**Session 3**

**Complex Patterns**

Create a new folder and place your libraries into it. Open up Geany and create a new file called start.py in the same folder. Type the following into start

#! /usr/bin/env python3

from blink import blink

while True:

 blink(1)

 blink(2)

 blink(3)

 blink(4)

 blink(5)

 print("Start again")

You should understand all this code. It lights up the yellow light for an increasing longer time and then repeats the sequence. You could just build complex light sequences using just these commands, however you will be writing blink a lot and might get confused in a long sequence.

Now let’s tackle how to write a long sequence. Using your current knowledge the only way to do this is to write blink many times. However with a bit more Python knowledge a much easier way opens up.

**Lists**

Programming languages have solved the problem of sequences by creating a data type called an array. In Python there are a number of versions of an array (sets, tuples, dictionaries and lists are the most common) and the one we will use in this course is the **List.**

A **List** is a simple sequence of data separated by commas. So

seconds = [1, 2, 3, 4, 5]

is a **List**. **seconds** is the name of the **List** and the sequence is defined using []. Inside the **List**, each element must be a Python data type, in this case we are using **Integers**. Now that we have a **List** Python has some very powerful tools to work with **Lists**. The first tool is a **for** loop.

You have seen a **while** loop (though we have currently only used these to make infinite loops), the other type of loop in Python is a **for** loop.

The difference between them is:

**while** loops – used where you work out where the end is as you go, these loops keep going until the end condition occurs.

**for** loops – these move through the sequence once and have a definite end. **for** loops have an added bonus of making each element of the sequence available to use

So let’s use a **for** loop to move through the **seconds** sequence. The code is

for second in seconds:

 blink(second)

Once you have tried this out let’s work out how it works.

**for second in seconds**:

**for** is a Python key word and is use to create the **for** loop. It means that we are going to move through a sequence doing something with each element at a time.

**second** is a variable name that will only exist inside the **for** loop. It will change value each time through the loop. The first time it will be the first element of the **List**, in this case **1**, the second time the second element in the **List**, in this case **2** etc.

**in** is a Python key word which means that we are going to work on the elements of the **List** that is provided. It can be used in expressions, in **for** loops and in **if**/**while** statements.

**seconds** is the name of the List which contains the elements.

**:** this starts the block of code which will run each time through the for loop. Once this block of code has finished running the **for** loop will fetch the next element in the **List**.

**for** loops must have a sequence like a **List** to work with (technically they require an iterable data structure), however in Python many things can act as **Lists**, including **Strings**, which we will work with later.

Now we have a way to move through **Lists** we can create multiple sequences. Let’s create a second sequence

seconds2 = [8, 6, 8, 4, 1]

Now to run both sequences we need two **for** loops

for second in seconds:

 blink(second)

for second in seconds2:

 blink(second)

This breaks a major principle of programming, **Don’t Repeat Yourself**. Both **for** loops use identical code with one small change. This is a classic case of repeated code. Many beginner programmers would be tempted to copy and paste the code, but this is a bad idea since it means that if you have errors in your code, you are copying the errors. When you come to fix the error you must remember where the code is. In small programs this is not such a problem, however imagine you had 1000s of lines of code.

So how do we fix this? If you need the same thing in two places in your code, create a function and call the function in both places. So let’s do that.

def sequence(seq):

 for second in seq:

 blink(second)

while True:

 sequence(lights)

 sequence(lights2)

This code should be clear to you, if not read you sessions notes for the first two sessions again.

This works well and allows you to build complex sequences of flashing lights. One issue is where should the function **sequence** be, should it be here or in the blink library? The answer to that is whether you will only use it once or many times. If you think it might be useful many times, then place it in the blink library.

**Creating sequences**

Sometime you want to create your own sequence and Python has some excellent tools to do this. One is called **List Comprehension**, but that is beyond the scope of this course, you could look this up if you are interested. However the basic technique is to use a **for** loop to run the code a certain number of times and then inside the loop to build a **String** or a **List** with what you want. Type in the following code

#! /usr/bin/env python3

for x in range(10):

 print(x)

and run it. The only thing new here is the **range** function which produces a sequence of numbers for the **for** loop, you can’t actually use that sequence directly. You could look up the way the **range** function work to get more information about it. The main aim of the range function is to run the loop that many times, giving you a different number each time that you could use if you wish. Inside the loop you can do anything you like.

There are two basic ways to build a sequence, build a **List** and build a **String**. Both of these are usually done inside the **for** loop. To build a list you use the **.append** function of **Lists**, to build a **String** you use **String** **concatenation** which uses the **+** sign. With both these techniques you usually start with an empty **List** or **String** outside the loop, then build it during the loop. For example

#! /usr/bin/env python3

new\_list = []

new\_string = ""

for x in range(10):

 new\_list.append(x)

 new\_string = new\_string + "Hello "

print (new\_list)

print (new\_string)

**String concatenation** is so common that Python has spent a lot of time and effort making this better and faster. There a lots of ways to do this in a very efficient manner (but which might be harder to understand), but I will leave it to you to find these.

**String Sequences**

You now need to bring into your folder the lcd library that you made during the practical. So far that library can display a word on the screen and can display a word and the light at the same time. Now we want the library to display a sentence, one word at a time. To do this we need to put the sentence into a **List** and use a **for** loop to display each element of the **List**. Let’s work on the second part of the problem first.

Create a new file and import your lcd library. Now type in the following:

sentence = ["This", "is", "a", "sentence. "]

while True:

 for word in sentence:

 lcd(word)

*If this does not work, Did you remember to import your lcd library functions? What did you call your functions?*

Already I am sure that you can see how this could be improved by writing another sequence function. However the problem at the moment is with the sentence as a **List**. (I will leave the issue of the full stop not displaying at the moment).

Writing sentences as **List**s is a painful way to do this. Surely Python has an easier way to create a **List** for us from a sentence. It would be much better if we could just do this.

sentence = "This is a sentence. "

while True:

 for word in sentence:

 lcd(word)

This actually works, but probably not the way you expected. Remember back to what **for** loops actually do, they move through sequences and I mention they actually move through Iterable Structures. A **List** is an Iterable Structure, but so is a **String**. In Python there a many of these so there are many ways to use **for** loops. Python is also very good at converting from one structure to the other when you need it. One of the most common problems in programming is moving through sentences and Python has an easy solution, it has a way of changing the **String** into a **List** based upon a defined deliminator, like a white space for sentences or a comma for excel files.

It does this by using the **Strings**’ internal methods, there a many of them but the one we want is the method **split**. **split** is a function and you must define the deliminator, however if you leave it out it defaults to whitespace. If you think about it you should know how this is done, we have actually done this ourselves already.

So this means the actual code we need is:

sentence = "This is a sentence. "

while True:

 for word in sentence.split():

 lcd(word)

This is so useful that it is worth making this a library function in the lcd library, but that is a task for your practical.

**Displaying two words at once**

I now want to move onto another problem. It’s nice that we are displaying one word at a time from a sentence, however the lcd display on Zumo has two lines and it would be better to use both of them. To do this we need to solve two problems, firstly how to get to the second line and secondly how to get to two words in the sequence. Let’s solve the second problem first.

Python’s **Lists** have a way to access the individual members of the **List**. Image we had a sentence

wolf = "What’s the time"

You know that **wolf** is a **String** and that **Strings** are a iterable structure and you can get to each letter using a **for** loop. Python also has a way to access each individual element or small sequences of elements since it automatically provides an index to each element within a **List**, actually it provides two indexes, depending on whether you are starting at the beginning or the end. The indexes work like this

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| W | h | a | t | ‘ | s |   | t | h | e |   | t | i | m | e |  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |  |
| -15 | -14 | -13 | -12 | -11 | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |  |

So to get to the **t** in Wha**t**’s you use the either of the following code:

wolf[3]

wolf[-12]

The name of the structure and inside the square brackets the index you are trying to get to. If you want as small section of the structure you need to provide two numbers, the starting number, a colon, the ending number(which must be one more than the actual index). So to get the **the** you need

wolf[7:10]

wolf[-8:-5]

wolf[7:-5]

wolf[-8:10]

This process is so common it is called **Slicing** and is one of the features that make Python so popular. Slicing also has a third number which can be used, but I will leave that for you to look up how that works.

For our problem we don’t need slicing but we do need to get to the index of our sentence once it is split up.

As we move through the sequence we want to display the elements at index 1, 2 then at 2, 3, then at 3, 4 etc until we reach the end of the string. Python will throw an error if we try to use an index that is too large.

Let’s build this solution first in pure Python and then move it onto the Zumo.

Create a new file and type in the following:

quote="Don't worry about the world coming to an end today. It is already tomorrow in Australia"

outcome = quote.split()

print(outcome)

If you run this you will see we have a **List** with each element in the **List** a word from the quote.

Now add in the following

for out in outcome:

 print(out)

This prints each word down the page. We are almost there, we just need to print the next word along side. To do this we need to get to the index of each word, not just the word itself, out of the **List** **outcome**. This is so common that Python has an easy way to do this, using the function **enumerate** which returns two things, the current index and the current element, we need to capture both. Change the **for** loop to:

for index, out in enumerate(outcome):

 print(out, index)

You now see each word and its index next to it.

We can now change this line so it prints the next word using the following code

 print(out, outcome[index+1])

This works but will throw an error on the last run through since the index runs out and we can’t add one to it. To solve this problem we need to know how long the **List** is and only print this line when there is still some index left. Python has a way to work out the length of a **List** with the function **len**. The code to do this is

len(outcome)

so we need to capture this in a variable and then use that information to check whether index is getting too big. This is done using an **if** statement, we used these in Session 1. So our code is

quote="Don't worry about the world coming to an end today. It is already tomorrow in Australia"

outcome = quote.split()

length = len(outcome)

for index, out in enumerate(outcome):

 if index+1<length:

 print(out, outcome[index+1])

This works but does not print the very last element by itself, so we need to add one last line outside of the for loop.

print(outcome[-1])

This is such a useful piece of code I am sure that we will use it often so it should be a function in the lcd library. However before we do this we need one last piece of information.

**Zumo LCD**

In our PiZumo library you have seen two functions, lcdClear and lcdPrint. There is another function which specifies exactly where the print will start. That function is

lcdGotoXY(col, row)

It requires two numbers, one for col (which stands for column) and one for row. There are only two rows the first row being 0 and the second row being 1. There are 8 columns, starting from 0 and going through to 7 so the command

lcdGotoXY(2,1)

will move the lcd cursor to the second position on the second row and the lcdPrint will start from there.

You are now ready for your practical. We have covered a lot of ground here and if you want more practise there are practical exercise available around **String** slicing and concatenation on the website.

**Glossary**

|  |  |
| --- | --- |
| **Concepts** | **Meaning** |
| Command | These are key words used by the language that perform a function for the language. It is also possible to create your own commands to be used in your programming. Most languages have libraries of commands that have already been built for you to use as well as those you build yourself. |
| Functions | The most common way to create your own commands. These can accept data, process data in some way and can return the changed data to be used in other parts of the program. In a well structured program almost everything is done in small single purpose functions. |
| Arguments | These are bits of data that are used by functions, so that they have information to work with. |
| Parameters | These are the place holder variables that are used in function definitions. These are replaced by the arguments that are used when the function is run. Often the two terms are used interchangeably.In Python Parameters can be given a default value, which means that an argument is not required when the function is called. |
| String | Data type, String refers to ordinary words.  |
| Integers | Data type, int refers to whole numbers.  |
| Float | Data type, Float used for decimals, but they are not accurate.  |
| Variable | Way of representing data for the program to work on. |
| Libraries | These contain commands that have been developed and tested and are ready to use. Many of the libraries have been written by the people who originally developed the language while other libraries have been developed by companies or individuals that use the language (Google, Yahoo, Apache etc have all developed extensive language libraries for a wide variety of languages). You can also develop your own library of commands. |
| Assignment | Uses = to give a variable name a value |
| Expression | An operation which must be true or false |
| Comments | These are notes for people reading the code. |
| Array | Structure used to group data together, in Python there are many types including List, Set, Tuple, Dictionary etc. |
| Object methods | All Python objects have built in methods to make them more useful. These include |
| String concatenation | Adding to strings together to make one string. This can be done using the + sign. Python also has some more efficient methods available. |
| Slicing | Python methods for getting inside sequences |
|  |  |
| **Python Commands** | **Meaning** |
| print | Python function which will show the data on the console |
| def | Python key word used to define your own functions |
| input | Python function which gets data from the keyboard |
| int | Python function which changes a String to its Integer value |
| if | Python keyword used to build decision making structures, must be followed by an expression which is either true or false |
| from … import | Python keywords used to bring in Python objects and function from other files for use in your current file. |
| while | Python keyword which creates a loop based upon the expression |
| True | Python keyword which always evaluates to true, there is also a keyword False. |
| for … in  | Python keyword which creates a loop based upon the iterable structure that it is given. in is also a Python keyword to look inside Arrays and can be used in expressions |
| range | Python function which is used to create a sequence of numbers |
| .append | Python method which adds an element to a list |
| .split | Python method which changes a string to a list |
| enumerate | Python function which is used in for loops to provide both the data and the index of the data |
| len | Python function which finds the length of a sequence |
|  |  |
| **Best Practise** |  |
| DRY Principle | Don’t Repeat Yourself. When you need the same code in two places make a function, never copy and paste. This way if there are errors they will only be in one place. Makes you code much easier to maintain and change later on. |