Name: Date:

Lesson 2: Function junction

Basic functions

A **function** in computer programming is a block of organised, reusable code that is used to perform a single related action. It is contained within a definition or **def** statement in Python which can be **called** with **parameters**. Let's start this lesson with a simple function that adds 3 to any number that is passed in. In the code sample below, note what happens if we try to call `addThree()` with a string parameter. We get an **exception** (error) raised by the interpreter because it cannot **concatenate** (add) an integer and a string:

```
>>> def addThree(n):
        return n+3

>>> addThree(7)
10
>>> addThree('monkey')

Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    addThree('monkey')
  File "<pyshell#2>", line 2, in addThree
    return n+3
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

Here's another function called `multiplyByTen()` that returns the input parameter multiplied by 10. Note how this function is able to operate with a string as input:

```
>>> def multiplyByTen(n):
        return n*10

>>> multiplyByTen(7)
70
>>> multiplyByTen('monkey')
'monkeymonkeymonkeymonkeymonkeymonkeymonkeymonkeymonkeymonkey'
```

TRY IT OUT #1: Try writing a function that returns the multiple of two numbers passed in as parameters. Can you also write a function that returns the cube of a number?

Lists

From Lesson 1 you recall that a **string** in Python represents a sequence of **characters** for example representing a word or a sentence or just a number of letters. In Python a **sequence** of anything is called a **list**. Here are a couple of lists – the first variable `foo` is a list of **integers**, the second list `bar` is a list of **strings**:

```
>>> foo=[1,2,3,4,5,6,7,8,9,10]
>>> bar=['doobie','doobie','doo']
```

A list contains individual items separated by commas and the whole list is surrounded by square brackets. An empty list is defined as just `[]` with no elements. You can add or remove items from a list using **append** and **remove**:

```
>>> bar.append("who")
>>> bar.append("are")
>>> bar.append("you")
>>> bar.append("you")
>>> bar.remove()
>>> bar
['doobie','doobie','doo','who','are','you']
```

You can **iterate** every item in the list using a `for` loop:

```
>>> bar=['doobie','doobie','doo','who','are','you']
>>> for el in bar:
        print(el)

doobie
doobie
doo
who
are
you
```

TRY IT OUT #2: Try building some strings. Can you build a list with both strings and integers in it? Can you think how you would reverse a list?

Functions of functions

Functions are very important in Python and can be combined in really powerful ways. There are also many **built-in** functions which you get for free. We met `range(a,b)` already in Lesson 1 which returns a list of integers from a to b. Another example of a built-in function is **sum** which takes a list and returns the sum of all the values:

```
>>> print(sum([1,2,3,4,5,6,7,8,9])
```

If you ever need to check how a built-in function works, you can try calling up its documentation which you can do by calling the function **doc string** using double underscore either side of 'doc':

```
>>> sum.__doc__
"sum(sequence[, start]) -> value\n\nReturns the sum of a sequence
of numbers (NOT strings) plus the value\nof parameter 'start'
(which defaults to 0).  When the sequence is\nempty, returns
start."
```

TRY IT OUT #3: Write a function that calculates the sum of the squares of the numbers from 1 to 10. You will need to first write a function called `square()` and then use `sum()` and `range()` with it.

Here's one way to do it using an empty list that you build up with append to eventually hold the squares before calling `sum()` at the end. Study it and make sure you understand how it works:

```
>>> def sumSquares(l):
      ls=[]
      for n in l:
         ls.append(n*n)
      return sum(ls)

>>> sumSquares(range(1,11))
385
```

There's an even way of doing this in just one line of code using the built-in **map** function which is called with TWO parameters – the function 'square' which **map** applies to every element in a list and the list itself, `[1,2,3,4,5,6,7,8,9]`:

```
>>> sum(map(square,range(1,11)))
385
```

Have a good look at that code - it's incredibly powerful stuff and a good example of how functions can be used **combinatorially** in Python.

Control flow

In Python, basic **control flow** is supported through an `if … else` statement. If a certain **condition** is satisfied, then the following code block is executed. Typical conditions involve comparison with **greater than** operator > or **less than** operator < or **equivalence** operator `==`. As with `for` loops, the code block starts with a `:` and is **indented**. Here is an example with a function called `biggest()` that will return the biggest of two input parameters. Note how this function is able to compare two strings and return the one which is bigger in alphabetical order:

```
>>> def biggest(a,b):
        if a>b:
           return a
        else:
           return b


>>> biggest(7,10)
10
>>> biggest('alpha','gamma')
'gamma'
>>> biggest('alpha','aardvark')
'alpha'
```

TRY IT OUT #4: Try writing a function that returns the biggest number in a list of numbers using an `if … else` block

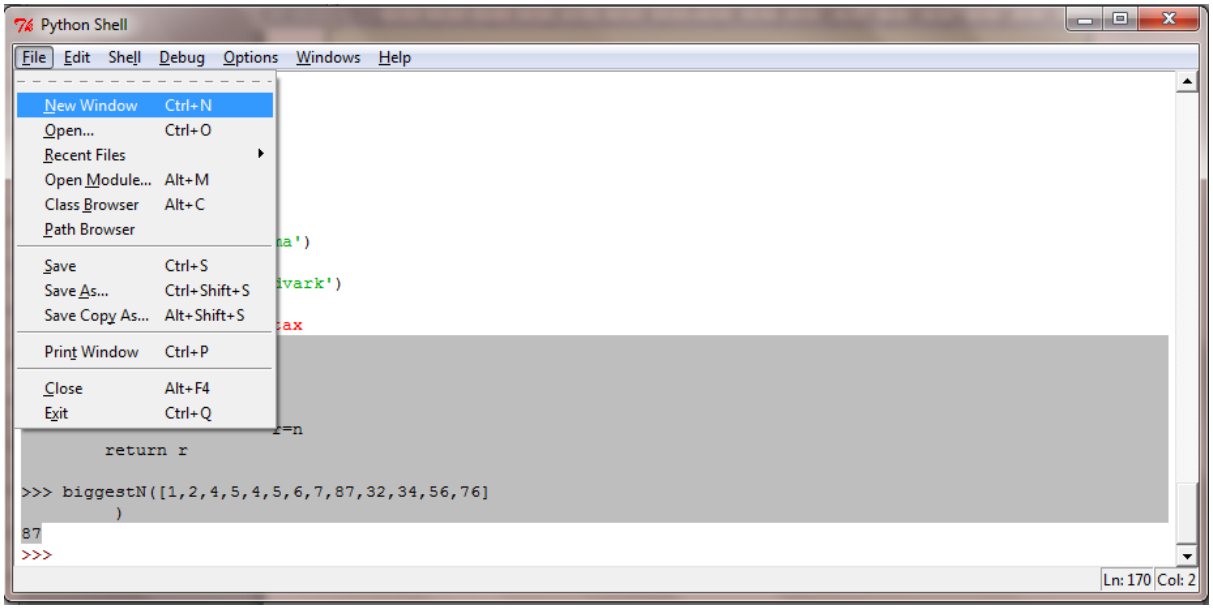
Here is a solution you should study and make sure you understand:

```
>>> def biggestN(l):
    r=0
    for n in l:
         if n>r:
              r=n
    return r

>>> biggestN([1,2,4,5,4,5,6,7,87,32,34,56,76])
87
```

IDLE edit mode

By this stage you're probably finding the interpreter environment difficult to work with now you're writing multi-line functions with `if … else` and `for` loops. Every time you make a mistake you have to start again with your code ☹ When we start developing larger pieces of code, we need to switch to developing code in **text files**. We can use IDLE to do this. If you click on **File -> New Window** you will launch an editor window.



Once you've written some code in that window, you will need to save it to a file using **File->Save**. The file should end with "`.py`". You should look to save your code to the Desktop and make sure you are able to copy it over to your network share. Once saved to a file, you can run the code

TRY IT OUT #5: Open an IDLE editor window. Copy over the `biggestN()` function code from the last section into that window and save the file to a file called "`test.py`" in your Desktop. Try running that file and seeing what happens. If you have problems running Python code written to a file it's often because you have something wrong with the spaces!

Here's what you should have in your file:

```
def biggestN(l):
    r=0
    for n in l:
        if n>r:
            r=n
    return r

result=biggestN([1,2,4,5,4,5,6,7,87,32,34,56,76])
print(result)
```

Bringing it all together ….

Well done for keeping up to the end of today's lesson. We'll finish with another Secret Code quest! Here goes:

SECRET CODE QUEST!

Write a function called `encodeString()` that takes a string parameter and returns a number list with every letter replaced by a code number. We can use a built-in function called `ord()` to generate the code number for every character in the string:

```
res=encodeString("Secret string!")
print(res)
```

```
[83, 101, 99, 114, 101, 116, 32, 115, 116, 114, 105, 110, 103,
33]
```

If you find that really easy, try writing the parallel `decodeList()` which takes the number list and returns the original string. To achieve that you will need to use another built-in function called `chr()` which converts a code number back into a character. Next week we'll study both `ord()` and `chr()` in more detail and learn how they can be used to recreate one of the oldest known military **ciphers** used by Julius Caesar's Roman army 2000 years ago!