Introduction to Computational Physics (PHYS105)

Lecture 5: Input and Output

Carl Gwilliam
(C.Gwilliam@liverpool.ac.uk)
UNIVERSITY OF LIVERPOOL

- Start recording + attendance code
- Moring everyone!
- This week we are going to talk about how to deal with input and output, both to/from the user and also files, and how to format that output nicely
    - As always, please feel free to interrupt with questions at any point

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

2

- OK, let's recap last week, where we learnt …
- Utilise …
  - …. equals (stress == not =), not equals ….
  - … the result of which is a boolean variable, T/F
  - … and which can be combined using logical operators: and (both input true), or (one input true), not (negates value)
- **As a reminder, here is the example we saw last week where want to take sqrt of list of numbers**
  - Mention difference to continue
  - Go over

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

3

- As usual, we will now look at the solutions to the formative problems and feedback on some common issues
- Ex1: The easiest way to test if a number is even is to check if the remainder when dividing by 2 is 0
  - **From Week 1 we know that we find the remainder using the % operator**
  - **We can then test if this is 0 using the == (not = 0) operator**
  - **At this point some people started to write if/else conditions, which is fine but not really needed** … the result of this will be a boolean variable (T/F)
  - Can just return this, without any if conditions -> print result -> true or false
- Ex2: Some people struggled with this.
  - **First let's write a for loop over numbers 0->10 using the range function, remembering the upper bound is exclusive so we need 11**
  - You know that you can access the given element of a list by passing the index in []
  - **Think what happens when loop: each iteration the variable n takes the values 0,1,2… 10 (as can see from the print out)**
  - We can simply use this as the index of the array and pass in square brackets: numbers[n] -> on first iteration this will be …
  - **i.e. do 1 iteration, then generalise using the loop variable (n in this case)**
- Ex3: Create a tuple using round rather than square brackets
  - Loop over it with a for in same way as a list

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.

4

- **Ex4: … and was to get you thinking about how the computer works at the same time as looking at loops**
  - **The variables are only stored to a certain precision, and it is good to be aware of this as can affect calculations in some cases**
  - **A good debugging strategy is always to print out variables in the loop (eps in this case) to see what is happening**
- Ex5: **We know the factorial of a number is the product of the numbers up to and including that number**
  - **We can calculate this simply with a loop if think about it in the correct way**
    - First write the while loop to loop while n is <= 10, remembering to update the value by 1 each time to prevent an infinite loop
    - **Each iteration the loop variable takes the numbers 1, 2 …**
    - **We can keep track of the product of the numbers seen by the loop so far using a variable**
    - Start with setting =1 **outside** the loop then multiplying by n each iteration and update the result i.e. assign back to same value
    - **The product so far IS the factorial up to that point -> print**
  - **Many problems can be solved like this so will use this kind of method several times**
- **Summative marks for the lecture 3 assignment released in canvas: 25 not submitted but average of those that did was 80%**
- **Any questions on the formative problems?**

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 54.59815003314236
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

5

- Let's start this week's topic …
- Precision
  - **For example, the output of the sqrt in slide 2 has a lot of precision we probably don't care about**
  - In labs, for example, often want to quote a value to a precision that is sensible given the uncertainties on the value
- Tabulate the results
  - **e.g. loop over numbers 1 to 6 and print the number itself, along with the sqrt, log and exp**
  - **If we do that with a bare print statement, we don't get proper columns as nums bleed into each other as you can see**

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" …

6

- f preceding the string **before (i.e outside) the quotes (point)**
- Expressions in CURLY braces
  - literal (e.g. 2.0) , variable
  - more complex expressions (as we will see)
  - **The f tells python to look for braces in the string and evaluate the contained code, replacing them in the string with the result**
- Suppose we have the variables … refer to code … and want to print them into a sentence
  - **If we wanted to do this up to now, we would have called print passing in the different variables as separate args (separated by commas)**
  - **With an f-string, however, we call the print function in the same way, but it just takes one "f-string" i.e quotes preceded by letter f**
  - **We then put each variable into this within braces at the correct position**
- Spurious space before the comma (point), **which comes from the default space added by print between each argument**

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

7

- **So we have the string, preceded by an f and within this we can have curly braces that contain the value or expression we want to print**
  - **This can be followed by a colon and then the specification of how we want to print it, still inside the curly braces**
- We can … Lets look at some of the most common type specifiers …
  - To put the output as a string we use s for string
  - For an integer we use d for decimal
    - Can also output in other formats e.g. binary (b), …
  - For a float we use f or if we want it in scientific notation i.e. 10^ **we use e since computers tend to display 10^ as e**
- Example: where we have now told it to print big as a float (:f), small as a float in scientific notation (:e), number as a decimal (:d) and string obv. as a string (:s)
  - Show binary as example
- Again … **But the format specification allows us to do a lot more …**
- **Any questions on f-strings and format specifications so far?**

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.
```

8

- Width in number of characters
  - **Padding the rest of the width with spaces (or another character if specified)**
  - **For example, here we have told python to print each in a field of width 12, so we get spaces if the output of the variable is shorter than that.**
  - **Not so useful in this case but can be used for aligning values in tables: by making the width larger that the widest print out we expect in any row**
- Precision
  - For example can tell it to print big to 2dp (12.2f) , and small to 3 dp (12.3e) -> see the difference to the previous result
  - Just give .2 or .3 with no whole number
  - Note that it rounds it nicely rather than simply truncating

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

```
[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.
```

Alignment	Symbol
Left	<
Centre	^
Right	>

- 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

9

- Left is a left arrow, right a right arrow and centred an up arrow (shift 6)
  - E.g. here we printed big left aligned, small centred and number right aligned
  - **You can see this changes where the value is wrt the space**
  - The default, if don't specify, is strings are left aligned and numbers right aligned
- Go over example: Create a numpy array of 10 angles angles from 0 to 360
  - print a header, with the title of each column given as a literal string (could be a var). For the format we obviously want a string (s), and we chose right aligned (>) in a field of width 10
    - **Note: need to use different quotes (single and double) for the outer and inner string so python can tell difference, but can be either way**
  - print a line: here we see that multiplying a string by a number repeats it that many times (which is kinda intuitive)
  - Then loop over our angles using the for loop saw last time, and print an f-string
    - We give each thing we want to print in braces, with the braces representing each column: for the angle a simple

variable, while for the others it's the actual calculation, showing the braces can contain more complex expressions**.**

- **(Best not to let this get too long as becomes hard to read)**
- Each is printed in the same field width of 10 **to line up with the headers**, with no dp for the angle (10) and 2 dp for the trig results (10.2)
- We use the default right alignment (since we have not specified)
- Get a nicely aligned table with a reasonable precision → very useful for presenting data/results

- **Any questions on width, precision and alignment?**

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

10

- Here are the corresponding examples to the above f-string ones.
  - They look every similar, except there is no f at the start.
  - **But you see that the braces now contain nothing before the colon**
  - **Instead, they just have the format specifier**
  - **The variables are then filled in from those in the format function, which have to …**
- **…cumbersome, in particular it is easy to forget which corresponds to which**
- You'll practice both in the problems class
- **Any questions on formatting in general?**

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'>
```

- Go over **example**
  - Ask the user for their name, passing the question as an argument to the input function
  - Store the result as a variable and print it out
  - We then ask them their favourite number and do the same, this time using an f-string
- **When python gets to each input call, it presents a box and waits for user input**
  - **Enter value and press return**
- **So we see that it prompts for the first question (point)**
  - **Only when have entered that and pressed return does it print the result (point) and the ask the second (point)**
- **We also see that the type of num in the second case is a string (point)**
  - This is because the return from input is always a string; if you want it as some other type (e.g. for calculations) you must explicitly convert it
  - As you'll see in this week's notebook

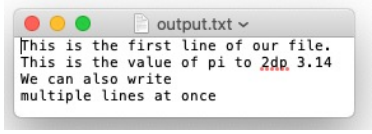
Reading and writing files

- 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

12

- **This is very useful to be able to read in experimental data or write out results to store them permanently or to be able to input them somewhere else for example**
  - **For NumPy arrays of data we can use the savetxt and loadtxt we saw, but we might want to store other information.**
- Takes two args: the first is the name of the file and the second is an optional mode, which defaults to read (point to code)
- We can write …
- Let's look at an example
  - Open a new file called output.txt for writing (**passing "w" as the mode**) and assign it to the variable fout
  - Can call write function as many times as we want using fout.write (same as calling np.cos for example)
  - **Can use normal strings or if want to format the output f-strings or format(), that we just learnt about, as input (e.g. pi to 2dp)**
  - **Here I have printed one line per write statement so we need to have \n at the end for a new line (it's not done automatically)**
  - **Can write multiple lines at once using multiple \n within a single write statement**
  - Close function to close the file
- If we open the file with some editor we can see the expected output …

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
- 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
- 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

```
[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
```

```
[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
```

13

- Can read back file using read (point to example)
  - Open for reading passing the file name and ”r” as the mode since only want to read (**could omit as r is default**)
  - Assign this to a variable, which I’ve called fin
  - Call the read function and assign the result to a variable -> print out
  - Close again
- Data sequence can iterate over
  - Open as before and assign to a variable
  - loop over file using for loop with same syntax as saw last week …
  - Each iteration the variable is that line of text and we can do what we want with it e.g. print
- Can see in the output that …
- Show tojenizing with split() e.g. ChatGPT
- **Any Questions on input and output?**

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)

14

- After first bullet: … and, of course, you will look at this in more detail in the notebook which has been released in canvas
- Congratulations … even if it may not feel like it sometimes
- **These will make heavy use of the numpy and matplotlib modules we have become familiar with**
  - **And introduce a new one, scientific python or scipy**
- **Any final questions?**