THE T-EXCHANGE ABSOLUTE BEGINNERS GUIDE TO PYTHON

This short beginner's guide is meant for those who would like to try Python, the programming language recommended by the makers of the Raspberry Pi. You don't however need a Raspberry Pi to use Python. It is available for a wide variety of operating systems. For the purpose of this short introduction I am using the Windows version, but it will work exactly the same when you use any of the Linux versions on the Raspberry Pi. This guide is a work in progress, so input and feedback is much appreciated. The documentation that comes with Python is excellent, so this guide is only to get you on the way. Once you get the basics, the rest is easily obtained from the documentation.

DOWNLOADING AND INSTALLING PYTHON (Windows Only)

Python is free of charge and you can find it at www.python.org in the Downloads section. The version I used was Python 3.2.3 Windows x86 MSI Installer. Of course you need to download a different version if you have a different operating system.

Installing is a breeze, just double click the MSI file and follow the instructions. No desktop icon is created, so you need to go to the Start menu to find it once it is installed. Go to the Python 3.2 folder in the Start menu (under All Programs). There you see various things including the Python Manuals. For now we are only interested in the program called EDLI (Python GUI). This starts the so called Python Shell. For the purpose of this guide, we shall only use the Shell as it also conveniently gives us access to an editor, but more about that later.

You should ideally create a folder in "My Documents" and call it something like "Python Work" so you have a place where you can store and easily find your own Python scripts.

When you start the Shell, i.e. EDLI (Python GUI) , then a window opens with the following text in it:

```
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

You are now all ready to go.

FIRST STEPS

The ">>> " indicates where the so called "command line" is. You can type all sorts of commands here. You could try "copyright" as suggested in the text. The result is a list of information.

You can use this Python Shell as a calculator. For example try:

```
>>> 4 * 4
16
>>>
```

You can also store information in “variables” like this:

```
>>> import math
>>> a = 3
>>> b = 4
>>> math.sqrt(a**2 + b**2)
5.0
>>>
```

The “import math” command you only need to do once. This imports the math module (which comes as standard with Python) and provides all sorts of additional math capability. It will be available until you close the Shell. “math.sqrt(x)” takes the square root of x. the “**” symbol means “powered by”. So in english, what you have done is:

1. Load the math module
2. Store value 3 in a variable called a
3. Store value 4 in a variable called b
4. Calculate the square root of the sum of a squared and b squared.

Once calculations become bigger and you also want to start manipulating text and collections of numbers and text and perhaps even the content of data files, then simply typing things in at the command line becomes quickly impractical. That’s where so called scripts come in.

YOUR FIRST PYTHON SCRIPT

A script is nothing more than a list of Python instructions which are saved in a text file that is then interpreted by the Python Shell when you ask to “run” it. To create such a text file all you need to do is go to the Shell click on “File” and “New Window” in the menu. A new window will appear called “Untitled”. It is always good practice to straight away save the new file by going to “File” and “Save As”. Let’s call this file “hello.py” and save it in the ”Python Work” folder you hopefully created earlier (otherwise you may want to do it now). Please note that the extension “.py” is NOT automatically appended when you save the file, so you need to add it yourself!

Now type the following in the new window:

```
print ("hello world!")
```

You will notice that the text automatically changes colour when it recognizes specific commands, functions and types of information, etc. You will also see some hints while you type. All these things are aimed to help creating and debugging scripts more quickly and easily.

Now click on “Run” and “Run Module” or press F5. You will be asked if it is o.k. to save your work first, so click “OK”. The Python Shell will come to the foreground again and if all is well you should see:

```
>>> ========================== RESTART ============================
>>>
hello world!
>>>
```

Congratulations you just wrote and executed your first Python script!

Go back to your script (you can find it in the taskbar when clicking on the Python logo). You can always re-open your scripts by going to “File” and “Open” or look in “Recent Files”. You may need to browse to your “Python Work” folder to find your saved scripts.

A BIT MORE ABOUT VARIABLES

Here are a few examples of all the things you can do with variables. We’ve already seen that you can store numbers in variables, but it is also possible to store text, or collections of information (more later).

Here are some examples. We are working directly in the Shell for this, as it is a bit quicker to try things out this way:

```
>>> text = "hello world!"
>>> text
'hello world!'
>>>
```

This is an example of a string. You may also put the string in single quotes ‘ ‘ the choice is yours. Notice that I didn’t need to use “print()” to show the content of the variable “text”. That is because I am working directly in the Shell. In scripts you will need to use “print()”.

It is possible to refer to specific parts of a string. This is called indexing. To select a single character from the string we can use:

```
>>> text[0]
'h'
>>> text[5]
' '
>>> text[-1]
'!'
>>>
```

The first character in a string has index [0]. Using negative indices works back from the end of the string where [-1] is the last character in the string. You can also get a range of characters using:

```
>>> text[:5]
'hello'
>>> text[6:]
'world!'
>>> text[6:8]
'wo'
>>> text[:-2]
'hello worl'
>>>
```

Basically [a:b] means “start at a, for b characters”, remembering that the first character has index 0, and the negative indices count from the end of the string.

Python supports so called *iterables*, which are collections of information such as lists, tuples, sequences (including ranges) sets and dictionaries. We're not worried about most of those for now, if you are interested there is plenty more about this in the Manuals in The Python Tutorial section. But here are a few examples:

```
>>> names = ["bill", "howie", "maarten"]
>>> names
['bill', 'howie', 'maarten']
>>> names[1]
'howie'
>>> names[2][0:3]
'maa'
>>>
```

This is an example of a list. You can mix the type of information in a list, so it doesn't need to be all strings. And as you can see the indexing works as well, but in this case on more than one level!

Here is an example of a tuple:

```
>>> coord  =  1, 2, 5
>>> coord
(1, 2, 5)
>>> x, y, z = coord
>>> x
1
>>>
```

In this case we stored some coordinates in coord. You will have noticed how easy it is to put the values from coord into individual variables again. This is known as *sequence unpacking*. You can also unpack a sequence at once, allowing things like:

```
>>> a, b, c = 5.5, 6.3, 7.8
>>> b
6.3
>>>
```

And this is also a very important type of variable, the range variable:

```
>>> n = range(10)
>>> n
range(0, 10)
>>> m = range(10,20,2)
>>> m
range(10, 20, 2)
>>> list(m)
[10, 12, 14, 16, 18]
>>>
```

The function range() creates a range of numbers which is very useful for repeat structures (explained later). The syntax of the function is **range([start], finish, [step])**, where the finish itself is

NOT included and the start and step are optional. The default for start is 0 and that for step is 1. I used the square brackets here to show that the information is optional (you will find that this notation is used in most literature). The square brackets should of course not be included in the actual code. As you can see if you call up a range variable, it simply shows it as a range. This is more efficient. In the example I have used the function **list()** to create a list containing the range of numbers in m to show that that range indeed contains all numbers.

You can also "nest" all sorts of variables like this:

```
>>> hotchpotch = [[1, "hello", 3.14], (234.5, 34.7), range(20, 40)]
>>> matrix = [[3, 4], [2, 6]]
>>> p, q = hotchpotch[1]
>>> q
34.7
>>> list(hotchpotch[2])[:5]
[20, 21, 22, 23, 24]
>>> hotchpotch[0][1][3:]
'lo'
>>> [[row[i] for row in matrix] for i in range(2)]
[[3, 2], [4, 6]]
>>>
```

Given the explanations so far, you should be able to follow everything I've done all the way until the last expression. The expression `[[row[i] for row in matrix] for i in range(2)]` is a so called nested *list comprehension*, which I have included purely to show the real power of Python. This expression transposes the matrix (I added this especially for Howie's benefit).

In addition to the above, there are almost limitless functions and so called methods you can apply to variables. Here are just a few examples:

```
>>> names = ["bill", "howie", "maarten"]
>>> names.pop()
'maarten'
>>> names
['bill', 'howie']
>>> names.append("john")
>>> names
['bill', 'howie', 'john']
>>> names.index("howie")
1
>>> len(names)
3
>>>
```

list.**pop([i])**, is a so called method for the object "list" (in our case the list called "names"). It removes the last item in the list and returns the value of that last item. list.**append(x)** ads x (in our case the string "john") to the list. list.**index(x)** returns the index of the first item with value x. **len(a)** is a function that returns the length (number of items or characters) of a list, tuple, range or string with name a.

PROGRAMS AND SCRIPTS

So far I have used the word script, rather than program. The reason is that programs are normally referring to compiled executables, whereas scripts are text files that are interpreted. Since we use the Python Shell to interpret our "programs", we should really refer to them as scripts. You can, however, compile Python Scripts into programs, but I won't discuss that here.

SCRIPTING STRUCTURES

There are three basic scripting structures you need to know: sequence, decisions and loops.

The **sequential** structure of Python scripts are is straightforward:

Start with this,
then do this,
and then do this,
etc…

So in order to work on a variable, you must declare it **before** you refer to it otherwise you get an error. This may seem logical, but it is the cause of many bugs in scripts and programs!

Start a new script and call it "sequence.py". Here is an example of a sequential script.

```
# this script lets you find a telephone number from a list
directory = ["bill", 2345, "howie", 6367, "maarten", 9854]
entries = int(len(directory) / 2)
print("There are", entries, "names in the directory")
find = input("Find the number for: ")
number = directory[directory.index(find) + 1]
print("The number for", find, "is", number)
```

You can add comments to your script using the # symbol. Protocol is that you should always put the comment on a separate line (although you can also put it after a statement). The function **int(x)** returns the integer value of x. Because a division turns any number into a floating point number, the expression `len(directory) / 2` would have resulted in 3.0, the **int()** function makes it into 3.

The **decision structure** in Python uses the **if** statement. The general structure is:

If a condition is true
    Do this
    etc…
Otherwise if another condition is true
    Do this
    etc…
Otherwise
    Do this
    etc…
Once the above is completed continue with this

Here is an example you can try. You could call this script “decision.py”.

```
# this script works out the kind of number you typed
number = int(input("Give us a number: "))
symbols = ""
print("The number you typed is", end=" ")
if number > 0 and number <= 10:
    print("greater than zero but smaller or equal to 10")
    symbols = "0 < n <= 10"
elif number == 0:
    print("equal to zero")
    symbols = "n = 0"
elif number > 10:
    print ("greater than 10")
    symbols = "n > 10"
else:
    print("less than zero")
    symbols = "n < 0"
print("In symbols this is", symbols)
```

You have noticed that when you typed the text automatically indented. This is extremely important. Python expects code that is grouped in a specific structure to be indented to the same level. The recommended indentation is 4 spaces. If you don’t indent properly you will get an error. A good example is the last print statement. If you had indented it 4 spaces, it would have belonged to the statements under else, and the text only shown if n < 0.

If you use the **input()** function to get a number (integer) always use the **int()** function as shown. The **input ()** function by default stores input in a string. Omitting this would have caused and error.

You have noticed that I used == for “equal to”. This is because a single = is of course already used to assign values to a variable. Here is an overview of all possible tests:

Operation	Meaning
<	strictly less than
<=	less than or equal
>	strictly greater than
>=	greater than or equal
==	equal
!=	not equal
is	object identity
is not	negated object identity

You can also create conditions by including so called Booleans such as the **and** in the example. The effects of those are:

Operation	Meaning
`x and y`	**True** if both x and y are **True**
`x or y`	**True** if x is **True** or y is **True** or both are **True**
`not x`	**False** if x is **True**, and **True** if x is **False**

You can store test results into variables as well, and use the values **True** and **False** directly (make sure you use a capital at the start and the rest lower case). Try this in the Shell:

```
>>> 1 == 0
False
>>> 1 > 0
True
>>> a = True
>>> not a
False
>>> b = False
>>> a or b
True
>>> a and b
False
>>>
```

Clever or what?

Finally the **loop structure** comes in two main forms. One uses the **for** statement and one the **while** statement. Let's start with the easy one and that is the **while** statement. Generally it works like this:

While a certain condition is true
    Do this
    Then do this
    etc…
Continue with this once the condition is no longer true

Let's augment our "sequence.py" script a bit so it now looks like this and save it as "directory.py"

```
# this script lets you find a telephone number from a list
directory = ["bill", 2345, "howie", 6367, "maarten", 9854]
entries = int(len(directory) / 2)
print("There are", entries, "names in the directory")
find = input("Find the number for (type 'stop' to exit): ")
while find != "stop":
    number = directory[directory.index(find) + 1]
    print("The number for", find, "is", number)
    find = input("Find the number for (type 'stop' to exit): ")
# this is the end of the while loop
print("Thank you for using directory enquiries...")
```

Notice that I have added the while statement after the input to make sure it is available to check the condition. I have added another input to ask for the next name, and see how I have added that to the very end of the loop to make sure when we input “stop” the script isn’t going to look for that in the directory (which would cause an error).

The **for** statement has the following general effect:

Take the first item from a list
    Do this (with the item from the list)
    Then do this (with the item from the list)
    etc…
    If not at the end of the list, take the next item from the list and do the above again
Once the above is completed continue with this

Let’s change our directory script again as follows:

```
# this script lets you find a telephone number from a list
names = ["bill", "howie", "maarten"]
numbers = [2345, 6367, 9854]
entries = len(names)
print("There are", entries, "names in the directory. They are:\n")
# the \n at the end of the above string adds a blank line
for n in names:
    print(names.index(n) + 1, n)
print("")
# the above prints a blank (empty) line at the end of the list
find = input("Find the number for (type 'stop' to exit): ")
while find != "stop":
    entry = names.index(find)
    print("\nThe number for", find, "is", numbers[entry], "\n")
    find = input("Find the number for (type 'stop' to exit): ")
# this is the end of the while loop
print("\nThank you for using directory enquiries...\n")
```

Please note that I have put the names and numbers in two different lists now and changed the “number” variable to “entry”.

You will also notice that I have added the odd “\n” in the text. This is a so called escape code. This one means “add a new line”. It makes the output a bit easier to read.

The **for** statement simply steps through each name in the names list and puts the name in n, until all names in the list have been exhausted in turn. Here is another good example of why it is important to keep an eye on your indentation. If I would have indented the second print command under **for**, then a blank line would have been added under each name!

For statements always steps through lists, but remember you can also create a list using the range variable. So you can use ranges in **for** loops too!

```
# times tables
y = int(input("Give a number: "))
for x in range(11):
    print(x, "*", y, "=", x * y)
```

And you can of course have any kind of range as input list. Try for instance to change the above to:

```
# times tables
y = int(input("Give a number: "))
for x in range(20,42,2):
    print(x, "*", y, "=", x * y)
```

NESTING THINGS

It is possible to "nest" various structures. We have already seen that the sequence structure goes inside both decision and loop structures, but you can also put loop structures in decision structures and vice versa, or even loop structure in loop structures and decision structures in decision structures.

Let's put a decision structure in a loop structure. We are back in our "directory.py" script and make a final addition:

```
# telephone directory enquiries
names = ["bill", "howie", "maarten"]
numbers = ["2345", "6367", "9854"]
def listnames():    # defines a function to print the names
    """Print the list of names"""
    entries = len(names)
    print("There are", entries, "names in the directory. They are:\n")
    for n in names:
        print(names.index(n) + 1, n)
    print("")
listnames()
find = input("Find the number for (or type stop or add): ")
while find != "stop":
    if find == "add":    # do the following when you typed "add"
        newname = input("Give a new name: ")
        newnumber = input("Give a new number: ")
        names.append(newname)
        numbers.append(newnumber)
        listnames()
        find = input("Find the number for (type 'stop' to exit): ")
    else:    # do the following when you did NOT type "add"
        try:    # captures errors for the statement below
            entry = names.index(find)
        except ValueError:    # do this when there was an error
            print("Name is not in the list, please try again...")
            find = input("Find the number for (type 'stop' to exit): ")
        else:    # do this when there was NOT an error
            print("\nThe number for", find, "is", numbers[entry], "\n")
            find = input("Find the number for (type 'stop' to exit): ")
print("\nThank you for using directory enquiries...\n")
```

Note that I have changed the telephone numbers in the “numbers” list to strings instead.

The first thing you have noticed is that I have added a block that starts with **def.** This defines a function. A function is a block of code you can call anytime you like from anywhere in the script (after the definition block). It saves a lot of typing since I am using this function more than once. The function itself does not do anything until you “call” it, hence the **listnames()** on line 11. I have called this function again on line 19. The first line under the **def** statement is a so called documentation string and should always have three quote marks either side. It is optional, but good practice to add one.

In the **while** loop we have now built in a decision structure to see whether we typed “add”. If so, then we can now add a new name and number to the directory, if not we assume you typed a name.

The **try** statement is a bit little bit like **if**, but allows you to do something about Python errors. By adding **try** before code, that code will no longer cause a system error. You can then use the **except** statement to deal with the error yourself in the script. In this case the fact that the name you typed was not in the list would have produced a ValueError. We intercept this and instead ask kindly to try again. If there was no error (**else** statement) then we simply continue with producing the required result, in this case showing the required telephone number. You will find that error interception using **try** is a very common way of dealing with errors in Python scripts.

ONE FOR THE ROAD

Here is a little script that some of you might find interesting:

```
# prime numbers
print("This script calculates the prime numbers in a specific range.")
print("Press Ctrl+C any time to stop the script")
noRange = True
while noRange:
    a = int(input("Please give the start of the range (positive whole number): "))
    b = int(input("Please give the end of the range (positive whole number): "))
    if (a <= 0 or b <= 0):
        print("Only positive numbers please!")
    elif a >= b:
        print("The second number needs to be bigger than the first!")
    else:
        print("\nThe prime numbers in this range are:")
        noRange = False
# this is where the actual calculations start
for n in range(a, b):
    for x in range(2, n):
        if n % x == 0:
            break
    else:
        print(n, end=" ")
print("\n")
```

Here I have used a **while** loop to make sure people type in some usable numbers (not full proof, but another example of how you could deal with testing if input is correct instead of using **try**).

The two nested **for** loops are doing the business here.

There are a few additional concepts here. The first one is **break**. This statement breaks the current loop you are in. I.e. it stops it. Because this happens in the inner of the two **for** loops, it only breaks out of the inner loop and then goes to the next item from the list in the outer **for** loop. This break happens when the remainder of the division of x by y (that is what the % means) is zero, i.e. when obviously the current number in the range is not a prime number, otherwise the remainder would have been non zero.

The second concept is the use of the **else** statement in the **for** loop. Code that comes after this statement is only executed on normal completion of the loop, i.e. it is ignored when you use **break** to stop the loop.

Can you now work out how it works?

Finally, you have seen me using `end=" "` in the **print** statement. Normally a print statement is completed with new line. If you don't want this, i.e. if you want to continue on the same line, then you can replace this new line by something else with **end=**. In our case we have a space, so that the next prime number simply comes after the previous with one space in between. By replacing it with `end=", "` there would also be a comma between the numbers.

FINALLY

This completes the absolute beginners guide. Of course it only covers the tip of the iceberg, but hopefully in the future I will be able to bring some additional guides such as on how to work with modules, data files and classes, etc. I may even put the odd "exercise" on the website for people to get their teeth into.

TIP! When you start making scripts for yourself, always try and write it first in English. You can then identify the sequence, decision and loop structures, and start putting together the script (= coding) from there. I may in the future show some more design techniques.

Maarten de Vries, July 2012