Week 8: Modules

Intended learning outcomes

You have now covered the basics of programming as described in the second lecture (variables, loops, conditional statements, etc) and will spend the rest of the course consolidating these structures and seeing further how programming can be useful in mathematics. By the end of this class, you will be able to:

- import, and use functions from, Python modules;
- generate random numbers in Python;
- use familiar mathematical functions and constants in Python.

1 Python modules

In Week 6 you were introduced to the idea of a function in Python and saw that there were a few built-in functions that you had already met. However, there are many more functions that you can load and use in Python.

In LaTeX you load *packages* to do various things; in Python the equivalent things are called *modules*. Both LaTeX and Python have large numbers of packages/modules, written by other people, that you can use. This week we will look at the `random` and `math` modules. In future weeks we will look at the `numpy`, `scipy` and `matplotlib` modules.

Whereas in LaTeX you load a package with `\usepackage{packagename}`, in Python you load a module with `import module_name`.

2 The `random` module

As you encountered in the mini-project, the `random` module allows you to generate random numbers to simulate random events such as throwing a dice.[1] In Google Colab (or the console in Spyder), type the following:

```
>>> import random
>>> print(random.random())
```

This should give you a number of type `float` between 0 and 1. Repeat the last line. You should get a different number! Repeat five or six times to convince yourself that you are getting something seemingly random.

When you have imported a module, the general way to call a function from it is `module.function()`, where `module` is the name of the module and `function` is the name of the function. Here, all functions will be preceded by `random`.

Before you use the random number generator in a program you should issue `random.seed()`, as this randomizes the randomizer based on the time on the computer's clock. You don't need to worry about this. Just do it at the beginning of your program to make your numbers more random!

The two most common commands for generating random numbers are `random.random()` and `random.randrange()`. The first gives a float with values between 0 and 1 according to the *uniform* distribution (i.e. all numbers are equally likely). To get a random float between 0 and n you just scale accordingly.

```
>>> print("Random number between 0 and 10:", 10*random.random())
```

The function `random.randrange()` will give you a random *integer* in the given range, where the syntax for the range is the same as for the `range()` command. So `random.randrange(1, 11)` gives a random integer between 1 and 10 inclusive; and `random.randrange(10, 21, 2)` will give a random *even* integer between 10 and 20 inclusive — remember that the 2 is the step size. Here we can simulate rolling a dice.

[1] Actually the numbers are only *pseudo*random, as they are generated *deterministically*, but as far as we are concerned in this course, they are random!

```
>>> dice_score = random.randrange(1, 7)
>>> print(dice_score)
```

In MAS113 you encounter the Law of Large Numbers, which roughly says that if you repeat an experiment a large number of times, then the relative frequency (or proportion) of the result x in the experiment should be approximately the probability that x occurs in a single experiment. We can test this in the case of a single roll of a dice by simulating many trials of this with Python.

We will simulate rolling a dice many times and record how many times each number from 1 to 6 comes up by storing it in a list called `frequency`, so that `frequency[i]` will be the number of times that we have rolled the number `i`. Note that this means that `frequency[0]` won't actually be used.

```
1  import random
2  random.seed()
3
4  NO_OF_TRIALS = 100
5
6  frequency = [0, 0, 0, 0, 0, 0, 0]
7
8  for i in range(NO_OF_TRIALS):
9      dice_score = random.randrange(1, 7)
10     frequency[dice_score] = frequency[dice_score] + 1
11
12
13 print("\nscore\trelative frequency")
14 print("-" * 26)
15 for i in range(1, 7):
16     print(i, frequency[i] / NO_OF_TRIALS, sep="\t")
```

Let's analyse the code line-by-line.

Line 1 imports the `random` module that we will use to simulate the dice roll.
Line 2 randomizes the randomizer.
Line 4 sets the number of times that we will roll the dice. (This is a constant, so we name it in capitals.)
Line 6 initializes each element in the list of frequencies to zero. (We could instead have written the list as `[0] * 7`.)
Line 8 starts the loop that performs the trial the required number of times.
Line 9 sets `dice_score` to be a random integer between 1 and 6.
Line 10 increments the count corresponding to the score we have just rolled.
Line 13 prints the header of the table, with the columns separated by a tab.
Line 14 prints a horizontal line of 26 minus signs at the top of the table.
Line 15 sets up the loop to run through the six possible scores we had.
Line 16 prints out each line in the table, with the entries separated by a tab.

Run the program. The Law of Large Numbers states that if we do a large number of trials, the relative frequency of each number should be roughly the probability of rolling that number. What is the probability of rolling each number? Do the relative frequencies look right? Increase the number of trials to 1,000 and run the program. Try increasing powers of 10. What do you think?

Adding print statements. Do you understand what the program is doing in Lines 8–10? If not, you can add some print statements to show you what is happening. This is a useful technique when you are debugging or trying to see what a program is actually doing. Put the following in Lines 11 and 12, making sure that they are indented to be part of the `for` loop, and change the number of trials to something small like 20:

```
11     print("dice score =", dice_score, end=";  ")
12     print("frequency list =", frequency)
```

Run the program. If you do not understand what you are seeing, then ask!

When you understand what is going on, you can remove these lines again, or just "comment them out" by putting a `#` at the beginning of both lines, so that the computer ignores them. You should also increase `NO_OF_TRIALS` again.

Being graphical. We can represent the relative frequencies in a much more graphic way. We can convert the relative frequencies into percentages by multiplying by 100. Then we can print that number of asterisks (remembering to convert the number to an integer first). Change Line 16 to the following:

```
16      print(i, "*"*int(100*frequency[i] / NO_OF_TRIALS), sep="\t")
```

Exercise 8.1. Change the program so that it simulates rolling *two* dice and taking their sum. (This is *not* the same as picking a random number between 1 and 12.) You should only have to change lines 6, 9 and 15.

Can you work out the probability of rolling a score of 2 with two dice? Of scoring 3? 4? Do these agree with what you see in a large number of trials?

3 The `math` module

The `math` module includes functions for familiar mathematical functions such as trigonometric functions, the logarithm function and the exponential function. In the console you can calculate simple things. Remember first to import the math module and to prefix the commands with `math`. Try the following in the console:

```
>>> import math
>>> print(math.pi)
>>> print(math.e)
>>> print(math.log(10))
>>> print(math.sin(1))
>>> print(math.radians(180))
>>> print(math.degrees(math.pi))
```

As is sensible for mathematicians, the `math.sin()` function takes angles in *radians* and `math.log()` means the *natural* logarithm, or logarithm to base e.

Other functions, e.g. inverse trigonometric functions and hyperbolic functions, are also included in the `math` module: a full list of commands can be found at `https://docs.python.org/3.8/library/math.html`.

It is important to remember that arithmetic with floats is *not* exact: the decimal expression are just numerical approximations to the real numbers. What is $\sin(\pi)$? What does Python calculate it to be?

```
>>> print(math.sin(math.pi))
```

What does the answer even mean? Python can represent floats using *scientific* or *exponential* notation, so for example `1.455e6` means 1.455×10^6, i.e. $1,455,000$, and similarly `1.455e-4` means 1.455×10^{-4}, i.e. 0.0001455.

```
>>> print(1.455e6, "\t", 1.455e-4)
```

So `math.sin(math.pi)` calculates a very, very small number. This will generally be fine for everyday computations, for example, plotting graphs. If you need more precise arithmetic then Python has modules available.

Now let's tabulate some values of $\sin(x)$ for $x \in [0, 2\pi]$. This is the kind of thing you would do if you wanted to plot the graph. We will split the interval $[0, 2\pi]$ into N equally sized subintervals and calculate the function on the end points of the intervals, in other words we will calculate $\sin(2\pi i/N)$ for $i = 0, 1, \ldots, N$. Here is a program to do this:

```
1  import math
2
3  NO_OF_INTERVALS = 16
4  HORIZONTAL_LINE = "-" * 48
```

```
5
6   print(HORIZONTAL_LINE)
7   print("x\t\t\tsin(x)")
8   print(HORIZONTAL_LINE)
9   for i in range(NO_OF_INTERVALS + 1):
10      x = 2 * math.pi * i / NO_OF_INTERVALS
11      print(x, math.sin(x), sep="\t")
12  print(HORIZONTAL_LINE)
```

Whilst the program does tabulate the values, the output is not ideal. The values are of different lengths so some things are in the wrong columns, there is too much precision in the values, and it would be better not to have scientific notation for the really small values (i.e. it would be better to see `0.00000` rather than `1.2e-16`). To improve this we can use the `str.format()` method, which allows us to format numbers to a fixed number of decimal places.

Replace line 11 with the following, rather cryptic looking line – this will make the output much more structured:

```
    print("{0:f}\t{1: f}".format(x, math.sin(x)))
```

The syntax of the `str.format()` method is quite complicated, so either google it or don't worry about it for now. We will come back to it.

Exercise 8.2. You can plot the graph of the sin function in a similar way to how we gave a visual representation of the relative frequencies of dice throws. The idea is that we scale the function appropriately so that we can print an appropriate number of text characters.

The output of the $\sin(x)$ function is a number between -1 and 1; we want to scale it to a number that we can print as a number of characters, so between 0 and 60 sounds about right. If we use $30(1 + \sin(x))$ we will get a number between 0 and 60. Change line 12 of the above program so that it prints out that many asterisks. (You might want to change the number of intervals to 40.)

The output that you get is more like a bar chart than a plot of a function. You can alter this by printing the correct number of spaces followed by a single asterisk, so your print statement looks like `print(" " * n + "*")` where `n` is the number of asterisks you were printing. Note here that `+` means the concatenation of the two strings, so the result will be something like `"                    *"`.

Homework

There is no assessed homework for the next lab, as the mini-project deadline is this week.

1. Finish off this sheet.

2. (*Quick review.*) In the following program, what values will `x` take? What will the output look like? Type in the code and check your answer.

```
import math

NO_OF_INTERVALS = 40
for i in range(NO_OF_INTERVALS + 1):
    x = 4 * (i / NO_OF_INTERVALS - 1/2)
    y = math.exp(-x**2)
    print("{0:f}".format(x), "*"*int(40*(y)), sep="\t")
```

3. (*Optional.*) The Birthday Paradox states that if you have 23 people in a room then there is greater than a 50:50 chance that at least two people share a birthday. Write a program that verifies this, by randomly selecting 23 dates, checking if there are two which coincide, repeating this process a large number of times, and printing out the proportion of times that there was success.