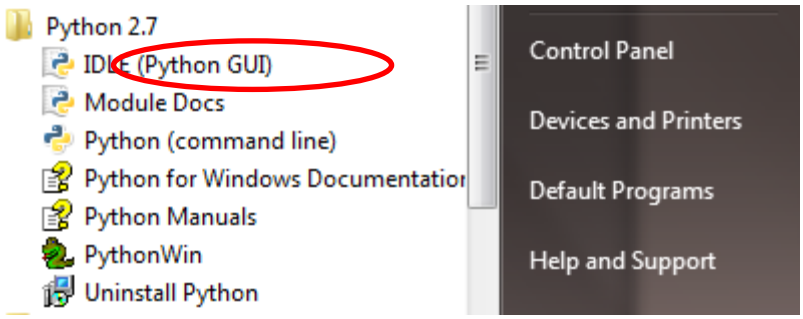
Name: Date:

Lesson 1: Strings and things

Getting started with the IDLE interpreter

On the Windows Start menu, you should find under the Python icon in "All Programs" an item called **IDLE** which will get you into the Python **interpreter** environment:



If you're having problems finding it, try typing "IDLE" into the Windows "*Search programs and files*" box:

Once you launch IDLE, you should see a >>> prompt which is where you enter Python code in single lines followed by an ENTER key:

```
Python 2.7.3 (default, Apr 10 2012, 23:31:26) [MSC v.1500 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

Congratulations! You're ready to write code! Let's start by typing the following **string** followed by ENTER**:**

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

You will notice that IDLE uses different colours for different text. For example our string is in green and our output is blue. If you're interested to find out what the colours mean you can try clicking the following menu options in IDLE: **Options -> Configure IDLE -> Highlighting**

TRY IT OUT #1: Try printing out your name and any other interesting strings you can think of.

If you make a mistake such as misspelling the Python **keyword** `print()` and calling it `pront()` instead, you will see that Python will raise an error indicating that `pront` is not defined:

```
>>> pront("Hello World")

Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    pront("Hello World")
NameError: name 'pront' is not defined
```

Simple Maths

We can do basic maths in the interpreter using **integers** like 1, 2, 3 and **operators** like + and -. When you press the ENTER key at the end of each line of code, the interpreter will output the answer. Here are a couple of simple examples:

```
>>> 1+1
2
>>> 1-1
0
```

In Python * is the multiplication operator:

```
>>> 2*2
4
>>> 2*2-2
2
```

TRY IT OUT #2: Try out some maths in the interpreter with +, - and *

Variables

If you want your Python code to remember something so you can use it later, you have to give that thing a name. Python will handle creating a location for that thing in computer memory. You assign a value to a name using an = sign. The name is often referred to as a **variable** in computer programming languages. It can be any set of characters (other than spaces) which are suitable for a name. Variables will be of different type depending on assignment. Often in Python we use `i` or `n` to refer to integer variables and `s` for string variables. The type of variable can be changed at any time through another assignment as a variable is just a name.

Now let's try creating a **string** variable in Python which represents a sequence of **characters** for example representing a word or a sentence or just a number of letters. You can use either double quotes " " or single quotes `''` to contain a string:

```
>>> s="Hello World!"
```

In this code, `s` is our variable which has been set to the string "`Hello World`". Remember we could have used any sequence of letters to represent our string. eg. `myStringThing="Hello World"`. Now we can print `s`:

```
>>> print(s)
'Hello World'
```

How long is this string?

```
>>> len(s)
11
```

TRY IT OUT #3: Create some strings and measure their lengths. Can you think of some ways you can make your strings longer without typing them?

You can add and even multiply strings. You do this using the Python + and * operators:

```
>>> s="Hello" + " World"
>>> print(s)
Hello World
>>> print(s*2)
Hello WorldHello World
```

Did you notice how we **reassigned** variable s? And did you see that `s*2` string thing? We're multiplying a string by 2!

String slicing

You can also **slice** a string into separate pieces. Before you do so, it's very important to understand that strings start from 0 and go up to len-1. So a string of length 5 has 5 separate **elements** which go from 0,1,2,3,4. For example the string "`Hello`" would consist of five characters "`H`", "e", "`l`", "`l`", "`o`" at **offsets** 0,1,2,3,4 in the string. To slice a string we use a colon to separate the start and end point for the slice:

```
>>> s="Hacker T. Dog"
>>> print(s[0:1])
H
>>> print(s[7:9])
T.
>>> print(s[10:])
Dog
>>> print(s[-3:])
Dog
```

Wait a moment, what was that last thing?! We used a negative value to indicate we're counting *backwards* from the end of the string. There's another fun thing you can do with slices - change the **increment** for the loop which is 1 by default. You do this with a second colon**:**

```
>>> s="Hacker T. Dog"
>>> print(s[::2])
Hce .Dg
```

TRY IT OUT #4: Try out some cool slices of string forwards and backwards if you like and with different increments.

You can loop over a string character by character. This is called **iteration** and is a very important operation in all computer programming languages. In Python it is done with a `for` loop:

```
>>> for c in s:
...     print(c)
...
H
a
c
..
```

There are a couple of important things to pick up here. First of all, the `for` loop line ends with a colon. Secondly, the code to run within the `for` loop is **indented** with a TAB character

represented here with an **ellipsis** … . The TAB is something that IDLE automatically does for you after you press carriage return. IDLE will also raise an error if you miss the colon:

SyntaxError: invalid syntax

Indented code blocks are a key feature of Python that you will need to get comfortable with as you progress.

Introduction to functions

It would be useful to package code up so you could reuse it a bit like you can reuse and combine different types of lego brick. Python lets you do that with a **function**. We'll do a lot more with functions in the next lesson but for now here's how you **define** and **call** a function in the interpreter. Note how can pass in **parameters** to a function and get the function to **return** values – in this function we are passing in `n` (an integer) and returning `n*n`:

```
>>> def square(n):
...     return n*n

>>> square(4)
16
```

Python has a number of built-in functions which you can use. For example `range(a,b)` returns a list of integers from `a` to `b-1`. We can use range in a `for` loop to **iterate** a loop counter which is normally called `i`. The loop counter runs from `a` to `b-1`. If `range()` is called with just ONE parameter, it sets `a` to 0 and `b` to that one parameter.

TRY IT OUT #5: Let's get loopy again! Try and write a `for` loop that prints the 7 times table. To do this you will need to use the `range()` built-in function.

Here's the solution:

```
>>> def SevenTimesTable():
...     for i in range(1,13):
...         print(i*7)
...
>>> SevenTimesTable()
7
14
21
28
35
42
49
56
```

```
63
70
77
84
```

We can use a parameter to **generalise** the above code to an `nTimesTable` function:

```
>>> def nTimesTable(n):
...     for i in range(1,13):
...         print(i*n)
...
```

It's worth spending some time studying this code – it's the most complicated code you've seen yet and it contains most of the concepts we've explored.

Bringing it all together ….

We've reached the end of the lesson material for today. Well done for keeping up to here! All that remains is our first Secret Code quest! Here goes:

SECRET CODE QUEST!

Write a function called `reverseString()` that takes a string parameter and returns a reversed string. You could use a loop or you could use string slicing. We can use this function to write or **encode** our first secret messages to each other – they're all backwards! You could write out your encoded messages on paper and use Python to help you **decode** them back to their unreversed form. Can you think how you would write a decode function?

```
>>> reverseString("Good Luck!")
!kcuL doog
```