CONTROL STATEMENTS

DR IAN CORNELIUS

HELLO

- Learning Objectives
    1. Understand what a control statement is
    2. Demonstrate the ability to use control statements

INTRODUCTION TO CONTROL STATEMENTS

- Typically, statements in code will be executed sequentially
- There are some situations which requires a block of code to be repeated
  - i.e. summing numbers, entering multiple data points, capturing user input
- Control statements, otherwise known as **loop statements** are required
- Python have two loop structures:
  - `while` - conditional loops
  - `for` - counter controlled loops

STRUCTURE OF A LOOP

- Loop structures can be likened to a conditional statement
  - they run on a `True` or `False` set of values
  - the loop will continuously loop until the condition is `True`
  - the loop will terminate when the condition is `False`
- Loops can run for a desired length of time
  - or until a user-defined flag terminates it
- Loops are great for re-using code
  - limiting the number of statements that are required
  - re-uses the same conditional arguments for testing instead of hundreds

`WHILE` LOOP STATEMENTS (1)

- `while` loops, are loops that will execute zero or more times before it is terminated
- The `while` loop structure:

```
while variable < value:
        ...
        variable += 1
```

- If you are doing an incremental loop, you need to manually increase the variable
  - hence the `variable += 1`

`WHILE` LOOP STATEMENTS (2)

- Initially `x` is `0` and the loop will increment `x` for each iteration
- This is repeated until `x < 10`

```
x = 0
while x < 10:
    print(x, end=" ")
    x += 1
```

```
0 1 2 3 4 5 6 7 8 9
```

WHILE LOOP WITH A BREAK STATEMENT (1)

- `break` statements can be used to stop the loop if a condition is evaluated to `True`

```
x = 0
while x < 10:
    print(x, end=" ")
    if x == 5:
        break
    x += 1
```

```
0 1 2 3 4 5
```

`WHILE` LOOP WITH A BREAK STATEMENT (2)

- Infinite loops can be constructed by using a `True` value after the `while` keyword
- Will continue incrementing `x` until it reaches a certain value
  - in this instance `x` must be equal to `10`
- If there is no condition to check in the loop it will continue incrementing

```
x = 0
while True:
    print(x, end=" ")
    if x == 20:
        break
    x += 1
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

`WHILE` LOOP WITH CONTINUE STATEMENT

- `continue` statements can stop the current iteration and continue onto the next

```
x = 0
while x < 10:
    x += 1
    if x == 5:
        continue
    print(x, end=" ")
```

```
1 2 3 4 6 7 8 9 10
```

`WHILE` LOOP WITH AN ELSE STATEMENT

- `else` statements can be used to executee a block of code when a condition has been met
  - that is the condition is no longer `True`

```
x = 0
while x < 10:
    print(x, end=" ")
    x += 1
else:
    print("\n\nx is no longer less than 10")
```

```
0 1 2 3 4 5 6 7 8 9

x is no longer less than 10
```

`FOR` LOOP STATEMENTS

- A `for` loop is a loop that is designed to increment a counter over a given range of values
- They are best suited for problems that need to iterate a specific number of times
  - i.e. looping through a directory or set of files
- Considered to be a **pre-test** loop
  - they check their condition before execution
- `for` loops are useful because…
  - they know the number of times a loop should be iterated
  - they use a counter
  - require a `False` condition to terminate the loop

LOOP STRUCTURE (1)

- The structure of a `for` loop consists of the following:
  1. Initialisation of a counter
  2. Test the counter variable:
     a. less than: `start < stop`
     b. greater than: `start > stop`
  3. Update the counter variable

```
for variable in range(x, y, step):
        ...
```

LOOP STRUCTURE (2)

- `for` loops can also be iterated forwards by using a positive step value

```
for x in range(1, 10, 1):
    print(x, end=" ")
```

```
1 2 3 4 5 6 7 8 9
```

LOOP STRUCTURE (2)

- `for` loops can also be iterated backwards by using a negative step value

```
for x in range(10, 1, -1):
    print(x, end=" ")
```

```
10 9 8 7 6 5 4 3 2
```

NESTED LOOPS

- These are loops that are located inside the body of another loop
  - consist of an inner (inside) and outer (outside) loops

```
for i in range(1, 4, 1):
        print(f"Iteration {i}: ")
        for j in range(1, 4, 1):
            print(i * j, end=" ")
        print("\n\n\n")
```

```
Iteration 1: 1 2 3
Iteration 2: 2 4 6
Iteration 3: 3 6 9
```

- The inner loop will go through all the repetitions for each repetition of the outside loop
  - the inner loop repetitions will complete sooner than the outside
- Nested loops are necessary for when a task performs a repetitive operation, and that task itself needs to be repeated

LOOPING THROUGH OBJECTS (1)

- `for` loops are great for looping through various objects:
  - i.e. strings, lists, tuples or dictionaries

STRINGS

- Strings are iterable, as they consist of a sequence of characters

```
title = "4061CEM"
for letter in title:
    print(letter, end=" ")
```

```
4 0 6 1 C E M
```

LOOPING THROUGH OBJECTS (2)

LISTS

- Lists are also iterable, as such they can be looped through

```
module = [4061, "Programming and Algorithms", "Ian Cornelius"]
for item in module:
    print("\n\n", item)
```

```
4061


Programming and Algorithms


Ian Cornelius
```

LOOPING THROUGH OBJECTS (3)

DICTIONARIES (I)

- Dictionaries are also iterable, but can be looped through in a variety of ways

ITEMS OF A DICTIONARY

- Returns the key and value of each item in a dictionary

```
module = {"code": 4061,
             "title": "Programming and Algorithms",
             "leader": "Ian Cornelius"}
   for key, value in module.items():
       print("\n\n", key, "=", value)
```

```
code = 4061
title = Programming and Algorithms
leader = Ian Cornelius
```

LOOPING THROUGH OBJECTS (4)

DICTIONARIES (II)

BY A KEY OF THE DICTIONARY

- Returns the key of a dictionary, which can be used to access the item in a dictionary

```
module = {"code": 4061,
             "title": "Programming and Algorithms",
             "leader": "Ian Cornelius"}
   for key in module.keys():
       print("\n\n", key, "=", module[key])
```

```
code = 4061
title = Programming and Algorithms
leader = Ian Cornelius
```

LOOPING THROUGH OBJECTS (5)

DICTIONARIES (III)

VALUES OF A DICTIONARY

- Returns all values in the dictionary, but not the key associated to it

```
module = {"code": 4061,
          "title": "Programming and Algorithms",
          "leader": "Ian Cornelius"}
for v in module.values():
    print("\n\n", v)
```

```
4061


Programming and Algorithms


Ian Cornelius
```

LOOPING THROUGH OBJECTS (6)

DICTIONARIES AND LISTS

```
module = {"code": 4061,
             "title": "Programming and Algorithms",
             "leader": "Ian Cornelius",
             "team": ["Terry Richards", "Daniel Goldsmith"]}
for key in module.keys():
    if type(module[key]) != list:
        print("\n\n", key, ":",  module[key])
    else:
        print("\n", key, ":", end=" ")
        for i in range(len(module[key])):
            print('\t', module[key][i], end=" ")
```

```
code : 4061
title : Programming and Algorithms
leader : Ian Cornelius
team : Terry Richards Daniel Goldsmith
```

INFINITE LOOPS

- Loops must have a way of terminating, otherwise the loop will continue to repeat until it is manually interrupted
- Infinite loops can occur when you forget to write code inside the loop to terminate the loop
  - i.e. there is no condition for a `boolean` expression to evaluate to `False`
- You must remember to include some condition to break out of a loop

```
x = 1
while x == 1:
    y = input("Enter a number: ")
    print(f"The number you entered is {y}!")
```

```
x = 1
while x == 1:
    y = input("Enter a number (Type 'q' to quit): ")
    if y == 'q':
        break
print(f"The number you entered is {y}!")
```

GOODBYE

- Questions?
  - Post them in the **Community Page** on Aula
- Contact Details:
  - Dr Ian Cornelius, ab6459@coventry.ac.uk