University Of Portsmouth
BSc (Hons) Computer Science
First Year

Programming

M30299
September 2022 - May 2023
20 Credits

Thomas Boxall
up2108121@myport.ac.uk

Contents

Module Introduction

20-09-2022 14:00 Nadim & Matthew PK2.23

Module Aims

This module will build up programming skills either from scratch or from where you are currently.
It will give you the basic knowledge; guidance, help and feedback to help develop programming skills.

Importantly, this module is 40 credits. It spans across the entire year.

Programming

Programming is the process of constructing computer programs, this encompasses analysing the problem, designing the algorithm, implementing the algorithm and testing the algorithm.
We write the programs in a programming language.
For the first $\frac{3}{4}$ of the year, we'll use Python 3 and for the final $\frac{1}{4}$ of the year, we'll use Dart. Dart is similar to Java. We will be the first cohort to use Dart.
Programming is a skill, which can only be developed through practice and should be fun! Having a good understanding and ability to program is important later during in the course and for careers.

Module Organisation

For this module, there will be content shared on Moodle (notes for lectures and videos complementing the notes) and timetabled sessions (in some, fundamental ideas will be covered which will make it possible to complete the weekly worksheets). Worksheets will be released weekly onto Moodle, these should be completed before the practical class of the following week.
Monday at 3pm in RB LT1 is the tutorial class. You need to go through the notes on Moodle before the sessions.
Practical classes are 1 hour 50 minute sessions in a computer lab. The main purpose of these is to get feedback on the worksheets.

Support

The academic tutors (Xia and Eleni) can be booked on Moodle.
There are drop-in sessions on Monday in the FTC. This session is optional and is designed for targeted questions or issues which can't be resolved in the tutorial/ practical classes.

Out Of Class Work

Should be spending about 8 hours per week outside of timetabled sessions working on this module. This includes working through the worksheets.

Assessments

There are three types of assessment used throughout the year

- 5x 30min programming tests (held in class, weighted 5% each)
- 2x 60min Computer based multiple choice tests (weighted 15 % each, one in January and one in May/June)
- 2x large programming assignments (weighted 20% and 25% respectively)

The programming tests will be based off of the previous weeks worksheets. There will be a practice test in week 3 (so we can understand how they work)
Each of the programming assignments will have a few weeks in which they can be worked on.

Resources

To write and execute Python programs, the recommended IDE is Pyzo. Other IDEs can be used however no support for configuration will be provided.
We will be using Python 3.x NOT Python 2.x.
The recommended book is called 'Python programming: an Introduction to Computer Science 3rd Edition'. There are a number of copies available in the library. Its ISBN number is '9781590282755'.

Writing Simple Programs

26-09-2022 15:00 Nadim RB LT1

This lecture introduces the basic steps involved in programming and provides some additional information about each stage.

Stages of Algorithm Design

When presented a problem to solve programmatically, the first stage to doing so is to understand the problem and to ensure that this understanding is correct. To aid this, it can be useful to work out how the user interacts with the system, through listing the user inputs and outputs to screen. At this stage, it can also be beneficial to make a note of some inputs and their expected outputs as this can be used to test the program at the end of development.
The next stage is to design an algorithm that accomplishes the task.

> **Algorithm**
>
> A detailed sequence of actions which acomplish a task. Cna be written in plain English or any other language.

The next stage is to implement the algorithm. This is where the plain English algorithm is converted into programming statements which can be executed by the machine.
The final stage is to test the program. This can be done ith the data noted down in stage one.

Key Program Concepts

In programming, there are a number of key concepts. These will be illustrated using examples written in Python 3.

Statements

Every line of a program is called a command or statement. These are executed (carried out) one after the other (there are ways in which the flow of the program can be altered, but this will be covered at a later date). Program execution ends after the last statement is executed.

Variables

A variable is a name for a part of the computer memory where a value is stored. The variables have names in the programs.
Statements in the program may create a new variable, use the value of a variable or change the value of a variable.

Assignment Statements

Assignment statements are used to assign a value to a variable. The syntax is as follows:

- The variable appears on the left hand size of the =

- The right hand side of is an expression, which has a value

```
LANGUAGE: Python3
variableName = expressionWhichHasAValue
```

Assignment statements are executed in two steps. First they evaluate the expression on the right hand side then second, assign the value to the variable on the left hand side.
If the variable on the lft hand side doesn't already exist, then it is created. If the variable exists already, its old value is replaced.

Numeric and String Values

Numeric values are numbers. They do not need any demarcation. For example, `2.2` is a numeric value.
String values are strings of characters. These can be any character. Strings need to be encased in single quotes or double quotes. Both are valid, however they can't be mixed. Lines 1 and 2 in the following example are valid, however line 3 is not.

```
LANGUAGE: Python3
validStringOne = "I'm in double quotes, notice I can use single quotes where I like!"
validStringTwo = 'Im in single quotes, notice I cant use single quotes in the string.'
invalidString = "Im not valid'
```

Arithmetic Expressions

Standard arithmetic expressions can be formed using `+`, `-`, `*`, `/` and `()`. Expressions are evaluated to give a value, this is commonly stored in a variable or outputted directly to the user.

Built-In Functions

Python has a number of built-in functions. These are algorithms which are part of the Python language. They can be accessed by using its name. Sometimes they have parameters, sometimes they return a value and sometimes they do both. Common examples of built-in functions are shown below.

```
LANGUAGE: Python3
print("I display information to the user")
variable = input("I allow the user to enter text, then I store it in the variable")
```

Example Execution

See Week 1, lecture 01c slides on Moodle for a detailed look at how programs execute and how the variable contents change.

Example programs from Lecture

Program 01

This program introduces a count-controlled loop (for loop) and the print statement.

LANGUAGE: Python3

```
total = 0
for i in range(34):
    #print("banana")
    #print(i)
    total = total + i

print("The total is: ", total)
```

This program should output the following

LANGUAGE: Unknown

```
The total is:  561
```

The two commented out lines (lines which begin with the #) symbol can be un-commented so that they run.

Program 02

This program introduces the concept of `input()`, `int()` and subroutines.

LANGUAGE: Python3

```
def simpleProgram():

    value = int(input("Please enter a whole number: "))

    for loopCount in range (value):
        print(loopCount)

#################

simpleProgram()
```

The program should output the following.

LANGUAGE: Unknown

```
Please enter a whole number: 12
0
1
2
3
4
5
6
7
8
9
10
11
```

The number 12 on line one is entered by the user.

COMPUTING WITH DATA AND NUMBERS

03-10-22 1500 Nadim RB LT1

Data and Data Types

There is a lot of data which programs have to process. Different types of data are stored as different 'Data Types', this allows them to be processed differently; an example of this is numerical data. In programming, we commonly distinguish between two different types of numerical data: integers (whole numbers, eg 55, 77, 88, -5) and fractional number (or floating point numbers, eg 4.6, 7.00956, -9.89). Words and other multi-character statements can be written within strings and truth values are stored as booleans.
All data values belong to one single data type and in some contexts, we use Class rather than data type.

Python Data Types

Python has all of the common data types within it. Each of the data types have a specific keyword:

Type	Python Keyword	Example
Integer	`int`	`33`
Fractional	`float`	`2.3`
String	`str`	`"Spam"`
Boolean	`bool`	`True`

Operations on Data Types

Data types have operations associated with them, some of these are language specific functions however the majority are universal across most programming languages.
For example, `int` and `float` both have the operations `+`, `-`, `*` and `/`.
Numeric data types follow the operator precedence rules, as a human would with mathematical equations. They follow BIDMAS. Where two operators have equal precedence, the calculations are carried out from left to right.

Type Conversions

It is important to be able to convert between different data types. The following example code shows the different functions.

LANGUAGE: Python3

```python
# convert 5 (int) to float, equals 5.0
floatVariable = float(5)

# convert 4.5 to int, this truncates, so will equal 4
intVariable = int(4.8)

# convert 6.8 to a string, equals "6.8"
strVariable = str(6.8)
```

It can be useful to find out what data type a variable or value is. To do this, use the `type()` function as seen below.

LANGUAGE: Python3

```python
print(type(44)) # outputs <class 'int'>
print(type("Banana")) # outputs <class 'str'>
print(type(4.67)) # outputs <class 'float'>
```

User Input

The `input()` function, returns a string. This can be really useful if we want to do something with a string. However, if we want to do something with a number, this is less useful. We can use the `float()` or `int()` functions to convert into floats or integers respectively. Examples of this can be seen below.
There is another function which can be used. The `eval()` function returns either a float or integer depending on the value passed into it. It can be useful in situations where the value entered by the user could be either floating point or integer.

LANGUAGE: Python3

```python
# convert to float
floatInput = float(input("Enter a float here: "))

# convert to an integer
intInput = int(input("Enter an integer here: "))
```

Arithmetic Operations

Where an arithmetic operation involves both a float and integer, the integer is automatically converted to a float then the operation is carried out. For example, in the operation `7+1.5`, the `7` would be converted to `7.0`. Therefore, the calculation would then be `7.0 + 1.5 = 8.5`.

Division

The `/` operator always performs floating point division, hence `11 / 4 = 2.75`.
The `//` operator performs integer division, where it is given two integers as inputs, the result will be a truncated integer; as seen in the following example `11 // 4 = 2`
The `%` operator gives the remainder of an integer division, hence `11 % 4 = 3`.

Issues with Floating Point Arithmetic

Floating point numbers are represented within the computer using a fixed number of space (64 bits), this means that there is a limit to the range and accuracy of the number which is able to be stored.
There are some numbers, 0.1 for example, which are unable to be represented within this size limit in binary, this can lead to issues with the value of a float number after performing mathematical operations on it.
This problem is true of all programming languages that use floating point numbers.

Python's Numeric Functions

There are a number of useful built-in functions in Python which help with maths.
The `round()` function takes a float as a parameter and returns the rounded value to the nearest int. It takes a second optional parameter which allows you to specify the number of digits after the decimal point to round to, as seen below.

LANGUAGE: Python3

```
intRound = round(5.6) # equals 6

floatRound = round(6.3345742, 3) # equals 6.335
```

The `abs()` function returns the absolute value of a number which is passed in as a parameter.
The `pow()` function takes two parameters, the first being the number and the second being the power of it we want to calculate, as seen below

LANGUAGE: Python3

```
powerTwo = pow(2, 3) # equals 8
powerThree = pow(3, 2) # equals 9
```

This function is the same as the `**` operator.

Math Module

Sometimes things we want to do mathematical things in Python which the base language can't do. To be able to do this, we import a library. This is some pre-written code which we can use in our programs.
To be able to use the math library, we first have to import it

LANGUAGE: Python3

```
import math
# or alternatively, if the line above doesn't work, use line below
from math import *
```

The math module provides a number of useful things including some constants (eg, `math.pi`) and mathematical functions (eg, `math.sqrt()`).

GRAPHICS, OBJECTS AND HIGH QUALITY CODE

10-10-22 15:00 Nadim RB LT1

Graphics

Graphics Introduction

Python, by default, does not contain a graphics system. We have to load the graphics code into the program, much the same as we do for the maths module. The graphics module we will be using was written by John Zelle. This defines the new classes which we have to use. The line of code shown below needs to be used at the top of the working python file to import the graphics module.

LANGUAGE: Python3

```
from graphics import *
```

The graphics module does not come pre-installed to Python 3 (like math) does. The package needs to be downloaded and saved either to where Python expects to find its modules or to the directory in which the file which uses it is saved.

Using the Graphics Module

Now we have loaded the graphics module, we can use it. To start with, we need to create a graphics window. We should assign it to a variable so that we can access it later and use it. The code to do this is shown below.

LANGUAGE: Python3

```
win = GraphWin("frameTitle", width, height)
```

The `GraphWin()` constructor has a number of optional parameters. Where these are omitted, the window will default to be 200px by 200px.
There are a number of different shapes available through the module.
To create a point object (which we need for a whole host of different things), you have to instantiate an object; the syntax for this is shown below.

LANGUAGE: Python3

```
p = Point(10,20)
```

Now we have a point (currently completely independent of the window we created earlier), we can do things with it. For example, we can draw it on the window, set its outline then move it to a different coordinate on the window.

LANGUAGE: Python3

```
p.draw(win)
p.setOutline("red")
p.move(40,10)
```

Notice how when we want to do something with `p`, we use the identifier (`p`) followed by a dot (`.`) followed by the name of the method which we want to apply to it (eg `draw()`).

We can also create circles (and lots of other shapes too)! The creation process for this is much the same as for a point. The syntax for this is shown below.

LANGUAGE: Python3

```
1 c = Circle(Point(10,10),30)
2 c.setFill("blue")
3 c.draw(win)
```

Notice how on line 1, we use a point to declare the coordinates of the circle.

Accessing Information

So far, the methods we have looked at manipulate the data, they set information. We can use get methods to get information about the various objects we are currently using. For example, we can use `getX()` to get the x coordinate of an object.

High Quality Code

> **High Quality Code**
>
> Code that is readable and code that is correct.

Readable Code

Program code is considered to be readable code where it can be easily understood by anyone who is familiar with programming in the language used but not necessarily familiar with what the code is supposed to be doing. This is important because in industry; software is often written and maintained by teams of people, the later can sometimes involve different people to the former.
To write readable code, it is important to name everything (functions, variables, etc...) with sensible names, use whitespace, write documentation (comments throughout the code or an accompanying document) and avoid over-complicating the code/ write repetitive code.

Good Names

Names of variables and functions must be legal. This means they must begin with either a letter or underscore, and only consist of letters, numbers and underscores. Also, they must not be keywords.
It is recommended to stick to one style of variable naming, for example `camelCase`.
When choosing names, it is good to choose something that relates to what the variable will be storing (eg `name` for the name of a user). Try to avoid abbreviations and using single letter names (apart from where it would be silly not to use them).

Whitespace

Where there is a block of code (e.g., functions, loops), these must be indented, with a `tab`.
There are a number of other standard conventions for whitespace: leave blank lines between functions; use a single space either side of an assignment operator; use a single space after commas.
Do not put whitespace between function names and brackets or before colons.

Length of Lines of Code

It is recommended to use 80 characters as a limit on how long a line of code can be. This makes the program easier to read and means that code will not be cropped or wrapped when you print it.

Documentation

When writing code, it is very good practice to document what your code should be doing. This helps when you return to your code in the future or if someone else has to do something with your code, it will help you understand what is going on with it. Documentation can be done in the form of comments.

> **Comment**
>
> A line of code which is ignored when the program is run. It allows developers to annotate their code.

It is a common misconception that more comments mean better code. This is not the case. In fact, if the code is really well written then comments shouldn't be needed.

Testing

When writing programs, it is a good idea to design test data (inputs which you can enter into the program where you know what the output should be so you can tell if the programme is working properly or not) before you begin programming. This allows you to test your program at various stages of development to make sure that your program is working correctly.

COMUTING WITH STRINGS & FILES

17-10-22 15:00 Nadim Zoom

Strings

Strings are kinds of sequences, there are other kinds of sequences which we will come across later in Python.

String Operations

There are a number of different operations we can perform on strings. The + operator concatenates two strings together and the * operator allows a string to be repeated multiple times; both can be seen in the program below.

LANGUAGE: Python3

```
words = "Hello"
print(words+ "there")
print(words * 3)
```

LANGUAGE: Unknown

```
Hellothere
HelloHelloHello
```

The function `len(stringName)` returns the number of characters in a string.

String Indexing

A string is a sequence of characters, each of the characters can be accessed individually using its index. Indexing begins at the first character, which has the index 0. Moving through the string, the indexes increase. We can access individual characters using the index notation, as seen below.

LANGUAGE: Python3

```
phrase = "Alright Dave?"
print(phrase[4])
```

LANGUAGE: Unknown

```
g
```

Python strings can also be indexed with negative indices where `-1` is the position of the final character, `-2` is the position of the penultimate character, and so on.

String Slicing

As well as being able to access individual character, we can access sub-sets of characters, also known as substrings. To do this, we use the notation `string[start:endPlusOne]`. This gives us a substring starting at position `start` and ending one position before `endPlusOne`.

String Methods

Along with the `len()` operation described earlier, there are a number of other useful string methods built in to python. `stringName.upper()` converts all letters within the string to uppercase. `stringName.replace(old, new)` replaces all the occurrences of `old` with `new`. `stringName.count(` counts all the occurrences of `toFind` within the string. `stringName.split()` splits the string into separate items in a list, split where the spaces were in the string.

String Formatting

Often programs need to display nicely formatted outputs. This can be achieved using the `.format()` method. The `.format()` method takes parameters of variables which need to be inserted into the string the method is applied to. Within the string, curly braces are inserted which contain the index of variable to be inserted within the `.format()` command; this can be seen below.

LANGUAGE: Python3

```
a = 12.55
b = 4
myString = "It will cost {0} pounds for {1} bottles of wine".format(a, b)
print(myString)
```

LANGUAGE: Unknown

```
It will cost 12.55 pounds for 4 bottles of wine
```

We can use the `.format()` method to format numbers and spaces too. This is done within the curly braces, where we add a colon then the formatting definition. The number before the decimal point is the total number of characters to include and after the decimal point tells python to use 2 decimal places. This can be seen below.

LANGUAGE: Python3

```
myString = "It will cost {0:10.2f} pounds for {1} bottles of wine".format(a, b)
print(myString)
```

LANGUAGE: Unknown

```
It will cost      12.55 pounds for 4 bottles of wine
```

Where the number before the decimal point is greater than the total length of the data to be inserted, Python pads out the gap with spaces.
If the number before the decimal point is smaller than the length of the data to be inserted, Python will ignore the number before the decimal point and will format the data as specified after the decimal point.
We are able to control where the padding text is using < (padding to the right), > (padding to the left) and ^ (equal padding each side). An example is shown below. The number is the total number of space allocated to the padding and the data to be inserted.

LANGUAGE: Python3

```
print("Here is a {0:^8} for you!".format("WORD"))
```

```
LANGUAGE: Unknown
Here is a   WORD   for you!
```

Sequences

Strings and lists are both examples of sequences, as a result of this, they share many properties. One such property being the ability to loop through all the indices within the sequence and perform an action with it. Another such property is the ability to concatenate, index and slice sequences.

Basic File Processing

This section will only introduce the processing of basic text files, which contain sequences of characters.
Text files are generally a few lines long, with each line ended by a special newline character. In python, this character is `\n`.
When Python reads in a text file, it reads it in as a single string, for example

```
to
be or not
to be
```

would be read in as

```
"to\nbe or not\nto be\n"
```

Basic file handling

In the following examples, the text file we are using will be called `myfile.txt` and that it is in the current directory.
To use the file, we first have to open it, as part of this we associate a variable with it and we have to declare the mode which we want to open the file in. The basic syntax is as follows

```
LANGUAGE: Python3
variableName = open(fileName, mode)
```

In our example of wanting to open ”myfile.txt” to read, we would use the following syntax

```
LANGUAGE: Python3
inFile = open("myfile.txt", "r")
```

After we have processed the file, we have to close it. This ensures the correct correspondence between the file variable and what is actually on the disk.
The syntax to close the file is `.close()`

Reading Data From A Text File

To read data in form the file, we have two options. We can either use `.read()` which reads the entire file’s contents into a single variable or use `.readlines()` which reads the file line by line into a list where each line is a different element in the list. There is also a `.readline()` method which only reads a single line at a time, this can be used where the file to be read in is very large and it would be detrimental to the memory of the system to read the whole file in at once. We can also use a for loop to iterate through the file, reading it line by line.

Writing to files

When we want to write something out to a file, we first have to open it for writing. This is done by using `mode` as `"w"`. This will either create the file, or if the file already exists, destroys its contents. We can then use a print statement to write out to the file, as seen below

LANGUAGE: Python3

```
print(contentToWrite, file=variableNameOfFile)
```

We then have to remember to close the file, this will ensure that the data is written to disk.

DEFINING FUNCTIONS

24-10-22 15:00 RB LT1 Nadim

Concept of Functions

In the programs we have written this far, we have been using functions to contain many smaller programs within one file. However, most programs out there in the 'real world' are longer than the programs which we've written so far.
Typically a program is a collection of several funciton definitions. Functions help us break large problems into smaller parts; improve the readability of code; and avoid repetition whereby we write similar code over and over again.

Breaking a large problem down using Functions

Often, when we are designing solutions to large problems we will break the problem down into smaller sub-problems. These sub-problems are much easier to solve, making the overall problem easier to solve. Some of the sub-problems which we first come up with, might be able to be broken down further into even smaller sub-problems, it is often these which can become programmed functions.

Parameters

When calling a function, it is useful to be able to pass data into the function so that it can perform an action on this. To do this, we use something called a parameter. In the example code shown below, the function (`greet()`) takes a single parameter, this is used in the print statement. Note that where the function is called from, we must supply an argument in the brackets or we will get an error.

LANGUAGE: Python3

```
1 def greet(name):
2     print("Hey there " + name + "!")
3
4 greet("Dave")
```

Example Execution

In Lecture 08d notes on Moodle, there is an example line-by-line execution of a function with parameters.

Returning Values

Whilst it can be useful to have a function that does something, and doesn't feed back into the main program, this isn't useful in reality. Functions also have the option to return a value at the end of execution to where they were called from in the main program. There are examples of this throughout Python which we have used already (for example, `float()`). The code below shows an example of a function which takes a parameter and returns a value.

LANGUAGE: Python3

```
def addTwo(x):
    y = x + 2
    return y

a = 3
b = addTwo(a)
print(b)  # outputs 5
```

Note that the `return` keyword defines what will be returned to the main program and that within the main program, there needs to be somewhere for the returned value to go otherwise the program will throw an error.

Returning Multiple Values

We may also find it useful at times to write functions which return multiple values. The code snippet shown below demonstrates how this is done.

LANGUAGE: Python3

```
def sumDiff(n1,n2):
    return n1+n2, n1-n2
s, d = sumDiff(10,3)
print(s)  # outputs 13
print(d)  # outputs 7
```

Future Weeks

The concepts introduced in this week are used throughout the upcoming weeks and in the coursework.

In Class Test

For the in class test this week, we will be instructed to download a python file from Moodle which will contain a stick man.
We will have to do something to the stick man (for example, add a top hat and a cane). This will be worth 6 marks. The sheet may not be photo copied in colour, pay attention to the colours written in the document.
The remaining marks in the test will come from making other things appear and move on the screen. This will be worth 4 marks.
Code quality, as well as outputs, will be assessed. We will not be marked down for lack of comments at this stage however we may loose marks if the code is inefficient or repetitive. This would be one or two marks at most.

Decision Structures, IF Statements and While Loops

07-11-22 1500 Nadim RB LT1

Up to this point in the course, all the code we have been written is executed sequentially, one line after the other. In this lecture, we'll be looking at how we can control the flow through a program using decision structures and how we can use loop structures that allow us to execute statements repeatedly.

Decisions

Algorithms can contain decisions. A commonly used decision structure is called an `if statement`. If statements take a condition (the thing which the output is dependent on) and they can have a number of possible outputs. An example of a simple if statement is shown below.

LANGUAGE: Python3

```
x = 45
if (x >= 40):
    print("Your value of x is bigger than 40")
```

Flowcharts can also be used to represent decision structures.

Conditions

Conditions are expressions of the data type `bool` of Boolean. This data type has just two values, `True` and `False`.
We can form Boolean expressions using the following operators.

Syntax	In English...	Code Example	Explanation
==	Equal-to	`X == Y`	Returns True if X is equal to Y; otherwise returns False.
!=	Not-equal to	`X != Y`	Returns True if X is not equal to Y; otherwise returns False.
>	Greater-than	`X > Y`	Returns True if X is greater than Y; otherwise returns False.
>=	Greater-than or equal-to	`X >= Y`	Returns True if X is greater than or equal to Y; otherwise returns False.
<	Less-than	`X < Y`	Returns True if X is less than Y; otherwise returns False.
<=	Less-than or equal-to	`X <= Y`	Returns True if X is less than or equal to Y; otherwise returns False.

Multi-Way Decisions

We will often need to have multiple outcomes from a decision structure. There are two additional bits we can use in the `if` statement, `elif` and `else`.

`elif`

ELse-IF structures allow us to have multiple conditions in one if statement.

`else`

Else structures will be executed if no `if` or `elif` conditions are true. In the example below, a comment is outputted to the user based on the mark they entered.

LANGUAGE: Python3

```
mark = int(input("Enter your mark: "))
if mark >= 70:
    print("Clever Cloggs")
elif mark >= 60:
    print("Try harder")
elif mark >= 50:
    print("Really, this is the best you could do?")
elif mark >= 40:
    print("Did you even try?")
else:
    print("Failure")
```

Designing Decision Structures

When designing decision structures, it is most efficient to write the if statement such that the code executed inside the if statement is executed in rarer cases than not executing it. For example, a kebab shop where orders over £50 get a 10% discount, the discount application code would be placed inside an if statement, rather than always being applied then if the order is under £50, the discount is reversed.

Review of For Loops

For Loops are used to iterate through a sequence of values. A for loop includes a loop variable, a sequence and a body.
In most `for` loops we have written so far, we have been using the `range()` function to allow us to iterate through a series of numbers. The `range()` function is actually more complex than it looks. It can take three arguments, `range(m, n, s)` where `m` gives the start of the sequence, `n` the step after the stop point and `s` give steps. This can be seen in action, in the code below.

LANGUAGE: Python3

```
for x in range (5, 0, -1):
    print(x)
```

LANGUAGE: Unknown

```
5
4
3
2
1
```

Nesting For Loops

We are able to nest `for` loops inside each other. When doing this, its important to ensure that the two loops have different loop variables, for example `x` and `y`.

While Loops, Booleans and Further Loops

14-11-22 15:00 Nadim RB LT1

While Loops

Whilst it can be useful to iterate a defined number of times, often we need to iterate until a condition changes. The time taken for this condition to change may be different every runtime. For this reason, there's another type of loop. The `while` loop iterates until a condition is met. The basic syntax is as follows.

```
while condition:
    statement(s)
```

At every run of the loop, the condition is checked and if the statement is `True`, the body of the loop is executed, then the loop returns to the top to check the condition again and if the condition is `False`, the body is skipped and the loop terminates.
When designing a while loop which iterates until the user instructs it to stop; the user can directly change the loop condition variable inside the loop. However, this isn't great; it would be better to use a sentinel loop.

Sentinel Loop

Sentinel loops are loops in which the question is asked before the loop is entered, then the loop begins with the processing before asking the question at the end. This reduces unnecessary processing whereby if the loop is about to be exited from or it doesn't need to be entered, the processing of that input doesn't need to happen. The basic pattern is outlined below

```
value = input()
while value != sentinel:
    process value
    value = input()
```

Boolean Operators

Python includes the `and`, `or` and `not` boolean operators. These work exactly the same as the logic gates do, except a 1 is represented by `True` amd a 0 is represented by `False`.
If we are writing a condition which uses multiple statements, joined by a boolean operator, all statements have to be written in full.

Break

The `break` statement allows us to exit a loop as and where we want. The following code is an example of input validation within a while loop.

LANGUAGE: Python3

```python
def getOneToTen():
    while True:
        number = int(input("Enter a number: "))
        if number >= 1 and number <= 10:
            break
        print("'Thats not between 1 and 10")
```

```
7     return number
```

The decision to use a `break`, as above, or to use a different condition for the loop and change that to exit the loop is up to the developer and their opinions on readability of code.

DESIGN AND SIMULATION

21-11-22 15:00 Nadim RB LT1

Coursework is now available on Moodle in the General Section. There is a FAQs document linked below it. The coursework is due on 13th December at 11pm.

Introduction

Up to this point, most of the programs we have been writing have been fairly short. This lecture will introduce a concept called 'Top Down Design' whereby a big problem (ie a complete program) is broken down into a series of smaller sub-problems, which are easier to solve. The idea of top-down design is to express a solution to a large problem in terms of smaller sub-problems.

Generating Random Numbers

We can use the `random` module to generate a random number as can be seen in the example below.

LANGUAGE: Python3

```python
from random import *
print(random()) # outputs 0.95310838187740532
```

Process of Top-Down Design

The first step to top-down design is to decide the inputs & outputs for the program. We then look at the main program flow, which will probably follow the following

1. Get input from the user
2. Run the main processing code on those inputs
3. Output something to the user

First Level Design

At the first level of design, we write the `main()` function. To do this, we assign names to the functions which we have designed in the step above. As part of this, we work out what parameters these functions will need and what values they will return.

Second Level Design

Now we are left with sub-problems. For each, we know what parameters they will take and what values they will return. As part of doing this we might realise that there is a sub-sub-problem somewhere. This will probably be a bit of a sub-problem which is more complex than the rest. At this stage we can just fill in parameters and the function name as we did for first level design.

Third Level Design

We now are left with sub-sub-problems. For each of them, we know what parameters they will take and what values they will return. As part of doing this, we might realise that there is a further problem (for example, deciding if a game has been won or not) which we will solve outside of this function. At this stage we can write the returned values, function name and any parameters which it will take.

Fourth Level Design

Problems at this stage will probably be very small and quick to implement. If there are any further sub-problems they pose, then more levels of design can be added as needed.

Execution

As our program is now written in its own file, we can execute it by calling the first-level design function at the bottom of the file. All the other functions can come anywhere above this line.

LECTURE: Using Lists, Tuples and Dictionaries

28-11-22 15:00 RB LT1 Nadim

NB: These notes were types up during January 2023 in preperation for Teaching Block 2. The lecture did not cover this content at the time as coursework support was given during lectures and practicals for the rest of Teaching Block 1 after it was released.

Storing Data

Often when programming, our programs need to process large collections of data of the same type. This may include: words in a document, temperatures for each day in a year, and marks from student's work.
There are a number of different data structures which can be used to store data. Three of these will be explored in this lecture.

Lists

Some languages, for example Java or C#, use the term 'array' instead of 'list'.
An example of creating two different types of list can be seen below.

LANGUAGE: Python3

```python
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
seasons = ["spring", "summer", "autumn", "winter"]
```

List Indexing

Lists are indexed in the same way as strings. They start with position `0`. Lists can also be indexed using negative indices.

LANGUAGE: Python3

```python
print(seasons[0])  # outputs 'spring'
print(fibonacci[8])  # outputs 21
print(seasons[-1])  # outputs 'winter'
print(fibonacci[-2])  # outputs 21
```

Basic List Operations

Lists, like strings, have operators for concatenation `+` and repetition `*`.
We can find the length of a list with the built-in function `len()`

LANGUAGE: Python3

```python
print(fibonacci + [55,89])
# outputs [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

print([10, 4] * 3)
# outputs [10, 4, 10, 4, 10, 4]

print(len(seasons))
# outputs 4
```

List Slicing

We can use list slicing to get sub-lists from the lists. This behaves in the same was as string slicing and uses the same syntax. The substring contains everything from and including the first value up to and not including the final value. Where the first or last value is omitted, it is assumed to be the beginning/ end of the list.

LANGUAGE: Python3

```
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
print(fibonacci[3:7])
# outputs [2, 3, 5, 8]

seasons = ["spring", "summer", "autumn", "winter"]
print(seasons[1:-1])
# outputs ['summer', 'autumn']
print(seasons[2:])
# outputs ['autumn', 'winter']
print(seasons[:-2])
# outputs ['spring', 'summer']
```

Iteration through a list

We can iterate (*loop*) through the elements in a list using a for loop.

LANGUAGE: Python3

```
for season in seasons:
    print(season, end=" ")

# outputs spring summer autumn winter
```

If we wanted to iterate through the indices of a list, we can use the verb|len()| function to get that value.

Membership Checking

It can be useful to know if a value appears in a list. This can be achieved using the `in` operator. `in` returns `True` if the value appears and `False` if the value does not.

Changing an element of a list

Lists are *mutable*, this means we can *alter* their elements. We can change list elements using assignment statements with list indexing.

LANGUAGE: Python3

```
shopping = ["jam", "eggs", "margarine", "sugar"]
shopping[2] = "butter"
print(shopping)
# outputs ['jam', 'eggs', 'butter', 'sugar']
```

List Methods

`append` can be used to add a new value to the end of a list.

LANGUAGE: Python3

```
shopping.append("eggs")
print(shopping)
# outputs ['jam', 'eggs', 'butter', 'flour', 'eggs']
```

`remove` can be used to remove the first occurrence of an element.
`index` can be used to get the first position of the first occurrence of a value.
`sort` method sorts the list into order.

Storing Objects in a list

We can store any kind of data in lists, not just numbers and strings. Objects get stored in the same way as any other data type.

Mixed-Type Lists

Lists *can* contain different types of data. This can make code difficult to understand therefore it it not regarded as best practice.

Nested Lists

Lists can be nested within other lists. This can be *really* useful.

LANGUAGE: Python3

```
matrix = [[1, 2], [3, 4]]
print(matrix[1][0])  # outputs 3
```

Tuples

Sometimes its useful to collect two or more related items of information together. A tuple can be used for this purpose.

LANGUAGE: Python3

```
exampleTuple = ("Dave", 44)
```

We use parentheses rather than square brackets to denote a tuple compared to a list. Elements can be indexed using the same notation as for lists and strings.
Tuples are *immutable*, this means we *can't* change the elements after assignment.
Tuples are often used to return multiple values form a function and we can store tuples in a list.

Dictionaries

Dictionaries can be regarded as unordered collections of data, whose values are indexed by key. Dictionaries are sometimes called mappings, hashes or associative arrays. Dictionary literals are written as aa sequence of `key:value` pairs within braces { & }.
The example below shows a mapping from string keys to integer values.

LANGUAGE: Python3

```
shopping = {"eggs" : 2, "ham" : 4, "jam" : 1}
```

We use the key within square brackets to access the value this key maps onto

LANGUAGE: Python3

```
print(shopping["ham"])  # outputs 4
```

We can use the key within square brackets to update a value in a dictionary, as we would in a list. We can also add new keys and values to a dictionary using the key in square brackets and assigning a value to it.
We can test if a particular key appears in a dictionary using the `in` operator.
We can delete entries using Python's built in `del` command.
We can obtain lists of all the keys and all the values in a dictionary using the `keys()` and `values()` methods inside the `list()` function.
We can iterate through the keys using a for loop, and use this to access the value (with square brackets).