**Loops and lists**

Lists are a data structure in python that can hold multiple values of varying data types. Lists are also known as ‘arrays’ in other programming languages. It is important to note that each value in a list is known as an ‘element’ and each one has associated with it an ‘index’, which is an integer.

**List.py**

days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]

print(days[0], days[4])

Lists are declared much like variables and don’t have to initially contain anything, for example:

days = []

would initialise an empty list to which we can add values later.

The elements can vary in type as I’ve said, so a list may contain a string, integer, float and boolean. They can even contain other lists.

To access elements of a list we reference the list’s name and then the index associated with the element’s position within it. In the above example you should see ‘Monday’ and ‘Friday’ outputted. Most programming languages zero-index which is to say they start counting from 0 for lists. Therefore ‘0’ is ‘Monday’, ‘1’ is ‘Tuesday’ and so-on.

Let’s add some interactivity to list.py.

choice = int(input("Which day of the week? 1-7: "))

# We subtract one because arrays are zero-indexed

print("Day:", days[choice - 1])

With this we can select any day of the week that we want. It is important to force our user input to be an integer value since that’s how list elements are accessed. Any other value would throw an error.

Before we move on to loops let’s have a look at a 2 dimensional list.

**2d\_list.py**

person\_one = ["James", 26]

person\_two = ["Bill", 23]

person\_three = ["Andrew", 34]

# Declare empty people list

people = []

# Append each person to the people list

people.append(person\_one)

people.append(person\_two)

people.append(person\_three)

print("People list:", people)

print("2nd person's name:", people[1][0])

print("3rd person's age:", people[2][1])

This time we have two indices to reference. The first selects the relevant sub-list and the second selects the appropriate element within that list. We can also use ‘len()’ to get the number of elements that are contained within a list. This even works if we did ‘len(people[2])’ to get the length of a sub-list.

Another built-in function we are using here is ‘.append()’. It allows you to add a new element to the end of a list.

There is a lot we can do with lists but it would be a bit much to detail all the methods here. It is likely that we will encounter some of these things in later exercises.

**Loops**

The ‘for’ loop is the first loop we will learn and it is quite useful for iterating over lists or sequences of numbers. Let’s see what we can do using our ‘days’ list from earlier.

**For\_in.py**

days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]

for day in days:

print("\nDay:", day)

The line ‘for day in days:’ is basically saying ‘for each element in the list’ and allows us to execute some code for each element present. In this example we are simply printing out each day one at a time.

Interestingly, strings are similar to lists. Add the following to the above script (the ‘for’ loop should be indented at the same level as the print statement above).

for letter in day:

print(letter)

The script now prints out each day then each letter of that day individually.

We also have another kind of loop available to use, the ‘while’ loop. This executes a piece of code over and over until a condition becomes True.

**More\_loops.py**

text = ""

# While text is less than 10 characters long

while(len(text) < 10):

text = text + "X"

print(text)

my\_list = []

# Do something an arbitrary number of times

for x in range(10):

my\_list.insert(0, x)

print(my\_list)

Hopefully it is clear from the example that a ‘for’ loop is better suited for executing a piece of code an arbitrary number of times since you don’t need to declare a separate ‘counter’ variable to track which loop it is on. However, a ‘for’ loop cannot test for a condition to be True and this is where we can use a ‘while’ loop. There is nothing wrong in using a ‘while’ loop for everything but it is neater to use ‘for’ loops where you can.

Also you may notice we are using another built-in function here called ‘.insert()’. This allows us to insert a value at a specific index in our list. Here we are inserting each number at the start of the list. This function along with ‘.append()’ can be very useful when working with lists.

**Reverse.py**

# Define our string reversing function

def reverse\_string(string):

# Set position to 0, new string to a blank string

position, new\_string = 0, ""

# While position does not exceed the length of the string

while(position < len(string)):

# Add the letter at the current position in the string to the

# start of our new string

new\_string = string[position] + new\_string

# Add 1 to position, i.e. go to the next position in the string

position += 1

# Return our new string

return new\_string

# Print out the output of our functions

print(reverse\_string("Hello"))

print(reverse\_string("This is a longer string than the previous one."))

This should be a more interesting piece of code to write. In it we define a function that is given a string. It then loops through the string and creates a new reversed string from it.

Loops can also be nested which allow us to carry out rather complex operations. The next example is quite large but it brings together a number of concepts we’ve learned recently. I have extensively commented the example and also provided a detailed trace of the program which can be found in the file ‘diamond trace’ on MyLearning.

This program allows the user to output one or more diamonds in the console window. Each diamond is made up of two triangles joined together, but could also be thought of as a string that grows and shrinks depending on what loop we are on. Therefore to create a single diamond we have to double the user’s input. If they specify that they want one diamond, we need to double it to 2 so that we can grow the shape and then shrink it. This is also a good example of using the modulus operator to determine whether the loop we are on is an even or odd number which gives us two alternating states. On even numbers we grow the first triangle and on odd numbers we reverse the previous operation.

**Diamonds.py**

''' This function outputs a series of diamonds.

It takes two arguments:

loops - how many diamonds to create

size - how big the diamonds should be

'''

def diamonds(loops, size):

# These are the characters our diamond and spaces will be made up of

char\_block = "#"

char\_space = " "

# Initialise the variables that will physically represent our

# diamond and surrounding space

text = ""

space = ""

# Fill space with 'size' number of chars

while(len(space) < size):

space = space + char\_space

# Print out first line of space chars for neatness

print(space \* 2)

# While we haven't reached the max number of loops

for x in range(loops):

# If the remainder of count / 2 is equal to 0

if(x % 2 == 0):

# While the string length is less than our desired size

# We divide the text length by 2 otherwise the diamond is too small

while((len(text) / 2) < size):

# Add two X's and surround it with spaces

text = text + (char\_block \* 2)

# Remove a space on either side as the shape grows

space = space.replace(char\_space, "", 1)

print(space + text + space)

else:

# While there are characters in the string

while((len(text) / 2) > 0):

# Remove an X from text and print out the string

text = text.replace(char\_block, "", 2)

# Add spaces on either side as the shape shrinks

space = space + char\_space

print(space + text + space)

# Main execution function

def main():

# Ask the user to define loop count and diamond size

# We multiply loops by 2 because a diamond is made of

# 2 triangles

loops = int(input("How many diamonds to create? ")) \* 2

size = int(input("How big should the diamonds be? "))

# Execute diamonds function

diamonds(loops, size)

main()