

1. Basic Concepts

1.1 What is a Program?

Believe it or not, you've programmed before! When? Every time you've given instructions of some sort. If you've ever given someone directions, you've programmed before. If you've ever told someone how to make toast or boil the kettle, or how to feed the dog or make a phone call, you've programmed before.

This is because a program is simply a **set or sequence of instructions on how to perform a particular task.**

If you are telling someone how to boil water in the kettle, to use one of our examples, you are "programming" because you are giving them **a sequence of instructions** on how to perform the task of boiling water in the kettle:

- 1.) Fill the kettle with water
- 2.) Plug the kettle in
- 3.) Switch the kettle on
- 4.) Wait for water to boil

Can you think of how to make these instructions more precise? What if the kettle is already full of water?

1.2 Okay, but what is a Computer Program?

Similarly, a computer program is **a set or sequence of instructions telling the computer how to perform a particular task.**

In most cases, this task involves doing something to **input data** (information that we give the program) in order to produce **output data** (information that the program gives us)

input --> program --> output 

We can imagine a basic program that subtracts one given number from another. This program consists of only two instructions, namely:

- 1.) Ask for two numbers, x and y
- 2.) Subtract Number y from Number x

or rather:

- 1.) Input two numbers, x and y
- 2.) $x - y$

Let's think a bit about this program.

- What input does this program need?

Our program needs two numbers as input. 5 and 3, 2 and 1.0, 100 000 000 and 7.5, and 1/8 and 57 are all examples of satisfactory input.

It is important to realise that the program determines what input is satisfactory, and what input will cause error. With our subtraction program, if we give something like "pineapple" or "B" as input, it will cause error. After all, what does it mean to subtract "pineapple" from "B"?!

- What does it *do* with this input (what *task* does it perform)?

Our program subtracts one given number from another. If we want to perform a *different* task (**add** two numbers together, for example), we would have to write a new program, like this:

1.) Input numbers x and y


2.) $x + y$

- What output does this program produce?

Our original program gives us the **difference** between the two numbers we gave as input.



Again, if we want our program to produce a different output (the **sum** of two numbers, for example), we would have to write a new program. We cannot change how the program works (what task it performs) by giving it different input.

Technically speaking...

The instructions that we give a computer fall into two groups: **expressions** and **statements** 

Expressions are instructions that **produce a result**. For example, the instruction

2 times 5

is an expression because carrying out this instruction produces a result: 10. We say that this expression **has a value** (of 10). Expressions -- or rather the **values** they have -- are useful in our programs because the program **more data** to work with. **Did you notice?** In our program example, instruction (2.) is an expression, since it produces the difference of the two numbers x and y.  

Statements, on the other hand, are instructions that **do not produce any result**.

For example, the instruction

Display the word "Hello" on the screen

does not produce any result; it simply shows the word "Hello" on our computer screen. No further data or information is produced when this instruction is carried out.

In a program, expressions can be combined with other expressions, to form **complex expressions**. We combine expressions using **operators** like sum (+), difference (-), product (*), quotient (/), and so on. An example of a complex expression would be


$$[(2 + 3.5) / 8] * 25$$

Can you break this complex expression into its step-by-step single expressions?

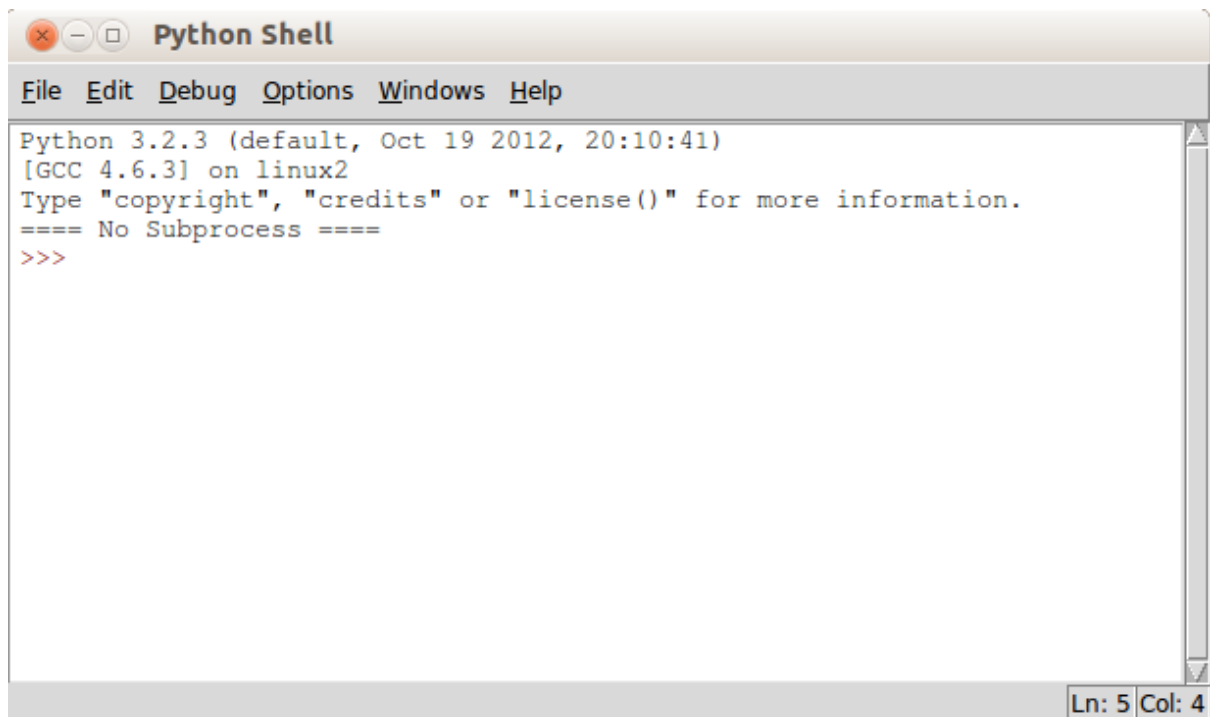
2. Getting Started with Python

Python is a programming language. A programming language is any language we use to give the computer instructions.

2.1 Starting IDLE

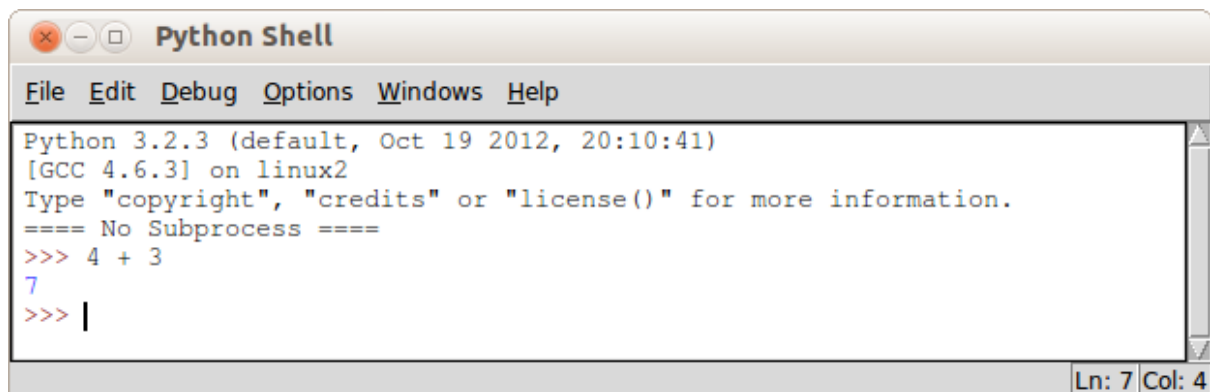
To get started with Python, we shall use a program called IDLE which includes a Python interpreter 

- 1.) On the Desktop, you will see a link called "Python"
- 2.) To open the Python interpreter, click on **Run > Python Shell**
- 3.) This should bring up the Python Shell, which looks something like this:



```
Python 3.2.3 (default, Oct 19 2012, 20:10:41)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>>
```

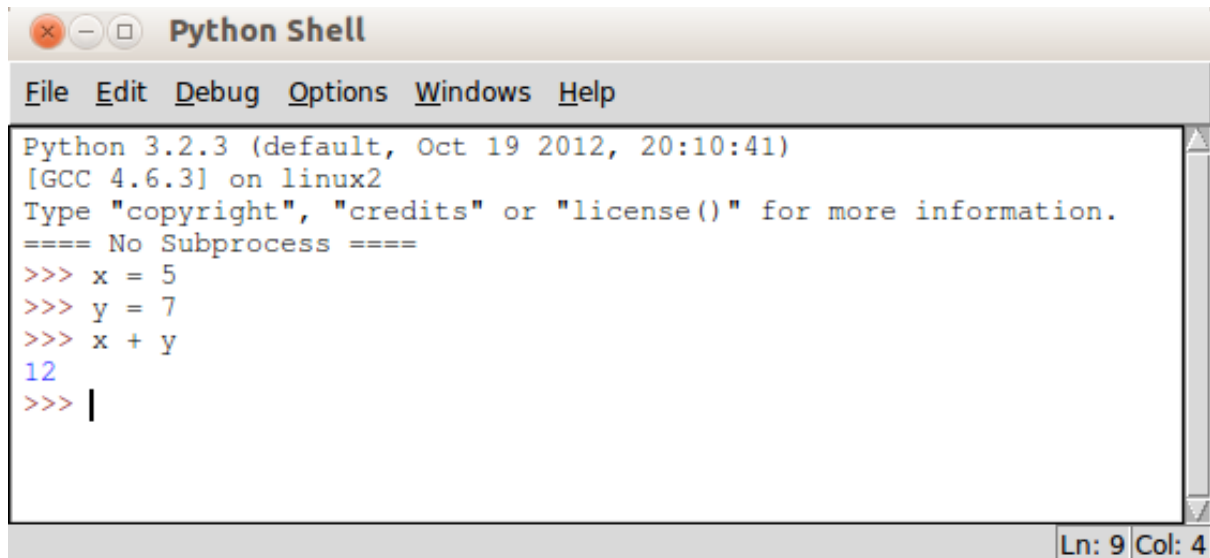
The Python shell is a bit like a calculator. You can enter Python expressions and see their results immediately. **Try it now!** Use the Python shell to add two numbers together:



```
Python 3.2.3 (default, Oct 19 2012, 20:10:41)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>> 4 + 3
7
>>> |
```

2.2 More than a Simple Calculator

If I told you that $x = 5$ and $y = 7$, and asked you to give me the sum of x and y , I'm sure you would know what to do. The Python shell is the same! Look at what happens if we type in $x = 5$ and $y = 7$, and then type in $x + y$:



```
Python 3.2.3 (default, Oct 19 2012, 20:10:41)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>> x = 5
>>> y = 7
>>> x + y
12
>>> |
```

Ln: 9 Col: 4

We can also type in longer expressions, for example, $a = 3 - 1$. **Try it!**

When we tell Python that $x = 5$, what we are doing is **assigning the value 5 to the variable called x** . In programming, a variable is a kind of placeholder for a value. We can easily change the value of a variable. Type the following into the Python shell and see what happens:

```
a = 5
a = 7
b = 2
a + b
```

The shell returns 9. This means that, when Python calculates $a + b$, it takes the new value of a which is 7, and not its old value of 5. This is important. When we change the value of a variable, its old value falls away.

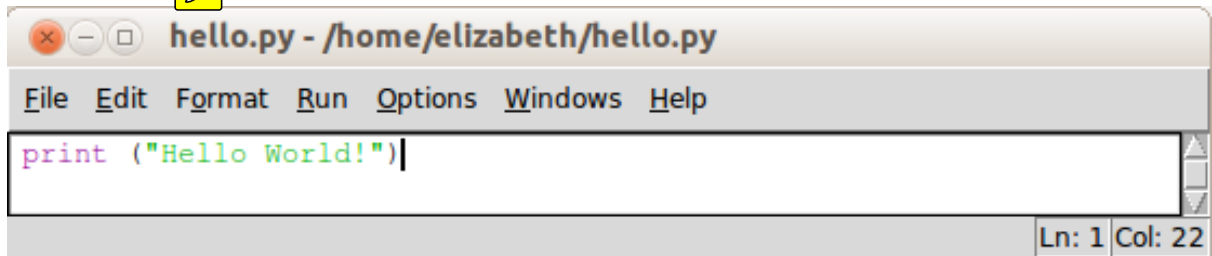
2.3 Running a Saved Python Script

The Python shell is fun for trying out quick calculations or simple one- or two-line programs. Still, we have to keep re-entering our program's instructions every time we want to run it.

Instead, we can save our program's instructions in a **text file**. Let's write our first .py file!

- 1.) From the Python Shell, click **File > New Window**

- 2.) Then click **File > Save**
- 3.) Navigate to `/home/username/Desktop`
- 4.) Save the file as **hello.py**
- 5.) Type in following:



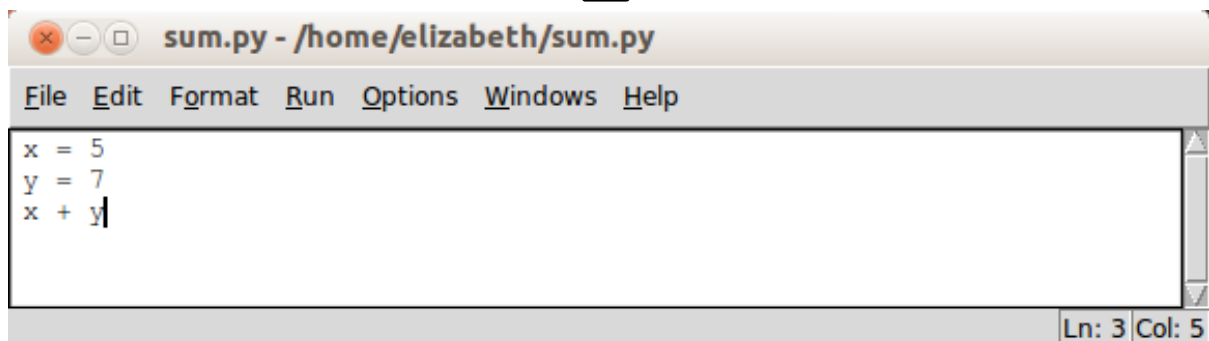
```
print ("Hello World!")
```

- 6.) Press **F5** or click **Run > Run Module**

Congratulations! You've just written and run your first Python program!

2.3.1 print()

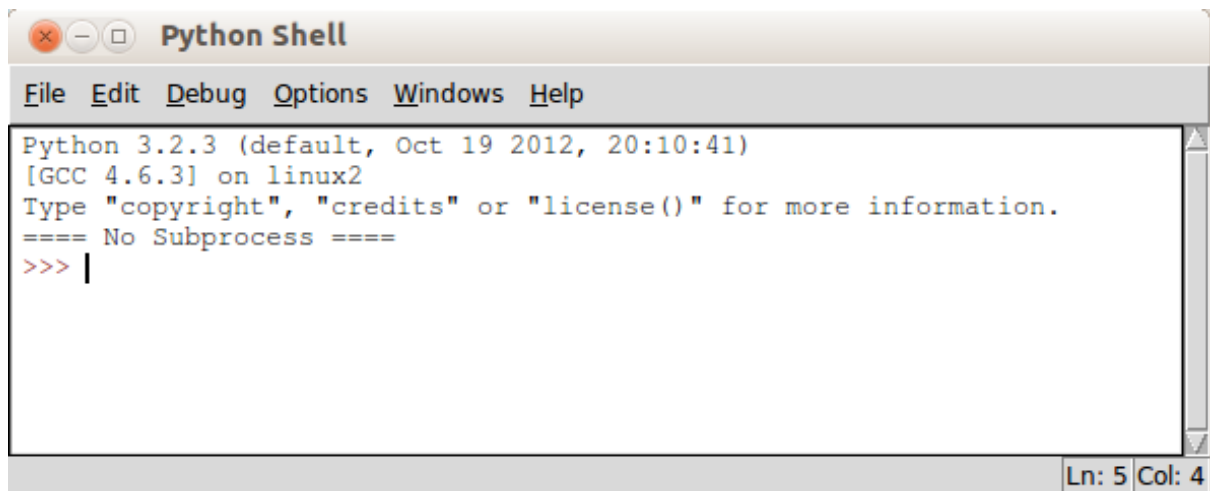
Remember our earlier `x + y` program? Let's save those instructions to file. Open a **New Window** and save it as **sum.py**



```
x = 5
y = 7
x + y
```

Now, it would be reasonable of us to expect that, if we ran **sum.py** we would get the same result as before. Python would give us the answer (12). Let's try it. Press **F5** or click **Run > Run Module** to run the program.

What happens?



```
Python 3.2.3 (default, Oct 19 2012, 20:10:41)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>> |
```

Ln: 5 Col: 4

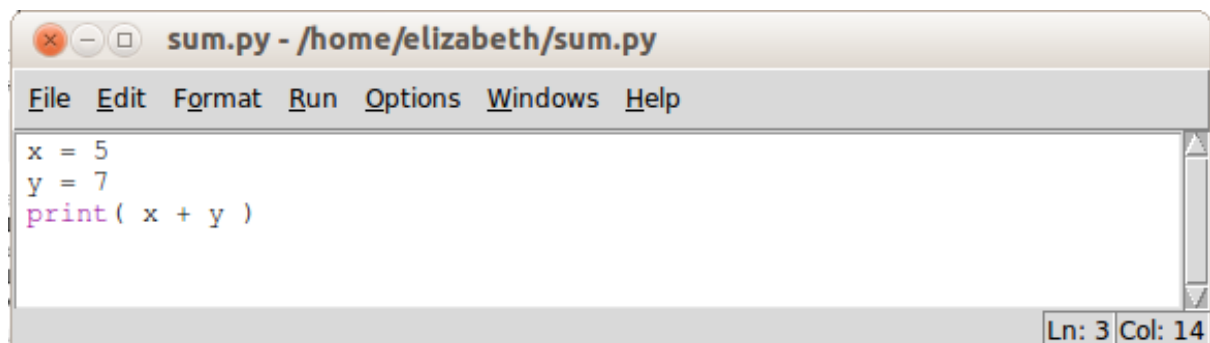
No output? Nothing?!

Why is this? Why did it work before? It worked before because we were typing into the shell itself. When we use the shell to **run a Python program** (a .py file) things are slightly different.

Here, the difference is that we must **explicitly tell the computer to output the result** of our `x + y` program. We must instruct the computer to show us the answer to `x + y`.

How do we tell the computer to output something? We can do this with a **print statement**. If you remember, we used one in our Hello World program (hello.py).

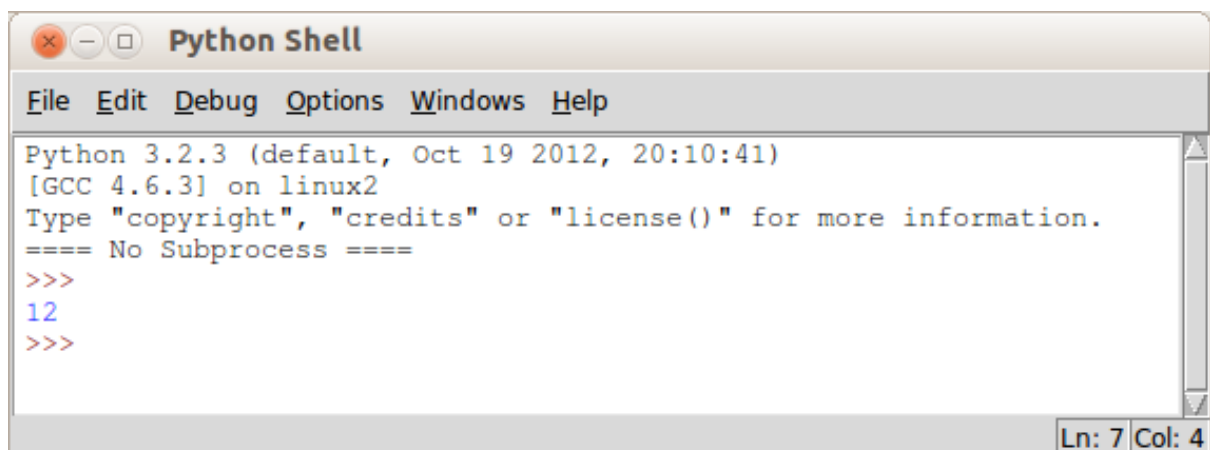
Let's modify the code in our sum.py file so that it looks like this:



```
x = 5
y = 7
print( x + y )
```

Ln: 3 Col: 14

And phew! It works!





```
Python 3.2.3 (default, Oct 19 2012, 20:10:41)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>>
12
>>>
```

Ln: 7 Col: 4

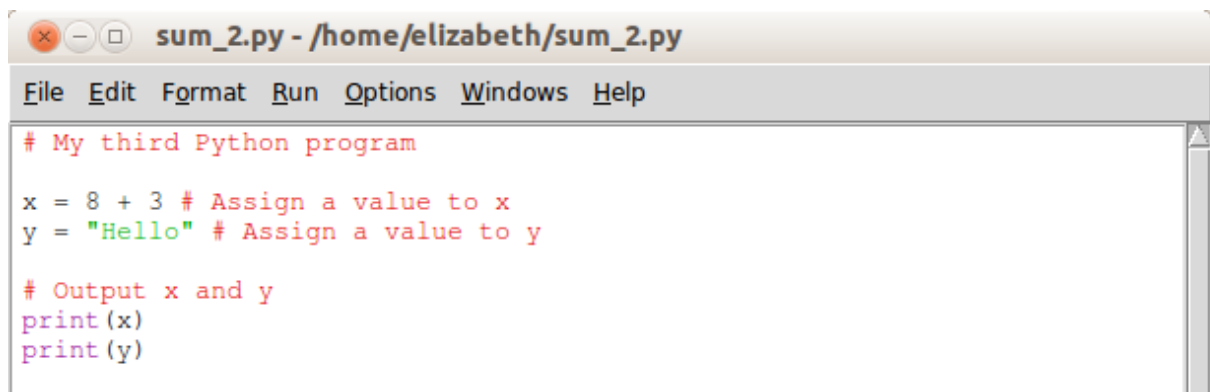
We'll come back to this useful print statement in the next section.

2.4 Two More Things...

There are two quick things for us to note at this point. The first is **comments** and the second is **case sensitivity** 

Comments are text that we can put into our program files. We can think of comments as little notes to explain our code and to make it easier for us to understand. Comments are ignored by the computer. 


In Python, a comment is any text that starts with a hash symbol: # Here is an example of a program with comments:




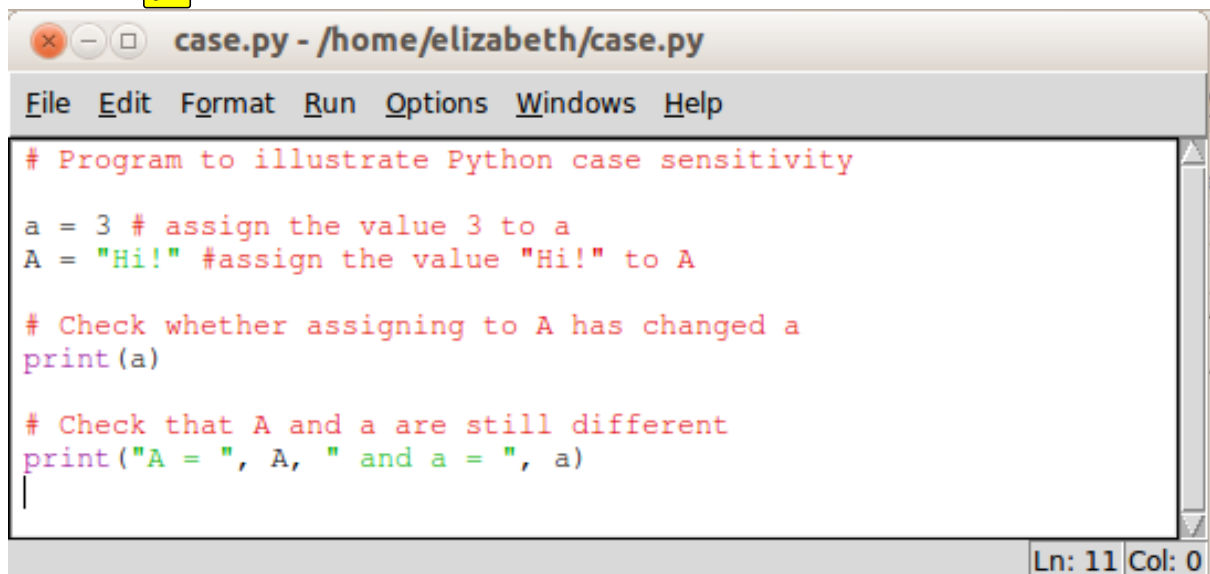
```
# My third Python program

x = 8 + 3 # Assign a value to x
y = "Hello" # Assign a value to y

# Output x and y
print(x)
print(y)
```

Comments are very useful for making our code more readable to other programmers (and to ourselves). It is a good habit to use comments in our programs. 

The second thing to note is that Python is **case sensitive**. This means that "A" and "a", and "x" and "X", are different things in Python. To illustrate, try out this program: 



```
# Program to illustrate Python case sensitivity

a = 3 # assign the value 3 to a
A = "Hi!" # assign the value "Hi!" to A

# Check whether assigning to A has changed a
print(a)

# Check that A and a are still different
print("A = ", A, " and a = ", a)
```

Ln: 11 Col: 0

What output does this program produce?

2.5 Exercises

- 1.) Start the Python shell (interpreter)
- 2.) Try using the interpreter as a calculator
- 3.) Write the Hello World program
- 4.) Write the program to add two numbers
- 5.) Write your own program that prints out your name
- 6.) Look at the following lines of code:

```
a = 9
```

```
b = 3
```

```
a/b
```

6 a.) What is the output if these lines are entered into the **Python shell (interpreter)**?

6 b.) What is the output if these lines are **run from a .py file**?

6 c.) Add another line of code to your .py file from (b.) so that it now shows us the output

3. Strings and Basic Output

3.1 The `print()` statement

In the previous section, we used `print()` to display our programs' output on the computer screen. Let's take a closer look at the `print()` statement.

In the Python **shell**, type the following:


```
print(1)

print(173 + 92)

print("hello")
```

Notice that the `print()` statement shows us the value of the expression inside its brackets. Instead of showing `173 + 92`, it outputs 265.

But what if we *wanted* to output `173 + 92`?

One way we could do this is to put `173 + 92` inside quotation marks, like so: `print("173 + 92")`. In programming, we call this kind of value a **string** 

What is a string?

A string is, quite literally, a *string* of characters. The string `"173 + 92"` is a string of the characters `"1"`, `"7"`, `"3"`, `" "`, `"+"`, `" "`, `"9"`, and `"2"` (notice the spaces). The string `"hello"` is a string of the characters `"h"`, `"e"`, `"l"`, `"l"`, and `"o"`.

What really matters when we're dealing with strings is that **they must be enclosed in quotation marks**. Try out the following:

```
print("173 + 92")          print("hello")

print(173 + 92)            print(hello)
```

If we print `173 + 92` without quotation marks, we get 265 instead of `173 + 92`. If we print `hello` without quotation marks, we get an error.


Another way we could output `173 + 92` instead of 265 is to use commas, like this `print(173, "+", 92)`. The `print()` statement can output multiple separate values, provided they are separated by a comma. Type out the following in the shell:

```
print("Jane has", 7, "apples")

print(5, "plus", 3, "is", 5 + 3)
```

These examples show us that:

(1.) We can output the values of many expressions using a single `print()` statement, as long as we put a comma between the expressions

(2.) The `print()` statement  puts a single space between the expressions we give it.

Using `print()` in this way can be very useful. Consider the following:

```
print("Hello, my name is Benji")          name = "Benji"
print("Benji is my name")                  print("Hello, my name is", name)
                                           print(name, "is my name") 
```

In both cases, the output we get is the same

```
Hello, my name is Benji
Benji is my name
```

The output is the same, but the second case is preferable. Think about it: if we wanted to change the name from `Benji` to another name, in the first case we would need to change both `print()` statements:

```
print("Hello, my name is Georgie")
print("Georgie is my name")
```

In the second case, however, we only need to change one line:

```
name = "Georgie"
```

The usefulness of this will become clearer in the next section, when we look at user input. For now, try out the following in the Python shell:

```
print("I'm learning Python!")

lang = "Python"
print("I'm learning", lang, "!")

print("Jane has 7 apples")

a = 7
print("Jane has", a, "apples")

print("Jane has 7 + 3 apples")
```

```
a = 7 + 3

print("Jane has", a, "apples")
```

Let's look at another feature of the `print()` statement. Type out the following in the shell:

```
print("This is on one line")

print("And this is on the next line!")
```

This shows us that the Python `print()` statement **ends on the next line**. This is clearly illustrated by an empty `print()` statement:

```
print("An empty print statement...")

print()

print("...means a blank line!")
```

CHECK this for Python 3.0?



One way to stop Python from ending a print statement with a new line is to put a comma just before the closing bracket, like so:

```
print("This is on one line" , ) vs end="" ?

print("And this is on the SAME line!")
```

3.4 Strings continued...

As we mentioned, Python treats things wrapped in quotation marks as **strings**. Either double quotation marks `" "` or single quotation marks `' '` can be used.

3.4.1 Multi-line Strings

All of our strings so far have been one-line strings. Sometimes however we might want to break our string across two or more lines, as in a poem:

```
Twinkle, twinkle, little bat!
How I wonder what you're at!
Up above the world you fly,
Like a tea tray in the sky.
```

As such we might prefer a multi-line string. We can form a multi-line string by using **triple quotation marks**, meaning three double or three single quotation marks to indicate the beginning and the end of the string. For example:

```
"""This is an

example of a multi-line
```

```
string!"""
```

If we do not use triple quotation marks in this case, we will get an error. Try it and see.

3.4.2 Escape characters

Another way that we can display strings across multiple lines is by using **an escape character** like so:

```
"This is an\n example of a multi-line\n string!"
```

The `\n` is an escape character. Basically, an escape character is something that we include within a string - *within* its opening and closing quotation marks - that tells Python how to display the string. If you print out the above string, you'll notice that the `\n` does not appear in the output.

There are many kinds of escape characters. Try out the following:

```
\t    \b    \\    \" or \'
```

e.g.) `print("What does \t do?")`

3.5 Exercises

HINT ! Is your program not working? Here are two things to check:

- 1.) Have you put quotation marks around your strings?
- 2.) Have you closed the parentheses (round brackets) of your `print()` statement?

1.) In the Python **shell**, do the following:

a.) Print (output) your first name

b.) Output your surname (family name)

c.) Output your first name followed by a space followed by your surname

d.) Create a **variable** called `firstname` (Note: no space!) and put your first name into it

e.) Create a variable called `surname` and put your surname into it

f.) etc etc

4. Basic Input

So far, our programs have been quite uninteresting. They consistently produce the same output (give the same results) because they consistently act on the same input.

This is because our input values have been set directly in our programs. In our programs, we have specified, say, that `x = 7` or that `name = "Benji"`. When input values are fixed in this way, we say they are **hard-coded**.

What if, rather than hard-coding our input values, we could tell our program to get input from somewhere? The simplest place for a program to get input is the **keyboard**, in the form of text entered by the user.

4.1 `input()`

In Python, the `input()` statement allows us to get input from the keyboard. Type the following into the Python shell – remembering the quotation marks!:

```
input("What is your name?")
```

What happens? Python asks you for your name! Type it in.

But now what? In order to do something with this keyboard input, we need some way to store or remember it so that we can use it later. We can do this with an **assignment statement**:

```
name = input("What is your name?")
```



Here we are assigning the input we get from the keyboard – your name, in this example – to the variable `name`. Now we can check if Python has remembered:

```
print(name)
```

And voila! It has.

4.1.1 `input()` and numbers

Now let's get some numerical input from the user:

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

This asks the user to enter their age. Now let's see how old they'll be next year.

```
print("Next year you'll be", age + 1)
```

Oh dear! We get an error. This is because Python *treats all input as strings*. Not to worry, though, because all we have to do is tell Python that this particular input **is a number and not a string**. We do this like so:

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

Remember the double closing brackets at the end!

Now we can try our print statement again:

```
print("Next year you'll be", age + 1)
```

And yay! It works.

Introducing Functions

You might notice that the `input()` statement is similar to the `print()` statement. Both use parentheses (brackets) in the same way, and both require "quotation marks" around their text. This is because `input()` and `print()` are **functions**.

In programming, a **function** is a defined set of instructions that performs a particular task. As we know, the `print()` function performs the task of showing output on the screen. The `input()` function performs the task of getting user input.

When we type statements like `print("Hello World")` or `print()`, we say that we are "calling" the print function. Calling a function executes its particular set of instructions.

This will become clearer in the later sections, when we write our own functions.

4.2 Exercises

1. a.) Using the `input()` statement, prompt (ask) the user for a number and assign that number to the variable `z`. *Remember, to assign a value to a variable, use the following format: `z = ...`*

1. b.) Print `z`

2. a.) Write a program that asks the user for their name and assigns it to the variable `username`

2. b.) Using the `print()` statement, output "Hello" followed by the user's name

2. c.) Write a program that asks the user for their age and assigns it to the variable `age`. *Remember to use `int(input())`!*

2. d.) Using the `print()` statement, output "Your current age is" followed by the user's age

3. a.) Write a program that asks the user to enter two numbers and prints the **sum** of those two numbers. *Hint: use two input statements and two variables*

3. b.) Write a program that asks the user to enter three numbers and prints the **sum** of those numbers.

3.c) Write a program that asks the user to enter two numbers and prints the **product** of those two numbers.

4.a) Bonus challenge: Write a program that converts temperatures from Fahrenheit (F) to Celcius (C). Ask the user for a temperature in Fahrenheit and output the Celcius result. *Remember, $1^{\circ}F = 9/5 * (1^{\circ}C + 32)$*

4.b) Bonus challenge: Write a program that converts distance in miles to kilometres. Ask the user for a distance in miles and output the distance in kilometres. *Remember, $1km = \sim 0.62 \text{ miles}$*

4.c) Double bonus question: Write a program that asks the user for some text and a number. Then, output the text a number of times equal to the number the user entered. For example, if the user enters "Hello" and 3, your output should be HelloHelloHello. *Hint: have a look at section 5.2.2*

5. Variables, Assignment and Operators

In the last sections, we have spoken a bit about variables and assignment statements. Here we are assigning the value 56.8 to the variable `x`:

```
x = 56.8
```

The following is another example of an **assignment statement**:

```
greeting = "Hello World"
```

Here, we are assigning the value "Hello World" to the variable `greeting`. In the preceding section, we used an assignment statement to save user input:

```
name = input("Please enter your name")
```

These examples give us some idea of how assignment statements and variables work. Let's step it up a bit. Can you guess what the following will do?:

```
name = "James"
```

```
name = "Jimbo"
```

Here we assigned "James" to the variable `name`, and then assigned "Jimbo" to **the same variable name**. Let's print `name` to see what its value is:

```
print(name)
```

As we see, the value of `name` is "Jimbo" and not "James". In fact, "James" has disappeared! This is because assigning a new value to a variable will destroy the previous value of that variable.

*One way we might understand variables is to think of them as boxes that can **hold only one item or value at a time**. In our example, we put "James" into the variable box first. If we then want to put "Jimbo" into the box, before we can do this we must take "James" out because the variable box stores only one thing at a time.*

With this in mind, try to guess what the following would produce:

```
greet = "Hi!"           a = 2
greet = "Hello"         a = 2 + 1
print(greet)            print(a)
```

Still, is there any way we can assign a new value to a variable **but somehow keep the old value**? The answer is yes, there is a way! Have a think about the following lines of code:

```
x = 6
```

```
y = x
```

```
x = "Hello"

print(x)

print(y)

print("x =", x, "and y =", y)
```

Here, we have assigned the new value "Hello" to the variable `x` **AND** we have managed to keep its old value, 6. Let's look at this step-by-step:

1.) `x = 6`

We create a new variable, called `x`, and assign it the value of 6

2.) `y = x`

We create another new variable, this time called `y`, and assign it a value of... well what exactly? In programming, variables are taken to be their current values. So `x` is considered to be 6, as this is its current value. Assigning `y` the value of `x`, in this case, assigns `y` the value of 6.

3.) `x = "Hello"`

We assign a new value to `x`.

4.) `print("x =", x, "and y =", y)`

Here we output the current values of `x` and `y`.

Step (2.) is the important step. Here we assign `x`'s current value to a new variable `y`. This allows us to assign a new value ("Hello") to variable `x` without losing the old value (6).

5.1 Incrementing variables

Let's look at one more trick with variables. Consider the following:

```
x = 1

x = x + 1
```

What is the value of `x` now? The value of `x` is now 2. In programming terms, we have increased, or **incremented**, the value of `x` by 1. Incrementing variables is very useful for loops, which we'll get to in a later section. For now, let's get play a bit. Type the following into the Python shell:

```
x = 0

print(x)

x = x + 1
```

```
print(x)

x = x + 1

print(x)

x = x + 1

print(x)
```

Can you see a pattern? Incrementing variables is very useful in programming. Most often, we use incrementation as a counter. You may have noticed this behaviour in the code above. Each time, `x` is getting bigger by 1,

Try out these in place of `x = x + 1`:

```
x = x + 2

x = x + 5

x = x - 1
```

In this last example, `x = x - 1`, we are decreasing the value of `x`. In programming lingo, we say that we are **decrementing** `x`. When we decrement variables, their value gets smaller. We use this if we need to count *backwards*.

Shorthand for incrementation

Incrementating (and decrementing) variables is very common in programming. For this reason, Python gives us a short way to express incrementation. Instead of writing `x = x + 1`, we can instead write `x += 1`. These two statements are equivalent. The same is for decrementing variables. Instead of `x = x - 1`, we can write `x -= 1`.

5.2 Operators

Of course, it is not enough to just assign values to variables. We want to *do* things to these variables. For interesting programs, we want to manipulate and change our program variables. We can manipulate variables using the various operators in Python.

5.2.1) Mathematical Operators

To change number values in Python (as in mathematics) we can use addition (+), subtraction (-), multiplication (*), and division (/) operators. These operators might look a bit different than we are used to. For example, to express multiplication in Python we use an `*` rather than the times sign, `x`.

Here are some common mathematical operators in Python (*assume `x` and `y` are integers or decimal point numbers*):

Addition: `x + y`

Subtraction: $x - y$

Multiplication: $x * y$

Exponents: $x ** y$ (Raising x to the power of y)

Division: Python has two operators for division, x / y and $x // y$. Try $5 / 2$ and $5 // 2$ to see what the difference is.

Modulo: $x \% y$ (The remainder of x divided by y)

The rules of BODMAS (Brackets Of Division / Multiplication, Addition / Subtraction) also apply to mathematical expressions in Python.

5.2.2) String Operators

We can also manipulate string values. Here are two common string operators (*assume s and t are two strings, and i is any whole number*):

Concatenation: $s + t$

Concatenation joins two strings together. For example: `"user" + "name"` gives us the string `"username"`

Repetition: $s * i$

This repeats `str1` i number of times. For example: `"dog" * 3` gives us the string `"dogdogdog"`

String Indexing

Recall that a string is simply a string of characters. We can access the individual characters of a string by **indexing**: `s[i]`. For example, `"Hello"[2]` gives us the character `"l"`.

Wait a minute... isn't `"e"` the second character of `"Hello"`, not `"l"`? It is, but in programming, **we always start counting from 0 rather than 1**. The characters in the string `"Hello"` are indexed as follows:

"H"	"e"	"l"	"l"	"o"
0	1	2	3	4

So, to index the character `"e"` in `"Hello"` we would use `"Hello"[1]`. Try this in the shell!

Now, how would you index the character `"o"` in `"Hello"`? You'd be right if you answered `"Hello"[4]`. But there is another way we can index the last character of a string. We can use the index `-1`, for example `"Hello"[-1]`. Try it! This is especially useful because we can index the last character of a string **without** knowing how many characters are in the string.

String Slicing

We can also access multiple characters from a string by string **slicing**: `s[i:j]`. Here, the numbers `i` and `j` are any whole numbers. `i` gives us the starting index of the slice and `j` gives us the index of the character just after the stopping point of the slice.

Let's look at the string "Hello World":

"H"	"e"	"l"	"l"	"o"	" "	"W"	"o"	"r"	"l"	"d"
0	1	2	3	4	5	6	7	8	9	10

For example, `"Hello World"[1:4]` gives us the string `"ell"`. You might have expected this to give us the string `"ello"`, but remember `j` is the index of the character just after the stopping point. We start at index `i` and continue to index `j - 1`.

Look at the following slices on the string "Hello World":

`"Hello World"[0:5]` gives us `"Hello"`

`"Hello World"[:5]` gives us `"Hello"`

`"Hello World"[6:10]` gives us `"World"`

`"Hello World"[6:]` gives us `"World"`

`"Hello World"[3:-1]` gives us `"lo Worl"`

As you might notice, `"Hello World"[0:5]` gives us the same slice as `"Hello World"[:5]`. This shows us that, if `i` or `j` is 0, we can just leave them out. The last example shows us that we can use negative indexing in slices.

Try these slices in the shell:

`"Hello World"[-5:]`

`"Hello World"[:-4]`

`"Hello World"[-5:-3]`

`"Hello World"[-7:-2]`

We can also change the step of the slice. For example, `"Hello World"[0:8:2]` gives us the string `"Hlow"`. This is the slice from index 0 to index 7, "Hello wo", counted (or sliced) in steps of 2: `"Hello wo"`.

We can think of string slicing like this: `string[start:stop:step]`

Try out the following in the shell:

```
s = "The quick brown fox"

s[4:]

s[4:9:]

s[4:9:2]

s[:2]

s[4:15:3]
```

5.3 Exercises

1.) What does the following statement do?: `x = 75`

^^ More assignment / variable examples?

2. a.) If `i = 8`, what is the value of `i / 3`? *Use the shell if you're not sure!*

2. b.) What is the value of `i // 3`?

2. c.) What is the value of `i % 3`?

3.) Write a program that asks the user for a number from 1 to 12. Output the times table for that number. *For example, if the user choses 3, output 3 6 9 15 ...*

4.) Write a program that outputs the letter "z" 1000 times on a single line

5. a.) If `s` is a string with the value "Harry's Hippie Hoedown", what is the value of: `s + ": tickets only R5"`? *Use the shell if you're not sure!*

5. b.) Using the same string `s`, what is the value of: `s + ": tickets only R" + "5" * 3`?

6. a.) Bonus question: What is the value of "ABBA was a Swedish band popular during the 80s"[0:4]?

6. b.) Bonus question: What is the value of "ABBA was a Swedish band popular during the 80s"[-15:-7]?

6. c.) Double bonus question: If `s` is a string with the value of "ABBA was a Swedish band popular during the 80s", what is the value of

"BAAB" + `s[4:11]` + "Danish" + `s[18:24]` + "un" + `s[24:-3]` + "90s"?

6.) Conditionals

Often, our decisions in life depend on certain conditions or circumstances. For example, whether or not I study this afternoon might depend on whether or not I have remembered my textbook. Consider the following statements:

If it is Tuesday, I shall go to tennis practice.

If there are more than five people, it's a party!

If I am tired or if it is past 11pm, I shall go to bed.

As these conditional statements show, we make decisions and change our behaviour based on certain conditions.

In our first example, I shall go to tennis practice **if it is Tuesday**. This is an **if**-condition. Whether or not I go to tennis practice depends on – is conditional upon – whether or not it is Tuesday.

In our second example, if there more than five people present

We can use similar conditional statements in our programs. With conditional statements, we can tell the computer to perform certain instructions if certain conditions hold. In other words, **we can program the computer to make simple decisions.**

(Still to finish)