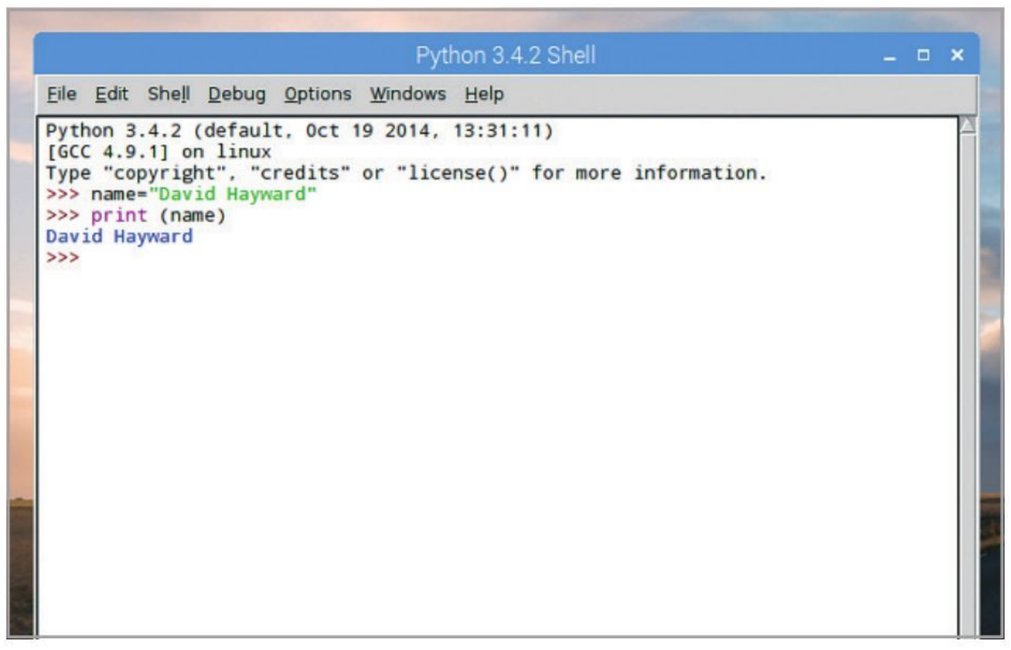
# Working with Variables

We've seen some examples of variables in our Python code already but it's always worth going through the way they operate and how Python creates and assigns certain values to a variable.

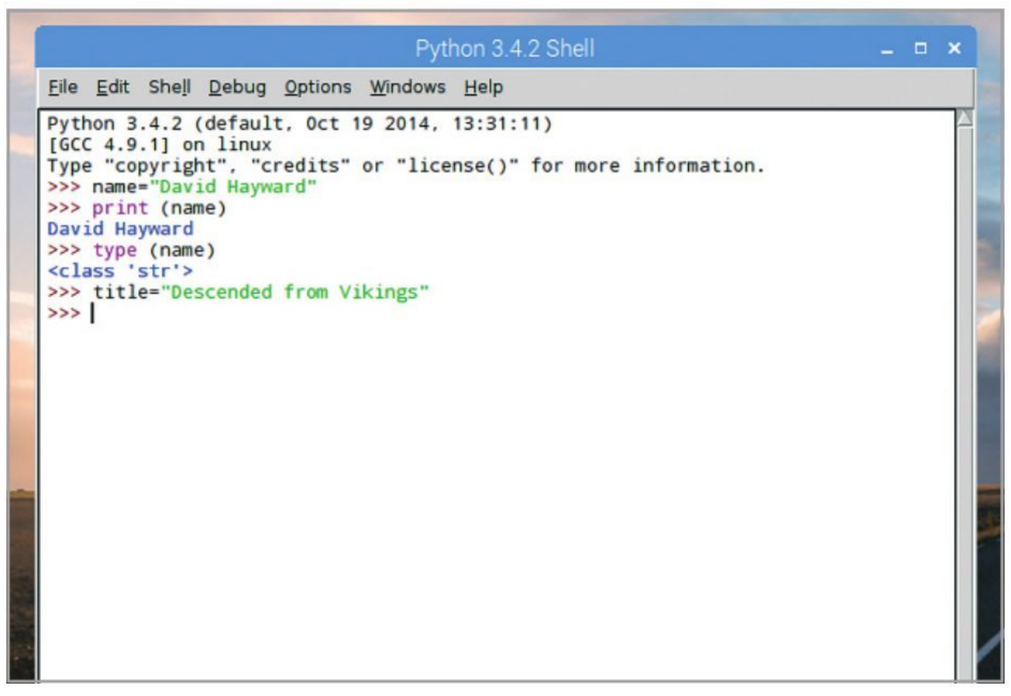
## VARIOUS VARIABLES

**You'll be working with the Python 3 IDLE Shell in this tutorial. If you haven't already, open Python 3 or close down the previous IDLE Shell to clear up any old code.**

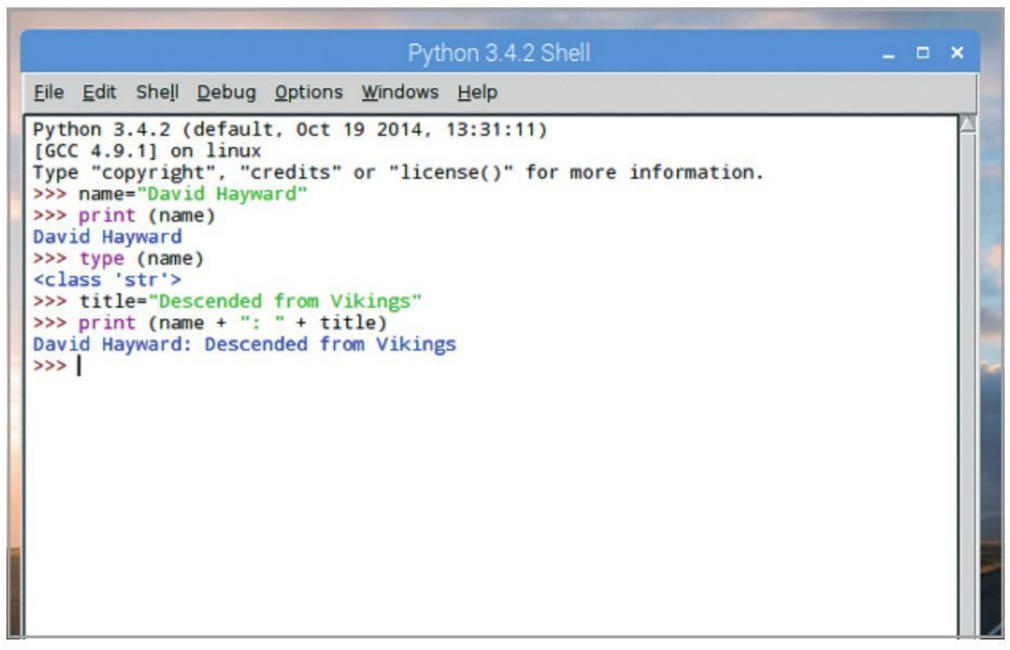
**STEP 1** In some programming languages you're required to use a dollar sign to denote a string, which is a variable made up of multiple characters, such as a name of a person. In Python this isn't necessary. For example, in the Shell enter: `name="David Hayward"` (or use your own name, unless you're also called David Hayward).




**STEP 2** You can check the type of variable in use by issuing the **type ()** command, placing the name of the variable inside the brackets. In our example, this would be: `type (name)`. Add a new string variable: `title="Descended from Vikings"`.




**STEP 3** You've seen previously that variables can be concatenated using the plus symbol between the variable names. In our example we can use: `print (name + ": " + title)`. The middle part between the quotations allows us to add a colon and a space, as variables are connected without spaces, so we need to add them manually.




**STEP 4** You can also combine variables within another variable. For example, to combine both name and title variables into a new variable we use:

```
character=name +
": " + title
```

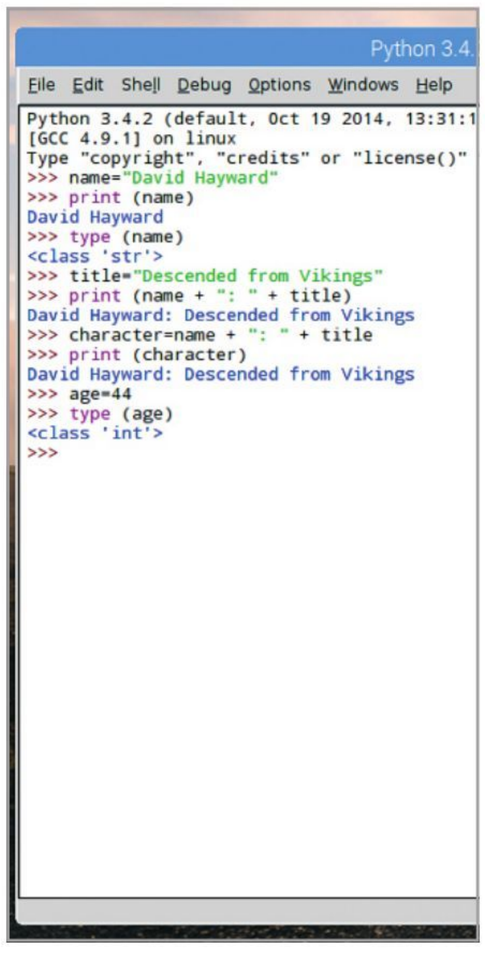
Then output the content of the new variable as:

```
print (character)
```

Numbers are stored as different variables:

```
age=44
Type (age)
```

Which are integers, as we know.



**STEP 5** However, you can't combine both strings and integer type variables in the same command, as you would a set of similar variables. You need to either turn one into the other or vice versa. When you do try to combine both, you get an error message:

```
print (name + age)
```

```
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> name="David Hayward"
>>> print (name)
David Hayward
>>> type (name)
<class 'str'>
>>> title="Descended from Vikings"
>>> print (name + ": " + title)
David Hayward: Descended from Vikings
>>> character=name + ": " + title
>>> print (character)
David Hayward: Descended from Vikings
>>> age=44
>>> type (age)
<class 'int'>
>>> print (name+age)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    print (name+age)
TypeError: Can't convert 'int' object to str implicitly
>>>
```

**STEP 6** This is a process known as TypeCasting. The Python code is:

```
print (character + " is " + str(age) + "
years old.")
```

or you can use:

```
print (character, "is", age, "years old.")
```

Notice again that in the last example, you don't need the spaces between the words in quotes as the commas treat each argument to print separately.

```
>>> print (name + age)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    print (name + age)
TypeError: Can't convert 'int' object to str implicitly
>>> print (character + " is " + str(age) + " years old.")
David Hayward: Descended from Vikings is 44 years old.
>>> print (character, "is", age, "years old.")
David Hayward: Descended from Vikings is 44 years old.
>>>
>>>
```

**STEP 7** Another example of TypeCasting is when you ask for input from the user, such as a name. for example, enter:

```
age= input ("How old are you? ")
```

All data stored from the Input command is stored as a string variable.

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> age= input ("How old are you? ")
How old are you? 44
>>> type(age)
<class 'str'>
>>>
```

**STEP 8** This presents a bit of a problem when you want to work with a number that's been inputted by the user, as age + 10 won't work due to being a string variable and an integer. Instead, you need to enter:

```
int(age) + 10
```

This will TypeCast the age string into an integer that can be worked with.

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> age= input ("How old are you? ")
How old are you? 44
>>> type(age)
<class 'str'>
>>> age + 10
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    age + 10
TypeError: Can't convert 'int' object to str implicitly
>>> int(age) + 10
54
>>>
```

**STEP 9** The use of TypeCasting is also important when dealing with floating point arithmetic; remember: numbers that have a decimal point in them. For example, enter:

```
shirt=19.99
```

Now enter `type(shirt)` and you'll see that Python has allocated the number as a 'float', because the value contains a decimal point.

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> shirt=19.99
>>> type(shirt)
<class 'float'>
>>>
```

**STEP 10** When combining integers and floats Python usually converts the integer to a float, but should the reverse ever be applied it's worth remembering that Python doesn't return the exact value. When converting a float to an integer, Python will always round down to the nearest integer, called truncating; in our case instead of 19.99 it becomes 19.

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> shirt=19.99
>>> type(shirt)
<class 'float'>
>>> int(shirt)
19
>>>
```

# User Input

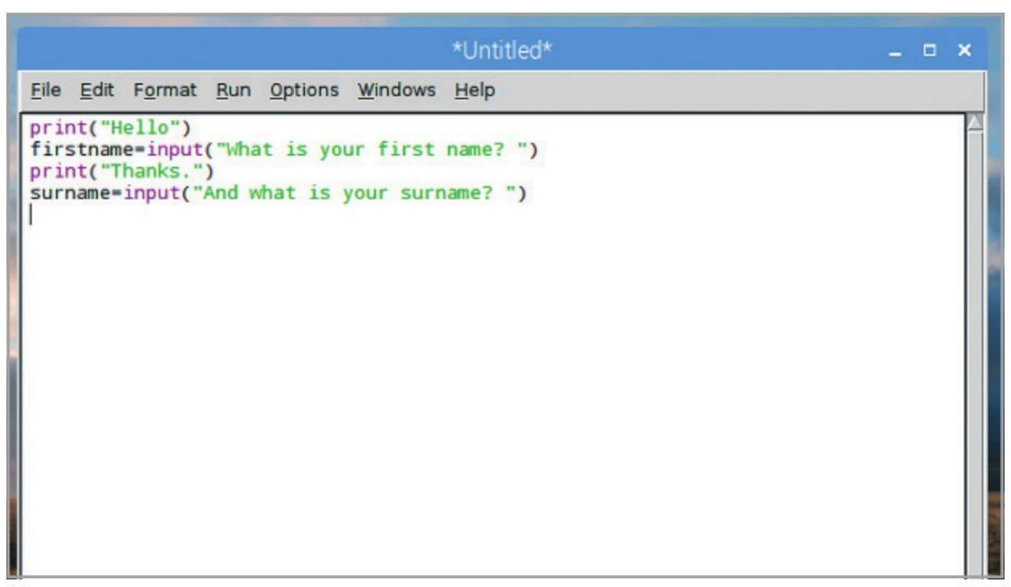
We've seen some basic user interaction with the code from a few of the examples earlier, so now would be a good time to focus solely on how you would get information from the user then store and present it.

## USER FRIENDLY

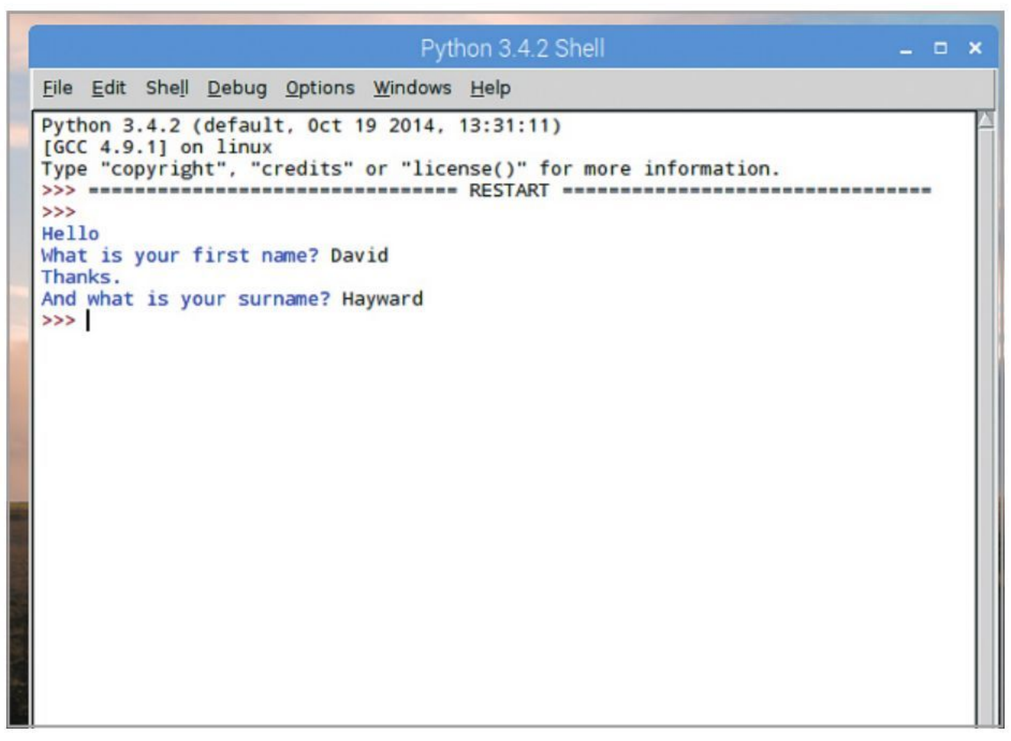
**The type of input you want from the user will depend greatly on the type of program you're coding. For example, a game may ask for a character's name, whereas a database can ask for personal details.**

**STEP 1** If it's not already, open the Python 3 IDLE Shell, and start a New File in the Editor. Let's begin with something really simple, enter:

```
print("Hello")
firstname=input("What is your first name? ")
print("Thanks.")
surname=input("And what is your surname? ")
```

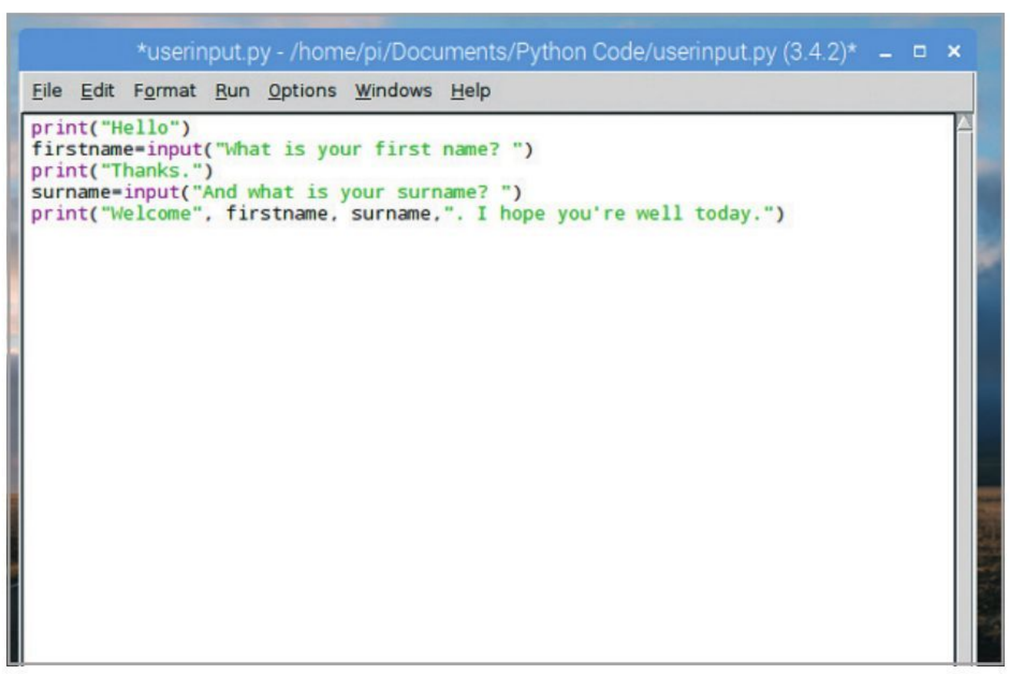



**STEP 2** Save and execute the code, and as you already no doubt suspected, in the IDLE Shell the program will ask for your first name, storing it as the variable firstname, followed by your surname; also stored in its own variable (surname).



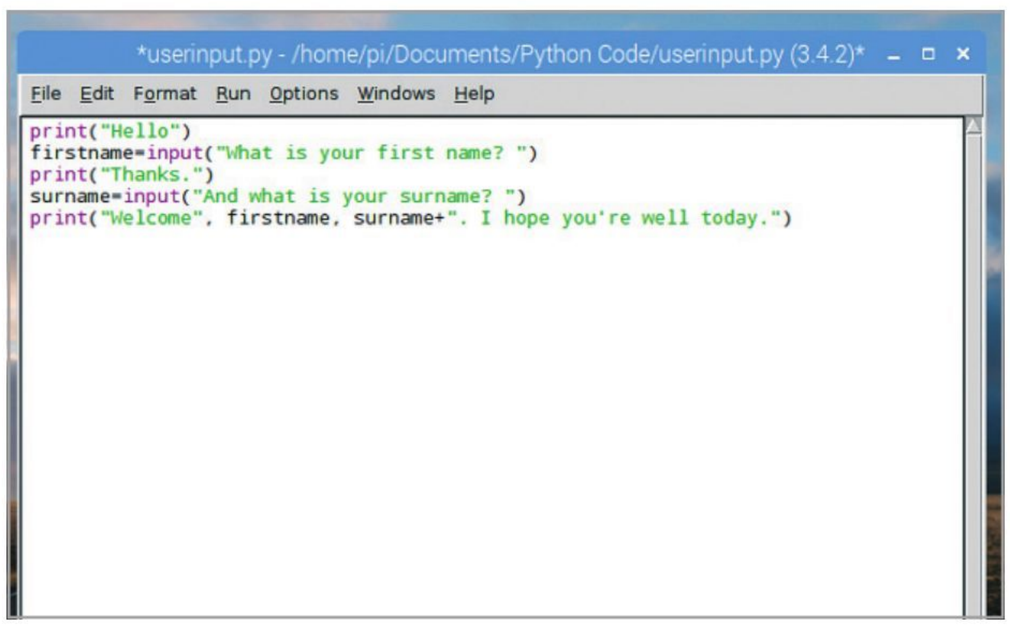

**STEP 3** Now that we have the user's name stored in a couple of variables we can call them up whenever we want:

```
print("Welcome", firstname, surname, ". I hope you're well today.")
```




**STEP 4** Run the code and you can see a slight issue, the full stop after the surname follows a blank space. To eliminate that we can add a plus sign instead of the comma in the code:

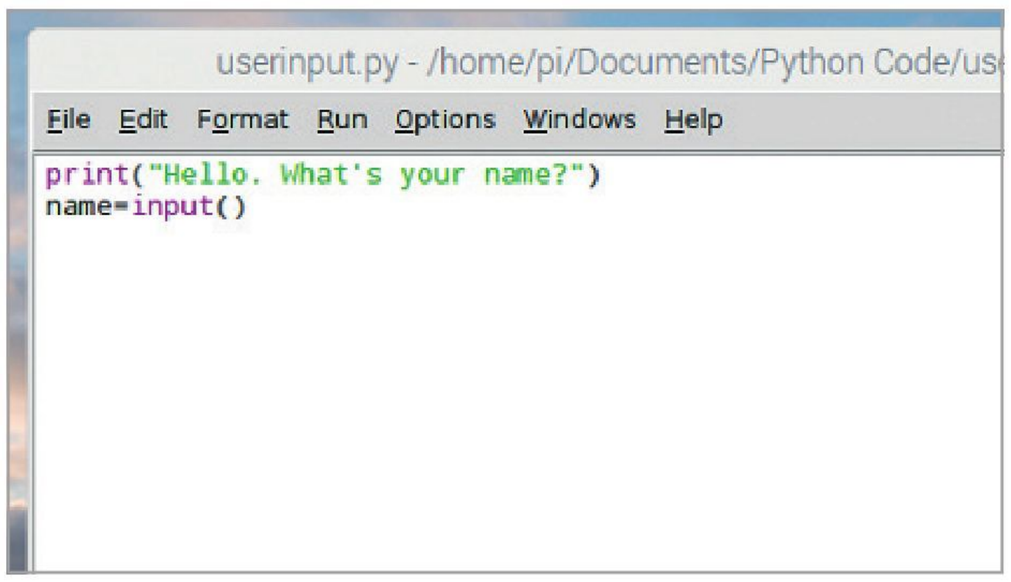
```
print("Welcome", firstname, surname+". I hope you're well today.")
```



## STEP 5

You don't always have to include quoted text within the input command. For example, you can ask the user their name, and have the input in the line below:

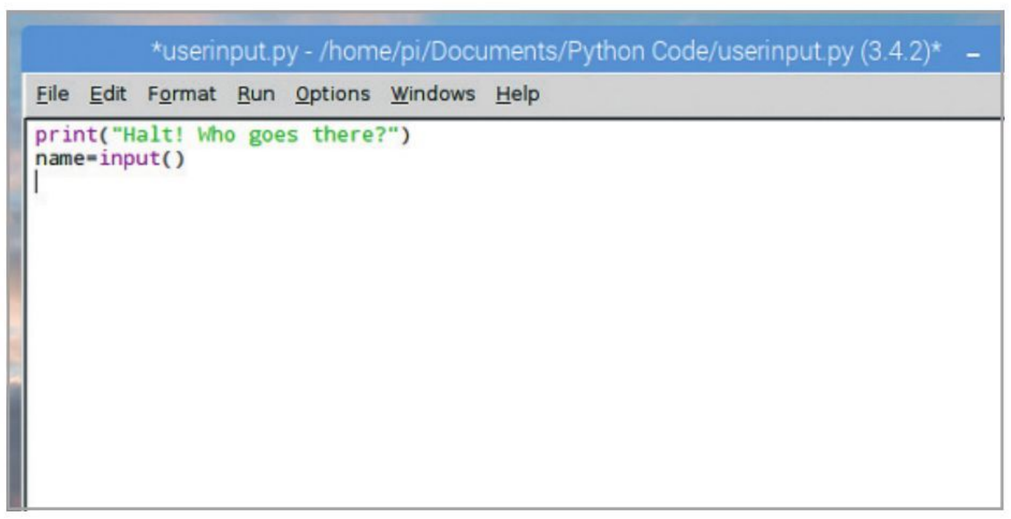
```
print("Hello. What's your name?")
name=input()
```




## STEP 6

The code from the previous step is often regarded as being a little neater than having a lengthy amount of text in the input command, but it's not a rule that's set in stone, so do as you like in these situations. Expanding on the code, try this:

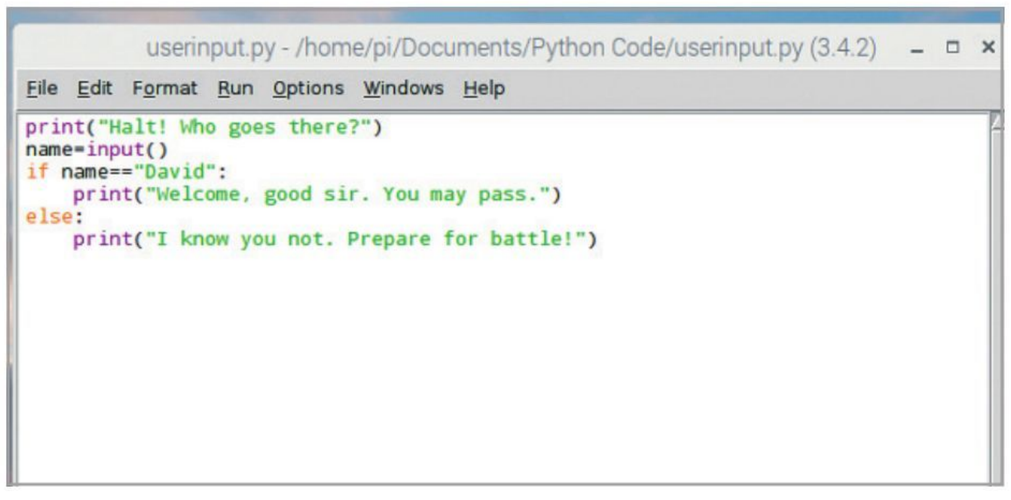
```
print("Halt! Who goes there?")
name=input()
```




## STEP 7

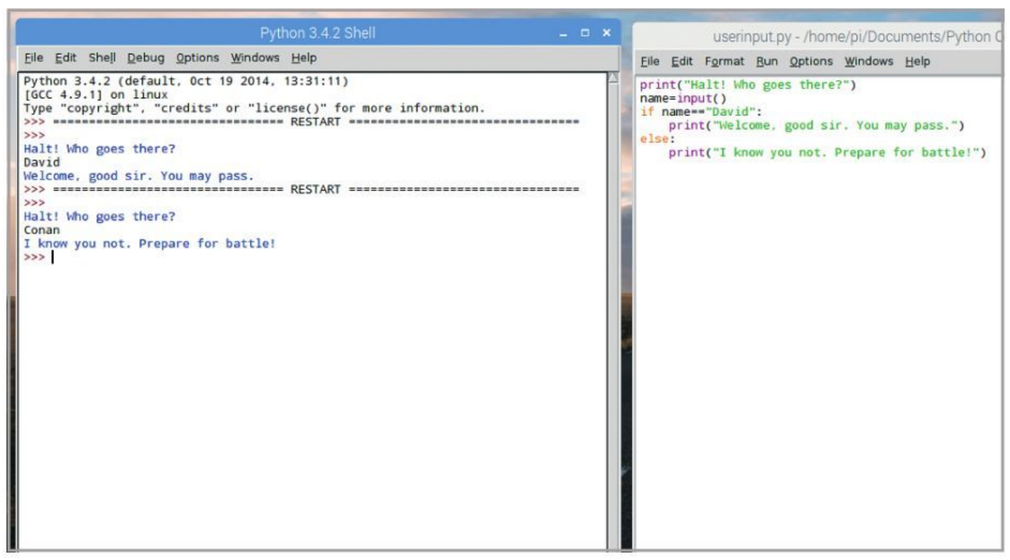
It's a good start to a text adventure game, perhaps? Now you can expand on it and use the raw input from the user to flesh out the game a little:

```
if name=="David":
  print("Welcome, good sir. You may pass.")
else:
  print("I know you not. Prepare for battle!")
```




## STEP 8

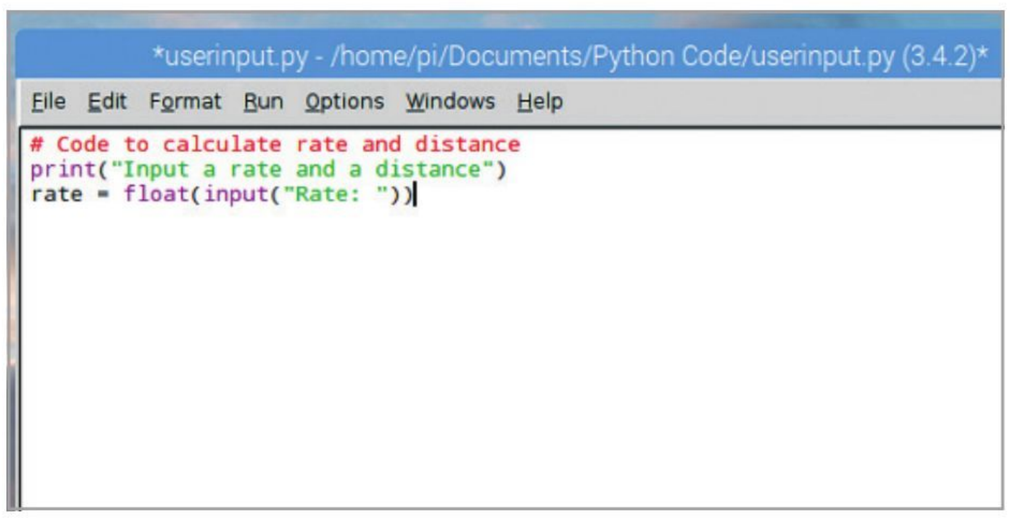
What you've created here is a condition, which we will cover soon. In short, we're using the input from the user and measuring it against a condition. So, if the user enters David as their name, the guard will allow them to pass unhindered. Else, if they enter a name other than David, the guard challenges them to a fight.




## STEP 9

Just as you learned previously, any input from a user is automatically a string, so you need to apply a TypeCast in order to turn it into something else. This creates some interesting additions to the input command. For example:

```
# Code to calculate rate and distance
print("Input a rate and a distance")
rate = float(input("Rate: "))
```

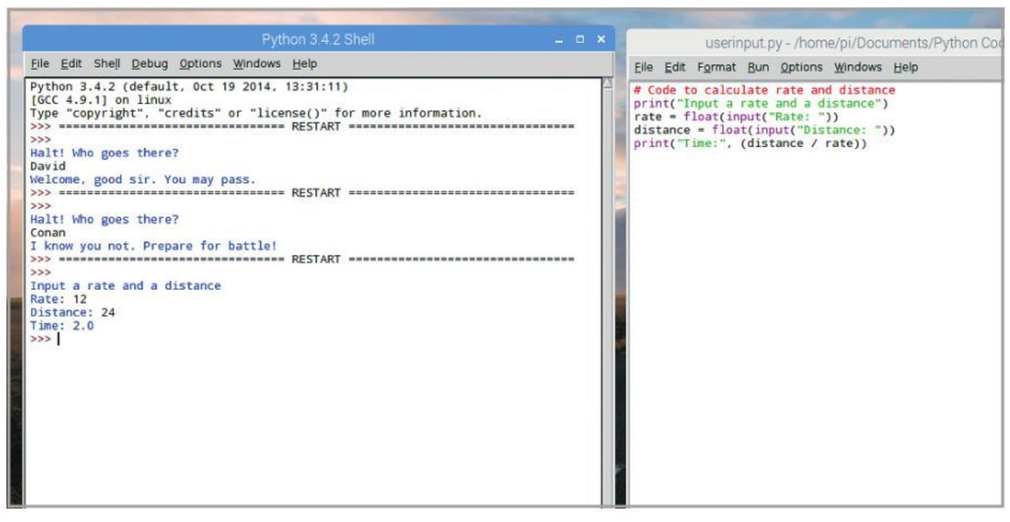



## STEP 10

To finalise the rate and distance code, we can add:

```
distance = float(input("Distance: "))
print("Time:", (distance / rate))
```

Save and execute the code and enter some numbers. Using the float(input element, we've told Python that anything entered is a floating point number rather than a string.



# Creating Functions

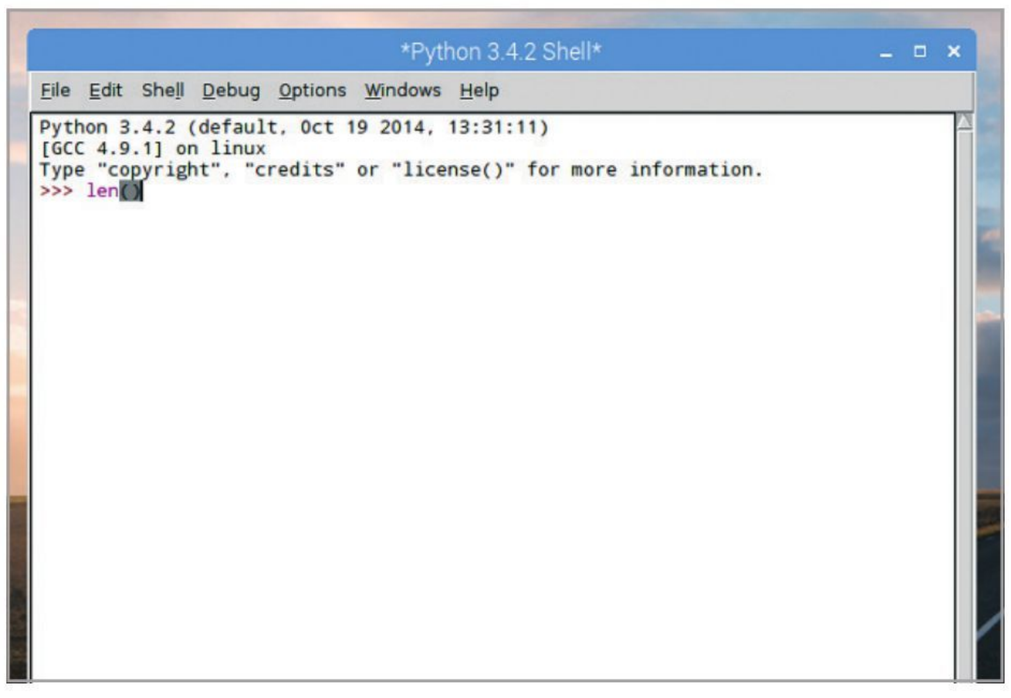
Now that you've mastered the use of variables and user input, the next step is to tackle functions. You've already used a few functions, such as the print command but Python enables you to define your own functions.

## FUNKY FUNCTIONS

**A function is a command that you enter into Python to do something. It's a little piece of self-contained code that takes data, works on it and then returns the result.**

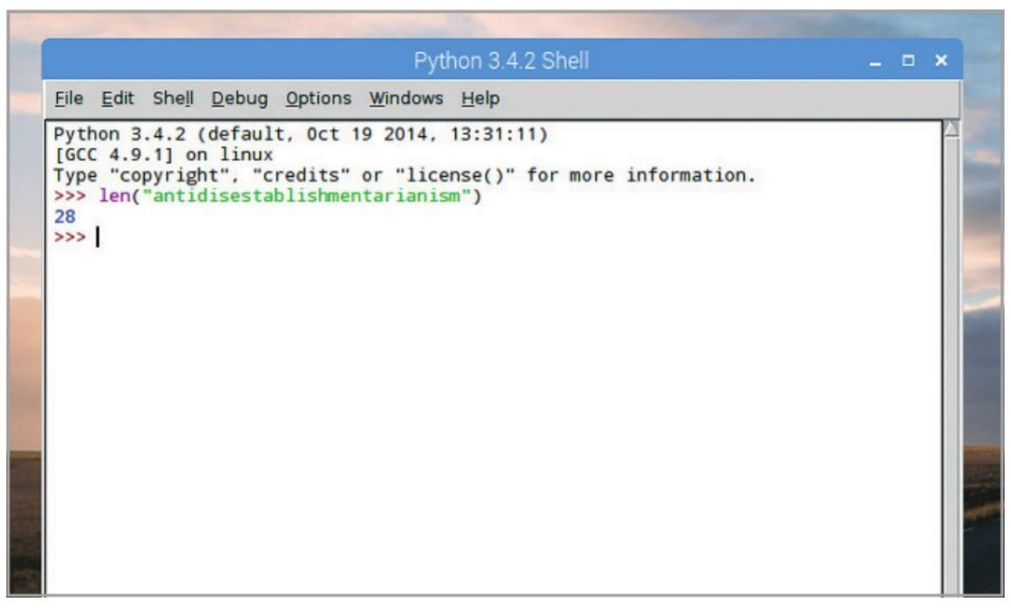
### STEP 1

It's not just data that a function works on. They can do all manner of useful things in Python, such as sort data, change items from one format to another and check the length or type of items. Basically, a function is a short word that's followed by brackets. For example, **len()**, **list()** or **type()**.




### STEP 2

A function takes data, usually a variable, works on it depending on what the function is programmed to do and returns the end value. The data being worked on goes inside the brackets, so if you wanted to know how many letters are in the word antidisestablishmentarianism, then you'd enter: `len("antidisestablishmentarianism")` and the number 28 would return.

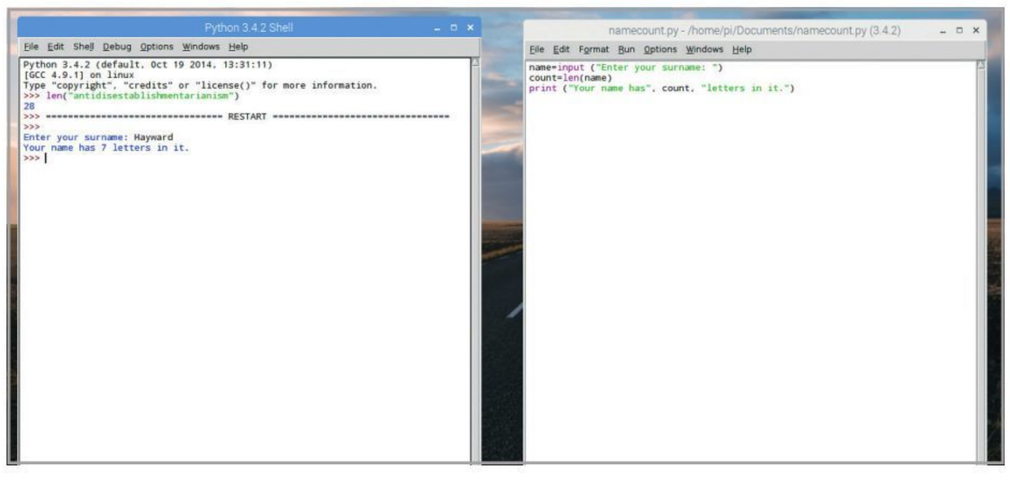



### STEP 3

You can pass variables through functions in much the same manner. Let's assume you want the number of letters in a person's surname, you could use the following code (enter the text editor for this example):

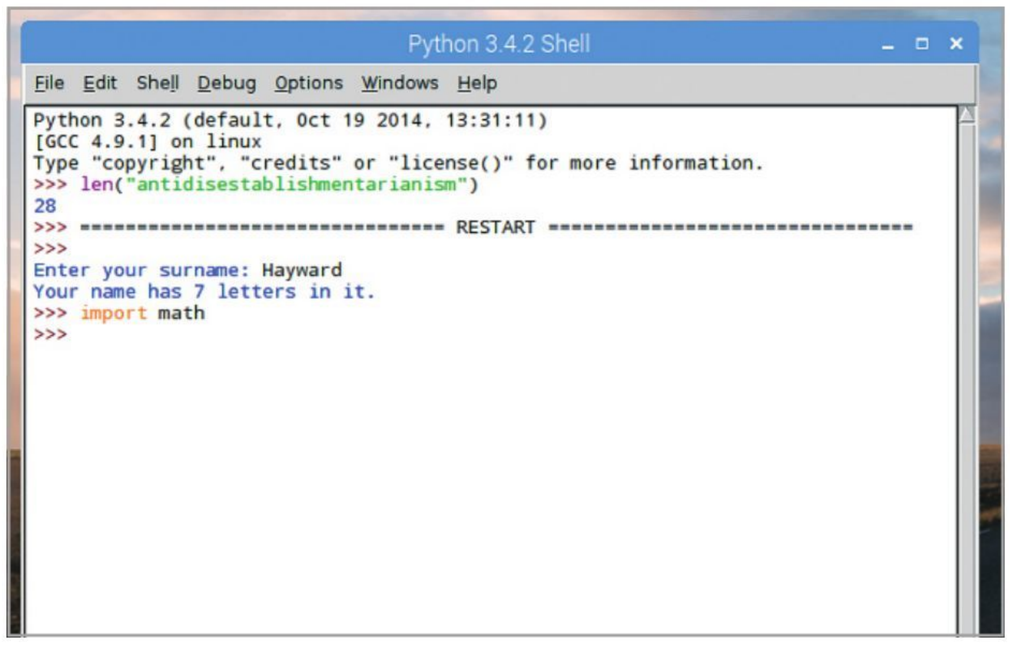
```
name=input ("Enter your surname: ")
count=len(name)
print ("Your surname has", count,
"letters in it.")
```

Press F5 and save the code to execute it.




### STEP 4

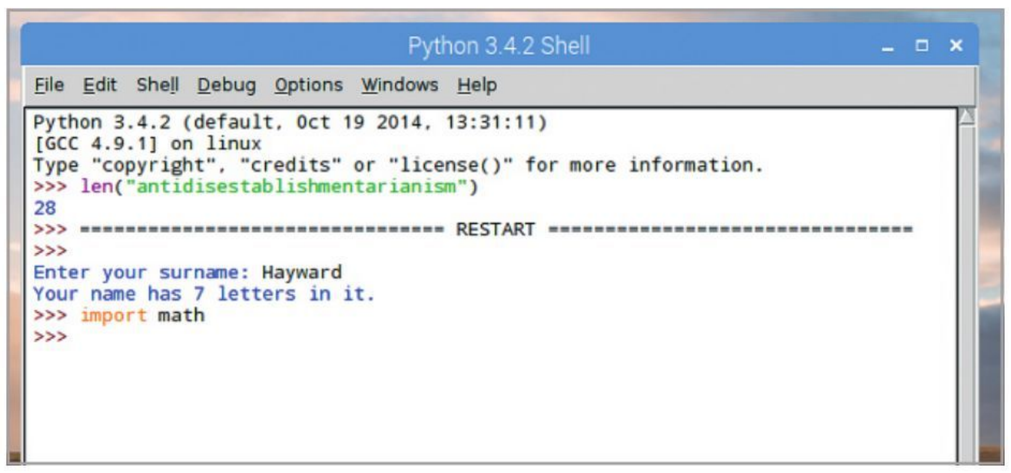
Python has tens of functions built into it, far too many to get into in the limited space available here. However, to view the list of built-in functions available to Python 3, navigate to www.docs.python.org/3/library/functions.html. These are the predefined functions, but since users have created many more, they're not the only ones available.



**STEP 5** Additional functions can be added to Python through modules. Python has a vast range of modules available that can cover numerous programming duties. They add functions and can be imported as and when required. For example, to use advanced Mathematics functions enter:

```
import math
```

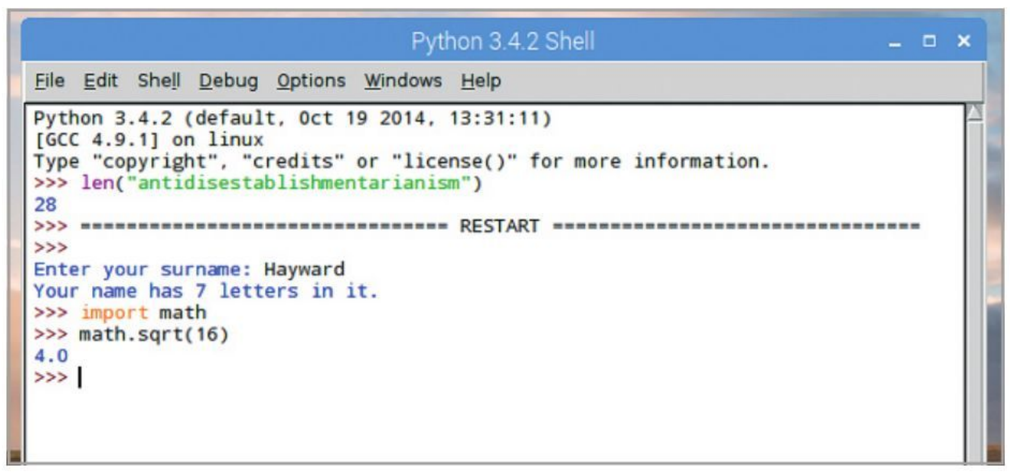
Once entered, you have access to all the Math module functions.




**STEP 6** To use a function from a module enter the name of the module followed by a full stop, then the name of the function. For instance, using the Math module, since you've just imported it into Python, you can utilise the square root function. To do so, enter:

```
math.sqrt(16)
```

You can see that the code is presented as module.function(data).



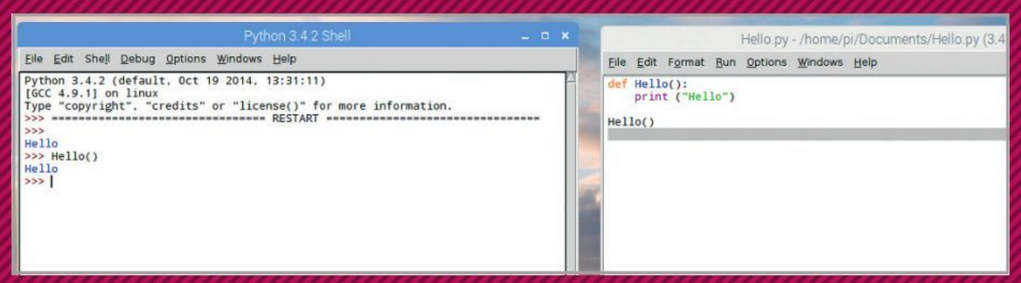

## FORGING FUNCTIONS

**There are many different functions you can import created by other Python programmers and you will undoubtedly come across some excellent examples in the future; you can also create your own with the def command.**

**STEP 1** Choose File > New File to enter the editor, let's create a function called Hello, that greets a user. Enter:

```
def Hello():
   print ("Hello")

Hello()
```

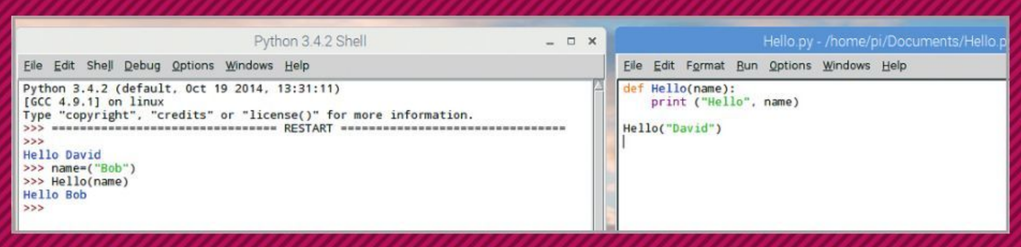
Press F5 to save and run the script. You can see Hello in the Shell, type in Hello() and it returns the new function.




**STEP 2** Let's now expand the function to accept a variable, the user's name for example. Edit your script to read:

```
def Hello(name):
   print ("Hello", name)

Hello("David")
```

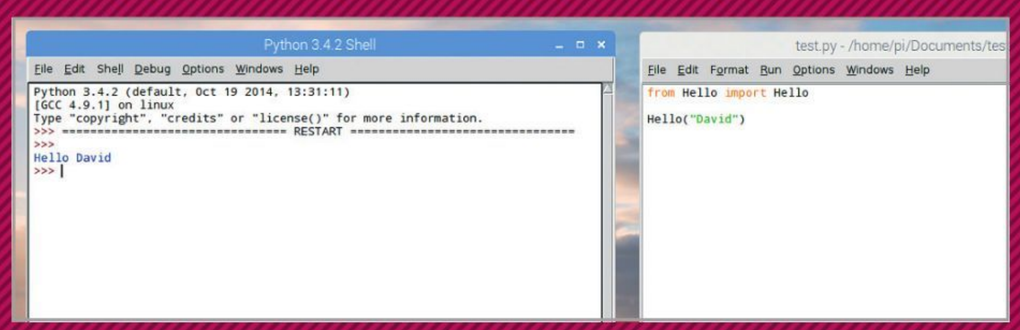
This will now accept the variable name, otherwise it prints Hello David. In the Shell, enter: name=("Bob"), then: Hello(name). Your function can now pass variables through it.




**STEP 3** To modify it further, delete the Hello("David") line, the last line in the script and press Ctrl+S to save the new script. Close the Editor and create a new file (File > New File). Enter the following:

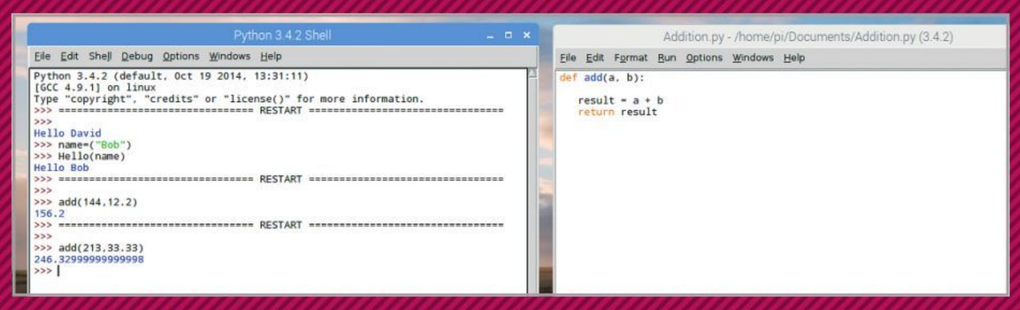
```
from Hello import Hello

Hello("David")
```

Press F5 to save and execute the code.




**STEP 4** What you've just done is import the Hello function from the saved Hello.py program and then used it to say hello to David. This is how modules and functions work: you import the module then use the function. Try this one, and modify it for extra credit:

```
def add(a, b):
   result = a + b
   return result
```



# Conditions and Loops

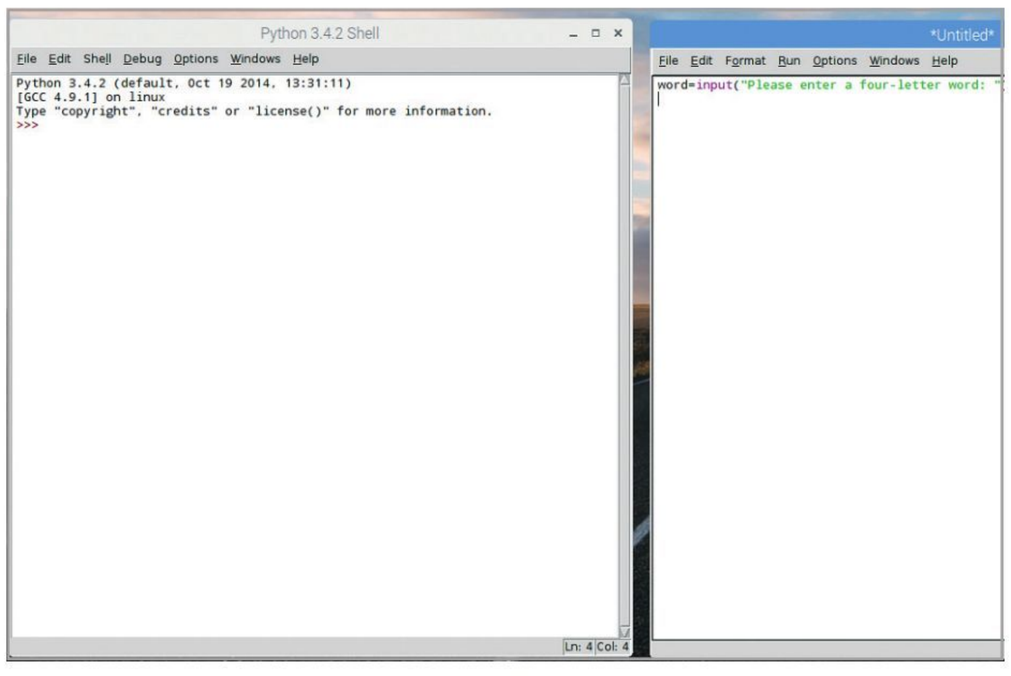
Conditions and loops are what makes a program interesting; they can be simple or rather complex. How you use them depends greatly on what the program is trying to achieve; they could be the number of lives left in a game or just displaying a countdown.

## TRUE CONDITIONS

**Keeping conditions simple to begin with makes learning to program a more enjoyable experience. Let's start then by checking if something is TRUE, then doing something else if it isn't.**

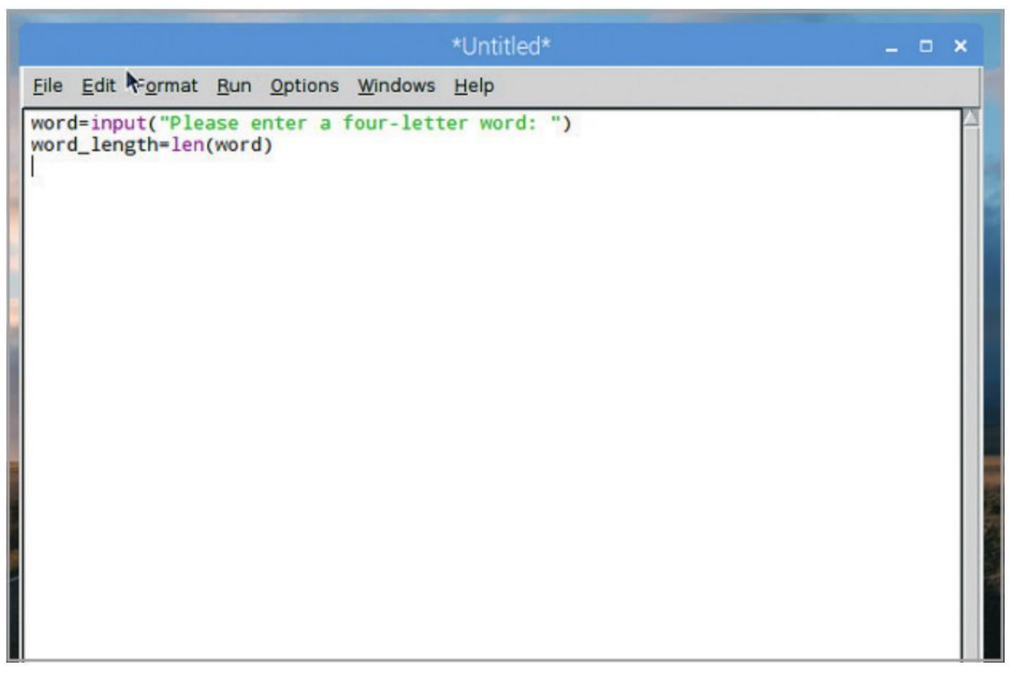
**STEP 1** Let's create a new Python program that will ask the user to input a word, then check it to see if it's a four-letter word or not. Start with File > New File, and begin with the input variable:

```
word=input("Please enter a four-letter word: ")
```




**STEP 2** Now we can create a new variable, then use the len function and pass the word variable through it to get the total number of letters the user has just entered:

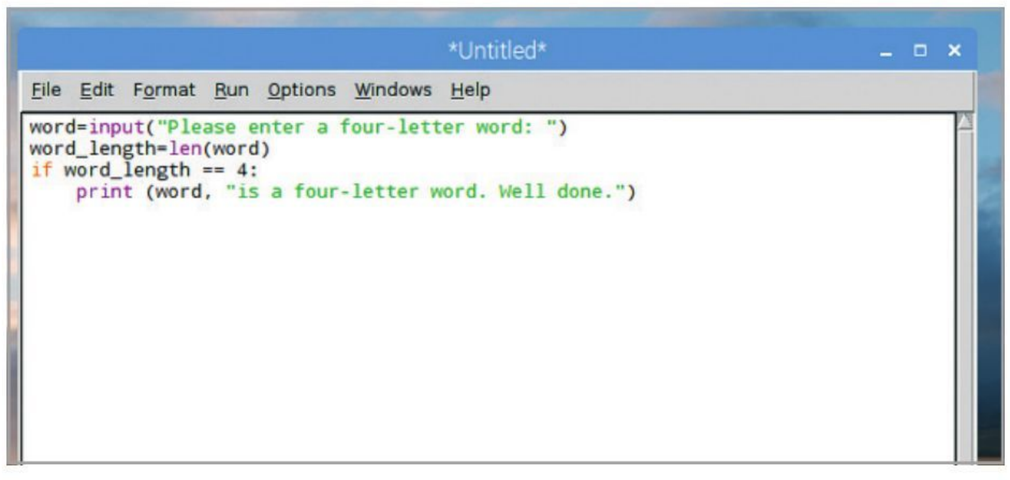
```
word=input("Please enter a four-letter word: ")
word_length=len(word)
```




**STEP 3** Now you can use an if statement to check if the word_length variable is equal to four and print a friendly conformation if it applies to the rule:

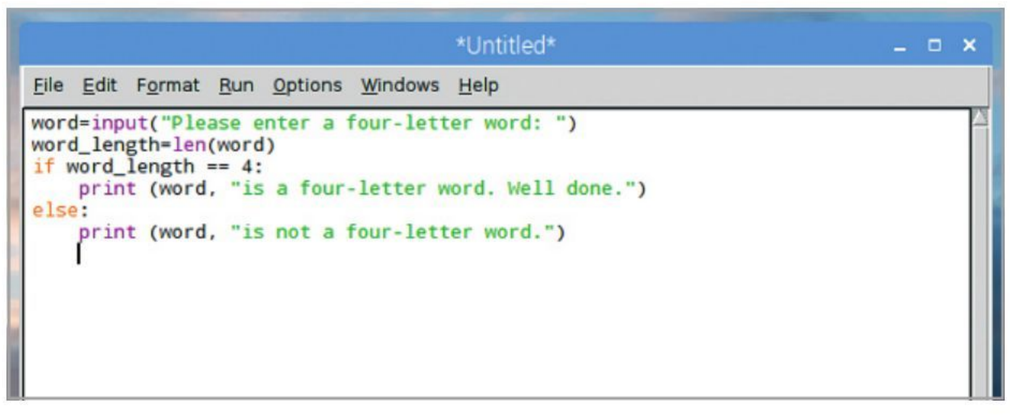
```
word=input("Please enter a four-letter word: ")
word_length=len(word)
if word_length == 4:
   print (word, "is a four-letter word. Well done.")
```

The double equal sign (==) means check if something is equal to something else.

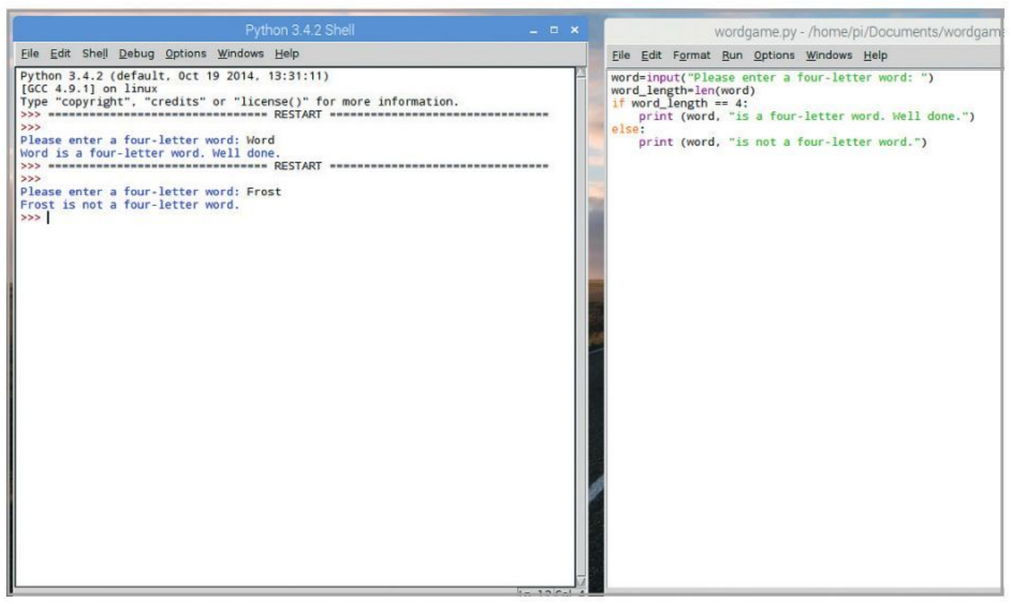



**STEP 4** The colon at the end of IF tells Python that if this statement is true do everything after the colon that's indented. Next, move the cursor back to the beginning of the Editor:

```
word=input("Please enter a four-letter word: ")
word_length=len(word)
if word_length == 4:
  print (word, "is a four-letter word.
Well done.")
else:
   print (word, "is not a four-letter word.")
```

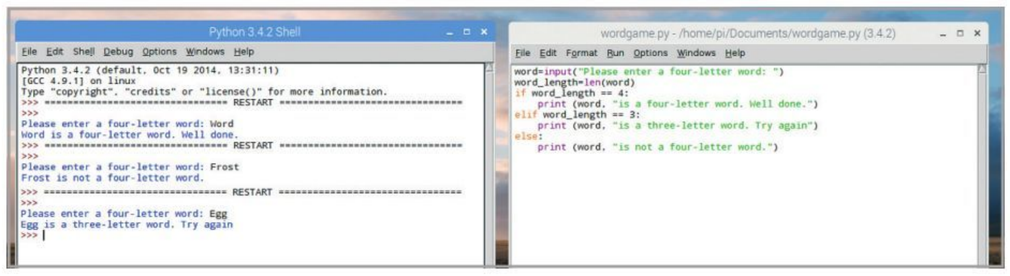


**STEP 5** Press F5 and save the code to execute it. Enter a four-letter word in the Shell to begin with, you should have the returned message that it's the word is four letters. Now press F5 again and rerun the program but this time enter a five-letter word. The Shell will display that it's not a four-letter word.




**STEP 6** Now expand the code to include another conditions. Eventually, it could become quite complex. We've added a condition for three-letter words:

```
word=input("Please enter a four-letter word: ")
word_length=len(word)
if word_length == 4:
   print (word, "is a four-letter word.
Well done.")
elif word_length == 3:
   print (word, "is a three-letter word. Try again.")
else:
   print (word, "is not a four-letter word.")
```

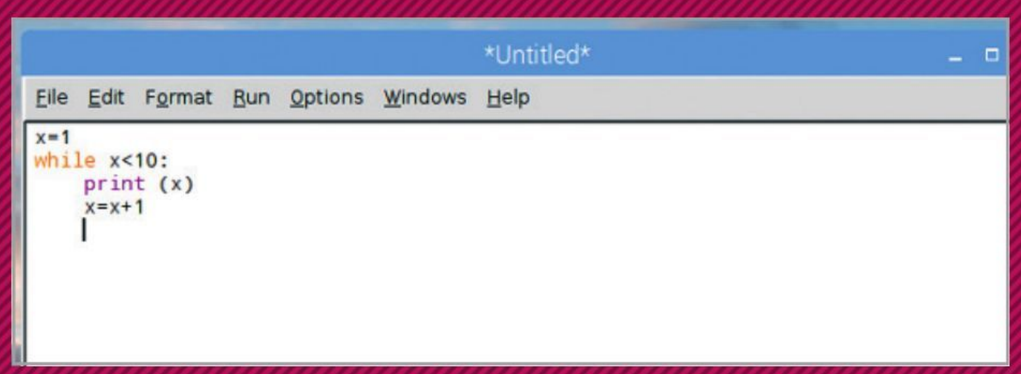



## LOOPS

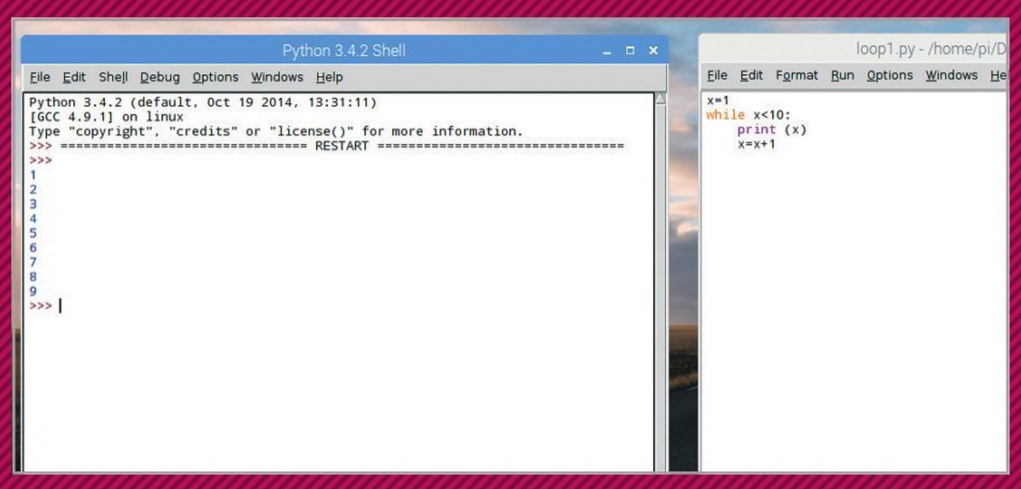
**A loop looks quite similar to a condition but they are somewhat different in their operation. A loop will run through the same block of code a number of times, usually with the support of a condition.**

**STEP 1** Let's start with a simple While statement. Like IF, this will check to see if something is TRUE, then run the indented code:

```
x = 1
while x < 10:
   print (x)
   x = x + 1
```

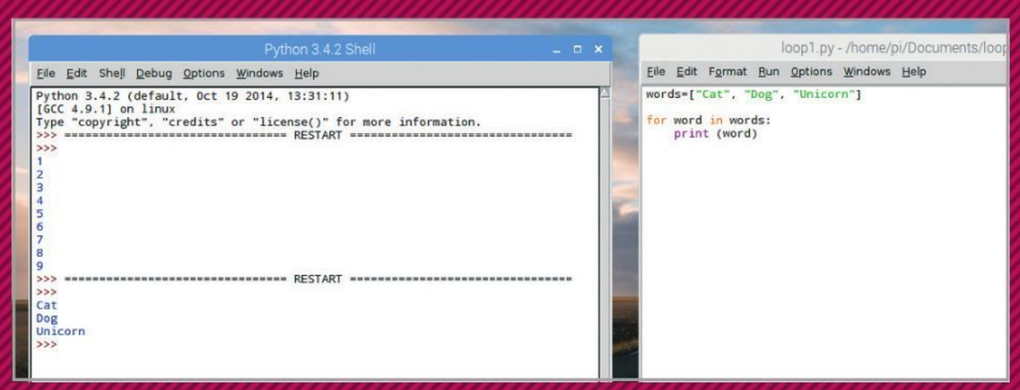



**STEP 2** The difference between if and while is when while gets to the end of the indented code, it goes back and checks the statement is still true. In our example x is less than 10. With each loop it prints the current value of x, then adds one to that value. When x does eventually equal 10 it stops.




**STEP 3** The For loop is another example. For is used to loop over a range of data, usually a list stored as variables inside square brackets. For example:

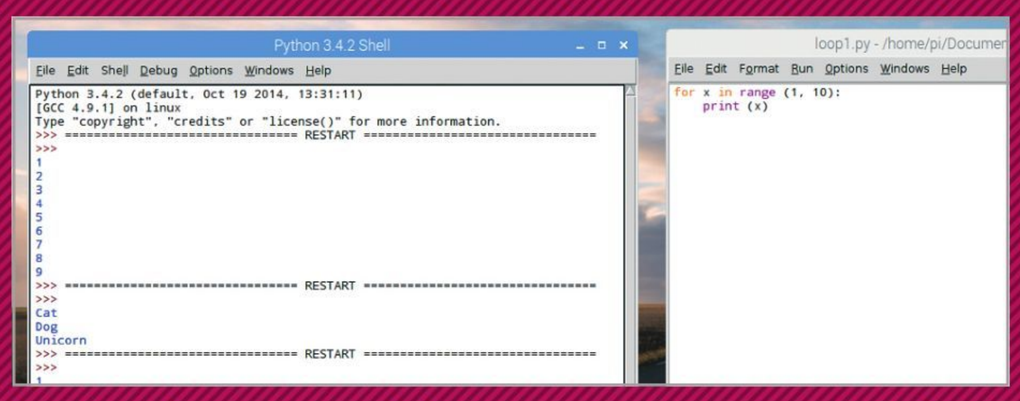
```
words=["Cat", "Dog", "Unicorn"]

for word in words:
   print (word)
```




**STEP 4** The For loop can also be used in the countdown example by using the range function:

```
for x in range (1, 10):
   print (x)
```

The x=x+1 part isn't needed here because the range function creates a list between the first and last numbers used.



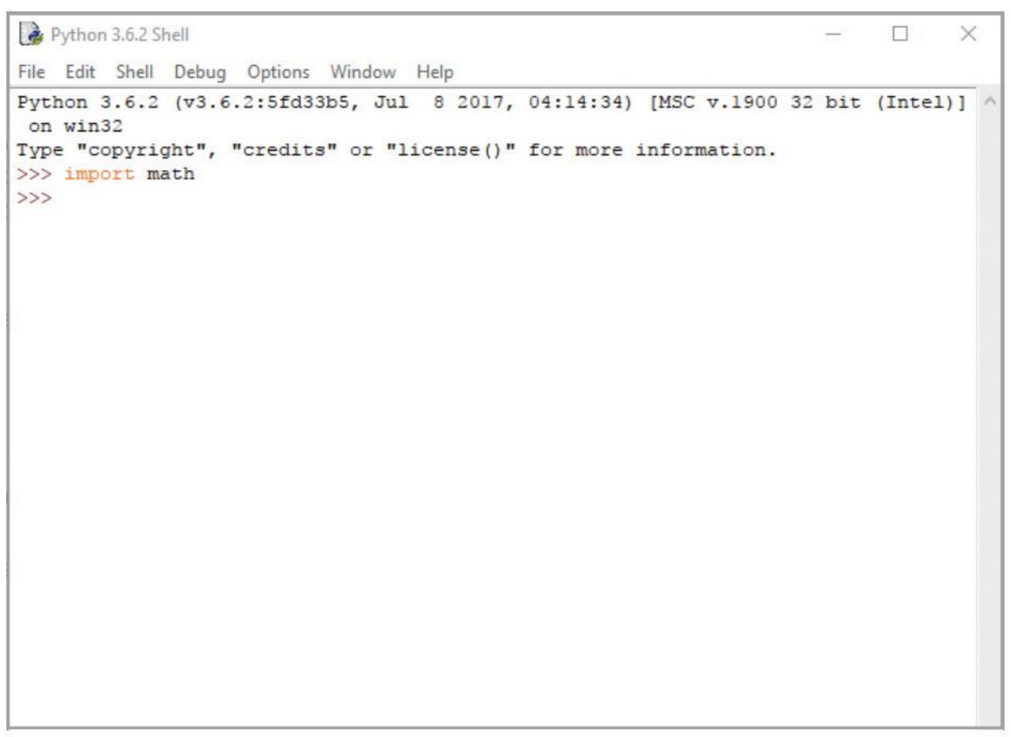
# Python Modules

We've mentioned modules previously, (the Math module) but as modules are such a large part of getting the most from Python, it's worth dedicating a little more time to them. In this instance we're using the Windows version of Python 3.

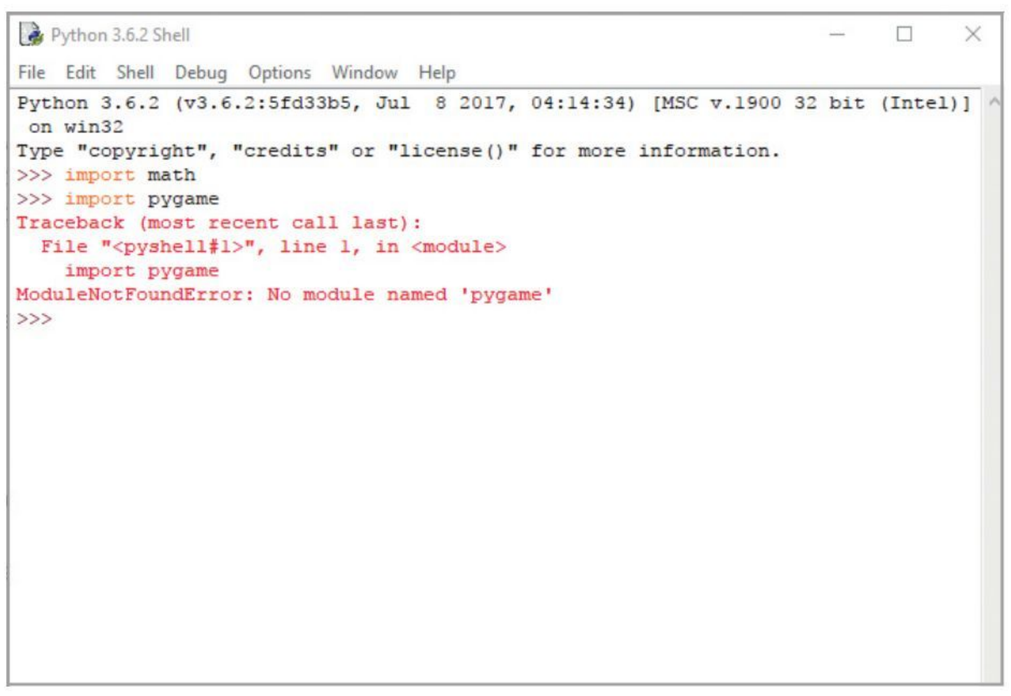
## MASTERING MODULES

**Think of modules as an extension that's imported into your Python code to enhance and extend its capabilities. There are countless modules available and as we've seen, you can even make your own.**

**STEP 1** Although good, the built-in functions within Python are limited. The use of modules, however, allows us to make more sophisticated programs. As you are aware, modules are Python scripts that are imported, such as import math.

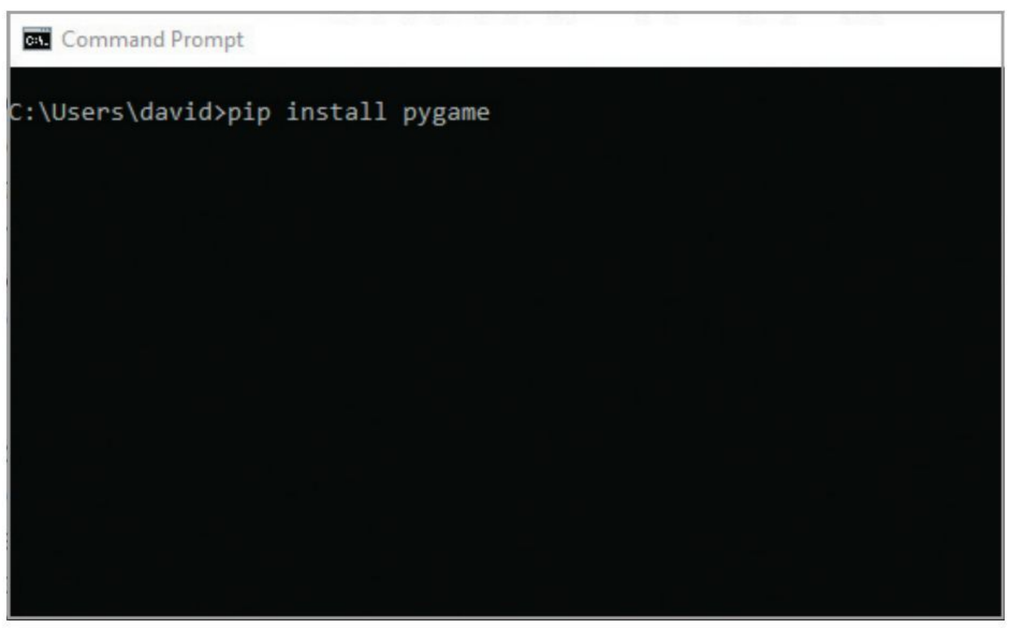



**STEP 2** Some modules, especially on the Raspberry Pi, are included by default, the Math module being a prime example. Sadly, other modules aren't always available. A good example on non-Pi platforms is the Pygame module, which contains many functions to help create games. Try: **import pygame**.

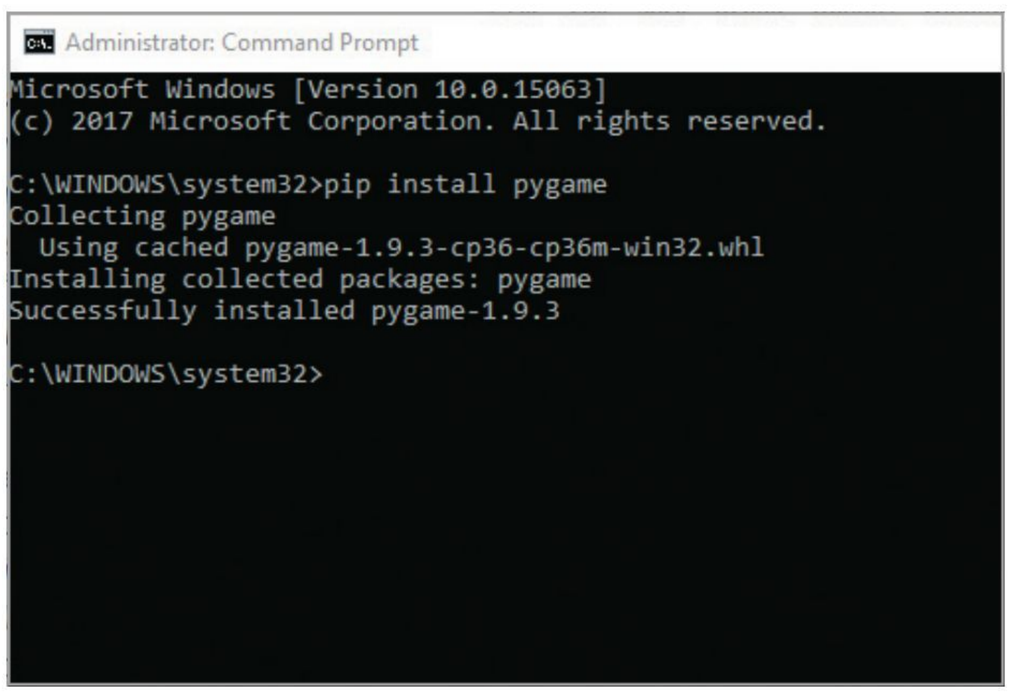



**STEP 3** The result is an error in the IDLE Shell, as the Pygame module isn't recognised or installed in Python. To install a module we can use PIP (Pip Installs Packages). Close down the IDLE Shell and drop into a command prompt or Terminal session. At an elevated admin command prompt, enter:

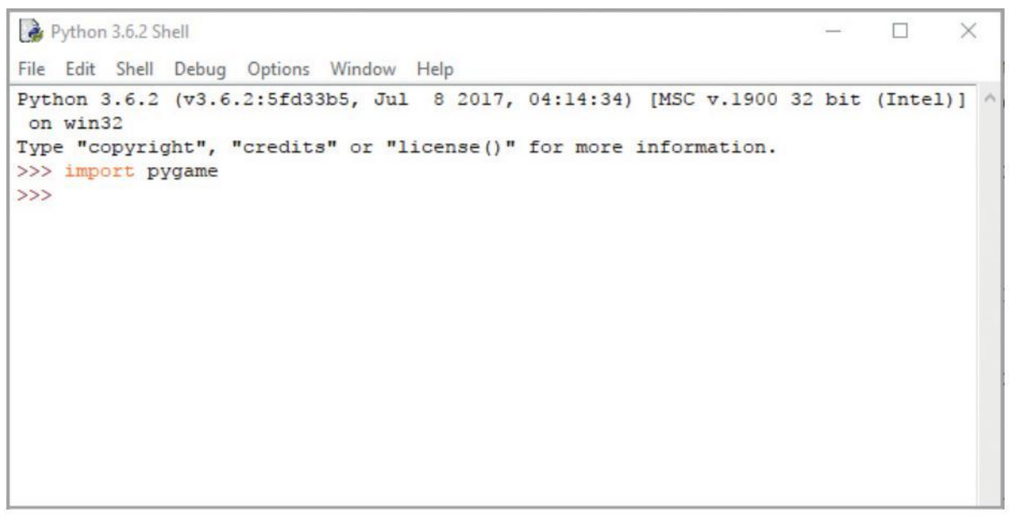
```
pip install pygame
```




**STEP 4** The PIP installation requires an elevated status due it installing components at different locations. Windows users can search for CMD via the Start button and right-click the result then click Run as Administrator. Linux and Mac users can use the Sudo command, with sudo pip install package.



**STEP 5** Close the command prompt or Terminal and relaunch the IDLE Shell. When you now enter: `import pygame`, the module will be imported into the code without any problems. You'll find that most code downloaded or copied from the internet will contain a module, mainstream of unique, these are usually the source of errors in execution due to them being missing.




**STEP 6** The modules contain the extra code needed to achieve a certain result within your own code, as we've previously experimented with. For example:

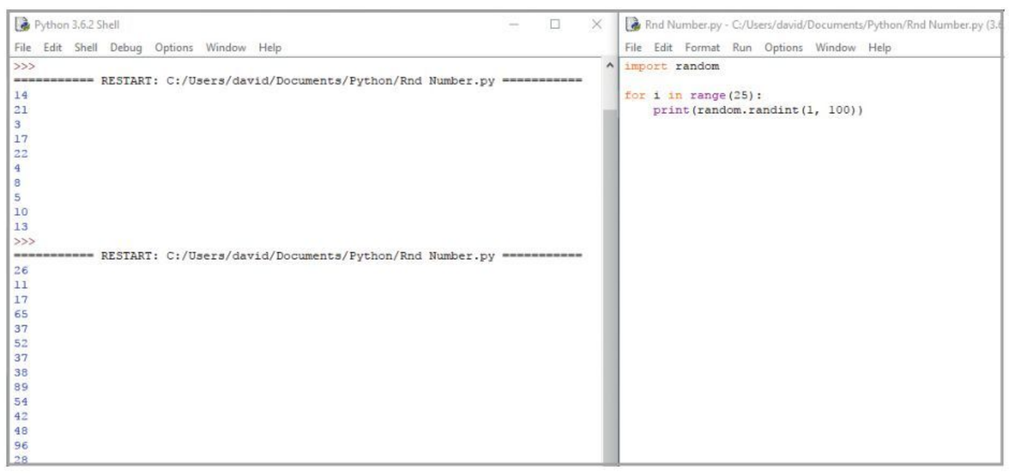
```
import random
```

Brings in the code from the Random Number Generator module. You can then use this module to create something like:

```
for i in range(10):
  print(random.randint(1, 25))
```

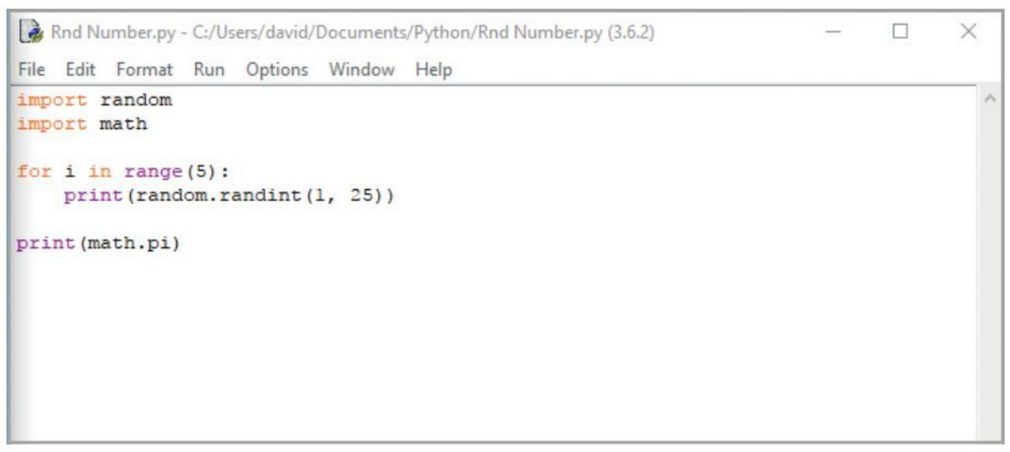

**STEP 7** This code, when saved and executed, will display ten random numbers from 1 to 25. You can play around with the code to display more or less, and from a great or lesser range. For example:

```
import random

for i in range(25):
  print(random.randint(1, 100))
```



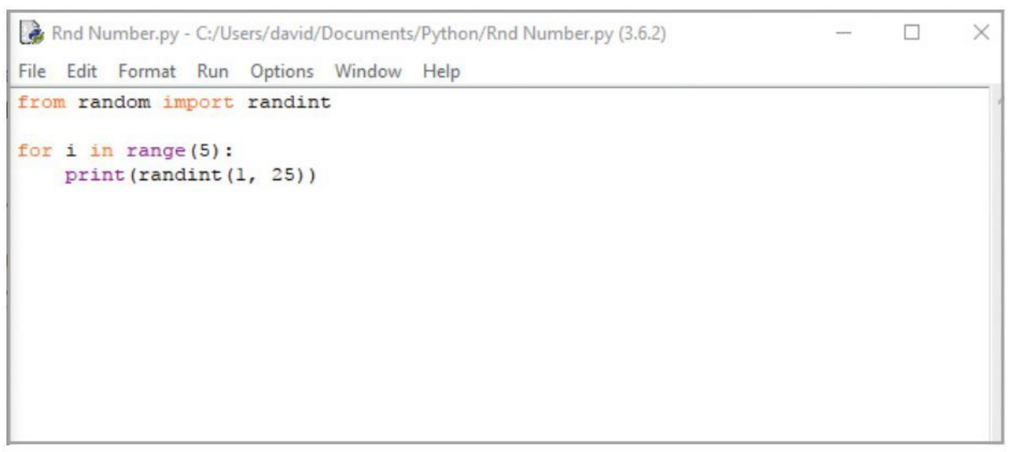

**STEP 8** Multiple modules can be imported within your code. To extend our example, use:

```
import random
import math

for I in range(5):
  print(random.randint(1, 25))

print(math.pi)
```




**STEP 9** The result is a string of random numbers followed by the value of Pi as pulled from the Math module using the print(math.pi) function. You can also pull in certain functions from a module by using the from and import commands, such as:

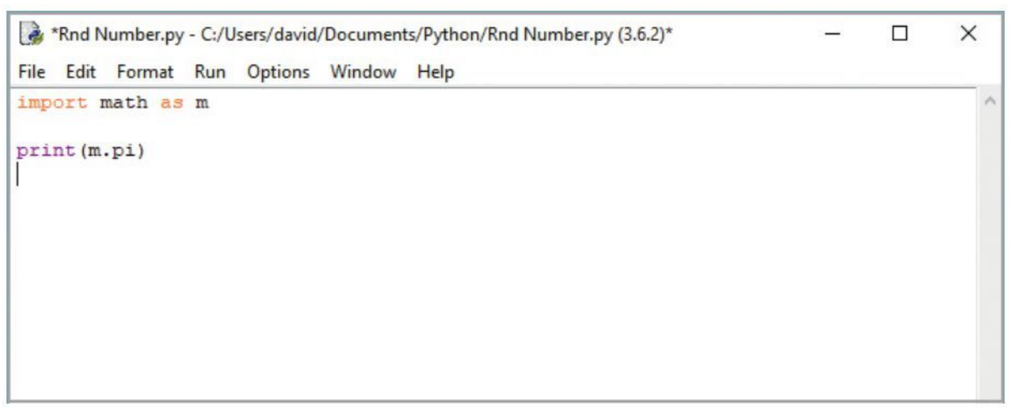
```
from random import randint

for i in range(5):
  print(randint(1, 25))
```




**STEP 10** This helps create a more streamlined approach to programming. You can also use Import module*, which will import everything defined within the named module. However, it's often regarded as a waste of resources but it works nonetheless. Finally, modules can be imported as aliases:

```
import math as m

print(m.pi)
```

Of course, adding comments helps to tell others what's going on.



# Python Errors

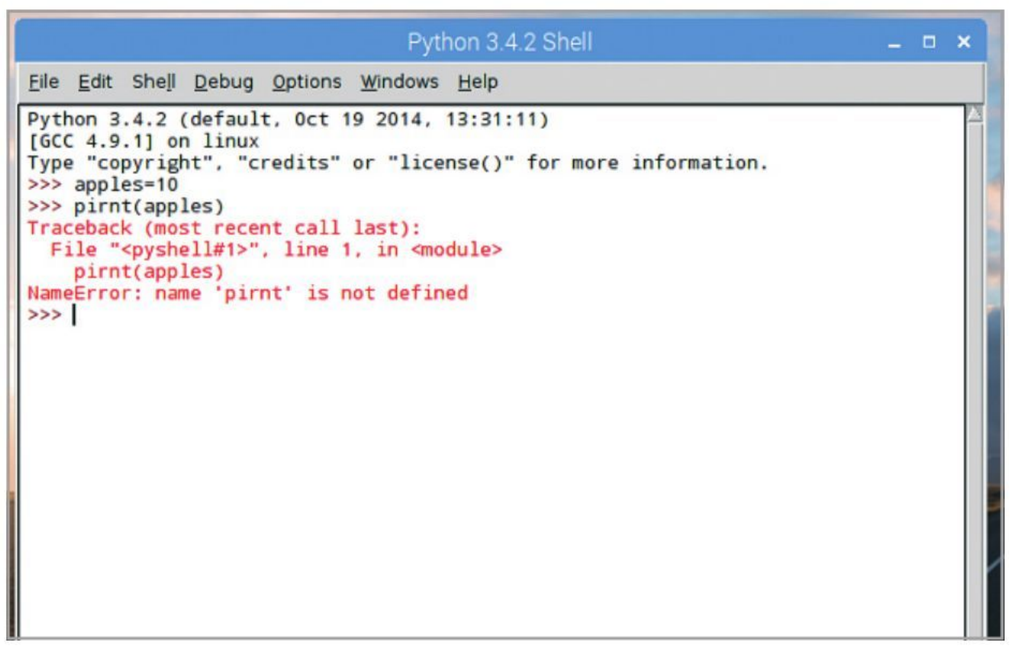
It goes without saying that you'll eventually come across an error in your code, where Python declares it's not able to continue due to something being missed out, wrong or simply unknown. Being able to identify these errors makes for a good programmer.

## DEBUGGING

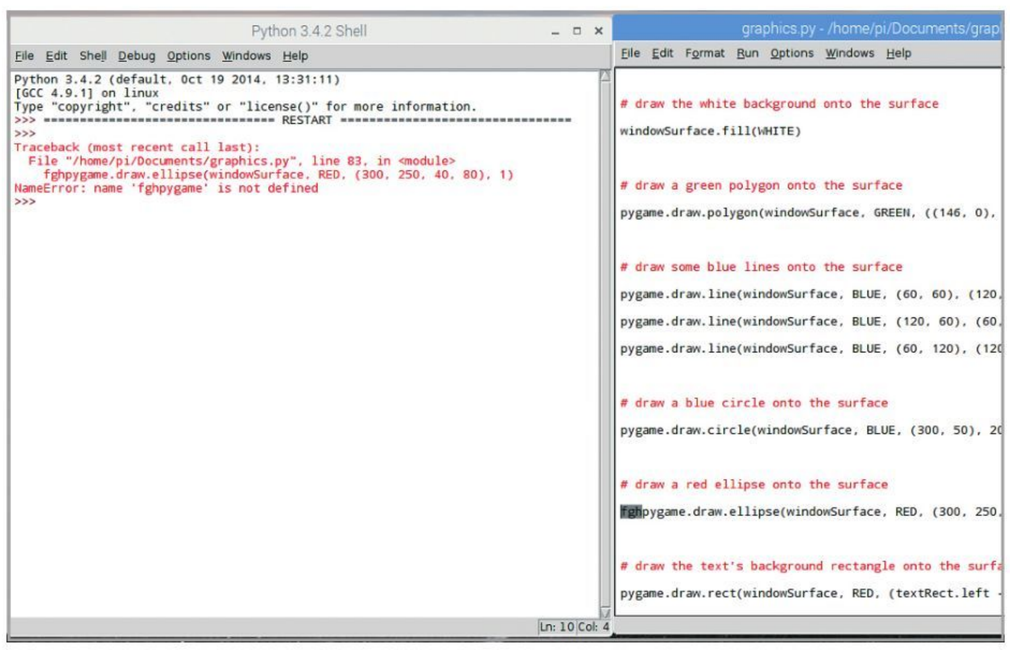
**Errors in code are called bugs and are perfectly normal. They can often be easily rectified with a little patience. The important thing is to keep looking, experimenting and testing. Eventually your code will be bug free.**

**STEP 1** Code isn't as fluid as the written word, no matter how good the programming language is. Python is certainly easier than most languages but even it is prone to some annoying bugs. The most common are typos by the user and whilst easy to find in simple dozen-line code, imagine having to debug multi-thousand line code.

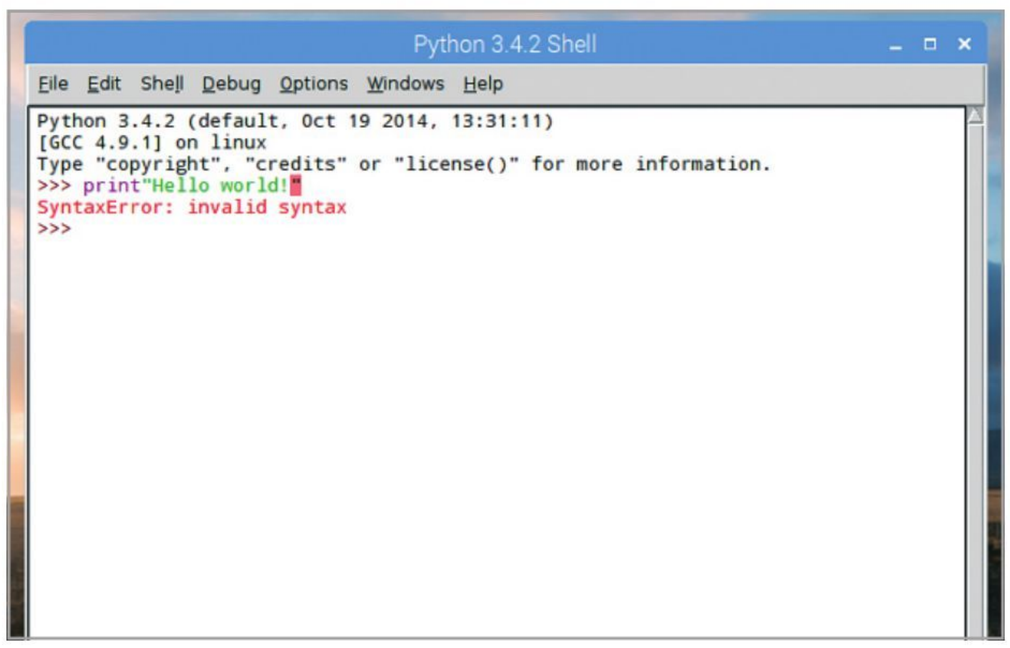

**STEP 2** The most common of errors is the typo, as we've mentioned. The typos are often at the command level: mistyping the print command for example. However, they also occur when you have numerous variables, all of which have lengthy names. The best advice is to simply go through the code and check your spelling.




**STEP 3** Thankfully Python is helpful when it comes to displaying error messages. When you receive an error, in red text from the IDLE Shell, it will define the error itself along with the line number where the error has occurred. Whilst in the IDLE Editor this is a little daunting for lots of code; text editors help by including line numbering.




**STEP 4** Syntax errors are probably the second most common errors you'll come across as a programmer. Even if the spelling is correct, the actual command itself is wrong. In Python 3 this often occurs when Python 2 syntaxes are applied. The most annoying of these is the print function. In Python 3 we use **print("words")**, whereas Python2 uses **print "words"**.



## STEP 5

Pesky brackets are also a nuisance in programming errors, especially when you have something like:

```
print(balanced_check(input()))
```

Remember that for every '(' there must be an equal number of ')'.

```
import sys

def balanced_check(data):
    stack = []
    characters = list(data)

    for character in characters:
        reference = {
            '(': ')',
            '{': '}',
            '[': ']'
        }
        if character in reference.keys():
            stack.append(character)

        elif character in reference.values() and len(stack) > 0:
            char = stack.pop()
            if reference.get(char) != character:
                return "NO"
        else:
```

## STEP 6

There are thousands of online Python resources, code snippets and lengthy discussions across forums on how best to achieve something. Whilst 99 per cent of it is good code, don't always be lured into copying and pasting random code into your editor. More often than not, it won't work and the worst part is that you haven't learnt anything.

8

You have a bare except clause; i.e.,

```
try:
    some_code()
except:
    clean_up()
```

The problem with a bare except is that it will catch *all* exceptions, including ones you really don't want to be ignoring (like KeyboardInterrupt and SystemExit). It would be much better if your except block only caught the specific exception you expect, and let all others bubble up as normal.

A few other general comments on your code:

- In line 200, you have this construction:

```
for letter in range(len(chosen_word)):
    if player_guess == chosen_word[letter]:
        word_guessed[letter] = player_guess
```

You're looping over the index variable, but also using the list element. It would be better to write:

```
for idx, letter in enumerate(chosen_word):
    if player_guess == letter:
```

## STEP 7

Indents are a nasty part of Python programming that a lot of beginners fall foul of. Recall the If loop from the Conditions and Loops section, where the colon means everything indented following the statement is to be executed as long as it's true? Missing the indent, or having too much of indent, will come back with an error.

File Edit Format Run Options Windows Help

```
word=input("Please enter a four-letter word: ")
word_length=len(word)
if word_length == 4:
print (word, "is a four-letter word. Well done.")
else:
       print (word, "is not a four-letter word.")
```

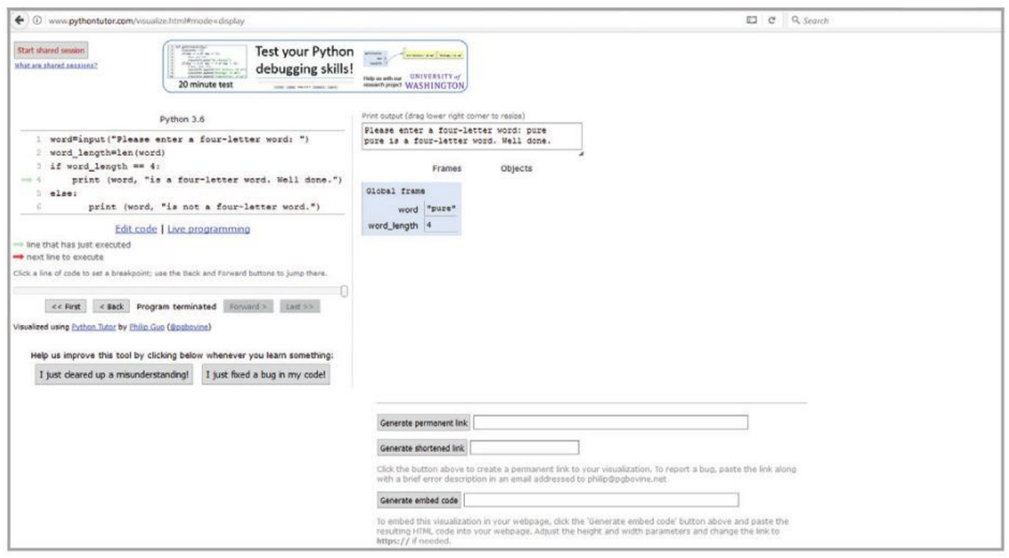
SyntaxError

expected an indented block

OK

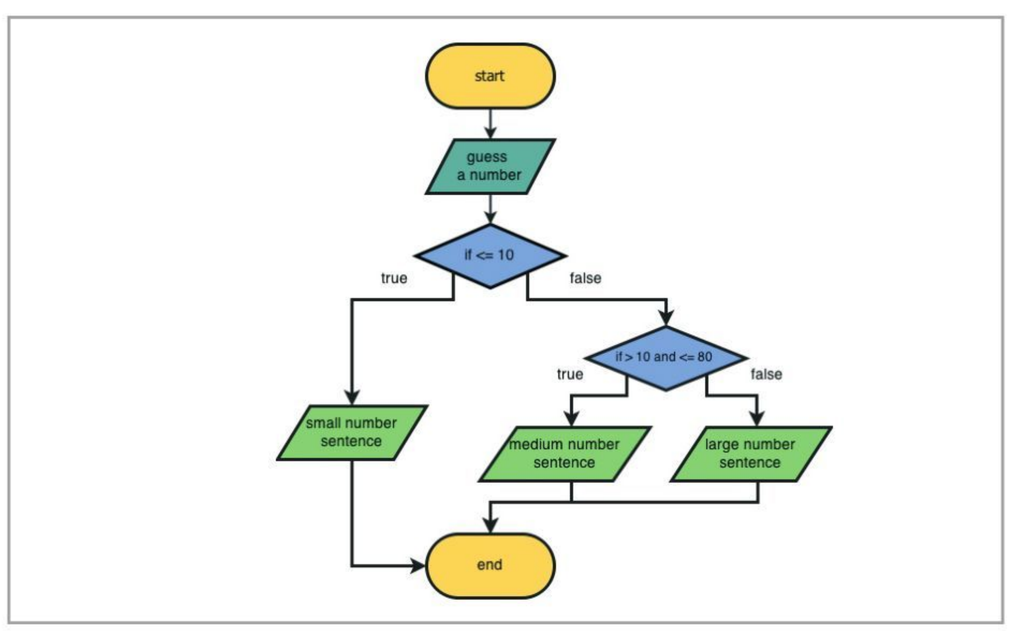
## STEP 8

An excellent way to check your code step-by-step is to use Python Tutor's Visualise web page, found at www.pythontutor.com/visualize.html#mode=edit. Simply paste your code into the editor and click the Visualise Execution button to run the code line-by-line. This helps to clear bugs and any misunderstandings.




## STEP 9

Planning makes for good code. Whilst a little old school, it's a good habit to plan what your code will do before sitting down to type it out. List the variables that will be used and the modules too; then write out a script for any user interaction or outputs.




## STEP 10

Purely out of interest, the word debugging in computing terms comes from Admiral Grace Hopper, who back in the '40s was working on a monolithic Harvard Mark II electromechanical computer. According to legend Hopper found a moth stuck in a relay, thus stopping the system from working. Removal of the moth was hence called debugging.

# Combining What You Know So Far

We've reached the end of this section so let's take a moment to combine everything we've looked at so far, and apply it to writing a piece of code. This code can then be used and inserted into your own programs in future, either part of it or as a whole.

## PLAYING WITH PI

**For this example we're going to create a program that will calculate the value of Pi to a set number of decimal places, as described by the user. It combines much of what we've learnt, and a little more.**

**STEP 1** Start by opening Python and creating a New File in the Editor. First we need to get hold of an equation that can accurately calculate Pi without rendering the computer's CPU useless for several minutes. The recommended calculation used in such circumstances is the Chudnovsky Algorithm, you can find more information about it at en.wikipedia.org/wiki/Chudnovsky_algorithm.

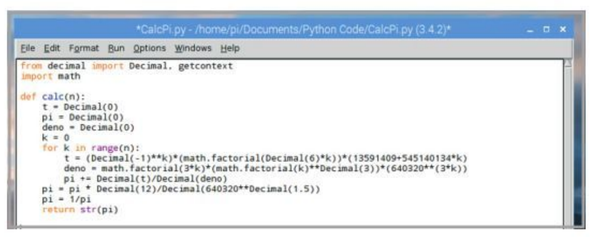
**STEP 2** You can utilise the Chudnovsky Algorithm to create your own Python script based on the calculation. Begin by importing some important modules and functions within the modules:

```
from decimal import Decimal, getcontext
import math
```

This uses the decimal and getcontext functions from the Decimal module, both of which deal with large decimal place numbers and naturally the Math module.

**STEP 3** Now you can insert the Pi calculation algorithm part of the code. This is a version of the Chudnovsky Algorithm:

```
def calc(n):
  t = Decimal(0)
  pi = Decimal(0)
  deno = Decimal(0)
  k = 0
  for k in range(n):
    t = (Decimal(-1)**k)*(math.factorial
(Decimal(6)*k))*(13591409 +545140134*k)
    deno = math.factorial(3*k)*(math.
factorial(k)**Decimal(3))*(640320**(3*k))
    pi += Decimal(t)/Decimal(deno)
  pi = pi * Decimal(12)/Decimal(640320**Decimal(1.5))
  pi = 1/pi
  return str(pi)
```



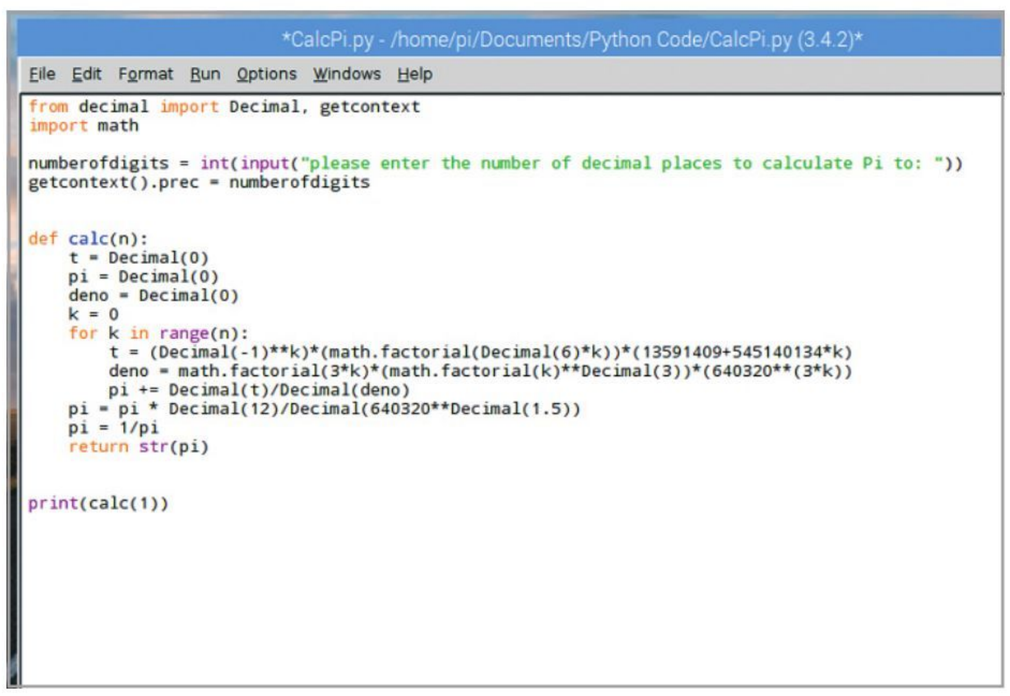

**STEP 4** The previous step defines the rules that make up the algorithm and creates the string that will eventually display the value of Pi, according the Chudnovsky brothers' algorithm. You have no doubt already surmised that it would be handy to actually output the value of Pi to the screen. To rectify that you can add:

```
print(calc(1))
```

**STEP 5** You can save and execute the code at this point if you like. The output will print the value of Pi to 27 decimal places: **3.14159265358973420766845359**1. Whilst pretty impressive on its own, you want some user interaction, to ask the user as to how many places Pi should be calculated.

**STEP 6** You can insert an input line before the Pi calculation Def command. It needs to be an integer, as it will otherwise default to a string. We can call it numberofdigits and use the getcontext function:

```
numberofdigits = int(input("please enter the
number of decimal place to calculate Pi to: "))
getcontext().prec = numberofdigits
```



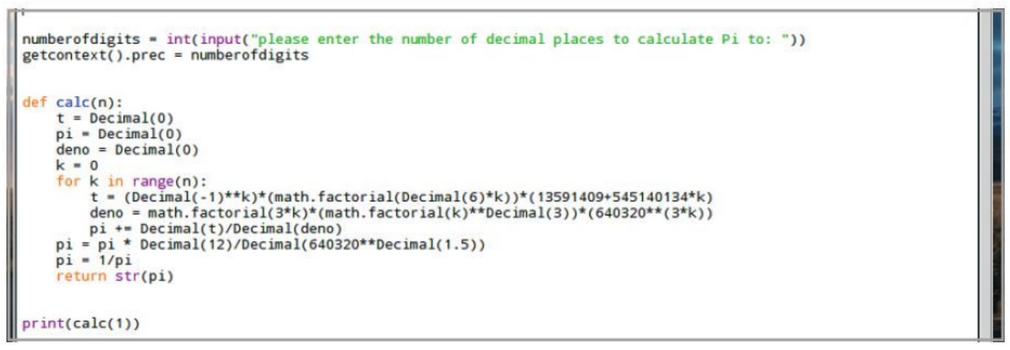
**STEP 7** You can execute the code now and it asks the user how many decimal places they want to calculate Pi to, outputting the result in the IDLE Shell. Try it with 1000 places but don't go too high or else your computer will be locked up in calculating Pi.

**STEP 8** Part of programming is being able to modify code, making it more presentable. Let's include an element that times how long it takes our computer to calculate the Pi decimal places and present the information in a different colour. For this, drop into the command line and import the Colorama module (RPi users already have it installed):


```
pip install colorama
```

**STEP 9** Now we need to import the Colorama module (which will output text in different colours) along with the Fore function (which dictates the foreground, ink, colour) and the Time module to start a virtual stopwatch to see how long our calculations take:

```
import time
import colorama
from colorama import Fore
```



```
numberofdigits = int(input("please enter the number of decimal places to calculate Pi to: "))
getcontext().prec = numberofdigits


def calc(n):
    t = Decimal(0)
    pi = Decimal(0)
    deno = Decimal(0)
    k = 0
    for k in range(n):
        t = (Decimal(-1)**k)*(math.factorial(Decimal(6)*k))*(13591409+545140134*k)
        deno = math.factorial(3*k)*(math.factorial(k)**Decimal(3))*(640320**(3*k))
        pi += Decimal(t)/Decimal(deno)
    pi = pi * Decimal(12)/Decimal(640320**Decimal(1.5))
    pi = 1/pi
    return str(pi)


print(calc(1))
```

**STEP 10** To finish our code, we need to initialise the Colorama module and start the time function at the point where the calculation starts, and when it finishes. The end result is a coloured ink displaying how long the process took (in the Terminal or command line):

```
from decimal import Decimal, getcontext
import math
import time
import colorama
from colorama import Fore
colorama.init()

numberofdigits = int(input("please enter the number of decimal places to calculate Pi to: "))
getcontext().prec = numberofdigits

start_time = time.time()
def calc(n):
    t = Decimal(0)
    pi = Decimal(0)
    deno = Decimal(0)
    k = 0
    for k in range(n):
        t = (Decimal(-1)**k)*(math.factorial(Decimal(6)*k))*(13591409+545140134*k)
        deno = math.factorial(3*k)*(math.factorial(k)**Decimal(3))*(640320**(3*k))
        pi += Decimal(t)/Decimal(deno)
    pi = pi * Decimal(12)/Decimal(640320**Decimal(1.5))
    pi = 1/pi
    return str(pi)

print(calc(1))
print(Fore.RED + "\nTime taken:", time.time() - start_time)
```