**Session 5**

**Parser**

A very common task in programming is taking information in one form and changing it to another so it can be analysed and used. To do this programmers write a type of program called a Parser. There are many around, including dealing with HTML data (Beautiful Soup), Excel data (xlrd, openpyxl), PDF data (PDFMiner, pyPDF), JSON data (json in the standard libraries), Word data (python-docx) etc. Programming languages (like Python, Javascript, Java etc) are also simply parsers, though they do use quite complex methods to analyse the data and change it into the form that the CPU can understand.

To make the Zumo work your lecturers have written two parsers, one for the Zumo written in C and one for the Raspberry Pi written in Python. These two parsers talk to each other so that you can write in Python and the Zumo can implement your instructions in C.

We are going to take some information that is written on a text file and use that information. Together we are going to write a parser for a calculator. We won’t be using our functions from Session 1 simply because we have a lot to learn and our files in Session 1 were not set up properly to be used as imports. (You could do this yourself, you now know how to construct a library). Also in this Session we will not finish the Parser, we will do just enough to get it working and show you how it works.

To create the Parser we will the Pseudo code approach explained in the last session. Briefly that approach involves writing out our instructions first in Pseudo code (in this case English) to work out how to get it to work and then turning this into Pseudo code into Python code.

Our task: To take a text file which has a list of calculations and process these in Python

We already have a text file created for you called inst.txt If you look at it in a text editor you see that it has 7 lines, some have calculations, some are blank and some have other data. The lines with calculations should also have spaces in the lines separating the operators and the numbers (like 3 + 5 not 3+5). Our Parser must be able to handle all three cases.

Parsers do four basic operations:

* Bring the data from the file into Python
* Go through the data line by line
* Examine the line to work out which command to use
* Assemble the pieces of the command and perform the action

So let’s write a Parser together. The first step is to create a new folder and a new file in that folder (called parser.py). In that file put in the four steps as comments

#Bring the data from the file into Python

#Go through the data line by line

#Examine the line to work out which command to use

#Assemble the pieces of the command and perform the action

Now let’s solve each operation. At the moment you don’t know how Python gets information from a text file, but you do know that it can be done. So you just need to learn the command. When you know that something can be done, simply google the question and look at stackoverflow or the Python documentation for the answer. The Python documentation (<https://docs.python.org/3/tutorial/inputoutput.html>) says the good practise is (7.2.1):

**>>> with** open('workfile', 'r') **as** f:

**...**  read\_data = f.read()

but does not really explain how it works.

**with** This activates a context manager. It means that the real command (**open**) has a context manager attached to it which opens and closes the operation properly, you don’t need to do it yourself. Other operations that often use context managers include opening and closing network connections and database connections. There is a way of creating your own context manager for an object, but that is beyond the scope of this course.

**open** This is a Python function which opens a text file. It takes two parameters, the name (and path) of the text file and the type of operation that you want to do. Both parameters are **Strings**, the second usually being 'r' for read, 'w' for write or 'a' for append (adding onto the end of the file).

**as** This is a Python keyword which creates an alias to be used. At this stage of your learning aliases are only applied to **with** and **import** commands.

**f** This is simply a variable name which will be used for the opened text file. It is not a very good name.

f.**read()** **read** is one of the methods that can be used on opened text files. It reads the entire file (which is called f) in as a **String** (and places it into the variable, in this case read\_data). Other operations are **write** (which writes out a **String** to a text file), **readline** (which reads the text file one line at a time, useful in loops) and **readlines** (which reads the entire text file into a list, line by line).

Now that we know how Python reads in a text file let’s do that with our text file and print out the results, we will use better variable names.

#Bring the data from the file into Python

with open('inst.txt', 'r') as file:

 data = file.read()

print(data)

For this to work our python file and the inst.txt need to be in the same folder. We now have the data from the text file in Python in a variable called data. Since we used the read function then data is a **String**. We have finished the first operation. We now need to do the second

#Go through the data line by line

Any time you want to go through something bit by bit you need a loop. In Python there are two types of loops, **for** and **while**. **for** loops work with Iterable Structures like **List** and have clearly defined end points, **while** loops work by first defining the end point as an expression and testing each time before running the block. Both can be used here but which is easiest?

**while loop** We need a test for this to work. We could test for the end of the file, and in most programming languages that is what you would need to do. However in Python there is a much easier way, and besides we already have the data as a **String** and the file is now closed, we would have to use the **while loop** while the file is still open (perhaps with the **readline** function).

**for loop** These work with Iterable Structures (like **List**), however Python has a huge range of options to convert data from one type into another (from **String** to **List**, from **String** to **Integer**, from **Integer** to **String** etc) and also a huge range of these Iterable Structures. In fact a **String** is already an Iterable Structure. Try this to see:

for x in data:

 print(x)

The outcome should be no surprise, we did this in Session 3.

We have a **String** called data so we could use **data.split()** to change it into a list. Let’s try that:

for x in data.split():

 print(x)

Great we have a list and with some very difficult programming we could parse this data. However the current data is split on white space, and these exist inside the lines as well as at the end of the lines. Maybe there is a better way to do this. **split** is a method that is available to all **Strings**, but there are other methods and one of these might suit our needs better in this case.

If you look again at the **String** methods in the Python documentation (<https://docs.python.org/3/library/stdtypes.html> at 4.7.1) you see there is a method called **splitlines()** which will do exactly what we want. So our command should be

for x in data.splitlines():

 print(x)

***NB.*** *Using string methods does not change the original string. If you do the following:*

cmds = data.splitlines()

print(cmds)

print(data)

*you will see that data has not been changed. This is why we need to capture the change in a new variable or use it immediately in a* ***for*** *loop.*

So our final code for this section is:

#Go through the data line by line

for cmd in data.splitlines():

 print("command is ",cmd)

We are now ready to solve the third problem: So far we have the following:

#This needs to be a function so that it can be used in a library.

#Bring the data from the file into Python

with open('inst.txt', 'r') as file:

 data = file.read()

#Go through the data line by line

for cmd in data.splitlines():

 print("command is ",cmd)

#Examine the line to work out which command to use

We currently have each line of the text file as a **String** in the variable cmd.

**Slicing**

We now need to look inside that string to work out the parts. Looking inside a variable to get to its parts is called slicing in Python which was introduced in Session 3. Slicing works on both Strings and Lists and is used to look inside to get the part of the structure that we are interested in.

Here we can use slicing to look inside the line, but which is easier **Lists** or **Strings**. We want to get to the operator and the numbers, but they are not always in the same place in a **String**. However if we turned the **String** into a **List** they should be. So let’s do that

cm = cmd.split()

Looking at the list this is created the numbers are always in index 0 and 2 while the operator is always in index 1. Now we have something we can predict and can work with.

So let’s complete the addition first

#Assemble the pieces of the command and perform the action

 if cm[1] == '+':

 answer = int(cm[0])+int(cm[2])

 print("The answer to", cmd, "is", answer)

This produces the following error:

builtins.IndexError: list index out of range

We actually get the right answer for the first line as you can see by the fact that that line printed before we got the error so our code is right. One line you might not understand, since we have not used this command since Session 1 is:

answer = int(cm[0])+int(cm[2])

cm[0] and cm[2] currently are **Strings** but we want them to be numbers so we can add them together. Python has a function called **int** (which we used in Session 1)which changes a **String** to its **Integer** value. This will also change a **Float** to its **Integer** value. Similar there are function **str** to change data into a **String** and **float** to change data into a **Float**. All these function do not change the original variable so the new data can only be used immediately or captured in a variable.

This line actually works, since we got the correct output on the first line so the problem is not there. To troubleshoot this problem let’s add a print statement

cm = cmd.split()

print(cm)

*Adding print statement through our code is a good debugging technique. It shows us exactly what is happening, especially inside loops where the variable will change data each run through the loop. Some IDEs also have the ability to step through your code line by line and let you examine exactly what is happening to all your variables.*

When we run this we see that the first line works perfectly, the second line does not produce anything and it is the third line, which is empty that creates the error. The error message also gives us a clue

list index out of range

The third line produces an empty list but we are trying to look at the second element. There is no second element so the index is out of the range of the list. This will occur on every blank line. Remember that one of our requirements was to handle blank lines.

This can be done with the following which must be the first **if** inside the loop

if cm == []:

 continue

**continue** is a Python keyword which can be used in loops. It means stop processing this current iteration and start the next one. There is also a keyword **break** which can be used in loops, **break** means to stop the loop entirely and do no more processing on it.

You should now know how to process subtraction, multiplication and division following the same pattern as addition.

Lastly we will process the line without a calculation, just with text.

if cm == []:

 continue

elif cm[1] == '+':

 answer = int(cm[0])+int(cm[2])

 print("The answer to", cmd, "is", answer)

else:

 print("I don't know how to do",cmd)

The only new code here is **elif** and **else**. This is used with if statements where you want to determine more than one outcome. It is used in situation where you want only one of many outcomes, the first is captured by the **if**, the rest by the **elifs** and the **else** captures all the other possibilities where the if or elif statement fails.

The full code so far is

#Bring the data from the file into Python

with open('inst.txt', 'r') as file:

 data = file.read()

#Go through the data line by line

cmds = data.splitlines()

for cmd in cmds:

 #Examine the line to work out which command to use

 cm = cmd.split()

 #Assemble the pieces of the command and perform the action

 if cm == []:

 continue

 elif cm[1] == '+':

 answer = int(cm[0])+int(cm[2])

 print("The answer to", cmd, "is", answer)

 else:

 print("I don't know how to do", cmd)

You should be able to finish this off yourself.

**Buzzer Playing Music**

The zumo has a buzzer which is able to play musical notes. To fully use this feature you need to have an understanding of music as the implementation uses musical octaves. The library code to activate this is:

buzzerPlayNote(musicalOctave, length\_of\_note, volume)

Where:

**musicalOctave** is also a library function. All the musical notes are functions which take a single parameter being the octave of the note. For example to play the musical note middle c would be to access the library function

note\_C(4)

**length\_of\_note** is an integer representing milliseconds. This is worked out using the musical length of the note as well as the tempo of the musical score.

**volume** is an integer between 0 and 15.

When you active the buzzerPlayNote function you should a single not for a defined time (usually using **sleep**). The buzzer only has a single speaker so two notes cannot be played together to make a harmonic note, the second will overwrite the first.

This is a sample of the first line of Greensleeves written for you. This is very badly written code since there is a lot of repeated code. You could easily create a small library to make the coding task easier for writing music.

However you could use this code to write a small buzzer sound effect.

**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. |
| Slicing | Python methods for getting inside sequences |
| Methods | Functions which are attached to objects and can be used to modify the object. In Python all the data types (Strings, Integers etc) are objects and all have methods attached. |
|  |  |
| **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 data 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 |
| assert | Python keyword which is followed by an expression, will throw an error if the expression is false |
| pass | Python keyword which is a placeholder, used to create function without filling in the exact implementation details yet |
| None | The value you turn from a function which does not contain a return statement. None evaluates to false but can also be tested for |
| with | Python keyword which activates the context manager attached  |
| open | Python function which opens a text file.  |
| as | Python keyword which creates an alias  |
| .read | Python method on open which the entire file in as a string  |
| continuebreak | **continue** is a Python keyword which can be used in loops. It means stop processing this current iteration and start the next one. There is also a keyword **break** which can be used in loops, break means to stop the loop entirely and do no more processing on it. |
| str | Python function which changes data into a string |
| float | Python function which changes data into a float value |
| elif else | Python keywords used with if to build if chains. else can also work with some other keywords outside the scope of this course. |
| .splitlines | String method which splits on the end of line |
|  |  |
| **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. |
| TDD | Test Driven Development a way of programming where you write your tests first. |
| Refactor | Once we have written our code and it works (with tests to prove it works) we can look at what we have done and make our code better. |
| Pseudo Code | Writing out in English inside comments what you want to do so that you can then turn this explanation into Python code. |
|  |  |