculates and returns the area of a circle with radius r
def area(r):
        import math
        return math.pi * ( r ** 2.0 )

# Calculates and returns the circumference of a circle with radius r
def circumference(r):
        from math import pi
        return 2.0 * pi * r
```

Inside our module, we have two functions, one to calculate the area of a circle and one to calculate the circumference. First off you'll note that we aren't actually importing anything in the module itself. We are importing the *math* library inside the functions themselves. This means that we don't actually load the module until it's required, saving the overhead of loading many modules that we may not even need. You may also notice that we only actually require the value of π from the *math* module, so loading the whole thing into cache is also a bit of a waste. In situations like this you can use the

```
from <module> import <function / value>
```

syntax to save loading the whole library. You'll also notice that instead of reqiring us to reference the module the function is in, this method simply imports the functions or variables into the global scope. You don't need to worry about modules imported this way adding overhead to every function call, as one modules are loaded a first time in a Python instance, they are essentially kept in memory until the program terminates, meaning subsequent imports are very fast.

Now that we have a circle library, let's import it and use it in a piece of code:

```
from circle import area, circumference

r = 3.2
a = area(3.2)
print('Area: ', a) # Area: 32.17
c = circumference(r)
print('Circumference: ', c) # Circumference: 20.11
```

You can see in this case we can import multiple parts of the module into the local scope by separating them with a comma. If we wanted to import everything within the module into the local scope, we could also accomplish that using:

```
from circle import *
```

but generally it's much safer to pick and chose only the functions you need, or just import the module itself. I hope this quick demo gives you and idea of just how powerful modules can be and how useful it can be to separate out core parts of your code into easily maintainable, reusable chunks. It may take a while to get your head around which parts could do with being spun out into a module and exactly how to do it, but trust me, keep trying. It pays off in the long run.

7 Writing Pythonic Code

In spoken language, there's a big difference between someone who can speak it and someone who speaks it like a native. In the same way, there's writing Python code, and then there's writing Pythonically. What is writing code pythonically? If you're familiar with other programming languages, the chances are that many of the topics and language features that we've been over are things you've seen in other languages and you feel write at home writing Python. However, Python introduces a whole bunch of little features that are designed to make programs easier to write, easier to read and all together more concise. Adopting these language idioms will make your code a little easier, and the Python pros will idolise you.

7.1 Inline If

If statements are great for deciding whether or not to run chunks of code, but sometimes when you need to do a single thing when some conditions are met within your code. This is particular useful if you need to assign a variable a different value depending on a value. The syntax for this is like so:

```
<expression-if-true> if <condition> else <expression-if-false>
```

It's very simple. If the condition is True, the first expression is evaluated and any value it generates is returned, and the final expression is evaluated and returns if the condition is False. Unlike a 'normal' if statement, the else clause is required in inline ifs, but you can just put the **None** literal here and it works just fine, but handling the False case is generally very useful. The best thing is that no matter which is evaluated, the value value produced by the expression is returned, meaning it can be assigned to a variable. Here are a few examples:

```
# We can make simple decisions and assignments much easier using this construct
a = 10
print('Hello, World') if a == 10 else None # Hello, World
c = True if a > 5 else False # a = True

# You can combine it with the 'in' to see if something is in an array or dictionary.
committee  = ['Ben', 'Marine', 'Mark', 'Matt', 'Sam']
name = 'Jim'
print('On committee') if name in committee else print('Not on committee')
# Not on committee
```

7.2 List Comprehensions

For loops are absolutely fantastic if you have a list that you want to run a large block of code on, but it's often the case that you want to initialise a list, or even initialise another list from the values of a first list. Using a real **for** loops to accomplish this is incongruous to say the least. Fortunately, one of the languages features of Python is **list comprehension**. This really neat feature lets you iterate through all of the values in a list and perform an operation on those values, generating a new list in a single line of code. The syntax of this are very simple and go like this:

```
<list> = [ <expression> for <variable> in <array / generator> if <conditional> ]
```

As you can see, this is like an assignment, a for loop and an if comparison all rolled into one. The whole thing is enclosed in square brackets. The expression and the for loop part are mandatory, however the if conditional is optional. The input to this is a pre-existing list or a generator, such as the **range** function. Here are some examples just to illustrate how it's used in practice:

```
# We can generate a new array using a generator, like the range function
array1 = [ x + 1 for x in range(10)  ] # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# We can generate a new list from the values of a previous list
array2 = [ i * 10 for i in array1 ] # [10, 20, 30, 40, 50, 60 70, 80, 90, 100]

# We can add in the conditional clause to filter off certain values
array3 = [ x for  x in array1 if x % 2 == 0 ] # [2, 4, 6, 8, 10]
```

7.3 Keyword Arguments

Sometimes, writing functions can be really hard, especially deciding on the order that you want arguments and how they should be tied into the code. In many other languages, the order or arguments is set and cannot be altered easily. However, Python has a remedy to this in the form of keyworded arguments, or kwargs as it's known in the language. Practically speaking, this is how you go about doing it:

```
def <function>(<argument1>, <argument2>, ..., <argumentN>, **kwargs):
        <expressions>
```

The way they are used is very simple. Traditional function definitions have a number of positional arguments, meaning that whenever you call a function, they will always be expected in the order they are defined within the definition. Keyword arguments are truly a panacea for both of these problems as they allow you to pass named variables into your functions as if you were passing in a dictionary of key value pairs. Not only does this allow you to pass them in using whichever order you wish, but you can also miss them out entirely. This is very useful if your function defaults to a sensible default value but you want the ability to build the ability to alter it into the interface for the function. Let's show you it in action.

```
# A demonstration of how to use keyworded arguments within your functions.

def greeter(name, **kwargs):
        greeting = kwargs['greeting'] if 'greeting' in kwargs else 'Hello, '
        ending = kwargs['ending'] if 'ending' in kwargs else '. '
        print('{}{}{}'.format(greeting, name, ending))

# First we can call it without a keyword argument
greeter('World') # Hello, World.
greeter('World', greeting='Howdy, ') # Howdy, World.
greeter('World', ending='! ') # Hello, World!
greeter('World', greeting='Sup, ', ending='? ') # Sup, World?
```

It's a rather pointless function, but it does a good job of illustrating how and why you might want to use keyword arguments. We have a greeter function that you pass in the name as the single positional argument, because this is something that you are going to want to set every time you call the function. You may have noticed that the inline if comes in particularly useful when using keyworded arguments, as it allows a simple, clear method of picking up the custom value set by a provided keyworded argument, but elegantly falling back to the built in default if it's absent.

7.4 Unpacking Return Values

Giving a function some data, sending it off to process it and then getting the resulting value is a core part of any good program. Leveraging the ability to structure your code in a convenient way is exceptionally powerful, but it's often troublesome to design the interface to this in a way that gives the flexibility that you want. Variable length non-keyworded and keyworded arguments allow an incredibly clean way to customise the input to your function, but what about the output? In most languages you are allowed to return a single variable from a function. Returning more than a single piece of data can be accomplished but it requires arrays, objects and other methods that just cause a lot of friction. To solve this problem, Python has a feature built into the language called **unpacking**. Let me show you how it works.

```
def pedantic_square_root(number):
        from math import sqrt
        root = sqrt(number)
        return (root, -root)

# We can give only one variable to assigne and
# we just get the tuple returned
root = pedantic_square_root(4)
print('{}'.format(root)) # (2.0, -2.0)

# However, because the function returns a tuple
# we can unpack them into two variables
root1, root2 = pedantic_square_root(4)
print('{} {}'.format(root1, root2)) # 2.0 -2.0
```

If you return a *tuple* from a function, it will simply return the tuple and assign it to that variable. However, if you separate your variables with a comma, the assignment operation will actually unpack the values from the tuple and assign the value to the corrsponding variable. Cool huh? Using this for returning multiple values from a function just makes everything easier.

7.5 Zipping and Enumeration

It's not uncommon to want to iterate over multiple arrays at once. Using a normal *for* loop isn't particularly useful, and using a *while* loop is just asking for trouble, so how can such a heady feat be accomplished? The great news is Python has two answers to this question **zip()** and **enumerate**. Let me show you quickly how these both work.

```
# Zip combines the two or more arrys into a single
# for loop, leting you easily iterate over the values

list1 = ['a1', 'a2', 'a3', 'a4', 'a5']
list2 = ['b1', 'b2', 'b3', 'b4', 'b5']

for l1, l2 in zip(list1, list2):
        print('{}, {}'.format(l1, l2))

# Will print:
# a1, b1
# a2, b2
# a3, b3
# a4, b4
# a5, b5
```

zip() simply iterates over the arrays as one and then lets python unpack the return value into each of the variables so that we can easily iterate over the values. Sometimes it's useful to get the index of the element of the list you're iterating over, which may well be the case in your science problem. In this case we use **enumerate()**:

```
# Zip combines the two or more arrys into a single
# for loop, leting you easily iterate over the values

list1 = ['a1', 'a2', 'a3', 'a4', 'a5']
list2 = ['b1', 'b2', 'b3', 'b4', 'b5']

for i, l1 in enumerate(list1):
        l2 = list2[i]
        print('{}, {}'.format(l1, l2))

# Will print:
# a1, b1
# a2, b2
# a3, b3
# a4, b4
# a5, b5
```

Instead of iterating over all of the arrays, **enumerate** only iterates over one, but it returns the index of the current array element, as well as the value. This means we can easily iterate over as many arrays as we want with corresponding elements without having to zip them. It's also useful for performing numerical integrations where knowing the adjacent elements is useful for calculating a value for the differential, i.e. dx.

8 Python 2 vs. Python 3

Throughout this guide I have been covering Python 3. This is because version 3 of the Python language is both the present and future of the language. However, I know there are some cases where you are likely to want to write Python 2 code, and it's generally nice to know the differences between the two in case you need them. In this section of the guide, I will take you through some of the key differences that you're likely to come up against when writing basic scientific Python. A comprehensive description of all of the differences between Python 2 and Python 3 is available at http://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html.

8.1 Print

The crucial difference is that in Python 3 *print* has been demoted. In Python 2 *print* is a statement, as opposed to merely a function. The reason for this, mainly because *print* as a function is much more flexible. Practically, the only difference to you will likely be this:

```
#Python 2
print 'This is a Python 2 print' # Will throw an error in Python 3
# Python 3
print('This is a Python 3 print') # Will work in Python 2
```

Long story short, just use the function version of **print** in whatever Python code you may write, as it's future proof and it will work on both Python 2.7 and Python 3 onward. If you want to delve a little deeper into what this change was made, there's a great little write up by one of the Python core developers, Brett Cannon, here: https://snarky.ca/why-print-became-a-function-in-python-3/

8.2 Integer Division

When writing scientific code for Python 2, this is definitely something to be aware of. When performing a floating point division, everything works perfectly, however if you do a division operation with two integers, the resulting value will also be an integer; which will lose all information of the fractional part of the result. This is not an issue in Python 3, however you can get around this in Python 2 in several ways:

```
# In Python 2:
print 1 / 2 # gives 0

# However, we can get around it using a type conversion:
print 1 / float(2) # gives 0.5

# Also, if we make one of the operands a float, it will
# output a float
print 1 / 2.0 # gives 0.5
```

Making a habit of making part of any division a floating point number will save anyone running your code on Python 3 some trouble, so it makes sense. Another way to instantly fix some legacy code would be to import *division* from the *future* module at the very start of your script, which will override the Python 2 '/' operator with the Python 3 version, like so:

```
from __future__ import division
# In Python 2:
print 1 / 2 # now gives 0.5
```

8.3 Unicode

Last, but certainly not least, Python 3's **str** class can now natively deal with unicode (utf-8) text. This may not be a big deal for a lot of you, but if you're writing code for a large international collaboration or community, and your code requires characters not within the normal ASCII character space, you'll be very grateful of it.

9 Conclusion

That just about gives you the basics you need to start writing scientific python code. Even though there's a lot of content contained within this little booklet, I've barely scratched the surface of what the actual language is capable of when importing other parts of the core libraries, let alone the vast and seemingly insurmountable ecosystem of community-made modules you can install with the click of a button. One of the other attractive features of Python is the plentiful documentation and code samples available on https://docs.python.org/3/, so make use of it and enjoy Python.