Lab2_Notebook

September 23, 2020

1 Lab 2: Program Flow: Loops and Conditions

In Lab 1 you learned some basic python tasks and wrote simple functions. This week you will start to learn about controlling the flow of your programs, allowing you to create much more complicated programs without much more effort. The two fundamental ways of doing this in programming are conditions and loops.

1.1 Loops: The While loop

If you want to perform a set of commands (code) over and over again, you could copy and paste the commands multiple times. This is ok for a small number, like last week's assignment, but will quickly get tedious and error-prone if you need to do the same task many many times. What if you wanted to create hundreds or thousands of objects? Computers are supposed to do repetitive tasks for us, not the other way around! In programming, you can get the computer to repeat a command or set of commands using loops. These come in several forms, but the simplest in Python is the `while` loop. This keeps repeating commands while a test is `True`. This can be used to repeat a command forever, as long as a condition is being met. Consider the following flow chart :

In Python, the example for the flow chart above would look as follows :

[1]:
```
again = 'yes'
while again == 'yes':
    print('Hello')
    again = input('Do you want to loop again? : ')
```

```
Hello
Do you want to loop again? : no
```

It will keep repeating this code until the user enters anything other than 'yes'.

What if we want to only repeat some commands a fixed number of times? The easiest way to do this is to use a `counter`. Create a variable which is going to count the number of times a loop has repeated, then keep repeating while that counter is less than the number of times we want to repeat

[2]:
```
count = 0
while count < 10:
    print(count)
    count = count + 1
```

```python
print('finished with count', count)
```

```
0
1
2
3
4
5
6
7
8
9
finished with count 10
```

The count starts at zero, and each time this loop repeats, it adds 1 to count. `while count < 10` means keep repeating while count < 10. As soon as count is equal to 10, the loop will end. This program will print the numbers 0 to 9, then when the loop finishes (where the indentation reduces) it will print `finished with count = 10` since 1 is added to count before exiting the loop.

These are not the only ways to use `while` loops, and you can have any test you like to determine when the loop finishes. When the condition is any test which has a True or False answer, and the loop keeps repeating as long as it is true. The last example tested if the "count" variable was less than 10 and kept repeating while that was true. Here is another example :

[3]:
```python
total = 0
while total < 20:
    num = int(input('Enter a number :'))
    total = total + num
    print('The total is', total)
print("Finished")
```

```
Enter a number :3
The total is 3
Enter a number :4
The total is 7
Enter a number :5
The total is 12
Enter a number :6
The total is 18
Enter a number :7
The total is 25
Finished
```

The above program creates a variable called total and assigns the initial value as 0. It asks the user to enter a number and will add it to the total. It will keep repeating this as long as the total is still below 20. When the total equals 20 or more, it will stop running the loop.

1.2 Conditions

Very often in programming, we want to be able to make decisions; to test if something happened and run one set of commands if it did, and another if it didn't. More generally, we want to know if a test is true or false as in the while loops above. The main tests we can use are:

Operator	Description
==	equal to
!=	not equal to
>	greater than
<	less than
>=	greater or equal to
<=	less than or equal to
and	both conditions must be met
or	either conditions must be met

The `and` and `or` operators are logical operators.

1.3 Loops: the For loop

Often we want to control a loop so that we know exactly how many times it goes round and controls a loop counter. The general structure of a `for` loop in Python looks like:

for < variable > **in** < sequence >: > < statement >

A common `for` loop has the form:

`for i in range (a,b,c):`

where `i` is the loop counter. The first time round the loop, `i` starts at the value `a` and increments in steps of `c` continues looping as long as `i` remains less than `b`. For example:

[4]:
```
for i in range(0,10,1):
    print(i)
```

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

Note that `i=10` is never reached. If the increment value is 1, then it does not need to be specified. Similarly, if the initial value is not specified, it is assumed to be 0. Here are some more examples:

[5]:
```
for i in range(1,10,2):
    print(i)
```

```
1
3
5
7
9
```

[6]:
```
for i in range(0,10,2):
    print(i)
```

```
0
2
4
6
8
```

[7]:
```
for i in range(10,-3,-2):
    print(i)
```

```
10
8
6
4
2
0
-2
```

1.4 Working with If statements

An `if` statement is a programming construct depending upon a logical condition which produces a *branch* such that the code does one thing if the condition is true and another thing if the condition is false. These are especially useful when combined with `for` loops and procedures. The general structure of an `if` statement in Python looks like

if $<$ logical condition which returns true or false $>$: $>$ $<$ statement if logical condition is true $>$

else: $>$ $<$ statement if logical condition is false $>$

More complicated branching statements can be built up using `elif`, short for 'else-if'.

if $<$ logical condition which returns true or false $>$: $>$ $<$ statement if logical condition is true $>$

elif $<$ second logical condition $>$: $>$ $<$ statement if first logical condition is false and second logical condition is true $>$

else: $>$ $<$ statement if first logical condition is false and second logical condition is false $>$

Such `elif` statements can be nested indefinitely if so desired.

```
[8]: for i in range(0,10):
         if i%2==0:
                 print(i, 'even')
         else:
                 print(i, 'odd')
```

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

```
[9]:  for i in range(-4,4):
         if i<0:
             print(i,'negative')
         elif i>0:
             print(i, 'positive')
         else:
             print(i, 'zero')
```

```
-4 negative
-3 negative
-2 negative
-1 negative
0 zero
1 positive
2 positive
3 positive
```

Using `if` statements in functions allows us to treat different arguments differently, for example we might want a function which acts on integers and halves the integer if it is even but multiplies it by three and adds one if it is odd, we could define such a function as follows:

```
[10]: def p(n):
          if n%2==0:
              return n/2
          else:
              return 3*n+1
```

```
[11]: print(p(4))
```

```
2.0
```

[12]:
```python
print(p(3))
```

```
10
```

Here are some examples which combine `if`, `for` and `while` loops :

[13]:
```python
num = int(input('enter a number between 10 and 20 : '))
if num < 10:
    print('too low')
elif num >= 10 and num <= 20:
    print('correct')
else:
    print('too high')
```

```
enter a number between 10 and 20 : 22
too high
```

The above program asks the user to enter a number between 10 and 20. If the user enters a number below ten, the output will tell the user that the number is too low. If the number is between 10 and 20 then the output says `correct`, and if the number is anything else then the output reads `too high`.

We can extend this to keep asking the user to input a number until they give one between 10 and 20 by using a `while` loop.

[14]:
```python
correct = False
while correct == False:
    num = int(input('enter a number between 10 and 20 : '))
    if num < 10:
        print('too low')
    elif num >= 10 and num <= 20:
        print('correct')
        correct = True
    else:
        print('too high')
```

```
enter a number between 10 and 20 : 22
too high
enter a number between 10 and 20 : 13
correct
```

We have started the code by assigning ‘correct’ to be False. We have then introduced a `while` loop that says while correct is False, the user should input a number. The code is the same as the example above except that if the user gives a value between 10 and 20, correct is now True and the while loop will end.

Another example would be to extend the maths question task from Lab 1. This time, we can ask Python to generate 5 different questions, and give the user a total mark out of 5 for their answers.

2 Practising with loops

Controlling loops, managing loop “counters” and performing running totals (see the command `score = score + 1` in the above code) are skills that you are likely to use a lot in scientific programming, so its good to see some examples and get some practice.

Here are some examples and a couple of practice tasks:

2.1 Example 1: Evaluating a series sum

Imagine a simple series sum of N terms….

$$y = 1 + 2 + 3 + \ldots$$

which can be written

$$y = \sum_{i=1}^{N} i$$

Lets do this with a `while` loop, for N= 10:

[20]:
```
N=10
y=0
i=0
while i < N+1:  # Note that N+1 is needed here, otherwise the final i=10 loop␣
 ↪would not be executed
    y = y + i
    i = i + 1   # This is our loop counter
print("y is", y)
```

```
y is 55
```

…and here’s the same thing with a `for` loop…

[16]:
```
N=10
y=0
for i in range(1,N+1): # Note that N+1 is needed here, otherwise the final i=10␣
 ↪loop would not be executed
    y = y + i
print("y is", y)
```

```
y is 55
```

2.2 Example 2: A more complicated series

Imagine another series sum, evaulated for N terms….

$$y = \frac{\pi}{2} + \frac{\pi}{4} + \frac{\pi}{6} ...$$

which can be written

$$y = \sum_{i=1}^{N} \frac{\pi}{2i}$$

Lets do this with a `for` loop, for N= 10:

[17]:
```
import math
N=10
y=0
for i in range(1,N+1):
    y = y + math.pi/(2*i)
print("y is", y)
```

```
y is 4.6008125746321955
```

2.3 Practising with loops - tasks for you!!

Here are a couple of practice tasks (not assessed - just for practice):

Task 1: Evaluate the following series for 5 terms, for k = 0.5, using a while loop:

$$y = k^{1/1} + k^{1/2} + k^{1/3} ... = \sum_{i=1}^{5} k^{1/i}$$

[18]:
```
# Insert code here
```

ANSWER: Should be 3.7122 (5 s.f.) for k=0.5, N=5

Task 2: Evaluate the following series for N terms (N defined in code) using a for loop:

$$y = 2 \times 4 \times 6 \times ... = \prod_{i=1}^{N} 2i$$

[19]:
```
# Insert code here
```

ANSWER: Should be 3840 for N=5

Solutions will posted on the VLE site for Lab 2

3 Debugging a piece of code

The term “bug” for a mistake or glitch in a machine first appeared in the late 19th Century, but in computing it’s said to come from an actual bug (a moth) which caused a computer to malfunction by getting fried in a relay.

When you’re writing programs (code), you are writing instructions which the computer can then interpret (through a compiler or interpreter to machine code). Computers need a precise set of instructions to follow, and can’t deal with ambiguities, so computer languages like Python work on a strict set of rules. Every command has to obey the rules of the language (syntax), but the commands have to fit together as well, so that your program makes logical (semantic) sense. The Python compiler can find “syntax errors” quite easily, but usually logical errors in your program will only appear when you run it and find it doesn’t do what you expect it to.

When Python finds an error, it will usually look something like this:

```
File “lab2.py”, line 10
print (10 3)
^
SyntaxError: invalid syntax
```

Read this message carefully, as it is trying to tell you where the error might be. The message consists of three parts:

The file name and the line number. The file name you probably already knew, but this is useful when you have bigger programs which are in many files. The line number is the important bit of information for now, as it tells you where in your code the error is. Note: It is quite common for an error to actually be on a previous line, particularly if some brackets are missing. Check the line before just in case.

The line of code that Python thinks the error is in. The claret (^) shows Python’s best guess for where the error is in the line. This is often correct, but not always so just use it as a starting point. If the error’s not at the point indicated, it’s almost certainly somewhere before that point.

The type of error which occurred. SyntaxError is a “syntax error”, which means an error in the grammar. Missing brackets, commas, and colons in the wrong place will usually lead to this type of error. Another common one is NameError, which happens when Python comes across a variable name it doesn’t recognise. If you see this, try checking the spelling carefully.

This week, you have an additional assignment - you have a task to debug a piece of code!!

[ ]:

```

```