INTRODUCTION TO PROGRAMMING WITH PYTHON

Contents

PART I: GETTING STARTED

Introduction

Like it or not, the work that we undertake for IPA involves a huge amount of time in front of our Lenovo machines. For some this is a pleasure, for others a chore. Some of us coming to IPA will be tech savvy. Some (like me) will be Luddites. Before leaving for Bangladesh to work for IPA I was given an iPad by my family, and it spent the first three weeks of its life with me in the box under the bed. 12 months later I had just completed my first serious computer program, that I believe has saved my project hundreds of man hours. The point of this anecdote is simply to demonstrate that programming can be done by anyone, and, given how working with computers is ubiquitous, there can be major benefits for everyone.

These notes serve as a guide to getting started building computer programs. The tools needed to be able to compute anything that is capable of computation are actually basic and few. These tools can of course be combined into algorithms of mind boggling complexity, but the foundations are simple, and readily learnable by everyone.

The notes take a practical approach inasmuch as they aim to show the programmer (you!) how to build programs whilst skimming over the technicalities of what is actually going on behind the scenes between the language itself and the interpreter. As the programs that the programmer wants to build become more sophisticated and efficiency becomes an issue, then these technicalities become very important indeed and more research will be needed by the programmer. Thankfully there are a huge number of resources out there for budding programmers (see below).

Resources

Resource	Description
Introduction to Computer Science (OpenCourseWare)	This undergraduate Computer Science course from MIT is hugely useful. It can be accessed online, but more conveniently through iTunes U. The materials consist of lectures, recitations (classes), lecture notes, example code, problem sets and exams. This is highly recommended.
Introduction to Computation and Programming Using Python (John Guttag, MIT Press)	This book was designed to accompany the above course, and provides a very clear and thorough introduction. The author places particular emphasis on learning to think like a computer scientist in terms of how to think about problems. I have borrowed heavily from this source for these notes.
docs.python.org	The official Python docs contain everything you could ever need to know about Python in as accessible language as possible.
Stackoverflow.com	If you have a problem, someone has had it before. This website allows users to post coding problems and receive help in finding solutions from the online community.

Other Tools

When you download the Python programming environment you have access to a large number of libraries. A library is a collection of modules that contain related functions and methods that have been written by the

Python programmers and organized together due to the purpose that they serve. The truly extraordinary thing about Python is the vast number of libraries that exist. Of particular note for those working for IPA:

Pandas	This is the Python statistical module. This is an incredibly powerful tool for data management and statistical analysis. It will very soon eclipse stata as the statistical package of choice for economists.
Time	The Time module allows the programmer to do very clever things with time. It is extremely handy for automation tasks.
DateTime	Do incredible calculations with dates with this module.

Additionally there is an excel add-in called **DataNitro** that allows the programmer to write Python script in Excel, and to combine the script with VBA script. This has proved hugely powerful in the work of projects who are reliant on external data provided in idiosyncratic formats by different partners/participants. It can also be used very effectively to create data entry tools using the excel structure.

The Python Language & Programming Environment

Python is a high level[1] programming language that has been designed according to a philosophy of code readability. It is a general purpose language and can therefore be used to write almost any sort of program.

Just like stata, Python is fussy about capitals, spacing, and (unlike many other languages) indentations. Therefore attention to detail in how the code is actually written is critical.

When Python is installed the programming environment is accessed from the Start Menu. The Python Command Line opens what is called a Python Shell. By entering code into this shell the programmer can communicate directly with the interpreter.

EXERCISE:
In the Python Shell enter:

```
print 'Hello World'
```

will cause the interpreter to produce the output:

```
Hello World
```

It would become absurdly tedious to try to write any serious program directly into the Shell. Instead programmers use the Python IDLE which is the integrated development environment in which programs are

[1] Languages can be high level or low level. A low level language gives performs operations by issuing instructions to the computer's interpreter at the machine level whereas high level languages use higher level abstractions. If you think of the substr command in stata, you understand what it does, but you do not see how the instructions are issued to the computer. This is an example of an even higher level language than Python.

written. The IDLE is simply an application and is opened by double clicking. Upon opening IDLE a Python Shell is opened into which commands and statements can be entered and objects outputted just as above.

A new editing window can be opened by clicking File – New Window in the Shell toolbar. The resulting editing window allows the programmer to write a program which can be run over and over again. If is helps, think of the editing window like a do-file editor in stata. In order to run a program that has been written in the editing window the code must be saved. Save the text as a .py file. In order to run the program either click Run – Run Module, or simply press F5.

NB: the code must always be saved before it is run, and IDLE will prompt the programmer to save the code each time the program is run (if there have been any changes since it was last run). If there is code that for whatever reason the programmer does not want to execute, then that code can either be deleted (not a good idea), or commented out (a good idea). In order to comment out code click Format – Comment Out Region. Whilst the program is being built it is better to comment out unwanted code, as once deleted the code must be saved before it is run, and often programmers will change their mind about which code is required.

When a program is run a new Python Shell will open in which objects that the programmer has instructed be returned will be outputted. It is into the shell that error messages will also be returned.

EXERCISE:
In the Editing Window enter:

```
A = 'Hello'
B = 'World'
print A + " " + B
```

Save this as a .py file and run the module. This will again cause the interpreter to produce the output:

```
Hello World
```

In this example we have created two variables, A and B, both of which are strings, and using the print command we have printed variable A, then a space character and then variable B.

Notice that these variables are now stored in the computer's memory. How do we know this? In the Shell (not the editor) enter:

```
C = A+B
C
```

This will cause the interpreter to produce the output:

```
'HelloWorld'
```

Now attempt one last exercise which should familiarize the programmer further with the IDLE environment, as well as introduce a function that allows the programmer to assign user input to variables used by the program.

EXERCISE:
In the Editing Window enter:

```
A = raw_input("Enter your first name ")
B = raw_input("Enter your second name ")
print "Your full name is: " + A + " " + B
```

Save this as a .py file and run the module.

In this example we have used the raw_input function to prompt the user to enter the components of his name, we have bound those inputs to the variables A and B, and then we have instructed the interpreter to return a string using the print statement.

Don't be alarmed if you do not fully understand what has happened here. This exercise was to enable you to get more comfortable with the IDLE environment before moving to the basic foundations of the Python language.

PART II: THE BULIDING BLOCKS

Objects

Objects are the basic structures which programmers handle, manipulate and return by writing Python code. In actual fact everything in Python is an object. Understanding object oriented programming becomes essential as the programs you wish to write become more complex, but for the purposes of these notes an intuitive approach is taken. The basic idea is to think of everything as being an object that can be passed around your program and acted upon and manipulated by other objects. All of these objects have a type, and what that type is dictates what can be done with and to that object. The most significant object types at this stage are:

int: This is Python speak for integer which has the normal meaning.

float: A float object represents a real number e.g. 2.8 or 30234.512 or even 3.142

bool: This is a Boolean value that can only have two values 'True' or 'False'

str: This denotes a string, textual information. A string is created by entering text within "" or ''. The ability to use either type of speech marks means that string with quotes are possible e.g. x = "He said 'Hello Rory'"

Later we will see many different types, but a lot can be achieved even with these basic types.

Numerical Operators

When working with numbers, types are important. Python takes a flexible approach to recognizing types, and if a type is not declared then it will make an assumption about the type of the object that you wish to create.

EXERCISE:
In the Python Shell enter:

```
A = 10
B = 4
A/B
```

Perhaps what the interpreter outputs surprises you! What has happened here is that when the variables A and B were created Python assumed they were being created as objects of type integer. The result of an operation on two integers is itself an integer, so the remainder is ignored. If either were a float the result would be a float. Now try this:

```
A = 10
B = 4.0
A/B
```

Notice now that the interpreter returns what you expected. This is because when the interpreter sees 4.0 it creates variable B as an object of type float. An alternative method is to declare the type using the float function: B = float(4)

If at any point you are unsure of the type of object you are working with you can use Pythons's built in function 'type'. So if you have a variable A, entering `type(A)` into the Shell will cause the interpreter to return the type of object A. The point is that when working with numbers, and in particular when doing arithmetic, programmers have to be aware of the different results that will be achieved when working with objects of different types.

Bearing the above in mind, the following demonstrate the numerical operators which are largely as expected.

Operator	Example	Explanation
+	A + B	If A and B are both integers then an integer is returned that is equal to the sum of A and B, if either is a float then a float is returned.
-	A – B	If A and B are both integers then an integer is returned that is equal to the value of A minus B, if either is a float then a float is returned.
*	A * B	If A and B are both integers then an integer is returned that is equal to the product of A and B, if either is a float then a float is returned.
/	A / B	If A and B are both integers then an integer is returned that is equal to A divided by B, if either is a float then a float is returned.
%	A%B	This returns the remainder when the number represented by A is divided by the number represented by B. It is referred to as "modulo".
**	A**B	If A and B are both integers then an integer is returned that is equal to the value of A raised to the power B, if either is a float then a float is returned.

NB See the Python documentation for more numerical operators and their uses.

Comparison operators take two or more objects, compare them and return a Boolean value;

Operator	Example	Explanation
==	A==B	Tests whether A is equal to B. An Boolean value is returned with a value 'True' if A equals B, otherwise 'False'.
!=	A!=B	As above, but is testing whether A and B are unequal.
>	A > B	Tests whether A is greater than B and returns the appropriate Boolean value
<	A < B	Tests whether A is less than B and returns the appropriate Boolean value
>=	A>=B	Tests whether A is greater than or equal to B and returns the appropriate Boolean value.
<=	A<=B	Tests whether A is less than or equal to B and returns the appropriate Boolean value.

Boolean

The operations that exist for the type bool are as follows:

Operator	Example	Explanation
and	X and Y	Returns a value of True if both X and Y are true, otherwise returns False
or	X or Y	Returns a value of True if at least one of X or Y are true, otherwise returns False
not	not X	Returns a value of True is X is False, otherwise returns True

Strings

Before diving into working with strings it is important to note that if an object is of type string, then very often it can be combined or manipulated as though it were of a numerical type, often with surprising results. These need to be borne in mind when working with strings as the following example will demonstrate:

EXERCISE:
In the Python Shell enter:

```
A = '3'
B = 4
A*B
```

Stata users would expect an error message here, but in fact there is nothing wrong in the way the language has been written. The output may or may not be what we require, but in any case this demonstrates the importance of type. In this example A is equal to the string of character '3', not the number 3.

If you have a string value that you wish to use in a calculation you can use the int function

```
A = '3'
B = 4
int(A) * B
```

And similarly we can use the string function if we want to use an integer as a string:

```
A + str(B)
```

In other words the str and int functions can be used to combine objects of different types.

As you will have noticed the + and * operators perform different functions depending upon the type of objects they are acting upon. Naturally certain types of use will produce errors:

```
A = '3'
B = '4'
A*B
```

A string is a sequence of characters of varying length. This means that the characters of a string can be iterated over in the same manner as other sequence types that we will see later on. The following operations can be used on all sequence types including strings:

Length: The length of a string can be determined using the len function. Examples are given below but the general format is `len(string)`.

Indexing: Indexing is used to return the character in a string at a particular index point. Somewhat confusingly (at least initially) the index begins at 0. So the first character in a string is at index point 0. The second character of the string is at index point 1 and so on. The general format is `string[index]`. The index point can be given using a positive integer, or a negative integer. The last character of the string is indexed by -1. The second to last character is at index point -2, and so on.

Slicing: Slicing is an extension of indexing and it allows the programmer to return a particular subset of a string. Slicing is achieved by entering `string[start:stop[:step]]`. The start point is the index value of the first character that you want to return, the stop point is the index value of the final character that you **do not** want to return. Step is optional and defines the direction and the size of the steps that is taken by the slice. If the step value is omitted it defaults to +1. In fact, as we will see in the examples the start and stop points are also optional.

EXERCISE (Length and Indexing):
In the Python Shell enter:

```
A = '123456789'
len('123456789')
len(A)
```

Notice that we can use len with the string itself, or with the variable name that points to that string. In both cases the length is returned as 9. Also notice that the length of the string is longer than the final index value, so if we were to enter:

```
A[9]
```

We will receive an error message to the effect that the string index is out of range. This is because the index values of a string begin at 0, so the last index value is in fact given by:

```
A[len(A)-1]
```
Or alternatively
```
A[-1]
```
If you are indexing using negative index values.

To be clear, the character 5 in the string A can be indexed both by `A[4]` and `A[-5]`

EXERCISE (Slicing):
In the Python Shell enter:

```
A = '123456789'
A[0:4]
```

This takes a slice of the string A starting at index position 0 and ending at the index position 4 – 1. This seems odd at first, however, it is in fact incredibly useful as the stop point ending one before the index value specified means we can write slices such as `A[0:len(A)]` and the returned string has the value that we intuitively would expect. If the stop point is omitted then the stop point defaults to the length of the string, such that a more efficient way to write `A[0:len(A)]` is simply to enter `A[0:]`. Indeed as the start point also defaults to 0 if omitted then we can also write `A[:]`.

In the above examples we have not included a step, and so the default step of +1 was used by the interpreter. The step defines the direction and the size of the steps taken by the slice:

```
A[::2]
```

The start position has defaulted to 0, the end position has defaulted to len(A) but the slice moves forward in steps of 2, as we have specified the size of the steps taken.

If we wanted the final four characters of the string in their natural order we could write (noting that the step parameter has been omitted and therefore defaults to +1):

```
A[5:]
A[5:len(A)]
A[-4:]
```

And in reverse order (i.e starting at the end of the string):

```
A[-1:4:-1]
A[-1:-5:-1]
```

One thing to note is then when moving backwards from the end of the string if no stop point is specified the stop point defaults to –len(string). This means that entering `A[-1::-1]` returns the entire string in reverse order.

Two very useful string operators are `in` and `not in`. These operators can be used to ascertain whether a particular character or sequence of characters exists within another string, and the appropriate Boolean value is returned.

EXERCISE:
In the Python Shell enter and observe the outputs:

```
MyString = 'Once upon a time there was a little boy called Jack'
'on' in MyString
'a' not in MyString
'on' and 'he' in MyString
'ja' in MyString
'Ja' in MyString
'w' and 'x' not in MyString
'w' in MyString or 'x' in MyString
```

Python has a mind-boggling number of built in functions and methods that can be used to manipulate strings, and indeed Python itself is generally thought of as being most powerful in its ability to work with strings. In general when I am working with data (from a management rather than analysis point of view) I tend to treat everything as a string.

Strings in Python are immutable, which means that once they are created they cannot be modified. So, in the slicing examples above, a slice of the string may be returned, the original string itself is unchanged.

Some of the most useful built in string methods and examples of their use are as follows (see the Python documentation for the complete list):

Method	**Example**	**Description**

str.capitalize()	`Lstring = 'esTer'`
	`Ustring = Lstring.capitalize()`
	`print Ustring`
		Returns a copy of the string with the first letter capitalized and the remainder in lower case

str.count(sub[,start[,end]])	`River = 'Mississippi'` `print River.count('is')`	Returns the number of non-overlapping instances of the specified substring in the range [start, end]. If the optional start, end arguments not given the default is [0:len(string)].

str.find(sub[,start[,end]])	`R = 'She sells sea shells on the sea shore'`
	`print R.find('sea')`
	`print R.find('Sea')`
		Returns the index value in the string where the specified substring is found. If it is not found the -1 is

		returned

str.isalnum()	`'12345'.isalnum()`
	`Num = '1234a'`
	`Num.isalnum()`
		Returns True if all characters in the string are alphanumeric and there is at least one character.

str.lower()	`name = 'niall'`
	`NAME = 'NIALL'`
	`print name == NAME`
	`print name`
	`print name == NAME.lower()`
		Returns a copy of the string with all characters converted to lowercase.

str.lstrip([characters])	`Spaces = '    IPA'`
	`NoSpaces = Spaces.lstrip()`
	`EG = http//:www.ipa.org`
	`EG.lstrip('pth:w./')`
		Returns a string with leading characters removed. The characters argument specifies the set of characters to be removed, it is not a sequence in that all combinations of those characters are removed. If no characters argument is given then the default is blank space.

str.replace(old, new[, count])	`Ages = 'Tom is 10, Anne is 12, Sarah is 15'`
	`print Ages`
	`Ages.replace('1', '2', 2)`
	`print Ages`
	`print Ages.replace('1', '2', 2)`
		Returns a copy of the string with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

str.rstrip([characters])		As lstrip but for trailing characters

str.split(separator[, maxsplit])	`Names = 'Rory, Cameron, Matt'`
	`NamesList1 = Names.split(',')`
	`print NamesList1`
	`NamesList2 = Names.split(',', 1)`
	`print NamesList2`
		Returns a list of words in the string using the separator as the delimiter string. If maxsplit is given, at most maxsplit splits are done. (For more on lists see below)

str.strip([characters])		As lstrip and rstrip but for both leading and trailing characters.
str.upper()		As lower but returns a capitalized string.

EXERCISE:
It is important to realize that when using the above methods the original string is not itself modified. In order to prove this to yourself enter the following into the Python Shell

```
MyString = 'The animals went in 2 by 2 hurrah!'
print MyString
print MyString.replace('2', '3')
print MyString
```

When you perform a string operation a modified copy of the string is returned. In order to capture this modification you must assign it to a new variable name

```
MyString = 'The animals went in 2 by 2 hurrah!'
MyNewString = MyString.replace('2', '3')
print MyString
print MyNewString
```

Creating Variables

Variables are a handy way of associating names to objects. Variable names are nothing more than names. You can name an object anything you like. Essentially what happens is that when an object is created in the memory of the machine providing a variable name creates a pointer from the variable name to the object in the memory.

There are some things to watch out for in naming objects as variables. Firstly, having multiple variable names pointing to the same object is called aliasing, and can cause major headaches. It is best avoided unless it is required for some specific reason. Secondly, as Python takes a dynamic attitude toward type (meaning that you can change the type at any time without declaring the change), then you have to be careful to keep track of what your variables are actually pointing to. For example if you enter x = 1, and then x = 'abc' the variable x will first point to an integer object and then a string object.

Lastly as a matter of style, when you write programs you will need to debug them. This means that you should make your code as readable as possible. Consider the following program that allows you to calculate length of the hypotenuse of a right angled triangle:

```
A = 3
B = 4
C = (A**2+B**2)**0.5
```

Now consider the following:

```
Side1 = 3
Side2 = 4
Hypotenuse = (Side1**2 + Side2**2)**0.5
```

What the above aims to illustrate is that when you come back to read your code some days, weeks or months later, the latter example will be much easier to understand.

If

So far everything we have seen would allow us to write on straight line programs. These could be useful in only a very limited set of circumstances, and frankly if they were all that computers could do the world would be in a sorry state. With 'if' you will be able to write what are called branching programs. These begin to open up millions more programming possibilities. The if is a conditional statement. They are of the following structure:

`if` *test*:
    execute this code
`else`:
    execute this code

There are three things to note. Firstly, the test must be a Boolean expression (i.e. capable of evaluating to true of False). Secondly, the code that is executed can be infinite lines of code or one line of code or anything in between. Thirdly, the indentation is obligatory. This is unusual. Stata users will be used to using { } to denote the branches of the commands. The indentation is automatically applied after the ':' which is also mandatory. Indentation will lead to code that looks like this for nested if statements:

```
if test1:
      print outcome1
elif test2:
      if test3:
            print outcome2
      else:
            print outcome3
else:
      print outcome4
```

Using `elif` is equivalent to writing 'else if' and therefore allows the programmer to add yet more branches. In fact the 'else' is optional meaning that:

```
if test1:
      print outcome1
```

Is exactly the same as writing

```
if test1:
      print outcome1
else:
      do nothing
```

EXERCISE:
Now we are going to write some real programs! In the Python Editing Window create a program that asks the user to enter a number and then evaluates whether that number is divisible by 3:

```
UserInt = int(raw_input("Enter an integer "))
if UserInt%3 == 0:
      print str(UserInt) + ' is divisible by 3'
else:
      print str(UserInt) + ' is not divisible by three'
```

Hit f5 to run the program. Notice that when using raw_input the entered data are considered strings. Therefore we have to use the int() function to type convert the input to an integer for the arithmetical operation and str() function when printing the user input.

Now let's do something a bit more complex. Ask the user to enter three unique integers and print the largest amongst them.

```
Int1, Int2, Int3 = int(raw_input("Enter the first integer ")),
int(raw_input("Enter the second integer ")), int(raw_input("Enter the third
integer "))

if Int1 > Int2 and Int1 > Int3:
      print str(Int1) + " is the largest of the entered integers"
elif Int2 > Int3:
      print str(Int2) + " is the largest of the entered integers"
else:
      print str(Int3) + " is the largest of the entered integers"
```

Now try this: write a program that asks the user to enter three unique integers. If none of the integers are odd numbers then return a statement to that effect. If any of the integers are odd then return a message to the user that tells them which is the largest of the odd numbers and how many of the numbers are odd. For example "5 is the largest of 2 odd numbers". A possible solution is found on the next page, but try your own program first.

SOLUTION:

```
Int1, Int2, Int3 = int(raw_input("Enter the first integer ")),
int(raw_input("Enter the second integer ")), int(raw_input("Enter the third
integer "))

OddInts = 0
Int1Odd = 0
Int2Odd = 0
Int3Odd = 0
if Int1%2 != 0:
    OddInts +=1
    Int1Odd +=1
if Int2%2 != 0:
    OddInts +=1
    Int2Odd +=1
if Int3%2 != 0:
    OddInts +=1
    Int3Odd +=1

if OddInts == 0:
    print "There are no odd numbers amongst the entered integers"
else:
    if Int1 * Int1Odd > Int2 * Int2Odd and Int1 * Int1Odd > Int3 * Int3Odd:
        print str(Int1) + " is the largest odd number of " + str(OddInts) +
" odd numbers"
    elif Int2 * Int2Odd > Int3 * Int3Odd:
        print str(Int2) + " is the largest odd number of " + str(OddInts) +
" odd numbers"
    else:
        print str(Int3) + " is the largest odd number of " + str(OddInts) +
" odd numbers"
```

Notice that when you wish to increment the value of an integer object you can write Var1 += 1, which is the same as writing Var1 = Var1 + 1.

Iterations

Being able to iterate, that is execute commands a number of times and/or on or with a number of objects, is the final tool that we need to add to the basic toolbox in order that new programmers can write meaningful programs. Often iteration is implicit; for example when the str.upper() (to return a string with each character capitalized) we are in effect telling the computer "for each character *char* in MyString, capitalize *char*," – but we do not actually need to specify the iteration as it is built into the method. However, if we had a number of strings and we wanted to apply the str.upper() method to each of them, one way to do this would be to iterate over the group of strings and execute the method on each string in turn.

Python has two statements that allow for iteration firstly we will consider the `while` statement. Like the `if` statement, the iteration mechanisms (often called loops) that can be constructed with the `while` statement begin with a test, and then blocks of statements that are executed depending upon the evaluation of the test. If the test evaluates to True then then the statements that follow the test are executed. Then the program returns to the test and performs the evaluation again and again if it evaluates to True then the statements that follow the test are executed. This repeats until the test evaluates to False. If the test never evaluates to False then the program will run to infinity and hence we will be stuck in an infinite loop.

EXERCISE:
The following is an example of how to use the while statement. In the Python Editing Window enter the following:

```
x = int(raw_input("Enter an integer "))
ItersRemaining = x
while ItersRemaining != 0:
      print "There are " + str(ItersRemaining) + " iterations remaining"
      ItersRemaining = ItersRemaining - 1
```

The test in the above example is whether ItersRemaining is not equal to zero. Each time that this test evaluates to True (i.e. when ItersRemaining does not in fact equal zero) the print command is executed. The value of ItersRemaining is then decreased by precisely 1. When ItersRemaining is equal to zero the test evaluates to False and the loop terminates.

Try running the above program again, and this time enter an integer value of -1. Now we have a problem! The program will run forever as ItersRemaining will never equal zero so the test will never evaluate to False. What might we do to overcome this problem? There are a number of solutions. Firstly we could write a test that evaluates the legitimacy of the input entered by the user before entering the loop. If the user enters a negative value the program could request that the input is re-entered until a positive value is entered. Alternatively, if this is too restrictive we could use the absolute value of x as the value of ItersRemaining. The first of these solutions is shown below:

```
x = int(raw_input("Enter an integer "))
while x < 0:
    print "Please do not enter negative integers"
    x = int(raw_input("Enter an integer "))
ItersRemaining = x
while ItersRemaining != 0:
      print "There are " + str(ItersRemaining) + " iterations remaining"
      ItersRemaining = ItersRemaining - 1
```

If using the absolute value of x is preferred then we would just write `ItersRemaining = abs(x)`. What should be clear is that the body of the loop should change one or more of the variables upon each pass through such that the while condition will at some point evaluate to False.

The following example uses the while statement to evaluate whether a user entered integer is a prime number

```
x = int(raw_input("Enter an integer "))
while x <= 0:
    print "You must select a positive integer. Please try again."
    x = int(raw_input("Enter an integer "))
divisor = 2
IsPrime = 0
if x == 1 or x == 2:
    print str(x) + " is a prime number"
else:
    while divisor <= x**0.5:
        if x%divisor == 0:
            print str(x) + " is not prime number"
            print "The smallest divisor is " + str(divisor)
            IsPrime+=1
            break
        else:
            divisor +=1
    if IsPrime == 0:
        print str(x) + ' is a prime number'
```

The above code is an example of an algorithm that is implemented using a while loop iteration. There are several methods for testing whether a number is prime. The method shown here is of the following format:

- X is the number we want to test
- Divide X by 2. If the result of this is an integer then X is not prime because 2 is a factor of X
- Divide X by 3. As above, if the result of this is an integer then X in not prime because 2 is a factor of X
- Continue dividing X by each number between 2 and $X^{1/2}$. If any of these divisions result in an integer then X is not prime as the program has found a factor.

The program executes this method in the following fashion:

- Firstly if the user is constrained to enter a positive integer. This is achieved using the first while statement in a manner similar to that already described.
- If the entered integer is found to be equal to 1 or 2 then a message is printed to notify the user that the entered integer is a prime number
- If the integer is not equal to 1 or 2 then the algorithm is executed.
- A variable called divisor is created that is initially equal to 2. While divisor is less than or equal to the integer value of the square root of x, then the loops tests whether X modulo the divisor is an integer. If it is then a print statement is returned to the effect that X is not a prime number. If X modulo the divisor is not an integer then the value of the divisor is increased by 1 until such time as the divisor is greater than the integer value of the square root of X.
- A variable called IsPrime is created which is initially equal to 0. If a factor is found then this increases to 1. Once the alogorithm has terminated if this variable is still equal to 0 then X is a prime number. This is evaluated in the last lines of the program and the appropriate message is returned to the user.

Notice the use of the break statement. In order for a number not to prime there has to be one factor that is not 1 or the number itself. Once a factor is found the number is definitely not prime. The break statement allows us to make the code efficient. As an example, imagine that we want to evaluate the number 1562500. The square root of this number is 1250 which means that the while loop will divide 1562500 by every number between 2 and 1250. However, clearly 1562500 is divisible by 2 and is not a prime number. The break statement comes at the end of the branch of the program which is followed when a number is not prime. If this branch is reached then the number is not prime, and the break statement forces the program to exit the loop. The point is that we only need to find one factor, and so to continue searching beyond that factor would be a waste of time. Of course the computer can do it in a fraction of a millisecond, but as the numbers get larger, and the programs more complex then such break statements can become useful from an efficiency point of view.

The `for` statement also allows for iterations. Unlike the while statement which can theoretically run infinitely, the for statement is used to execute blocks of code a fixed number of times. The general format is as follows:

```
for variable in sequence:
      code
```

The variable name can be anything although as mentioned above giving sensible names to variables can improve code readability. The sequence can be a number of different object types, and we will see this shortly. What in fact happens is that the variable name is bound to the first element in the sequence and the code is executed on that variable. Then the variable name is bound to the next element of the sequence and the code is executed on that variable, and so on until the sequence is exhausted, or a break statement is encountered.

The sequence that is iterated over can be an object such as a list, tuple or dictionary (which we will encounter in the intermediate course), or it could be a string or a named group of objects (such as integers). Technically speaking any object with an iterable method can be used in a for loop in Python. Often the sequence will be a range. Consider the following examples:

```
for j in range(1, 11):
    print "This is iteration number " + str(j)

for j in range(1, 11):
    for k in range(1, 6):
        print j * k
```

The range function used above creates a sequence that begins at the first integer specified and ending at the integer before the final integer specified. If no first integer argument is provided then the default is zero. The second example is a nested loop, first the program enters the first for loop by binding j to the integer 1, it then enters the second for loop and will totally execute that for loop before then returning to the outer loop and setting j equal to 2. Examine the output and make sure you understand what has happened.

In the following example we will see how to iterate over a string. The thing to remember is that a string is a series of characters, and therefore a for loop can iterate over each of those characters in turn. Similarly when we come to look at lists and dictionaries and tuples we can iterate over each element of those objects in a similar fashion.

```
for character in "Hello World":
    if character != " ":
        print character.upper()
```

In the above example the for loop iterates over each character in the string “Hello World”. If the character is not a blank then then upper method is applied to the character and it is printed.

EXERCISE: Take the following string which is a string of floating point numbers separated by commas and find the total adding them together.

```
total = 0
s = '1.23,2.4,3.123'
slist = s.split(',')
for elem in slist:
    total = total + float(elem)
print total
```

The above example splits the string into a list of numbers based upon the comma. The for loop then iterates over the elements of list and adds them together. You will learn more about lists in the intermediate course.

Finally, lets return to the while loop that tests whether a number is prime, and modify it using a for loop so that we can test a range of numbers.

```
x = int(raw_input("Enter a start point integer "))
y = int(raw_input("Enter an end point integer "))

for numbers in range(x, y +1):
    divisor = 2
    IsPrime = 0
    if numbers == 1 or numbers == 2:
        print str(numbers) + " is a prime number"
    else:
        while divisor <= numbers**0.5:
            if numbers%divisor == 0:
                print str(numbers) + " is not prime number"
                print "The smallest divisor is " + str(divisor)
                IsPrime+=1
                break
            else:
                divisor +=1
        if IsPrime == 0:
            print str(numbers) + ' is a prime number'
```

As should be evident, essentially all that has happened here is that we have wrapped the program in a for loop. There are a few important things to note:

- The range we have created goes from x to y +1. This ensures that the integer y is tested. Recall that the range function creates a sequence from the first integer value up to but not including the last integer value specified.
- The divisor and IsPrime variables must be contained within the for loop. This is because both of these variables need to return to their intial values each time the for loop is executed. If they were defined outside the for loop then the divisor and IsPrime variables would increase without resetting and this would lead to perverse results.

PART III: ADVANCED OBJECTS

Tuples

Tuples bare a resemblance to strings in that they are essentially an ordered sequence of elements, and they are immutable, i.e. they cannot be altered once they have been created. Where they differ from strings is that the elements of a string are always characters whereas the elements that make up a tuple can be any object. Therefore a tuple can have elements that are strings, integers, floats, other tuples, lists, dictionaries and so on, and there is no requirement that the elements of a tuple are all of the same type.

Creating tuples involves writing a list of elements each separated by a comma and enclosed in parentheses.

EXERCISE: in the Editing Window enter the following and run the program:

```
tuple1 = (1, 2, 3, 'Rory', 'Cameron')
tuple2 = ('J-Pal', 'IPA')
tuple3 = (tuple1, tuple2)
EmptyTuple = ()
SingleTuple = (50,)
tuple4 = tuple1 + SingleTuple

print tuple1
print len(tuple1)
print tuple3
print len(tuple3)
print tuple4
print EmptyTuple
print SingleTuple
print tuple2 + EmptyTuple + SingleTuple
print tuple1[3]
print tuple3[1]
print tuple3[1][1]
print tuple3[0][2:4]
```

The print statements and their outputs are described sequentially as follows:

The first print statement prints tuple1 which contains five elements, three of which are integers and two of which are string objects. The tuple is therefore of length 5 as the second print statement confirms.

The third print statement prints tuple3 which contains tuple1 and tuple2. Although tuple1 and tuple2 both contain multiple elements, tuple3 itself only contains two elements and is therefore of length 2 as the fourth print statement confirms.

The fifth print statement prints the EmptyTuple which is a tuple that contains no elements. This demonstrates the method for creating empty tuples.

The sixth print statement prints SingleTuple. Notice that to create a tuple with a single element a comma is still required. Had we instead written `EmptyTuple = (50)` this would have created only an object of type integer with a value of 50.

The seventh print statement demonstrates how tuples can be concatenated. Notice that the output is a tuple that has taken the elements from the tuples to be concatenated and placed them in a new tuple. In other words the tuples themselves are not added together but the elements from the tuples.

The eighth print statement demonstrates how to access individual elements of a tuple. Like the indexing demonstrated above in relation to string objects, the index values begin at zero. Therefore, this print statement prints the element of tuple1 that has an index position of 3.

When tuples contain other tuples if the index value specified indexes a tuple then the entire tuple will be returned. This is demonstrated by the ninth print statement. In order to access individual elements of tuples within tuples then double indexing is needed. First the tuple within the tuple is indexed, and then the element within the nested tuple. This is demonstrated by the tenth print statement.

In order to access multiple elements within a tuple a method that is functionally the same as string slicing is needed. In the final print statement a slice of the tuple within tuple3 is referenced first by referencing the nested tuple at index point 0, and then the elements at index points 2 and 3 are printed. The same rules regarding default positions apply as in string slicing (see above).

Tuple elements can be iterated over using the for statement in the expected way i.e. `for element in tuple`.

In order to test membership of a tuple the `in` statement can be used. A Boolean value is returned. So, in the above examples the expression `3 in tuple1` would return a Boolean with value True.

Lists

Like tuples, lists in Python are a sequence of objects, not necessarily of the same type. They are probably the most flexible type of object in that they are mutable, meaning that they can be modified. This means for example that a list can be created and then a group of objects can be iterated over and then added to the list conditional on meeting the conditions specified by the programmer.

Lists are created in exactly the same manner as tuples except that square brackets are used rather than regular parentheses. Indexing and slicing are performed in the same manner, and as such new examples will not be given here as to those types of operation.

The mutability of list objects is extremely useful, but can also cause confusion. When thinking about immutable objects we can 'pretend' to alter them by binding a new object to a variable name already in use. However, although the variable name may be the same, in the computer memory the objects are in fact different. When a list is mutated however, the object is in the fact the same and has been changed. The important of this can be seen from the following example:

```
Tuple1 = (1, 2, 3)
Tuple2 = (4, 5, 6)
print "The id of Tuple2 in the computer memory is " + str(id(Tuple2))
Tuple3 = (Tuple1, Tuple2)
print Tuple3
Tuple2 = Tuple2 + (7,)
print Tuple2
print Tuple3
print "The id of Tuple2 in the computer memory is " + str(id(Tuple2))

List1 = [1, 2, 3]
List2 = [4, 5, 6]
print "The id of List2 in the computer memory is " + str(id(List2))
List3 = [List1, List2]
List4 = List1 + List2
print List3
print List4
List2.append(7)
print List2
print List3
print List4
print "The id of List2 in the computer memory is " + str(id(List2))
```

The example is a tedious one, but it makes a serious point. In the Tuple example two tuples are created, and a third tuple is then created who's two elements are the first two tuples. We then modify Tuple2 by adding a fourth element, the integer 7. However, although it appears that we are modifying Tuple2, in fact we are creating a new object called Tuple2 which contains the elements of another object called Tuple2 as well as the integer 7. That they are different object can be seen by examining the id numbers as they are stored in the computer's memory. The id of an object is accessed by using the `id(x)` statement. This means that upon reprinting Tuple3 after Tuple2 has been 'modified', nothing has changed. Tuple3 contains the elements 1, 2, 3, 4, 5, 6 even after the change to the object Tuple2.

The exact opposite is true in the list example. In that example List3 contains two elements which are List1 and List2. When we first print List3 it contains the elements [[1, 2, 3], [4, 5, 6]]. Then we modify List2 by appending the integer 7. Now when we reprint List3 we see that it contains the elements [1, 2, 3,], [4, 5, 6, 7]]. Notice that the id numbers for List2 have not changed. List2 remains the same object but can truly said to have been modified. The variable name List3 points to two lists in the memory, so if either of those lists change then List3 will also change

Notice also that List4 has been created by adding together the elements of List1 and List2. After the modification of List2, the elements of List4 have not changed. This is because the variable name List4 does not point to the Lists themselves. Rather List4 simply created a new list by taking the elements of List1 and List2 and concatenating them.

A number of extremely powerful built in methods are provided for working with lists:

Method	Example	Description

list.append(x)	`IntsTuple = (1, 2, 5)`
	`List1 = [1, 2, 3, 4]`
	`print List1`
	`List1.append(IntsTuple[2])`
	`print List1`
		Adds an item to the end of an existing list

list.insert(i, x)	`List2 = [1, 2, 4, 5]`
	`print List2`
	`List2.insert(2, 3)`
	`print List2`
		Inserts x into the list at index position i.

list.remove(x)	`List3 = ["IPA", 'IPA', 'J-PAL']`
	`List3.remove('IPA')`
	`print List3`
		Removes the first instance of x from the list. If no such elements is contained in the list then an error is generated.

list.pop([i])	`List4 = [2, 4, 6, 8]`
	`List5 = [8, 10, 12, 14]`
	`print List4`
	`print List5`
	`List5.pop()`
	`List4.append(List5.pop(1))`
	`print List4`
	`print List5`
		Removes the item at the specified index position and returns it. If no index position is provided as an argument then the last element of the list is removed and returned.

list.index(x)	`A = [4.2, 9.66, 7.2, 9.66]` `print A.index(9.66)`	Returns the index position of the first element of the list which has the value x. An error is generated if there is no such item.
list.count(x)	`A = [4.2, 9.66, 7.2, 9.66]` `A.count(9.66)`	Returns the number of times that x appears in the specified list.

list.reverse()	`Phrase = ["Lists", "Love", "You"]`
	`Phrase.reverse()`
	`Print Phrase`
		Reverses the order of the elements in the specified list.

Now let's look at some of the things that can be achieved with lists. The most useful aspect of the list is that they can be modified on the fly, and then once the list is complete it can be used in further branches of the program. Firstly, lets extend the prime number calculator from the above example. In its current format the output is not very useful as it is simply a number of print statements that tell the user if the numbers in the given range are

prime. If the programmer wanted to actually do something with the prime numbers then this would not be terribly useful.

```
x = int(raw_input("Enter a start point integer "))
y = int(raw_input("Enter an end point integer "))
PrimeList = []

for numbers in range(x, y + 1):
    divisor = 2
    IsPrime = 0
    if numbers == 1 or numbers == 2:
        PrimeList.append(numbers)
    else:
        while divisor <= numbers**0.5:
            if numbers%divisor == 0:
                IsPrime+=1
                break
            else:
                divisor +=1
        if IsPrime == 0:
            PrimeList.append(numbers)
print "The prime numbers in the given range are " + str(PrimeList)

SqRtPrimes = []
for primes in PrimeList:
        SqRtPrimes.append(round(primes**0.5, 2))

print "The square roots of the primes numbers in the given range are " +
str(SqRtPrimes)
```

Now the prime numbers in the given range have been captured in a list, and in the above example have been iterated over and another operation has been performed with them, in this case the square roots have been found and placed in a distinct list. Notice the use of the round() function which has rounded the roots to two decimal places. A list and description of useful built in functions such as round can be found here: http://docs.python.org/2/library/functions.html

Sticking with the same theme lets now write a program that generates a random list of numbers that is of a length specified by the user, and the random numbers being drawn from a range set by the user. The program will find the prime numbers in both those lists then eliminate any non-unique numbers and evaluate which set of random numbers contained the most unique primes. This is a somewhat more advanced program, which uses methods and other operations not yet seen. This is a beginners course, but for those interested it will be worthwhile to investigate further any of the language that is not familiar.

```
import random
ListLength = int(raw_input("Enter list length "))
x = int(raw_input("Enter a start point integer "))
y = int(raw_input("Enter an end point integer "))
ParentList = [[], []]
ParentPrimesList = []

for Lists in ParentList:
        for Iters in range(1, ListLength + 1):
                Lists.append(random.randrange(x, y))

        PrimesInList = []
        for number in Lists:
                divisor = 2
                IsPrime = 0
                if number == 1 or number == 2:
                        PrimesInList.append(number)
                else:
                        while divisor <= number**0.5:
                                if number%divisor == 0:
                                        IsPrime += 1
                                        break
                                else:
                                        divisor+=1
                        if IsPrime == 0:
                                PrimesInList.append(number)
        ParentPrimesList.append(PrimesInList)

print "The first list of random integers is " + str(ParentList[0].sort())
print "The second list of random integers is " + str(ParentList[1].sort())
print "The prime numbers in the first list of random integers is " + str(ParentPrimesList[0])
print "The prime numbers in the second list of random integers is " + str(ParentPrimesList[1])

UniquePrimesParent = []
for PrimesList in ParentPrimesList:
        UniquePrimes = list(set(PrimesList))
        UniquePrimes.sort()
        UniquePrimesParent.append(UniquePrimes)

print "The unique prime numbers from the first list of random integers is " +
str(UniquePrimesParent[0])
print "The unique prime numbers from the second list of random integers is " +
str(UniquePrimesParent[1])

if len(UniquePrimesParent[0]) == len(UniquePrimesParent[1]):
        print "The number of primes in both sets of random integers is equal"
elif len(UniquePrimesParent[0]) > len(UniquePrimesParent[1]):
        print "There are more primes in the first random integer set"
else:
        print "There are more primes in the second random integer set"
```

Briefly we will look at some aspects of this program that may be unfamiliar:

- `import random` – random is a python module. There are hundreds of modules in the Python library but they are not always available immediately upon opening the programming environment. In some cases they must be imported.
- First a parent list is created which is a list the contains two empty lists. Then each of these lists is iterated over the following operations are made upon each of the lists in turn:
    - The list is populated with random integers according to the user entered specifications
    - An empty list called PrimesInList is created
    - The numbers within the integer list are determined to be prime or not and if they are then they are appended to PrimesInList.
    - After interating over the numbers in the integer list the PrimesinList object is appended to the ParentPrimesList object.
- The list of prime numbers drawn from the integer lists are then mutated into sets. A set is an advanced object type. Technically a set object is an unordered collection of distinct hashable objects. Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.
- The print statements allow the user to follow the progress of the program.

If you feel that the code is starting to look a little chaotic and messy, then you are absolutely right. What we ideally want to do is break this program down into its constituent parts. This is exactly what we will do when we look at functions in the intermediate course.

So far the types of code we have been looking at do not really do anything that someone working for IPA/J-PAL might actually be interested. So now, lets see an example of what can be achieved with lists that might actually be useful. In the folder that contains this document there is a file called UIDSearchProgram together with a stat file, and a text file, both called DataSet. If you open the .py file in the editor there are notes contained in the file that describe what is happening. In order to run the program you will need to save the text file dataset and make sure you alter the file path in the code.

This code is interesting but could be greatly improved upon by using the final advanced object type that we will look at in the Intermediate course; Dictionaries.

PART IV: FURTHER TOPICS

Introduction

The course up to this point has given the tools needed to create some basic although potentially powerful programs. However, in order to write the types of programs that might be truly useful to you and your projects you will need to look to the intermediate course (which is not yet created). Therefore, below are some suggested topics and a brief introduction to them.

Dictionaries

Dictionaries are like lists except that their index values need not be integers. They are therefore referred to as keys. So for example, the key might be the UID of an individual, and then the entry could be different types of data associated with that UID which can then be accessed using a method similar to indexing. They are an incredibly powerful objects .

Functions

Being able to write user defined functions massively extends the power of Python. It also enables the programmer to break down problems into small bite size chunks, where each chunk solves a particular aspect of the problem and then passes the result to the next function for more work. They are essential for complex programs.

Exception Handling

When errors are thrown up by a program the program effectively crashes. However, exception handling allows the programmer to take these errors and do something with them. This prevents the program from crashing, and acts as a flow of control mechanism just like the if statement.

Classes

Function belong to classes. Being able to write user defined classes takes the level of abstraction one step higher, and the use of classes is important the more complex your programs become.