FACTFILE:
GCSE DIGITAL TECHNOLOGY
Unit 4
DIGITAL AUTHORING CONCEPTS

Rewarding Learning

Data Programming Constructs 3

Learning Outcomes

Students should be able to:

Demonstrate understanding of and use the functionality of the following constructs in a programming language:

- controlling the flow of a program through sequence, selection and iteration;
- building reusable code that utilises user-defined functions or methods; and
- basic file handling.

Content

- Flow of Control:
  - sequence
  - selection
  - iteration
- User-defined functions
- File handling

Flow of Control

Computer programming involves breaking a problem task down into a collection of smaller steps, each of which is simple enough that it can be carried out by the computer. This is sometimes called *problem decomposition* and it is certainly an important part of computer programming. The programmer must also ensure that the simple steps are carried out in the correct order. This is sometimes referred to as specifying the *flow of control* of a program. Programming languages normally provide three broad ways to specify the flow of control through a program.

- **Sequence** – carry out the steps in a specified order – one after another.
- **Selection** – choose which step to carry out next depending on whether or not a particular condition is met.
- **Iteration** – carry out the steps repeatedly until some condition has been met.

Different programming languages take slightly different approaches to sequence, selection and iteration, but once you have mastered these concepts in one language, you will find it fairly straightforward to transfer your understanding to another.

The examples used in this fact file use the Python programming language. Further examples of Python code can be found in the following fact files.

- U4FF3: Data Types, Operators and Computational Thinking
- U4FF5: Data Programming Constructs 1
- U4FF6: Data Programming Constructs 2

Sequence

The simplest way in which control can flow through a program is that it starts at the beginning and works one step at a time until it reaches the end. In other words the steps are carried out in *sequence*. For example, the following Python program calculates the average height of Bill, David and Marie.

```
billsHeight = 170
davidsHeight = 168
mariesHeight = 165

totalHeight = billsHeight + davidsHeight +
mariesHeight
averageHeight = totalHeight / 3

print(averageHeight)
```

There are six steps in this program and they are carried out in order – top to bottom.

- The first three steps assign values to variables to represent the heights of Bill, David and Marie.
- The next two steps perform some calculations.
- The final step prints the result.

Sometimes the order doesn't matter – e.g. if the order of the first two lines was reversed, the program would still calculate the average correctly.

```
davidsHeight = 168
billsHeight = 170
```

Often, however, the order is important – e.g. the program above would not work correctly if the order of the calculation steps was reversed.

```
averageHeight = totalHeight / 3
totalHeight = billsHeight + davidsHeight +
mariesHeight
```

Selection

A programmer may sometimes wish to specify alternative paths through the program code, depending on whether or not some particular condition has been met. For example, the following Python program advises the user about drug dosage, depending upon their age.

```
age = int(input('Please type the patients age in
years: '))
if age <= 7:
    response = 'Take one tablet, twice a day.'
else:
    response = 'Take two tablets, twice a day.'
print(response)
```

The first line of the program prompts the user to type in her age; this value is then assigned to the integer variable, *age*. The last line of the program prints the program's response to the user. The four lines in between are where the selection occurs. These four lines make up a selection statement or, more specifically, an *if-else* statement – this is where the *selection* occurs.

There are two legs to the if-else statement but, for any given user, only one of them is actually selected to be executed. The selection depends on the value of the variable, *age*.

- If the variable, *age*, has a value that is less than or equal to 7, then the first leg is executed. This assigns the value, *'Take one tablet, twice a day.'*, to the variable, *response*.
- If the variable, *age*, has a value that is greater than 7, then the second leg is executed. This assigns the value, *'Take two tablets, twice a day.'*, to the variable, *response*.

It is possible to have a selection statement with more than two legs. For example, we might want to specify that children under 3 should not be given any tablets. In this case, the following program might be used.

```
age = int(input('Please type the patients age in
years: '))
if age < 3:
    response = 'Do not take these tablets.'
elif age <= 7:
    response = 'Take one tablet, twice a day.'
else:
    response = 'Take two tablets, twice a day.'
print(response)
```

You will note that there are three legs to the selection statement:

- The first leg is executed only if the variable, *age*, has a value that is less than 3. If the condition is met then the assignment is carried out and control passes directly to the print statement.
- The second leg is executed only if the variable,

age, has a value which is less than or equal to 7. Since any values less than 3 have already been caught by the first leg then the second leg is effectively only executed for values between 3 and 7. If the condition is met then the assignment is carried out and control passes directly to the print statement.

- The final leg has no associated condition but, since values of the variable, *age*, that are less than or equal to 7 have already been caught by one of the first two legs, we can be sure that control will only ever pass to this point for values that are greater than 7.

An if-else statement in Python can have as many legs as are needed:

- The keyword *if* must be used to introduce the first leg and it must be followed by a condition (e.g. age<3).
- Subsequent legs must be introduced by one of the keyword *elif* (pronounced else *if*) or *else*.
  - The keyword *elif* must be followed by a condition (e.g. age<=7).
  - The keyword *else* must not be followed by a condition, and it may only be used in the *final* leg of a conditional expression.

Iteration

A programmer may sometimes wish to have a section of code executed repeatedly – perhaps a fixed number of times, or perhaps until some condition has been met. Consider, for example, the program above that gives advice on drug dosage. In its current form, this program will only advise a single patient each time it is executed.

```
Please type the patients age in years: 2
Do not take these tablets.
>>>
```

What if we wanted to give the user the chance to receive advice for a second patient? – or a third, or fourth? The following program achieves this.

```
age = 99
while age > 0:
    age = int(input('Type the patients age, or 0 to
quit: '))
    if age == 0:
        response = 'Bye!'
    elif age < 3:
        response = 'Do not take these tablets.'
    elif age <= 7:
        response = 'Take one tablet, twice a day.'
    else:
        response = 'Take two tablets, twice a day.'
print(response)
```

The lines shown in **bold** are new code that has been added to achieve the desired iterative behaviour (i.e. to make the program loop). The remaining code is almost identical to the previous program. Here is what's new.

- The keyword, *while*, is what creates the loop (known as a *while* loop) and the condition (age>0) is what controls when the loop stops. All of the code in the while loop is repeatedly executed until the condition evaluates to *false* (i.e. until age=0).
- An extra leg has been added to the selection statement. This ensures that if the user has typed 0, the response string is assigned the value "Bye!".
- The very first line of the program assigns the value 99 to the variable, age. This is to ensure that it has a value before the condition of the while statement (age>0) is encountered – otherwise an error would be generated. We have chosen to give it the value, 99 but any value greater than zero would work equally well.

Here is how the program behaves.

```
Type the patients age, or 0 to quit: 2
Do not take these tablets.
Type the patients age, or 0 to quit: 4
Take one tablet, twice a day.
Type the patients age, or 0 to quit: 8
Take two tablets, twice a day.
Type the patients age, or 0 to quit: 0
Bye!
>>>
```

You will notice, in this interaction, that I have chosen to enter a range of ages (2, 4, 8 & 0) such that each leg of the select statement gets tested. It is always good practice to choose test data in this way, to ensure that all paths through your code are tested.

User-Defined Functions

Modern programming languages normally supply the programmer with some means of structuring a complex program into a number of simpler ones. It is easier for the programmer to understand and debug a program that has been structured in this way. These simpler programs may be called subprograms, procedures or functions.

Subprogram

"[A subprogram] is a set of program instructions performing a specific task, but which is not a complete program. It must be incorporated into a program in order to be used."

BCS Glossary of Computing, 13th edition, p 280.

Procedure

"[A procedure] is a subprogram that is generally written using a precise formal definition. The procedure is defined and given an identifier (or name). This identifier can be used subsequently just like any other program instruction.."

BCS Glossary of Computing, 13th edition, p 279.

Function

"[A function] is similar to a procedure but returns a single value. The name of the function is usually used as a variable having that value. Every time the name is used in the program, the function will be executed and the result will appear like any other variable."

BCS Glossary of Computing, 13th edition, p 279.

Subprogram is a generic term, but procedure and function mean specific things in the context of specific programming languages. Procedures and functions are similar to one another.

- Both procedures and functions may take parameters – this means that data items may be passed to them when they are invoked.
- Only functions, however, return a value – this value will be associated with the function name after it has been executed.

The following Python function takes three parameters (n1, n2 & n3 – assumed to be positive integers) and returns the biggest of them.

```
def biggest( n1, n2, n3 ):
    biggestSofar = 0
    if n1 > biggestSofar:
        biggestSofar = n1
    if n2 > biggestSofar:
        biggestSofar = n2
    if n3 > biggestSofar:
        biggestSofar = n3
    return(biggestSofar)
```

The following Python interactions illustrate how the function behaves.

```
>>> biggest(1,2,3)
3
>>> biggest(3,2,1)
3
>>> biggest(9,10,1)
10
>>>
```

The first and last lines of code (shown in **bold**) are important components of a function definition. These are the only ones that are of interest to us here.

- The keyword *def* indicates that the programmer is about to *define* a function.
- The name of the function is *biggest*, and it has three parameters – *n1, n2* and *n3*.
- The *return* statement, at the end of the definition, indicates that the value of the variable *biggestSofar* is to be returned as the value of the function.

File Handling

Sometimes you may wish to access data that is stored in a file. This may be because you want data that is entered at one session to be available to your program at a later session – perhaps even to a different user. Since programs may terminate and computers may be switched off between sessions, this data needs to be stored in non-volatile storage – typically your computer's hard disk. Most programming languages provide mechanisms to support file handling to facilitate this.

In Python the normal sequence of operations to access data in a file are as follows.

1. Instruct Python to open the file.
2. Access the data by either reading from the file or writing to it.
3. Instruct Python to close the file.

The following short program illustrates how to append data to a text file using Python.

```
userName = input('Please type your first name: ')
myFile = open('usernames.txt', 'a')
myFile.write(userName)
myFile.write('\n')
myFile.close()
```

Here is what is happening.

- Line 1 prompts the user to enter their name, which is then assigned to the variable, *userName*.
- Line 2 opens a file, or creates one if it doesn't already exist.
  - The name by which the file will be identified within the program is *myFile*. This is sometimes referred to as the *file handle*.
  - The operating system is instructed to use the name *'username.txt'* to identify the file. This is what will appear in a directory listing.
  - The value '*a*' indicates that the file is to be opened in *append mode*. This means that any data written to the file will be appended to it – i.e. added to the end of the file.
- Line 3 writes the value of the variable *userName* to the file.
- Line 4 writes a newline character to the file.
- Line 5 closes the file. This is important – failure to close files after use can generate errors and cause the program to behave unexpectedly.

In this case the file is opened in append mode. There are a range of alternative modes that might also be used, including the following.

Access Mode	Description
a	File is opened in *append mode* so that any data is written to the end of the file. No existing data is overwritten.
w	File is opened in *write* mode so that any data is written to the beginning of the file – overwriting any existing data.
r	File is opened in read mode so that data is read from the start of the file.

The following Python program illustrates how to read data from a file.

```
userName = input('Please type your first name: ')
myFile = open('usernames.txt', 'w')
myFile.write(userName)
myFile.write('\n')
myFile.close()

myFileAgain = open('usernames.txt', 'r')
nameInFile = myFileAgain.read(3)

print('File contains the name: ')
print(nameInFile)
```

Here is what is happening.

- Lines 1–5 are identical to the previous program, except that this time the access mode is '*w*' rather than '*a*'. In other words this is just writing some data to the file.
- Line 6 opens the file that has just been written to – but this time the access mode is 'r' so that we can read back the data that has just been written.
- Line 7 reads the first 3 characters from the file and assigns the resulting string value to the variable, *nameInFile*.
- Lines 8–9 print the resulting values.

The following interaction illustrates how this program behaves.

```
Please type your first name: Barney
File contains the name:
Bar
>>>
```

By changing the value of the integer parameter passed to read, we can have the name returned in full.

```
nameInFile = myFileAgain.read(10)
```

The resulting behaviour would look like this.

```
Please type your first name: Barney
File contains the name:
Barney
>>>
```

Resources

In addition to an excellent Python tutorial, there are very clear and accessible explanations and illustrations of sorting and searching algorithms at the TutorialsPoint website.

- BBC Bitesize, Algorithms & Control Flow: http://www.bbc.co.uk/education/guides/zrxncdm/revision
- TutorialsPoint, Python 3 Tutorial: https://www.tutorialspoint.com/python3/index.htm
- TutorialsPoint, Python 3 – Files I/O: https://www.tutorialspoint.com/python3/python_files_io.htm
- TutorialsPoint, Python 3 – Functions: https://www.tutorialspoint.com/python3/python_functions.htm