COMP 10280
Programming I (Conversion)

John Dunnion

School of Computer Science
University College Dublin

COMP 10280 Programming I (Conversion)/Lecture 10

Outline

Conditional statement inside iteration statement (1)

- The loop body can be a single statement or a number of statements
- This statement or one of these statements can be a conditional statement or another iteration statement

Counting different grades (1)

Write a program that counts different grades

Initialise all counters
Prompt the user for a grade
Read the grade
while *grade != "" (empty string)* ***do***
    if *grade == 'A'* ***then***
        Increment grade A counter
    else if *grade == 'B'* ***then***
        Increment grade B counter
    . . .
    else
        Print "Unknown grade entered."
    Prompt the user for another grade
    Read the grade
Print out the results
Program finishes

Counting different grades (2)

```
# Initialise all the counters
a_grades, b_grades, c_grades, d_grades, e_grades = 0, 0, 0, 0, 0

# Prompt the user for a grade
grade = input('Enter a grade (empty string to exit):  ')

while grade != "":
    print('Grade is:  ', grade)
    print()
    if grade == 'A':
        a_grades += 1
    elif grade == 'B':
        b_grades += 1
    elif grade == 'C':
        c_grades += 1
    elif grade == 'D':
        d_grades += 1
    elif grade == 'E':
        e_grades += 1
    else:
        print('Unknown grade entered.')
# Prompt the user for another grade
    grade = input('Enter a grade (empty string to exit):  ')
```

Counting different grades (3)

```
# Now print out the results
print('Number of A grades:', a_grades)
print('Number of B grades:', b_grades)
print('Number of C grades:', c_grades)
print('Number of D grades:', d_grades)
print('Number of E grades:', e_grades)

print('Finished!')
```

Nested loops

- We can also have an iteration statement appearing in the loop body of another iteration statement
- This construct is called a nested loop
- Loops can be nested indefinitely

Addition table (1)

Write a program that generates an addition table

Prompt the user for the size of the table
Read table_size
i = 0
while *i* $\leq$ *table_size* ***do***
    j = 0
    while *j* $\leq$ *table_size* ***do***
        Print i + j
        Increment j
    print a newline
    Increment i
Program finishes

Addition table (2)

```
# Prompt the user for the size of the table
table_size = int(input('Enter the size of the table (an int):  '))

i = 0
while i <= table_size:
    j = 0
    while j <= table_size:
        print(i + j, " ", end = "")
        j += 1
# Print a newline
    print()
    i += 1
```

Addition table (3)

```
>>>
Enter the size of the table (an int):  10
0 1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10 11
2 3 4 5 6 7 8 9 10 11 12
3 4 5 6 7 8 9 10 11 12 13
4 5 6 7 8 9 10 11 12 13 14
5 6 7 8 9 10 11 12 13 14 15
6 7 8 9 10 11 12 13 14 15 16
7 8 9 10 11 12 13 14 15 16 17
8 9 10 11 12 13 14 15 16 17 18
9 10 11 12 13 14 15 16 17 18 19
10 11 12 13 14 15 16 17 18 19 20
```

More nested loops

- Nested loops can have different limits
- The limit of one loop can be dependent on the counter of another

Multiplication table (1)

Write a program that generates a triangular multiplication table

Prompt the user for the size of the table
Read table_size
i = 0
while *i* $\leq$ *table_size* ***do***
    j = 0
    while *j* $\leq$ *i* ***do***
        *Print i * j*
        Increment j
    print a newline
    Increment i
Program finishes

Multiplication table (2)

```
# Prompt the user for the size of the table
table_size = int(input('Enter the size of the table (an int):  '))

i = 0
while i <= table_size:
    j = 0
    while j <= i:
# Inner loop:  Only go as far an the outer loop counter
        print(i * j, " ", end = "")
        j += 1
# Print a newline
    print()
    i += 1
```

Multiplication table (3)

```
>>>
Enter the size of the table (an int):  20
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
0 10 20 30 40 50 60 70 80 90 100
0 11 22 33 44 55 66 77 88 99 110 121
0 12 24 36 48 60 72 84 96 108 120 132 144
0 13 26 39 52 65 78 91 104 117 130 143 156 169
0 14 28 42 56 70 84 98 112 126 140 154 168 182 196
0 15 30 45 60 75 90 105 120 135 150 165 180 195 210 225
0 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240 256
0 17 34 51 68 85 102 119 136 153 170 187 204 221 238 255 272 289
0 18 36 54 72 90 108 126 144 162 180 198 216 234 252 270 288 306 324
0 19 38 57 76 95 114 133 152 171 190 209 228 247 266 285 304 323 342 3
0 20 40 60 80 100 120 140 160 180 200 220 240 260 280 300 320 340 360
```

For loops

- A `for` loop is traditionally used when you have a piece of code which you want to repeat a certain number of times
- Virtually every programming language has a `for` loop
- However, the for loop exists in many different flavours, ie both the syntax and the semantics differ from one programming language to another

Different kinds of `for` loop (1)

- Count-controlled `for` loop (Three-expression `for` loop) This is by far the most common type. This statement is the one used by C. The header of this kind of for loop consists of a three-parameter loop control expression. In general, it has the following form:

```
for (init; cond; inc)
    statement
```

where `init` is the initialisation part, `cond` determines a termination expression and `inc` is the incrementing expression, where the loop variable is incremented or decremented. An example of this kind of loop is the `for` loop in the programming language C:

```
for (i=0; i < n; i++)
```

This kind of for loop is not implemented in Python!

Different kinds of `for` loop (2)

- Numeric Ranges
  This type of `for` loop is a simplification of the first kind. It's a counting or enumerating loop, which starts with a start value and counts up to an end value, for example:

```
for i = 1 to 100
```

  Python doesn't have this type of loop either!
- Vectorized for loops
  - They behave as if all iterations are executed in parallel
  - The expressions on the right-hand side of all the assignment statements are evaluated before the assignments are executed

Different kinds of `for` loop (3)

- Iterator-based `for` loop
  - This is the `for` loop in Python
  - This kind of loop iterates over an enumeration of a set of items
  - It is usually characterised by the use of an implicit or explicit iterator
  - In each iteration step, a loop variable is set to a value in a sequence or other data collection
  - This kind of `for` loop is also available in most Unix and Linux shells

The `for` loop in Python

- The general form of the `for` loop in Python is as follows:

  for *variable* **in** *sequence*:
  statement(s)
- Recall that when describing the form of a statement, italics are used to describe the type of Python code that can occur at that point in the statement
- The variable following the keyword `for` is bound to the first value in the sequence and the statement block is executed
- The variable is then bound to the second value in the sequence and the statement block is executed again
- This process continues until the sequence is exhausted or a `break` statement in the statement block is executed

The `range` function (1)

- The sequence of values bound to the variable in a `for` loop is most often generated using the `range` function
- This returns a sequence containing an arithmetic progression
- The `range` function takes three integer arguments: `start`, `stop` and `step`
- It produces the progression `start`, `start + step`, `start + 2 * step`, ...
- If `step` is positive, the last element of the sequence is the largest integer `start + i * step` that is less than `stop`
- If `step` is negative, the last element of the sequence is the smallest integer `start + i * step` that is greater than `stop`

The `range` function (2)

- For example, `range(5, 40, 10)` produces the sequence `[5, 15, 25, 35]`
- `range(40, 5, -10)` produces the sequence `[40, 30, 20, 10]`
- If the first argument is omitted, the default is 0
- If the last argument (`step`) is omitted, the default is 1
- So `range(0, 3, 1)`, `range(0, 3)` and `range(3)` all produce the sequence `[0, 1, 2]`

The `range` function (3)

- In Python 2.x, the `range` function generates the entire sequence when it is invoked
- For example, `range(10000000)` produces a sequence of 10 000 000 integers, whether these are used or not
- The `xrange` function generates the values only as they are needed in the `for` loop
- In Python 3, the `range` function behaves in the way that the `xrange` function behaves in Python 2.x

Addition table using a `for` loop (1)

```
for i in range(0, 20):
    for j in range(0, 20):
        print(i + j, " ", end = "")
# Print a newline
    print()
```

Addition table using a `for` loop (2)

```
>>>
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38
```

Addition table using a `for` loop (3)

```
# Prompt the user for the size of the table
table_size = int(input('Enter the size of the table (an int):  '))

for i in range(0, table_size + 1):
    for j in range(0, table_size + 1):
        print(i + j, " ", end = "")
# Print a newline
    print()
```

Triangular multiplication table using a `for` loop

```
# Prompt the user for the size of the table
table_size = int(input('Enter the size of the table (an int):  '))

for i in range(0, table_size + 1):
    for j in range(0, i + 1):
        print(i * j, " ", end = "")
# Print a newline
    print()
```

Demonstrating the bevaviour of the `for` loop (1)

Consider the following program

```
# Demonstrating the behaviour of the for loop

for i in range(0, 20):
    print('Counter is:', i)

print('Finished!')
```

Demonstrating the bevaviour of the `for` loop (2)

This produces the following output

```
>>>
Counter is:  0
Counter is:  1
Counter is:  2
Counter is:  3
Counter is:  4
Counter is:  5
Counter is:  6
Counter is:  7
Counter is:  8
Counter is:  9
Counter is:  10
Counter is:  11
Counter is:  12
Counter is:  13
Counter is:  14
Counter is:  15
Counter is:  16
Counter is:  17
Counter is:  18
Counter is:  19
Finished!
```

Demonstrating the bevaviour of the `for` loop (3)

- Now consider the following program
- We want to skip the numbers from 6 to 10

```
# Demonstrating the behaviour of the for loop
# Trying to increment the counter
# This won't work...

for i in range(0, 20):
    print('Counter is:', i)
    if i == 5:
        print('Skipping next five...')
        i += 5

print('Finished!')
```

Demonstrating the bevaviour of the `for` loop (4)

This produces the following output

```
>>>
Counter is:  0
Counter is:  1
Counter is:  2
Counter is:  3
Counter is:  4
Counter is:  5
Skipping next five...
Counter is:  6
Counter is:  7
Counter is:  8
Counter is:  9
Counter is:  10
Counter is:  11
Counter is:  12
Counter is:  13
Counter is:  14
Counter is:  15
Counter is:  16
Counter is:  17
Counter is:  18
Counter is:  19
Finished!
```

Demonstrating the bevaviour of the `for` loop (5)

- In Python, the `for` loop loops over an iterable
- This is determined at the start of the loop
- Thus trying to modify the value of the iterator, for example by trying to execute `i += 1`, has no effect on the operation of the loop

Demonstrating the bevaviour of the `for` loop (6)

```
# Demonstrating the behaviour of the for loop
# Trying to change the value of the increment variable
# This won't work...
# We can change the value of the increment variable, ...
# ...  but it changes back!

for i in range(0, 5):
    print('Counter is:', i)
    i += 3
    print('Counter after modification:', i)

print('Finished!')
```

Demonstrating the bevaviour of the `for` loop (7)

This produces the following output

```
>>>
Counter is:  0
Counter after modification:  3
Counter is:  1
Counter after modification:  4
Counter is:  2
Counter after modification:  5
Counter is:  3
Counter after modification:  6
Counter is:  4
Counter after modification:  7
Finished!
```

Specifying a sequence as a literal

- It is possible to specify a sequence as a literal
- Elements of the sequence are given inside square brackets ([and]) and separated by commas
- [10, 20, 30]
- ['COMP 10280', 'COMP 20240', 'COMP 20270', 'COMP 30640', 'COMP 30680', 'COMP 47340']

Iterating through a sequence of strings (1)

```
# Iterating through a list of strings
# Strings are given in a literal

strings = ['aardvark', 'buffalo', 'cat', 'dog', 'elephant', 'fox',
           'giraffe', 'hyena', 'iguana', 'jackal', 'kangaroo',
           'llama', 'mouse']

for word in strings:
    print('The length of', word, 'is:', len(word))

print('Finished!')
```

Iterating through a sequence of strings (2)

This produces the following output

```
>>>
The length of aardvark is:  8
The length of buffalo is:  7
The length of cat is:  3
The length of dog is:  3
The length of elephant is:  8
The length of fox is:  3
The length of giraffe is:  7
The length of hyena is:  5
The length of iguana is:  6
The length of jackal is:  6
The length of kangaroo is:  8
The length of llama is:  5
The length of mouse is:  5
Finished!
```

Iterating through a string

- Even though a string does not have an `iter` method defined, it is still possible to iterate through it using the `for` command
- This will iterate through the string one character at a time

Program to iterate through a string (1)

```
# Iterating through a string
# User is prompted for the string

# Prompt the user for a string
string = input('Enter a string (Press "Enter" to finish):  ')

# Keep going as long as an empty string is not entered
while string != "":

# Iterate through the string
    for ch in string:
        print(ch)
    print('***')

# Prompt the user for another string
    string = input('Enter a string (Press "Enter" to finish):  ')

print('Finished!')
```

Program to iterate through a string (2)

This produces the following output

```
>>>
Enter a string (Press "Enter" to finish):  dog
d
o
g
***
Enter a string (Press "Enter" to finish):  Hello, world!
H
e
l
l
o
,

w
o
r
l
d
!
***
Enter a string (Press "Enter" to finish):
Finished!
```