Phys105-Week04

November 20, 2020

1 Introduction to Computational Physics - Week 4

1.1 Table of contents week 4

1.2 Introduction to week 4

This week we will look at how Python allows us to control the flow of programs and introduce boolean variables.

1.3 Python flow control

We have seen how we can do calculations in Python, how we can use library functions and how we can create our own functions. Now we look at how Python allows us to decide how we would like our program to "flow" from one calculation or statement to the next. This allows us to deal with situations where we want to do something different depending on whether a variable is positive or negative, for example. Several ways of steering programs are provided, including the `for`, `while` and `if` statements. We look at these in the following.

1.3.1 For loop

The `for` loop allows us to repeat sections of a program a specified number of times. The full syntax of the loop is shown below:

[1]:
```
# <!-- Student -->
start = 0
stop = 5
step = 2
#
for i in range(start, stop, step):
    print("Index",i)
print("Final value of index",i)
```

```
Index 0
Index 2
Index 4
Final value of index 4
```

The start of the loop is indicated by the `for` statement, which is followed by the name of the index (here we use `i`) which will steer how many times the loop is executed. The values `i` should take are determined by the statement `in range(start, stop, step)` and `i`, `start`, `stop` and `step` must all be integers. The `for` line is terminated using a colon (:).

When the program runs, the value of `i` is first set to `start`, then all the indented statements following the `for` statement are carried out. The value of `i` is then incremented by `step`, and if the resulting value is less than `stop`, the loop is executed again. This continues until adding `step` to `i` would give a value greater than or equal to `stop`.

If `step` is one (which it often is), you can use the simpler version of the `for` loop which omits the `step` parameter:

[2]:
```
# <!-- Student -->
start = 0
stop = 3
for n in range(start, stop):
    print("Index",n)
print("Final value of index",n)
```

```
Index 0
Index 1
Index 2
Final value of index 2
```

Again, the indentation indicates the body of the loop, i.e. the section of the program that is repeated. You can see that the loop doesn't run with `n = stop` and that the value of `n` on leaving the loop is the last "allowed" value.

1.3.2 Week 4 exercise 1

Write a program using a `for` loop that produces two columns, the left column being the integers 0 to 10 and the right column the words “zero” to “ten”.

Hint Use a list containing the strings “zero”, “one”, “two” etc. and recall that you can access the elements of a list using the syntax `list[index]`!

1.3.3 Week 4 exercise 1 answer

[3]:
```
# <!-- Demo -->
numbers = ["zero", "one", "two", "three", "four", "five", "six", "seven",␣
 ↪"eight", "nine", "ten"]
for n in range(0, 11):
    print(n,numbers[n])
```

```
0 zero
1 one
2 two
3 three
4 four
5 five
6 six
7 seven
8 eight
9 nine
10 ten
```

1.3.4 Loops and slices

In weeks 2 and 3, we have looked at how “slicing” can be used to access array elements. One of the examples we looked at was:

[4]:
```
# <!-- Student -->
import numpy as np
#
countArr = np.linspace(0, 10, 11)
print("countArr =",countArr)
print("countArr[3:6] =",countArr[3:6])
```

```
countArr = [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
countArr[3:6] = [3. 4. 5.]
```

This example is slightly modified below to illustrate how the parameters used in slicing are related to those in a `for` loop.

[5]:
```
# <!-- Student -->
countArr = np.linspace(0, 4, 5)
print("countArr =",countArr)
print("countArr[1:4:2] =",countArr[1:4:2])
```

```
countArr = [0. 1. 2. 3. 4.]
countArr[1:4:2] = [1. 3.]
```

1.3.5 Week 4 exercise 2

Explain the role of each of the three indices between the square brackets in the statement `countArr[1:4:2]`).

1.3.6 Week 4 exercise 2 answer

The indices are `[start, stop, step]`.

1.3.7 Alternative forms of the for loop

Loops can also be used to cycle through the elements in a list. An example is shown below:

[6]:
```python
# <!-- Student -->
loopList = ["one", "two", "three"]
for var in loopList:
    print(var)
print("End of loop, var is",var)
```

```
one
two
three
End of loop, var is three
```

1.3.8 Week 4 exercise 3

Can you cycle through a tuple in the same was as through a list? Write a program similar to that for the list to test this!

1.3.9 Week 4 exercise 3 answer

[7]:
```python
# <!-- Demo -->
loopTuple = ("A", "B", "C")
print(" ")
for var in loopTuple:
    print(var)
print("End of loop, var is",var)
```

```

A
B
C
End of loop, var is C
```

1.3.10 While statement

The `while` statement offers another way of repeatedly using a section of code. An example follows:

```
[8]: # <!-- Student -->
     #
     test = 0.3
     limit = 1.1
     step = 0.25
     while test < limit:
         print("test =",test)
         test = test + step
     print("Final value of test is",test)
```

```
test = 0.3
test = 0.55
test = 0.8
test = 1.05
Final value of test is 1.3
```

As in the case of the for loop, the line containing `while` finishes with a colon. The body of the loop, indicated by the indentation, is executed until the condition `test < limit` is false. (It won't execute at all if the condition is false the first time it is checked.) Something must be changed in the body of the loop to ensure that at some point `test < limit` becomes false, or the loop will run for ever. In the example above, the value of `test` increases by `step` each time the while loop runs, because we set `test = test + step`.

The condition that is tested has one of two values, `True` or `False`. If the statement `test < limit` is `True` execution continues, if it is `False`, it stops. As such logical conditions are used so frequently, Python has a data type, `bool` (short for *boolean*), which can take only the values `True` or `False`. We will learn more about boolean variables later.

1.3.11 If statement

The `if`, `elif`, `else` statement has the syntax illlustrated below:

```
[9]: # <!-- Student -->
     test = 2.3
     if test < 1.0:
         print("This is section A")
     elif test > 2.0 and test <= 3.0:
         print("This is section B")
     elif test > 3.0 and test <= 4.0:
         print("This is section C")
     else:
         print("This is section D")
     print("This is the end of the if statement, the value of test is",test)
```

```
This is section B
This is the end of the if statement, the value of test is 2.3
```

Again, the lines starting with the `if`, `elif` (short for *else if*) and `else` statements must end with a colon. The code that is executed when the conditions tested in each of these lines are `True` is indented. Only the first of the sections of the `if`, `elif`, `else` block for which the condition is met is executed. *(Note, Python doesn't check the logic of your control statements, so it won't warn you if the tests in your `if` block don't make sense!)*

Statements can consist of just an `if`, an `if` and an `else`, or an `if` and one or more `elifs`, or, as above, of an `if`, one or more `elifs` and an `else`.

Python allows you to write the above `if`, `elif`, `else` statement in a more natural way (closer to standard mathematical notation) as follows:

[10]:
```
# <!-- Student -->
test = 2.3
if test < 1.0:
    print("This is section A")
elif 2.0 < test <= 3.0:
    print("This is section B")
else:
    print("This is section C")
print("This is the end of the if statement, the value of test is",test)
```

```
This is section B
This is the end of the if statement, the value of test is 2.3
```

1.3.12 Continue and break

These statements allow you to modify the behaviour of a loop. In a `for` or a `while` loop, `continue` causes control to jump back to the beginning of the loop, without executing the statements after the `continue`. Two examples are shown below.

[11]:
```
# <!-- Student -->
for letter in "Constantinople":
    if letter in "a, e, i, o, u":
        continue
    print(letter)
print("Final value of letter is",letter)
```

```
C
n
s
t
n
t
n
p
l
Final value of letter is e
```

Note that the string "a, e, i, o u" above could be replaced by "a e i o u" or "aeiou" and the routine would still work. It is checking whether `letter` is in the string enclosed in quotes and as `letter` is never "," or " " (there are no commas or spaces in "Constantinople"), their presence in the string makes no difference!

Here's another example:

```
[12]: # <!-- Student -->
for i in range(0, 6):
    print(i)
    if i > 2:
        continue
    print(10*i)
print("Final value of i is",i)
```

```
0
0
1
10
2
20
3
4
5
Final value of i is 5
```

In contrast, `break` causes control to jump to the end of the loop:

```
[13]: # <!-- Student -->
for letter in "Constantinople":
    if letter in "a, e, i, o, u":
        break
    print(letter)
print("Final value of letter is",letter)
```

```
C
Final value of letter is o
```

```
[14]: # <!-- Student -->
for i in range(0, 6):
    print(i)
    if i > 2:
        break
    print(10*i)
print("Final value of i is",i)
```

```
0
0
1
10
2
20
3
Final value of i is 3
```

We see that, if the `continue` statement is used, the code in the `else` statement is executed. If the `break` condition is applied, the code in the `else` statement is not run: there is a difference between the code that is executed if a `while` loop runs completely, or if it terminates due to a `break` statement.

1.3.13 Precision of floating point numbers and flow control

Computers use binary representations of numbers and for `floats` this results in limited precision. Similarly, addition, subtraction and other operations cause a loss of accuracy. If you want to read more, see this article.

A consequence of this is that you should never rely on a `float` having exactly a particular value. For example, the following code is not likely to give the result you want:

```
if test == 0.1397:
```

Instead, you should use something like:

```
if np.abs(test - 0.1397) < 1e-10:
```

You can test how precise the representation of `floats` is using the following code.

[15]:
```
#<!-- Student -->
#
eps = 1.0
while eps + 1.0 > 1.0:
    eps = eps/2
eps = 2*eps
print("The precision of your computer is", eps)
```

```
The precision of your computer is 2.220446049250313e-16
```

1.3.14 Week 4 exercise 4

Explain the functioning of the `while` loop that determines your computer's precision!

1.3.15 Week 4 exercise 4 answer

In the `while` loop, for as long the computer can tell the difference between `eps` and `eps + 1`, it halves `eps`. The value of `eps` at the end of the loop is therefore half of the value where `eps` and `eps + 1` are indistinguishable. Multiplying this by two gives the required value.

Python provides a function which returns the precision with which floats are represented in the `sys` (short for *system*) package:

[16]:
```
#<!-- Student -->
import sys
print("Precision of float is",sys.float_info.epsilon)
```

```
Precision of float is 2.220446049250313e-16
```

Numpy also has a function, described here, which provides information on the representation of floating point numbers:

[17]:
```python
#<!-- Student -->
print("Precision of float is",np.finfo(float))
```

```
Precision of float is Machine parameters for float64
---------------------------------------------------------------
precision =  15   resolution = 1.0000000000000001e-15
machep =    -52   eps =        2.2204460492503131e-16
negep =     -53   epsneg =     1.1102230246251565e-16
minexp =  -1022   tiny =       2.2250738585072014e-308
maxexp =   1024   max =        1.7976931348623157e+308
nexp =       11   min =        -max
---------------------------------------------------------------
```

1.3.16 Week 4 exercise 5

Write a program using a `while` loop to calculate and print out the factorial of the first ten integers. The factorial of a number is given by $n! = n(n-1)(n-2)...3 \times 2 \times 1$, for example, $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$.

1.3.17 Week 4 exercise 5 answer

[18]:
```python
# <!-- Demo -->
number = 10
n = 1
factorial = 1
while n <= number:
    factorial = factorial*n
    print("Factorial",n,"is",factorial)
    n = n + 1
```

```
Factorial 1 is 1
Factorial 2 is 2
Factorial 3 is 6
Factorial 4 is 24
Factorial 5 is 120
Factorial 6 is 720
Factorial 7 is 5040
Factorial 8 is 40320
Factorial 9 is 362880
Factorial 10 is 3628800
```

1.3.18 Week 4 exercise 6

Write a program using a `for` loop to calculate and print out the quantity $\sum_{n=1}^{N} n$ with N taking the values 1...15.

1.3.19 Week 4 exercise 6 answer

[19]:
```python
# <!-- Demo -->
number = 15
print(" ")
total = 0
for n in range(1, number + 1):
    total = total + n
    print("Sum for",n,"is",total)
```

```
Sum for 1 is 1
Sum for 2 is 3
Sum for 3 is 6
Sum for 4 is 10
Sum for 5 is 15
Sum for 6 is 21
Sum for 7 is 28
Sum for 8 is 36
Sum for 9 is 45
Sum for 10 is 55
Sum for 11 is 66
Sum for 12 is 78
Sum for 13 is 91
Sum for 14 is 105
Sum for 15 is 120
```

1.4 Week 4 marks

Exercise	Mark	Comments
1	2	
2	1	
3	1	
4	2	
5	2	
6	2	
Total	**10**	

[ ]:
```
```