Phys105-Week05-Student

November 26, 2020

1 Introduction to Computational Physics - Week 5

1.1 Table of contents week 5

1.2 Introduction to week 5

This week we will learn about boolean variables and then look in a bit more detail at how output can be attractively formatted using print statements. If you have spare time, use it to complete any outstanding work from the previous weeks' computer classes. Once you have finished these (make sure you get help if you need it!), you can try the *Fizz buzz* problem. This has often been used as part of the selection process for people applying for computing jobs!

1.3 Boolean variables in Python

As there are many situations where expressions are either true or false, Python has `bool` variables which can ony take these two values. For example:

[3]:
```
# <!-- Student -->
boolVar1 = True
print("boolVar1",boolVar1)
```

```
boolVar1 True
```

Numbers can be compared in a variety of ways, producing the values `True` or `False`:

[4]:
```
# <!-- Student -->
print("2 > 3",2 > 3) # greater than
```

```
print("2 == 2",2 == 2) # equal to
print("2 >= 2",2 >= 2) # greater than or equal to
print("3 <= 2",3 <= 2) # less than or equal to
print("2 != 2",2 != 2) # not equal to
```

```
2 > 3 False
2 == 2 True
2 >= 2 True
3 <= 2 False
2 != 2 False
```

As we have mentioned before, there are some potential pitfalls here. While it makes sense to test if two integers are equal, this is very risky (that means don't do it!) for `floats`, as even real numbers that you know should be mathematically identical may be different on your computer because of errors in their calculation or representation. For `floats`, "equality" can be tested in the following way:

```
epsilon = 1E-10
boolVar = np.abs(floatX - floatY) < epsilon
```

Choose a sensible value of `epsilon`, dependent on the errors you expect in the calculation of `floatX` and `floatY`.

Boolean operators (`and`, `or`, `not`) can also be used to act on `bool` objects:

[36]:
```
# <!-- Student -->
print("True and True =",True and True)
print("not True =",not True)
print("False or True =",False or True)
```

```
True and True = True
not True = False
False or True = True
```

A nice feature of Python is the ease with which strings can be manipulated. For example, we can look if a particular character or word is `in` (or `not in`) a string as below. (Notice that "Mary" and "mary" are not the same; in Python, size matters!)

[5]:
```
# <!-- Student -->
testString = "Mary had a little lamb"
wordFound = "Mary" in testString
print("'Mary' is in '",testString,"' is",wordFound)
wordFound = "mary" in testString
print("'mary' is in '",testString,"' is",wordFound)
wordNotFound = "Wolf" not in testString
print("'Wolf' is not in '",testString,"' is",wordNotFound)
```

```
'Mary' is in ' Mary had a little lamb ' is True
'mary' is in ' Mary had a little lamb ' is False
'Wolf' is not in ' Mary had a little lamb ' is True
```

We will learn below how to print the above without the superfluous spaces after and before the quotation marks in *' Mary had a little lamb'*!

1.3.1 Week 5 exercise 1

Modify the program below so that it completes the sentence printed out in each row with the word `True` if `total` is even and `False` if it is odd.

[6]:
```
# <!-- Student -->
total = 0
for n in range(1, 21):
    total = total + n
    print("Total for",n,"is",total,"which is even. This statement is")
```

```
Total for 1 is 1 which is even. This statement is
Total for 2 is 3 which is even. This statement is
Total for 3 is 6 which is even. This statement is
Total for 4 is 10 which is even. This statement is
Total for 5 is 15 which is even. This statement is
Total for 6 is 21 which is even. This statement is
Total for 7 is 28 which is even. This statement is
Total for 8 is 36 which is even. This statement is
Total for 9 is 45 which is even. This statement is
Total for 10 is 55 which is even. This statement is
Total for 11 is 66 which is even. This statement is
Total for 12 is 78 which is even. This statement is
Total for 13 is 91 which is even. This statement is
Total for 14 is 105 which is even. This statement is
Total for 15 is 120 which is even. This statement is
Total for 16 is 136 which is even. This statement is
Total for 17 is 153 which is even. This statement is
Total for 18 is 171 which is even. This statement is
Total for 19 is 190 which is even. This statement is
Total for 20 is 210 which is even. This statement is
```

1.4 Pretty printing

We have already seen how we can improve the clarity of printed output by using `f-strings`. Here, we will first look at these in a bit more detail, then we will look at how `format` statements can be used. While `format` statements have long been a feature of Python, the more useful `f-strings` were first introduced in Python 3.6. This means that many programs still use `format` statements, so it is useful to know something about them, even if it is usually better to use `f-strings`!

1.5 Using f-strings

An example illustrating printing using the `f-string` syntax is shown below, together with the equivalent "plain" print statement. Notice how the `\b` control character must be used to produce a backspace in the latter case to ensure that there aren't spurious spaces in front of the comma and the full stop!

[9]:
```
# <!-- Student -->
big = 999.999999
small = 666.666666E-19
number = 33
string = 'letters'
print(f"Big is {big}, small is {small}, number is {number} and string is␣
 ↪{string}!")
print("Big is",big,"\b, small is",small,"\b, number is",number,"and string␣
 ↪is",string,"\b!")
```

```
Big is 999.999999, small is 6.66666666e-17, number is 33 and string is letters!
Big is 999.999999, small is 6.66666666e-17, number is 33 and string is letters!
```

The `f-string` allows us to include a *format field* in the string we want to write, denoted by curly brackets (braces), which indicates that a variable (or expression - see later!) should be incorporated into the string at that point. Python chooses sensible defaults for the format used for these variables, but symbols inside the braces can be used to specify the format required, e.g. `f` for float, `e` for scientific notation, `d` for integer and `s` for string. How these are included is illustrated below:

[41]:
```
# <!-- Student -->
big = 999.999999
print(f"Big is {big:f}, small is {small:e}",f"number is {number:d} and string␣
 ↪is {string:s}!")
```

```
Big is 999.999999, small is 6.666667e-17 number is 33 and string is letters!
```

Using an `f-string` above has helped improve the print statement output in some respects (we don't need to use the `\b` backspace character to ensure the comma and fullstop are in the right place). Another useful feature of the `f-string` is that we can steer the number of figures we want after the decimal point (for floats or exponentials) as follows:

[42]:
```
# <!-- Student -->
print(f"Big is {big:.2f}, small is {small:.2e} and number is {number:d}!")
```

```
Big is 1000.00, small is 6.67e-17 and number is 33!
```

Note that numbers are rounded sensibly. (You can't specify the number of figures after the decimal point for integers of course!)

We can also control the length of the number that is printed out (including the spaces in front of the number) by adding an integer in front of the decimal point in the format specifiers:

[43]:
```
# <!-- Student -->
print(f"Big is {big:12.2f}, small is {small:12.3e} and number is {number:12d}.")
```

```
Big is      1000.00, small is    6.667e-17 and number is           33.
```

The equivalent for strings is shown below:

[44]:
```
# <!-- Student -->
print(f"String is {string:12s}.")
```

```
String is letters     .
```

By default, numbers are aligned to the right (right justified) and strings to the left (left justified), but this can be changed using the characters `<`, `>` and `^` for left, right and central alignment, respectively, as illustrated below:

[45]:
```
# <!-- Student -->
print(f"Big is {big:<12.2f}, small is {small:^12.3e}, number is {number:>12d}␣
 ↪and string is {string:>12s}.")
```

```
Big is 1000.00     , small is  6.667e-17  , number is           33 and string is
letters.
```

In `f-strings`, the `f` can also be an `F`!

[46]:
```
#<!-- Student -->
nameJ = 'John'
nameF = 'Fiona'
year = 2016
print(F"Hello {nameJ}, did you know f-strings were introduced in {year}?")
print(F"No, {nameF}, I didn't!")
```

```
Hello John, did you know f-strings were introduced in 2016?
No, Fiona, I didn't!
```

The braces can contain expressions that are calculated at run time (i.e. when the print statement is executed), for example:

[47]:
```
#<!-- Student -->
x = 2
y = 7
print(f"The product of {x} and {y} is {x*y}. The larger of {x} and {y} is␣
 ↪{max(x, y)}.")
```

```
The product of 2 and 7 is 14. The larger of 2 and 7 is 7.
```

The following example illustrates how some of the above techniques can be used to align columns of numbers:

[48]:
```
# <!-- Student -->
nMax = 10
nMin = 0
print("Number Square   Cube")
for n in range(nMin, nMax):
    print(f"{n:6d} {n**2:6d} {n**3:6d}")
```

```
Number Square   Cube
     0      0      0
     1      1      1
     2      4      8
     3      9     27
     4     16     64
```

```
5     25    125
6     36    216
7     49    343
8     64    512
9     81    729
```

Using "tab" characters (`\t`) can also help with formatting columns of numbers (though you might have to do some tweaking to get things to line up as you want!). A tab is eight characters long. Here is an example:

[49]:
```
# <!-- Student -->
print("Months in the year:")
print("1 \t 2 \t 3 \t 4 \t 5 \t 6 \t 7 \t 8 \t 9 \t 10 \t 11 \t 12")
print("Jan \t Feb \t Mar \t Apr \t May \t Jun \t Jul \t Aug \t Sept \t Oct \t␣
 ↪Nov \t Dec")
```

```
Months in the year:
1       2       3       4       5       6       7       8       9       10
11      12
Jan     Feb     Mar     Apr     May     Jun     Jul     Aug     Sept    Oct
Nov     Dec
```

Note, there are lots more tools for writing things prettily, see here, for example.

1.6 Using format statements

Everything we have done using `f-strings` can also be done using the `format` syntax, as shown below. Instead of including the variables that are to be printed at the relevant place in the string, they are added (in the required order) in a `format` statement after the string. The *format field* in the string can be used to indicate that an `int`, a `float` or a `str` should be included at that point. The format field is denoted by curly brackets, with the symbols inside the brackets indicating the format of the variable that should be substituted at that point (e.g. `f` for float, `e` for scientific notation, `d` for integer and `s` for string, as for `f-strings`).

[50]:
```
# <!-- Student -->
print("Big is {:f}, small is {:e}, number is {:d} and string is {:s}!"
      .format(big, small, number, string))
```

```
Big is 999.999999, small is 6.666667e-17, number is 33 and string is letters!
```

The `format` printing method allows us to steer the number of figures we want after the decimal point (for floats or exponentionals) as follows:

[51]:
```
# <!-- Student -->
print("Big is {:.2f}, small is {:.3e} and number is {:d}.".format(big, small,␣
 ↪number))
```

```
Big is 1000.00, small is 6.667e-17 and number is 33.
```

We can also control the length of the number that is printed out:

[52]:
```
# <!-- Student -->
print("Big is {:12.2f}, small is {:12.3e} and number is {:12d}.".format(big,␣
 ↪small, number))
```

```
Big is      1000.00, small is    6.667e-17 and number is           33.
```

Alignment can be changed, as for `f-strings`:

[53]:
```
# <!-- Student -->
print("Big is {:<12.2f}, small is {:^12.3e} and number is {:<12d}.".format(big,␣
 ↪small, number))
```

```
Big is 1000.00     , small is  6.667e-17   and number is 33          .
```

1.6.1 Week 5 exercise 2

Use first the `f-string` and then the `format` syntax to print out "The number of eggs in a dozen (12) is more than ten.", but using a variable `dozen = 12` for the number in the brackets.

I think you'll agree that the `f-string` is much clearer! Further good news is that it is also faster (i.e. uses less computing time) than the `format` statement. (In fact, it is also faster that the "c-style" printing option that Python offers which we haven't looked at because it's horrid!)

1.6.2 Week 5 exercise 3

Using the tools we have seen in this Notebook, write a row containing the text strings "One", "two", "three"…"seven". Below it, write a row listing the days of the week. Line up the two rows so "One" is above "Monday", "two" above "Tuesday" etc.

1.6.3 Week 5 exercise 4

Reproduce the table above with columns Number, Square and Cube for integers from 0 to 9 inclusive using `format` statements. Make sure that your table is aligned correctly!

1.7 The Fizz buzz problem

Write a program that prints the numbers from 1 to 20. But for multiples of three print *Fizz* instead of the number and for multiples of five print *Buzz*. For numbers which are multiples of both three and five, print *Fizz buzz*.

No hints or anything to start you off, just have a go! When you have a working program, try googling *Fizz buzz Python* to have a look at how other people have solved this problem. There are lots of possible solutions!

And by the way, congratulations are in order. It's only your fifth week and you are tackling a problem that crops up in interviews for professional programmers!