M30299 – Programming
Lecture 12 – Using While Loops

Matthew Poole & Nadim Bakhshov
moodle.port.ac.uk

School of Computing
University of Portsmouth

2020/21

Introduction to lecture

- In the last lecture we reviewed `for` loops, and saw that these are used for repeatedly executing a section of code.
- For loops are suited to executing code a number of times that can be predicted **in advance** (i.e. just before the loop starts). For example, the body of:

```
>>> for i in [2, 9, 8, 1]:
        print(i)
```

is executed 4 times, and the body of:

```
>>> for i in range(n):
        print(i)
```

is executed `n` times (whatever the value of `n` is before the loop).

`while` loops

- Often programs need to execute a segment of code a number of times that is **difficult** or **impossible** to predict in advance.
- For example:
  - executing some code that asks the user for a month value (i.e. between 1 and 12) until they enter a valid value; and
  - repeatedly halving and displaying a variable's value until it falls below 1.
- For such situations, we need to use a different kind of loop, known as a `while` loop.

while loops

- Let's see an example while loop that deals with the second problem given above:

```
i = int(input("Enter a number: "))
while i >= 1:
    i = i / 2
    print(i)
```

`while` loops

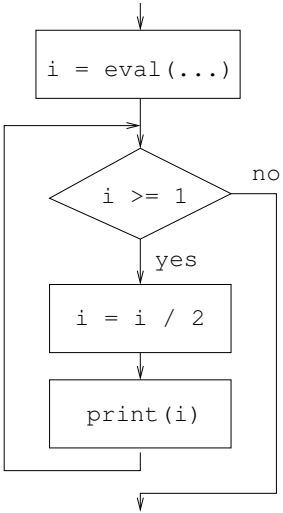
- We see that a `while` loop has the form:

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

  and is executed as follows:

- The boolean **condition** is checked (evaluated):
  - if it is `True`, the statement(s) in the **body** are executed, and then the whole loop is re-executed.
  - if it is `False`, the body is skipped and the loop terminates.

Flowcharts for `while` loops



Another `while` loop example

- The example below calculates how many years it will take for an investment to reach £1M, given an interest rate of 5.5%.

```
def yearsToBecomeMillionaire(amount):
    interestRate = 0.055
    year = 0
    while amount < 1000000:
        amount = amount * (1 + interestRate)
        year = year + 1
    return year
```

- We can't easily predict how many times the loop body will be executed (this is why a `for` loop should not be used here).

Design of a `while` loop

- Consider a function that adds up a sequence of numbers entered by the user.
- If we used a `for` loop to do this, the function would have to ask the use how many numbers there are at the beginning; e.g:

```
>>> addUpNumbers()
How many numbers are there: 2
Enter a number: 7
Enter a number: 4
The total is 11
```

Design of a `while` loop

- Let's write a version where this is not the case.
- We need some mechanism to allow the user to say that there are no numbers left.
- The easiest way (maybe not the best) is to ask the user if there are any more numbers after they enter each number; e.g.

```
>>> addUpNumbers()
Enter a number: 7
Any more numbers: y
Enter a number: 4
Any more numbers: n
The total is 11
```

- Let's consider how to design a function to perform this task.

Design of a `while` loop

- We should already realise that we need a variable that keeps a running total of the entered numbers (a good name is `total`).
- This variable needs to be created & initialised to 0 at the beginning, and its value displayed at the end.
- A first stab at an algorithm to solve the problem might be:

```
total = 0
while loop needs to continue
    number = int(input("Enter a number: "))
    total = total + number
    moreNumbers = input("Any more numbers? ")
print("The total is", total)
```

Design of a `while` loop

- The only thing left to deal with is the while loop condition.
- Clearly, this needs to use the value of the string variable `moreNumbers` (i.e. the user's response):
  - if the user responds with "y", we wish the condition to be true (i.e. so the loop body is re-executed); and
  - if the user responds with "n" (or anything that isn't "y") we wish the condition to be false (to exit the loop).
- We also need the condition to be true the first time it is evaluated (i.e. before the user has been asked for any input).
- We use `moreNumbers == "y"` as the loop condition, and ...

Design of a `while` loop

- ... initialise the variable `moreNumbers` before the loop to a value that satisfies this condition. We obtain:

```
def addUpNumbers():
    total = 0
    moreNumbers = "y"
    while moreNumbers == "y":
        number = int(input("Enter a number "))
        total = total + number
        moreNumbers = input("Any more numbers? ")
    print("The total is", total)
```

- The above function isn't yet perfect – its annoying to keep entering "y" to continue.

Sentinel loops

- A better solution is to allow the user, when prompted with:

  ```
  Enter a number:
  ```

  to be able to type a special value to signal the end.
- A special value that signals the end of a loop is called a **sentinel**, and loops that use this technique are called **sentinel loops**.
- The basic **pattern** for using sentinel loops for user input is:

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

- Notice here that the sentinel value is not processed.

Sentinel loops

- We need to choose a sentinel value for a new version of our `addUpNumbers` function.
- One option (maybe not the best) for the sentinel is 0, since the user probably wouldn't want to add a 0.
- We make changes to turn our code into a sentinal loop:

```
def addUpNumbers():
    total = 0
    number = int(input("Number (0 to stop): "))
    while number != 0:
        total = total + number
        number = int(input("Number (0 to stop): "))
    print("The total is", total)
```

Sentinel loops

- 0 is not really a good sentinel – let's change it!
- What would a better sentinel be? Possibly nothing - i.e. let the user press return without a value to denote the end.
- i.e. we need to modify the function so that it behaves as follows:

```
>>> addUpNumbers()
Number (return to stop): 7
Number (return to stop): 4
Number (return to stop):
The total is 11
```

- The problem now is that the **type** of the data values (int) differs from that of the sentinel value (which is what?)

Sentinel loops

- This problem can be solved by:
  - reading the user's input as a string (using `input`);
  - comparing this with the empty string `""`; and
  - converting it into an `int` using `int` inside the loop.
- This gives us an arguably better function:

```
def addUpNumbers():
    total = 0
    nStr = input("Number (return to stop): ")
    while nStr != "":
        number = int(nStr)
        total = total + number
        nStr = input("Number (return to stop): ")
    print("The total is", total)
```