CE151

Introduction to Programming
Part 3
While Loops and Functions

While Loops 1

Programs often require the repetition of execution of some fragment of code, for example the processing of input items one by one. On many occasions the number of times the execution should be repeated will differ according to the input data.

To perform such repetitions in Python (and many other high-level languages) we use a ***while loop***. This causes the execution to be repeated while some condition is true.

[The Python language also has another kind of loop which we shall introduce later.]

While Loops 2

Consider the task of washing the dishes. We could describe this task in the following way.

```
fill bowl with hot water
add washing-up liquid
while there are dirty dishes
  wash a dirty dish
  dry it
pour water from bowl into drain
```

The ***loop body*** (the indented "code") will be performed repeatedly until there are no more dirty dishes.

While Loops 3

Observe that we cannot determine by looking at the code how many times the wash-and-dry sequence will be performed; this will depend on the number of dirty dishes. Furthermore we do not even know if it will be performed at all – if there are no dirty dishes there will be nothing to wash. (In this case we should not waste resources by filling the bowl so before starting the task we should check if there are any dirty dishes!)

While Loops 4

While loops in the Python language have a similar structure as that used in the dish-washing example. The general form of a while loop is

```
while expression :
   statements
```

If the loop body is a single statement it may appear on the same line:

```
while expression : statement
```

The expression should ideally evaluate to either `True` or `False` but any expression is permitted. If its value is numeric, any non-zero value will be regarded as being true.

While Loops 5

The expression is evaluated and if its value is `True` (or regarded as being true), the loop body will be executed. After this has been done the whole process is repeated. This process continues until the value of the expression becomes `False`.

The expression used in a while loop should normally be some form of comparison or an expression built using one or more of the logical operators. We can then be sure its value is either `True` or `False`.

While Loops 6

The expression used to control a while loop should involve the use of at least one variable whose value could potentially be changed within the loop body. If this is not the case and the expression is true on entry to the loop it will remain true after each iteration and the loop will never terminate.

(There are rare occasions when such behaviour is desired – in this case a loop of the form `while True :` ..... would be used.)

While Loops – A First Example 1

As a first example we shall consider a program to display temperature conversions using a range specified by the user. Typical output would be:

```
11 Celsius is 51.8 Fahrenheit
12 Celsius is 53.6 Fahrenheit
13 Celsius is 55.4 Fahrenheit
14 Celsius is 57.2 Fahrenheit
15 Celsius is 59.0 Fahrenheit
```

If some of the values were less than 10 or greater than 99.9 the output would not look as neat; it would be better to produce some form of tabular output but we do not yet know how to perform string formatting.

While Loops – A First Example 2

We must first input the start and finish Celsius values, and then use a loop with the Celsius value starting at the start value, incrementing in each iteration and continuing until it is greater than the finish value.

```
print("Enter range in Celsius")
start = int(input("Start value: "))
finish = int(input("End value: "))

cels = start
while cels <= finish :
  fahr = cels * 9/5 + 32
  print(cels, "Celsius is", fahr, "Fahrenheit")
  cels = cels + 1
```

While Loops – A First Example 3

Note that when the body of the loop comprises more than one statement these statements must be written as a suite; they must be indented further than the line containing the `while` keyword and all be indented by the same amount.

If in our program the user supplies a start value that is greater than the finish value the loop condition would be false when it is first checked and the loop body would not be entered. In this case we might wish to output a warning message, or perhaps present the table in descending order. On the following slide we present a version that uses the latter approach.

While Loops – A First Example 4

```
print("Enter range in Celsius")
start = int(input("Start value: "))
finish = int(input("End value: "))

cels = start

if start > finish :
  while cels >= finish :
    fahr = cels * 9/5 + 32
    print(cels, "Celsius is", fahr, "Fahrenheit")
    cels = cels - 1
else :
  while cels <= finish :
    fahr = cels * 9/5 + 32
    print(cels, "Celsius is", fahr, "Fahrenheit")
    cels = cels + 1
```

While Loops – A Second Example 1

As another example we will write a program to calculate the average examination mark for a module. Since the program could be used for different modules we cannot specify how many students there are so we need some way of detecting the end of the input. If the data came from a file we could do this by detecting the end of the file but at the moment we know only how to obtain input from the keyboard. Hence we ask the user to enter a negative number after the last mark. Valid marks can be real numbers in the range 0.0 to 100.0, so we need to store the marks, and the running total, as values of type `float`. In order to calculate the average we need to count how many marks have been entered – the counter will be an integer.

While Loops – A Second Example 2

```
total = 0.0
count = 0
print("Enter marks one per line")
print("Use a negative number to end")
mark = float(input("Mark: "))

while mark >= 0 :
  total = total + mark
  count = count + 1
  mark = float(input("Mark: "))

print("The average mark is", total/count)
```

While Loops – A Second Example 3

Note that if the user types a negative number as the first entry, the loop body will not be entered and the variable `count` will still hold the value 0 when the division takes place. This will cause the program to terminate with an error so we should use an `if` statement to check if the value of `count` is 0 before attempting to output the result.

It should be observed that the first input had to be performed before entering the loop in order to have a value to use when checking the termination condition. If the code to perform the input is complex we can avoid duplication of code by using a different approach, as seen on the next slide.

While Loops – A Second Example 4

```
total = 0.0
count = 0
finished = False
print("Enter marks one per line")
print("Use a negative number to end")

while not finished :
  mark = float(input("Mark: "))
  if mark < 0 : finished = True
  else :
    total = total + mark
    count = count + 1

if count > 0 :
  print("The average mark is", total/count)
else:
  print("No marks entered")
```

Functions 1

Consider a recipe book. Several recipes may require the use of puff pastry. Instead of giving instructions telling the reader how to make puff pastry every time it is needed the author will probably provide a single recipe for puff pastry and then use a phrase such as "Prepare the puff pastry as described on page 27" whenever a recipe needs it.

A similar approach should be taken to program design. If some non-trivial task needs to be done in several different places in the program we should provide the code to perform the task as a ***function definition*** and then whenever the task needs to be performed we simply write one statement to invoke the function.

Functions 2

A function is a piece of code that is defined in one part of a program and used elsewhere. There are several reasons why large programs should make use of functions:

- program development becomes easier to control if a large program is broken down into several smaller tasks
- the same actions may need to be performed in several places; by encapsulating these actions inside a function we avoid having to write the code more than once; furthermore if we wish to change the way these actions are performed the change is needed in only one place
- the code in a function will usually be independent of the rest of the program so the function may be re-used in other programs

Functions 3

Suppose we have a program that requires user input using particular formats in several places. Each time an input value is obtained we must check that it is of the appropriate format. If it is not we need to output a message such as

```
Invalid input
Please try again
```

This output can be generated using two lines of code, but it is inconvenient to have to type these lines several times at different places in the program. Instead we can write a function to generate this output and call (or invoke) the function whenever we need the output.

Functions 4

To define the function we use a **`def`** statement:

```
def invalid() :
  print("Invalid input")
  print("Please try again")
```

The parentheses are necessary in all function definitions; in this case there is nothing inside them, but as we shall soon see in most function definitions there will be something inside them.

To invoke the function we simply write a statement containing its name followed by parentheses. When there is nothing inside the parentheses in the function definition there should also be nothing inside the parentheses in the call:

```
if month > 12 : invalid()
```

Functions 5

Some recipes in the book described on slide 16 may require more puff pastry than others, so the quantities of the ingredients listed on page 27 may have to be varied.

Assume that the recipe on page 27 produces 250g of puff pastry. A recipe may then say "Prepare 500g of puff pastry using the recipe on page 27." A user of the book would be unlikely to prepare two separate batches of 250g and then combine them; instead he or she would follow the recipe on page 27 but use twice the quantity of all of the ingredients listed on that page.

Functions 6

In the same way as the amount of puff pastry needed will vary between recipes, the values used in a function in a Python program are likely to vary when that function is called under different circumstances – the function will operate on different data each time it is invoked.

Arguments are used to supply data to a function; functions may have no arguments, one argument, or many arguments.

When a function is declared the names of its arguments are listed in the parentheses after the function name (separated by commas) e.g.

```
def printAverage(a, b, c) :
    ......
```

Functions 7

To call a function we need to supply values for the arguments, e.g.

```
printAverage(3, 17, 25)
printAverage(firstMark, secondMark, thirdMark)
```

In Python (unlike C or Java) we cannot specify in the function definition the types of the values that should be supplied as arguments. Hence documentation should be provided indicating what types of values are expected.

A function may have a documentation section similar to that for a program; if present this must appear immediately beneath the first line of the definition and have the same level of indentation as the function body.

Functions 8

Here is a possible definition for the **printAverage** function, complete with documentation, and some sample calls to the function.

```
def printAverage(a, b, c) :
    """
    prints average of three integers
    arguments: a:int, b:int, c:int
    """
    total = a + b + c
    print("Average is ", total/3)

printAverage(3, 5, 7)
printAverage(myAge, yourAge, hisAge)
printAverage(7*10, 9*9, 8**4)
```

Functions 9

Observe that we have already seen calls to functions: `print` and `input` are functions, as are `round` and `abs` that have been encountered in the lab exercises.

The functions that we have defined so far, along with `print`, are examples of what were called ***procedures*** in many earlier languages such as Algol and Pascal – they simply perform a task which may change the state or perform output.

It is also possible to write functions that perform some activity and generate a result – `input`, `round` and `abs` are examples of such functions. These functions are said to ***return*** a value.

Functions 10

We might prefer to write an average-calculation function which, instead of printing the average of the numbers, returns it to the caller. This will allow us to display averages with different layout or messages in different parts of the program.

The function returns a result through a statement comprising the keyword `return` followed by an expression whose value is to be returned.

The caller could store the returned result in a variable, supply it as an argument to another function or use it as part of an expression.

The following slide shows an average-calculation function that returns a result together with some sample calls.

Functions 11

```
def average(a, b, c) :
    """
    returns average of three integers
    arguments: a:int, b:int, c:int
    """
    total = a + b + c
    return total/3

print("The average is ", average(3, 5, 7))
averageAge = average(myAge, yourAge, hisAge)
s = average(7*10, 9*9, 8*4) + average(3, 5, myAge)
```

Functions 12

If a Python function does not explicitly return a result it will actually return the special value `None`. (Note that this value has its own unique type; it is not a number and is not equal to 0.)

Functions can be written that explicitly return a result in some cases but not in others (e.g. by placing a `return` statement in one of the two branches of an `if…else` statement but not in the other); care should be taken to avoid this when writing code, unless this is precisely what is required.

Functions 13

If a `return` statement is encountered while running a function, the value will be returned immediately and any subsequent statements in the function will not be reached.

The function on the following slide shows how this can be useful. It finds the smallest factor (other than 1) of a positive integer supplied as an argument. It does this by checking in turn whether the argument is divisible by each smaller number (starting from 2) and returns the first factor it finds.

If all numbers less than or equal to half of the argument have been tested and no factor has been found the argument must be prime so the smallest factor is the argument itself and this is returned.

Functions 14

```
def factor(n) :
  """
  finds smallest factor (>1) of a positive integer
  argument: n:int
  """
  if n<=0:
    print("Invalid argument")
    return None

  i = 2
  while i <= n/2 :
    if n % i == 0 : return i
    i = i + 1

  # no factor found - number is prime
  return n
```