Introduction to Computational Physics (PHYS105)

Lecture 5: Input and Output

Carl Gwilliam

(C.Gwilliam@liverpool.ac.uk)

Lecture 4: Recap

- Last week we learnt how to control the flow of our program's execution

- Executing code only if certain conditions are true
  - Using `if…elif…else`
- Repeatedly executing pieces of code
  - Either `for` a range or data structure or `while` a condition is true

- These both utilise
  - Relational operators

    `==, !=, >, <, >=, <=`
  - Boolean variables

    `True, False`
  - Logical operators

    `and, or, not`

```
[109]: import numpy as np

numbers = [2, 736, 100, -6, 999]

for n in numbers:
    if n >= 0:
        print ("Square-root of", n, "=", np.sqrt(n))
    else:
        print ("Cannot take the square root of a negative number", n)
```

```
Square-root of 2 = 1.4142135623730951
Square-root of 736 = 27.129319932501073
Square-root of 100 = 10.0
Cannot take the square root of a negative number -6
Square-root of 999 = 31.606961258558215
```

X	Y	X and Y	X or Y
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Lecture 4: Formative problem solutions

- Exercise 1: Function to test if odd or even
  - Check if the remainder when divide by 2 is 0
  - Result is a boolean (`True` or `False`)

```
[103]: def isEven(number):
           return number%2 == 0

       num = 422
       print ("Number", num, "is even = ", isEven(num))

       num = 7123
       print ("Number", num, "is even = ", isEven(num))
```

```
Number 422 is even =  True
Number 7123 is even =  False
```

- Exercise 3: Can we loop over a tuple
  - Yes, only difference is it's immutable

```
loopTuple = ("A", "B", "C")

for var in loopTuple:
    print(var)

print("End of loop, var is",var)
```

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

- Exercise 2: for loop to print number & name
  - We access the element of an array using `[]`
  - Simply pass the loop index into the brackets
    - On each iteration this takes the values 0, 1, ... 10 i.e. the elements of the list we need to access

```
[104]: 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
```

- If you are struggling with loops think what will happen each iteration & what you would write for that specific iteration, then generalise

Lecture 4: Formative problem solutions (2)

- Exercise 4: explain the calculation of the precision using a while loop

```
[106]: 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
```

  - Starts with `eps = 1.0`, so `eps + 1.0 > 1.0`
  - Enters loop and haves the `eps` value
  - Keeps going while the computer can tell the difference between `eps+1` and `1`
  - Eventually `eps` is so small that `eps+1` is the same as `1` to the precision stored
  - The condition in the `while` then fails
  - The precision is the last value where can tell the difference → twice the final `eps` value as have halved since then
  - Note: can print `eps` value in loop to see this

- Exercise 5: Factorial of each of first 10 numbers
  - The loop var `n` gives the current number
  - At each iteration we times the result so far by the current number, `n`, and reset the result

```
number = 10
n = 1
factorial = 1

while n <= number:
    factorial = factorial*n
    print("Factorial",n,"is",factorial)
    n = n + 1
```

$$n! = n(n-1)(n-2)\ldots 3 \times 2 \times 1,$$

```
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
```

  - Some people just printed 10 factorial rather than the factorial of each element up to 10.

Formatting

- We saw how to output results to the screen using the `print()` function already in lecture 1
  - However, as we have seen, the result can often be messy

Output of example on slide 2:

```
Square-root of 2 = 1.4142135623730951
Square-root of 736 = 27.129319932501073
Square-root of 100 = 10.0
Cannot take the square root of a negative number -6
Square-root of 999 = 31.606961258558215
```

```
[118]: print ("n \t sqrt(n) \t log(n) \t exp(n)")
       for n in range(1, 6):
           print(n, np.sqrt(n), np.log(n), np.exp(n))
```

```
n        sqrt(n)          log(n)          exp(n)
1 1.0 0.0 2.718281828459045
2 1.4142135623730951 0.6931471805599453 7.38905609893065
3 1.7320508075688772 1.0986122886681098 20.085536923187668
4 2.0 1.3862943611198906 54.598150033144236
5 2.23606797749979 1.6094379124341003 148.4131591025766
```

- Often, we want to output the results of calculations to a certain precision
  - E.g. in labs a precisions that is sensible given the uncertainties on the value

- Further, when working with arrays of data we often want to tabulate the results

- How can we do this?
  - There are two related ways: the newer “f-strings” and the somewhat older `format` function
  - Will show both since they are both still in use, but you should prefer “f-strings” in general

F-strings

- Formatted string literals, or f-strings for short, are identified by an `f` preceding the string
- Within such a string we can write python expressions in braces `f'String with {expression}'`
  - Such expressions can be a literal value, the name of a python variable or a more complex expression
- Let's see an example compared to the standard print syntax
  - Here the variables `big`, `small`, `number` and `string` have been passed into the f-string in braces

```
[119]: 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,", small is",small,", number is",number,"and string is",string,"!")
```

```
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 !
```

  - Apart from avoiding the spurious space before the comma, it may seem like we have not gained much
- But the power of f-strings is that we can also include instructions of how to format the result of the expression within the braces, which are hence known as a "format field" …

Format specifier

- The format specification is placed after the expression within the braces, separated by a colon

```
f'String with {expression : format_specifier}'
```

- We can use this to explicitly tell python how to display a given type using a "type specifier"

Type	Specifier	Meaning
String	s	String (this is the default)
Integer	d	Decimal integer
	b, o, x	Binary, Octal, Hexadecimal integer
Float	f	Floating point
	e	Float point in scientific notation e.g. 1e10

Common type specifiers (there are many others)

- So, in our previous example, we could have

```
[120]: 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!
```

  - Again, this looks pretty similar since the default python string representation does a good job

Width and Precision

- The type specifier can also be preceded by a number (e.g. `10.3f`)
- The whole part of the number specifies the width of the field the value is printed in

```
[122]: print(f"Big is {big:12f}, small is {small:12e}, number is {number:12d} and string is {string:12s}.")
```

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

  - This can be useful for aligning values in tables
- For float types (f, e, ...) the fractional part of the number specifies the precision in d.p.
  - Note, you can specify just the decimal part to give the precision w/o specifying the width

```
[124]: print(f"Big is {big:12.2f}, small is {small:12.3e} and number is {number:12d}.")
       print(f"Big is {big:.2f}, small is {small:.3e} and number is {number:d}.")
```

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

Alignment

- Within the width of the field, you can choose how to align the value
  - Default if not specified is strings (numbers) are left (right) aligned

Alignment	Symbol
Left	<
Centre	^
Right	>

```
[127]: 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.
```

- This is again useful for tabulation
  - For example, putting everything together we can print a table of trig functions to 2d.p.

```
[151]: import numpy as np
angles = np.linspace(0, 360, 10)

print(f'{"angle":>10s}{"sin":>10s}{"cos":>10s}{"tan":>10s}')
print('-'*40)
for angle in angles:
    print(f'{angle:10.0f}{np.sin(angle):10.2f}{np.cos(angle):10.2f}{np.tan(angle):10.2f}')
```

```
     angle       sin       cos       tan
----------------------------------------
         0      0.00      1.00      0.00
        40      0.75     -0.67     -1.12
        80     -0.99     -0.11      9.00
       120      0.58      0.81      0.71
       160      0.22     -0.98     -0.22
       200     -0.87      0.49     -1.79
       240      0.95      0.33      2.90
       280     -0.39     -0.92      0.42
       320     -0.43      0.90     -0.47
       360      0.96     -0.28     -3.38
```

Note: you can multiply a string by a number to repeat it

Format function

- The usage of the somewhat older `format()` function syntax is similar to the f-string
- Again, we specify the formatting in braces at the correct point in the string
  - With exactly the same syntax as above
- The difference is that we provide the variables via the `format` function called on the string
  - Rather than placing them directly in the braces within the string

```
[153]: print("Big is {:f}, small is {:e}, number is {:d} and string is {:s}!".format(big, small, number, string)
print("Big is {:.2f}, small is {:.3e} and number is {:d}.".format(big, small, number))
print("Big is {:12.2f}, small is {:12.3e} and number is {:12d}.".format(big, small, number))
print("Big is {:<12.2f}, small is {:^12.3e} and number is {:<12d}.".format(big, small, number))
```

```
Big is 999.999999, small is 6.666667e-17, number is 33 and string is letters!
Big is 1000.00, small is 6.667e-17 and number is 33.
Big is      1000.00, small is    6.667e-17 and number is           33.
Big is 1000.00     , small is  6.667e-17   and number is 33          .
```

  - The variables have to be passed to `format` in the same order you want them to appear
- Having the variables and the formatting specified in different places is cumbersome
  - Hence the preference for the newer f-strings

Keyboard input

- Now we have looked at how we format output we will look at to how to get user input

- This is done using the `input()` function
  - The argument is a prompt string printed to the user
  - The return is what the user has input

```
user_string = input("prompt")
```

- When python gets to each call of input it presents a box for the user to type their input

```
[*]: name = input("What is your name?")
     print(f"\nHello {name}, nice to meet you\n")

     num = input("What is your favourite number?")
     print(f"\nThe number {num} is read as a {type(num)}")
What is your name? |
```

```
[*]: name = input("What is your name?")
     print(f"\nHello {name}, nice to meet you\n")

     num = input("What is your favourite number?")
     print(f"\nThe number {num} is read as a {type(num)}")
What is your name? Carl

Hello Carl, nice to meet you

What is your favourite number? |
```

- After the input you must press return
- The result is always returned as a string
  - Must convert if needed (see notebook)

```
[158]: name = input("What is your name?")
       print(f"\nHello {name}, nice to meet you\n")

       num = input("What is your favourite number?")
       print(f"\nThe number {num} is read as a {type(num)}")
What is your name? Carl

Hello Carl, nice to meet you

What is your favourite number? 10

The number 10 is read as a <class 'str'>
```

Reading and writing files

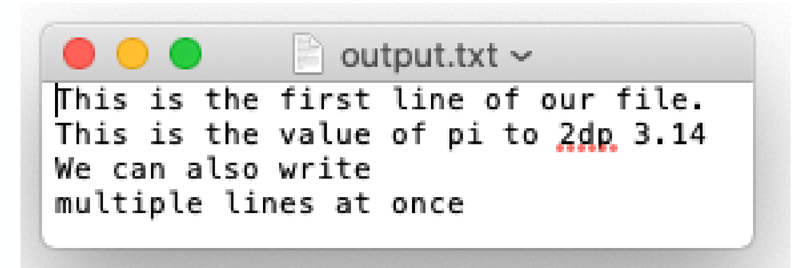
- Now we have seen how to input and output to the screen, what about reading/writing a file?

- In order to do this, we must first open a file using the `open()` function
  - This takes the name of the file and an optional mode
    - read = "r" (default), write = "w", both = "rw"
  - It returns a "handle" to the file that we can use to interact with it

```
file_handle = open('name', 'mode')
```

- We can write a string to the file using the `write()` function
  - Can use f-strings or format with the format specifiers above to format the output

```
[168]: fout = open('output.txt', 'w')
       fout.write("This is the first line of our file.\n")
       fout.write(f"This is the value of pi to 2dp {np.pi:.2f}\n")
       fout.write("We can also write\nmultiple lines at once\n")
       fout.close()
```



- Note: after finishing we must remember to `close()` the file

Reading and writing files (2)

- We read the file back in using the `read()` function
  - We must remember to open the file for reading first
  - And close again when we are done

```
[172]: fin = open('output.txt', 'r')
       text = fin.read()
       print(text)
       fin.close()
```

```
This is the first line of our file.
This is the value of pi to 2dp 3.14
We can also write
multiple lines at once
```

- This reads the whole file at one into a single string
  - What if we want to read it line-by-line?
  - For example, to allow processing each line

- We can easily do this with a for loop
  - Since the file handle provides a kind of data sequence we can iterate over directly

```
[176]: fin = open('output.txt', 'r')

       for line in fin:
           print(line)

       fin.close()
```

```
This is the first line of our file.

This is the value of pi to 2dp 3.14

We can also write

multiple lines at once
```

- Each line of the file is separated by a blank line
  - This is because `print()` writes a new line character at the end of each string, but this is also already present in the line read from the file → twice
  - You will see how to solve this in the notebook

Summary

- This week we have looked at input and output
  - How to receive input from the user
  - How to read/write a file
  - How to format our output
- You now know all the basics of python that you will need for this course
  - Of course, there are more advanced features (some of which you will cover in PHYS205)
  - And you need further practice
  - But you have learnt the basic building blocks
- For the rest of the course, we will focus on *using* python to perform useful calculations
  - Fitting experimental data
  - Numerical calculations
  - Monte Carlo methods

NumPy matplotlib

SciPy

- In doing so we will practice what we have learnt (+ see a few new features along the way)