**PYTHON Programming Exercises 2**

Last time we learnt some of the basics of PYTHON (including the print function; mathematical operators; defining variables; working with text strings; saving and executing your code as a .py file; the importance of using # comments to help us explain our code; Boolean expressions; and the use of ready-built code libraries (Math & Turtle Graphics) to help us out. In addition, we were introduced to some simple programming constructs such as the FOR loop (which we’ll consider in a bit more detail in Exercise Sheet 3). Each of these concepts will be reinforced in future exercises, as they form the basic building blocks for constructing some useful programs, i.e. the “nitty-gritty” elements of Python. (Feel free to take a couple of minutes opening up last week’s tutorial sheet to refresh your memory of these).

In today’s exercises, we will consider more of the nitty-gritty elements introduced in Lecture 2, including more working with Strings; Lists; and Tuples.

**Task 1: More Working With Strings**

We’ve already seen how to handle strings (concatenate as new variables; and working with different case types). Let’s recap and then take this a little further.

Open up a Python Shell Window (the one with the >>> Prompt). Simply type (**with your own First and Last names ALL in lower case**):

>>>name=”david”

>>>surname=”kidner”

>>>code=”is4s761”

>>>module=”principles of computing”

>>>print(name,surname,code,module)

>>>print(name+surname+code+module)

>>>student=name + “ “ + surname + “ “ + code + “ “ + module

>>>print(student)

>>>student=student.upper()

>>>print(student)

(N.B. You can use the UP arrow on the keyboard to re-select (or edit) an earlier command to save having to type it each time).

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters. However, Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets can be used to access elements of the string. OK, so how many characters are now in the student string? We can use the len (for length) command:

>>>print(len(student))

Can we print out the first character of our string? Type:

>>>print(student[1])

Remember, the first character in a string is at position 0, not position 1.

>>>print(student[0])

We can also output substrings between characters at different positions where [from : up to] defines the range:

>>>print(student[0:1])

Try these also:

>>>print(student[1:4])

>>>print(student[:30])

>>>print(student[10:100])

>>>print(student[0:len(student)])

>>>print(student[10:])

>>>print(student[-1])

>>>print(student[-9:-1])

>>>print(student[-9:])

>>>print(student[-9:len(student)])

>>>print(student[-0])

OK, assume the module code has now been updated from IS4S761 to CS4S761, so …

>>>print(student.replace(“I”,”C”))

Oooops! Or being a bit more specific …

>>>print(student.replace(“IS”,”CS”))

We could have checked this beforehand …

>>>print(student.count(“I”))

>>>print(student.count(“IS4”))

There are many other operators which work on Strings. These can be important when working with data (as we’ll see in later exercises) and when we need to search for specific instances of text, codes or data input. See: <https://www.programiz.com/python-programming/methods/string> for a list of some of these. In the meantime, here’s a few more to experiment with:

>>>print(student.index(“IS4”)) # Finds index of instance of IS4

>>>pos=student.index(“IS4”) # Saves index as a variable pos

>>>print(student[pos:pos+7]) # Prints Module Code

>>>code=student[pos-1:pos+8] # Assigns code to Module Code?

>>>print(code)

>>>code==”IS4S761” # Is code IS4S761? True or False?

When writing code in relation to text strings, we need to be careful of “white space”, i.e. blank characters. Try again with the spaces …

>>>code==” IS4S761 “

>>>code.lstrip()==”IS4S761 “ # Strip out left most space

>>>code.rstrip()==” IS4S761” # Strip out right most space

>>>code.strip()==”IS4S761” # Strip out all spaces

These types of string manipulation are useful when checking user input before it’s stored in a program, e.g. a postcode.

**EXERCISE 1:** Implement 4 or 5 other string operators from the list on <https://www.programiz.com/python-programming/methods/string>. For example, how would you test that the last 3 characters of the module code “IS4S761” is a number, or not a character string? How would you test that the module code starts with “IS”?

**Task 2: Working With Lists**

Lists are one of the most common types of data structures you will encounter in Python. A list is simply a collection of items, or data if you prefer, that can be accessed as a whole, or individually if needed. They are very flexible (and sometimes confusing), as a list can be strings, integers, and variables – or combinations of arbitrary values. You can even include functions in lists, and lists within lists.

Open up a Python Shell Window (the one with the >>> Prompt).

>>>numbers = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

>>>names = [“David”, “Gaylor”, “Mark”, “Sue”, “Nathan”, “Eric”]

>>>print(numbers)

>>>print(names)

>>>print(numbers[0:4])

>>>print(names[-2])

>>>print(len(names))

Essentially, we can access or index items in a list in the same way as we did for text strings. However, we can sometimes forget that we are not specifically dealing with text strings or the individual item in a list, for example: what command would you use to print the list element of “David” in uppercase?

>>>print(names[0].upper())

We can also easily add new items to a list in one of two ways:

>>>numbers=numbers+[55]

>>>numbers.append(89)

>>>print(numbers)

>>>names=names+[“Eleri”]

>>>names.append(“Jim”)

>>>print(names)

>>>everything = names + numbers

>>>print(everything)

In this case, we can join two lists of different data types. We can also remove items from a list in one of two ways:

>>>del everything[18] # delete by item/index number

>>>everything.remove(89) # delete a specific value

>>>del everything[0] # delete David

>>>everything.remove(“Jim”) # delete Jim

>>>print(everything)

If we try to put David back into the list:

>>>everything[0]=”David” # Insert David into 1st position

>>>print(everything) # Overwrites Gaylor!

>>>everything.insert(1, ”Gaylor”) # Slots Gaylor into 2nd position

Another useful function for lists is to “pop” (i.e. remove) the last item in a list. This works through a list and stores the last item in a variable for us to work with.

>>>print(names) # ['David', 'Gaylor', 'Mark', 'Sue', 'Nathan', 'Eric', 'Eleri', 'Jim']

>>>popped = names.pop()

>>>print(popped) # Jim

>>>print(names) # ['David', 'Gaylor', 'Mark', 'Sue', 'Nathan', 'Eric', 'Eleri']

By default, pop will remove the last item in the list, but can be indexed to remove any item in the list.

>>>popped = names.pop(2)

>>>print(popped) # Mark

>>>print(names) # ['David', 'Gaylor', 'Sue', 'Nathan', 'Eric', 'Eleri']

For some operations, we might prefer to work on a sorted (e.g. alphabetical order) list. This can be temporary ***sorted()*** or permanent ***sort()***. If you popped Jim and Mark from Names, let’s add them back and sort the list:

>>>names.append(“Jim”)

>>>names.append(“Mark”)

>>>print(names) #['David', 'Gaylor', 'Sue', 'Nathan', 'Eric', 'Eleri', 'Jim', 'Mark']

>>>print(sorted(names)) #['David', 'Eleri', 'Eric', 'Gaylor', 'Jim', 'Mark', 'Nathan', 'Sue']

>>>print(names) #['David', 'Gaylor', 'Sue', 'Nathan', 'Eric', 'Eleri', 'Jim', 'Mark']

>>>names.sort()

>>>print(names) #['David', 'Eleri', 'Eric', 'Gaylor', 'Jim', 'Mark', 'Nathan', 'Sue']

>>>names.sort(reverse=True)

>>>print(names) #['Sue', 'Nathan', 'Mark', 'Jim', 'Gaylor', 'Eric', 'Eleri', 'David']

>>>names.reverse()

>>>print(names) #['David', 'Eleri', 'Eric', 'Gaylor', 'Jim', 'Mark', 'Nathan', 'Sue']

Finally (for now), you can also use the list function to break a string down into its component characters. This can be useful when we are “parsing” text when searching for particular instances. Try these:

>>>myname=list(“David Kidner”)

>>>print(myname)

>>>months=list(“JFMAMJJASOND”)

>>>print(months)

**Task 3: Working With Tuples**

Tuples are very much identical to lists. However, as we’ve seen, where lists can be updated, deleted, or changed in some way, a tuple remains a constant. This is called immutable and they’re perfect for storing fixed data items. They are created in the same way as Lists, but we use normal curved brackets **()** instead of square brackets []. Try the following:

>>>fruit = (“banana”, “raspberry”, “orange”, “apple”)

>>>print(fruit)

>>>print(len(fruit))

>>>print(fruit.index(“raspberry”))

>>>print(fruit[1])

>>>fruit[1]=”cherry” # What Happens?

**THAT’S ALL FOR NOW!**