Loops (Iteration, Repetition)

Loops: Repeating parts of a program

So far, all our programs have carried out one major task such as converting a single quantity of metres to centimetres.

Frequently, we want to repeat such a calculation. Say we have thirty values for metres which we want to convert to centimetres. Using the program described earlier, we would have to run it 30 times to achieve the desired result.

Programming languages provide **loops to allow us repeat part of the program** as many times as we wish.

For example, in the conversion program we can write the program to repeat the process of reading a value to be converted and displaying the result, 30 times or any number of times.

This is called **looping** (or iteration).

The `while` loop

- There are a number of looping techniques, but basically all program looping can be performed using one looping construct called a **while** loop.
- Loops are **another form of conditional statement**.
- In the case of a loop, **we use the condition to decide whether to repeat a statement or not**.
- We repeat the statement based on the evaluation of the condition in the same way as for the if statement.
- The **action part of a loop** is referred to as the **loop body**.
- This may be a simple or block of statements (group of statements).
- The **loop body is executed only if the condition evaluates to true**, the condition is then re-evaluated to test if it is still true. If it is true, we repeat execution of the loop body and test the condition again. **This process continues until the condition evaluates to false.**

- In certain situations, the condition will never evaluate to false and the loop will **continue to execute endlessly.**

- Such a loop (usually the result of a programming error) is called an **endless** or **infinite** loop.

- An endless loop may be terminated by interrupting the program or switching off the computer, both of which terminate the program as.

- To interrupt a program, a combination of keys is pressed, such as pressing the control key and the C key simultaneously (denoted by Ctrl/C). The operating system will then the interrupt and terminates the program.

Write a program to implement a guessing game. The program “knows” a secret number and prompts the user to guess the number. It allows the user to keep guessing until they find the number.

```
# guess.py: Guess the secret number

secret = 4                    # Secret number the user has to guess
guess = -1

while guess != secret:
      guess = int(input("Guess the number between 1 and 10:  "))
      if guess != secret:
            print("\nWrong guess: ", guess)
      else:
            print("Well done !")
```

Running this program:

```
Guess the number between 1 and 10:  3
Wrong guess:  3
Guess the number between 1 and 10:  5
Wrong guess:  5
Guess the number between 1 and 10:  4
Well done !
```

- Modify the guessing game to allow **the user only 3 chances** to guess

```
# guess.py: Guess the secret number

secret = 4
guess = -1
num_chances = 1

while (guess != secret) and ( num_chances <= 3 ):
      guess = int(input("Guess the number between 1 and 10:  "))
      if guess != secret:
            print("\nWrong guess: ", guess)
            num_chances = num_chances + 1
      else:
            print("Well done !")

if num_chances > 3:
      print("Sorry you have used all of your guesses")
```

Running this program:

```
Guess the number between 1 and 10:  0
Wrong guess:  0
Guess the number between 1 and 10:  1
Wrong guess:  1
Guess the number between 1 and 10:  2
Wrong guess:  2
Sorry you have used all of your guesses
```

The statement

```
while (guess != secret) and ( num_chances <= 3 ):
```

The `while` above tests both conditions and only repeats if **both** conditions are true

```
# pay3.py: Calculate and display hourly pay for 5 workers

count = 1
hours_worked = float(input("\nEnter number of hours worked: "))

while count <= 5:
   if hours_worked > 100:
      print("\nHour worked too large:", hours_worked)
   else:
      rate_per_hour = float(input("\nEnter rate per hour: "))
      if rate_per_hour > 50:
         print("\nRate per hour too high ", rate_per_hour)
      else:
         pay = rate_per_hour * hours_worked
         print("\nPay = ", pay, "for ", hours_worked, "hours")
   hours_worked = float(input("\nEnter number of hours worked: "))
   count = count + 1
```

```
# calc4.py: Calculator program to add 2 numbers, 3 times
# Author: Joe Carthy
# Date: 01/10/2022

count = 1

while count <= 3:
  number1 = float(input("\nEnter first number: "))
  number2 = float(input("\nEnter second number: "))
  sum = number1 + number2
  print("\nThe sum of", number1, "and", number2, "is", 
sum, "\n\n")
  count = count + 1

print (“Finished summing\n”)
```

`calc4.py outputs`:

```
Enter first number:  4
Enter second number: 6
The sum of 4 and 6 is 10

Enter first number:  20
Enter second number: 30
The sum of 20 and 30 is 50

Enter first number:  65
Enter second number: 50
The sum of 50 and 65 is 115

Finished summing
```

Write a program to sum the integers 1 to 99 (i.e. calculate the sum of 1+2+3+...+99) and display the result.

```
# sum.py: calculate 1+2+3+.....+99
# Author: Joe Carthy
# Date: 01/10/2022

sum = 0                       # contains the sum we wish to compute
i = 1                         # the loop counter

while i <= 99:
      sum = sum + i
      i = i + 1

print(“Summation is: “, sum)
```

Executing this program produces as output:

```
Summation is: 4950
```

```
# calc6.py: Calculator program to add 2 numbers until 0 entered
# Author: Joe Carthy
# Date: 01/10/2022

number1 = -1    # Any non-zero value will do

while number1 != 0:
   number1 = float(input("\nEnter first number: "))
   if number1 != 0:
      number2 = float(input("\nEnter second number: "))
      sum = number1 + number2
      print("\n\nThe sum of", number1,"and",number2,"is",sum, "\n\n")

print("\n\nCalculator program terminated \n")
```

Running this program:

```
Enter first number: 3

Enter second number: 3

The sum of 3.0 and 3.0 is 6.0


Enter first number: 0

Calculator program terminated
```

Write a program to display 4 lines with 1 star (‘*’) character on line 1; 2 stars on line 2, 3 stars on line 3 and 4 stars on line 4. The output should appear as follows:

```
*

**

***

****
```

```
# tri.py: displays triangle composed of *'s
num_lines = 1

while num_lines <= 4:
   num_stars = 1
   while num_stars <= num_lines:    # inner loop
      print("*”, end = "")
      num_stars = num_stars + 1     #end inner loop

   print("\n")           # start new line
   num_lines = num_lines + 1     # end outer loop
```

Write a program to allow you enter as many numbers as you wish. The program should sum the numbers and calculate the average. You may use 0 as the value used to indicate that you are finished entering numbers. Such a value is called a **sentinel**.

```
# sum5.py: Sum numbers until 0 entered and display sum and average

sum = 0 # contains the sum we wish to compute
n = 0   # number of numbers user entered

num = float(input("Enter a number or 0 to quit: "))
while ( num != 0 ):
        sum = sum + num
        num = float(input("Enter a number or 0 to quit: "))
        n = n + 1      # count numbers entered
if n != 0:
        average = sum / n
        print("\n\nSum is: ", sum, "Average is:", average)
```

Outline

Iteration statement (1)

- Thus far, all of our programs have carried out one action or one major task
  - Calculating the area of various geometrical shapes
  - Calculating the total price including taxes
  - Checking if a year is a leap year
  - Checking if a password is correct
  - …

Iteration statement (2)

- We often want to carry out an action or a calculation a number of times
- For example, say we want to check 20 different years to see if they are leap years
- We would have to run our program 20 times!
- We would prefer to be able to run a program once and allow it to repeat the operation(s) the required number of times

Iteration statement (3)

- In our programs thus far, we have considered two types of statement:
  1. Sequence (or Sequential statement)
  2. Conditional statement (or Selection statement)
- With sequential statements, the first statement is executed, then the second, and so on in sequence
- With conditional statements, one of a number of statements, or none, is executed depending on the value of a controlling expression (condition)

Iteration statement (4)

- The third type of statement is the iterative statement (or repetition statement or loop statement)
- An iterative statement repeatedly executes a statement or a block of statements (called the loop body) while a condition is true or until a condition is met
- The condition is evaluated. If it evaluates to `True`, the statement in the loop body is executed
- The condition is evaluated again. If it evaluates to `True`, the statement in the loop body is executed again
- This process continues until the condition evaluates to `False`
- When the condition evaluates to `False`, the statement following the iterative statement is executed

Iteration statement (5)

- In certain situations, the condition will never evaluate to `False`
- In this case, the iteration statement will continue to execute endlessly
- Such a loop is called an infinite loop or endless loop
- This is often the result of a design or programming error!
- A program with an infinite loop may be terminated by interrupting the program or by turning off the computer
- Often a control sequence (for example CONTROL-C) can be used to interrupt a program
- An IDE will often provide a command to interrupt a program
- The interpreter or the operating system detects the interrupt and terminates the program

The `while` statement

- In Python, the `while` statement has the following form:
- **while** *Boolean expression*:
  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
- *Boolean expression* indicates that any expression that evaluates to `True` or `False` can follow the reserved word `while`
- *statement(s)* indicates that any sequence of Python statements can appear at that point

Summing numbers (1)

- Consider the following program:

```
number1 = 1
number2 = 2
number3 = 3

total = number1 + number2 + number3
print('Total␣is:', total)
```

- This produces the following output:

```
>>>
Total is: 6
```

Summing numbers (2)

- Consider the following variation:

```
number1 = 1
number2 = 2
number3 = 3

# Running total
total = 0

total = total + number1
total = total + number2
total = total + number3

print('Total␣is:', total)
```

- This produces the following output:

```
>>>
Total is: 6
```

Summing numbers (3)

- Now consider the following program that uses a `while` statement:

```
# Running total
total = 0

# Counter for loop
count = 1

while count <= 3:
    total = total + count
    count = count + 1

print('Total␣is:', total)
```

- This produces the following output:

```
>>>
Total is: 6
```

Summing numbers (4)

- The loop body is executed only if the condition (`count <= 3`) evaluates to `True`
  - Since we have initialised `count` to 1 the condition evaluates to `True` and the statements in the loop body are executed
  - The first value assigned to a variable is called the initial value
- In the loop body, a revised running total is calculated by adding the value of `count` to `total`
  - The variable `total` is assigned the value `total + count`, ie `total` is assigned the value 1 (0 + 1)
  - The variable `count` is then increased by 1, to 2
  - We then test the condition again

Summing numbers (5)

- The variable `count` is now 2 and the condition `(count <= 3)` remains `True` so we execute the loop body again
  - The variable `total` is assigned the value `total + count`, ie `total` is assigned the value 3 (1 + 2)
  - The variable `count` is then increased by 1, to 3
- The condition `(count <= 3)` remains `True` so we execute the loop body again
  - The variable `total` is assigned the value `total + count`, ie `total` is assigned the value 6 (3 + 3)
  - The variable `count` is then increased by 1, to 4
- The condition `(count <= 3)` is now `False` so we do not execute the loop body again
- Instead, execution continues with the first statement after the iterative statement

Summing numbers (6)

- Consider the following variation that lets us see what is going on:

```
# Running total
total = 0
print('Initial␣value␣of␣total␣is:', total)

# Counter for loop
count = 1

while count <= 3:
    total = total + count
    print('Current␣total␣is', total)
    count = count + 1

print('Total␣is:', total)
```

Summing numbers (7)

- This produces the following output:

```
>>>
 Initial value of total is: 0
Current total is 1
Current total is 3
Current total is 6
Total is: 6
>>>
```

Summing numbers (8)

- Each time we execute the loop body (“go around the loop”), we add `count` to `total` and add 1 to `count`
- The variable `count` is used in this example to control how many times we execute the loop
- After executing the loop three times, `count` will have the value 4
- Each time the interpreter executes the statements in the loop, the condition is tested. The statements in the body of the loop are executed only if the condition evaluates to `True`
- When `count` has the value 4, we “leave the loop” (“the loop terminates”)
- Execution continues with the next statement after the `while` statement (if any)

Summing numbers (9)

- Note that the condition is evaluated before the loop body is executed
- If the condition initially evaluates to `False`, the loop body will not be executed at all
- It is essential that the variables `total` and `count` are initialised to appropriate values for the loop to operate correctly
- As a general programming principle, all variables should be given correct values, usually at the beginning of a program or beginning of a block
- If you do not initialise a variable, you cannot be sure what value it may contain (sometimes it may contain 0 for numbers) and your program may not run correctly.
- This is a common source of errors for beginners
- Python does not allow you to use uninitialised variables!

Infinite loops (1)

- What would happen if the statement

  ```
  count = count + 1
  ```

  were omitted from the loop body ?
- The loop would never terminate, as `count` would always be less than 3
- We would have an infinite loop
- This is a very common error to make when using loops
- Such an error may be a logical error, a design error or a programming error and produces a runtime error
- These differ from syntax errors because the program will execute, but does not produce the expected results

Infinite loops (2)

- Consider the following variation:

```
# Running total
total = 0
print('Initial␣value␣of␣total␣is:', total)

# Counter for loop
count = 1

while count <= 3:
    total = total + count
    print('Current␣total␣is', total)

print('Total␣is:', total)
```

Infinite loops (3)

- This produces the following output:

```
>>>
Initial value of total is: 0
Current total is 1
Current total is 2
Current total is 3
Current total is 4
Current total is 5
Current total is 6
Current total is 7
Current total is 8
Current total is 9
...
```

Augmented assignment

- In Python, augmented assignment is the combination, in a single statement, of a binary operation and an assignment statement
- For example, executing the statement

  ```
  x += 1
  ```

  achieves a similar effect to that of executing

  ```
  x = x + 1
  ```

- Some differences:
  - `x` is evaluated only once
  - When possible, the operation is performed “in-place”, ie rather than creating a new object and assigning that to the target, the old object is modified instead

Using augmented assignment

- Re-writing the previous (working) program:

```
# Running total
total = 0
print('Initial␣value␣of␣total␣is:', total)

# Counter for loop
count = 1

while count <= 3:
    total += count
    print('Current␣total␣is', total)
    count += 1

print('Total␣is:', total)
```

Another while loop example

- Write a program that checks whether numbers up to a limit are divisible by 2 or 3:

```
# Defining how far we go
limit = 20
# Counter for loop
count = 1

while count <= limit:
    print('Number is:', count)
    if count % 2 == 0:
        print('Number is divisible by 2')
    if count % 3 == 0:
        print('Number is divisible by 3')
    print()
    count += 1

print('Finished!')
```

Definite and indefinite iteration

- Often it is not known in advance how many times to repeat a loop
- It is not possible to use `while` loops as presented in the examples, where it was specified exactly how many times to repeat the loop body
- We will now write a program that continues operating for as long as the user requires
- This program converts kilometres into miles
- The user may wish to enter one number or 1000 numbers
- This type of loop is sometimes referred to as indefinite iteration, as it is not known in advance how many times it will be repeated

Converting from km to miles

Write a program that converts from kilometres to miles

```
# Conversion from km to miles
km_to_miles_conv = 0.621371192

# Defining how far we go
limit = int(input('Enter the number of numbers
                           you wish to convert:  '))

# Counter for loop
count = 1

while count <= limit:
# Prompt the user for a number of km, convert the number
                           and print out both
    km = float(input('Enter the number of kilometres:  '))
    miles = km * km_to_miles_conv
    print(km, 'kilometres is', miles, 'miles')
    print()
    count += 1

print('Finished!')
```

Converting from km to miles

Write a program that converts from kilometres to miles

```
# Conversion from km to miles
km_to_miles_conv = 0.621371192

# Prompt the user for a number of km
km = float(input('Enter a number of kilometres you wish
                           to convert (negative number to exit):  '))

while km >= 0:
# Convert the number and print out both km and miles
    miles = km * km_to_miles_conv
    print(km, 'kilometres is', miles, 'miles')
    print()
# Prompt the user for another number of km
    km = float(input('Enter a number of kilometres you wish
                           to convert (negative number to exit):  '))

print('Finished!')
```

More indefinite iteration

- Another technique is to execute the loop body as long as the user doesn't enter a particular value
- For example, a program might continue unti a negative value is entered
- The next example illustrates a very common technique for controlling the number of times a loop is repeated in interactive programs
- Such programs use information from the user to control how they operate

`for` Loop

The **for loop** is used when we know the number of times we wish to repeat the loop body. We frequently know how often we wish to repat a statement(s). For example, we often use the **for** loop to process a list of items.

The for loop is often used in combination with the range() function.

The range() function

This function returns a sequence of numbers in a given range:

range(6) returns: 0,1,2,3,4,5 # integers from 0 up to but **not including 6**

range (1, 5) returns 1, 2, 3, 4 # from 1 up to but not including 5

`for` Loop

There are 3 forms of range :

`range (stop)` generate list from **0** to **stop**, ***not** including stop*

```
range (3) gives 0, 1, 2
```

`range(start, stop)` generate list from **start** to **stop**, **not** including stop

```
range (4,8) gives 4, 5, 6, 7
```

`range(start, stop, step)` generate list from **start** to **stop**, **not** including stop, by increments of size **step**

```
range (0, 12, 2)    # yields 0, 2, 4, 6, 8, 10
```

`for` Loop

The **for loop** is used when we know the number of times we wish to repeat the loop body. We frequently know how often we wish to repat a statement(s). For example, we often use the **for** loop to process a list of items.

We can re-write the program to sum the integers 1 to 99 using a for loop as follows

```
# sum3.py: Sum 1 + 2 + 3 + ... +99

sum = 0        # contains the sum we wish to compute
for i in range(1, 100):
   sum = sum + i

print("\nSummation is:", sum, "\n")
```

Write a program to read 10 integers, sum them and calculate the average. The program should display the sum and the average.

```
# sum3.py: Sum 1 + 2 + 3 + ... 10+

sum = 0                       # contains the sum we wish to compute
for i in range(1, 11):        # sum 1 to 10:
       sum = sum + i


average = sum / 10


print("\n\nSum is: ", sum, "Average is:", average)
```

```
Output:


Sum is:  55 Average is: 5.5
```

Modify the program tri.py to use **for** loops, to display 4 lines with 1 star (‘*’) character on line 1; 2 stars on line 2, 3 stars on line 3 and 4 stars on line 4.

```
#!/usr/bin/python3
# tri3.py: displays triangle composed of *'s

num_lines = 1

for num_lines in range (1, 5):                        # outer loop
  for num_stars in range (1, num_lines+1):            # inner loop
    print("*", end = "")
print("\n")                       # start new line
```

The inner loop displays `num_lines` stars on each line.

String Processing

In programming, we often wish to “process” the elements of a string in various ways. We will show how to access and process strings in the following examples. We usually need to know how long a string is when we are going to process it. The **`len()`** function gives us the length of a string e.g.

```
l =  len(“abcd”)
print (“l = “, l)


l = 4



print(len(“123456”))


outputs


6
```

Write a program to output the characters in a short string on separate lines.

```
# str.py: Output each characters on a newline

string = "abc"
length = len(string)

for i in range (0, length):
    print( string[i] )
```

outputs

```
a
b
c
```

In this example length is 3 and range (0, length) gives 0,1, 2.

The last element of a string is always at position length – 1 and strings always start at element 0.

Write a program to output the characters in a short string, separating them with the “&” character.

```
# str2.py: Output each characters of string followed by &

string = "abcdef"
length = len(string)

for i in range (0, length):
    print(string[i] + "&", end="")
print()
```

outputs

```
a&b&c&d&e&f&
```

Write a program to read a string and display it in reverse i.e. Joe is displayed as oeJ

```
# str3.py: Read a string and display it in reverse

string = input("Enter a string: ")
length = len(string)

for i in range (length-1, -1, -1):
    print(string[i],  end="")
print()
```

outputs

```
Enter a string: ABCDEF
FEDCBA
```

Pay particular attention to the range function used here `range (length-1, -1, -1):`

Note the first element of a string in Python is **element 0**.

In this example we entered a 6 character string.

This means the elements are from **0 up to 5** (not 6).

Thus to display the string backwards we need to **range from element 5 to element 0**.

This is why we subtract 1 from length in the range function.

Also we need to include element 0 in the output, so we need to set the stop value in range to be -1

e.g. range (5, 0, -1) will give us elements 5,4,3,2, 1 but NOT element 0.

To include 0 we need range(5, -1, -1) which means start at 5, step down by 1 each time and stop at -1 but **do NOT** include -1.

Outline

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

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]`

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!
```

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!
```