05 Library functions

September 13, 2020

1 Introduction

A feature of modern programming languages is an extensive library of *standard functions*. This means that we can make use of well-tested and optimised functions for performing common tasks rather than writing our own. This makes our programs shorter and of higher quality, and in most cases faster.

1.1 Objectives

- Introduce use of standard library functions
- Importing and using modules
- Introduction to namespaces
- Print formatting of floats

2 The standard library

You have already used some standard library types and functions. In previous activities we have used built-in types like `string` and `float`, and the function `abs` for absolute value. We have made use of the standard library function `print` to display to the screen.

Python has a large standard library. To organise it, most functionality is arranged into 'modules', with each module providing a range of related functions. Before you program a function, check if there is a library function that can perform the task. The Python standard library is documented at https://docs.python.org/3/library/. Search engines are a good way to find library functions, e.g. entering "Is there a Python function to compute the hyperbolic tangent of a complex number" into a search engine will take you to the function `cmath.tanh`. Try this link: http://bfy.tw/7aMc.

3 Other libraries

The standard library tools are general purpose and will be available in any Python environment. Specialised tools are usually made available in other libraries (modules). There is a huge range of Python libraries available for specialised problems. We have already used some parts of NumPy (http://www.numpy.org/), which is a specialised library for numerical computation.

The simplest way to install a non-standard library is using the command `pip`. From the command line, the library NumPy is installed using:

```
pip install numpy
```

and from inside a Jupyter notebook use:

```
!pip install numpy
```

NumPy is so commonly used it is probably already installed on computers you will be using. You will see `pip` being used in some later notebooks to install special-purpose modules.

When developing programs outside of learning exercises, if there is a no standard library module for a problem you are trying to solve, search online for a module before implementing your own.

4 Using library functions: `math` example

To use a function from a module we need to make it available in our program. This is called 'importing'. We have done this in previous notebooks with the `math` module, but without explanation. The process is explained below.

The `math` module (https://docs.python.org/3/library/math.html) provides a wide range of mathematical functions. For example, to compute the square root of a number, we do:

[1]:
```
import math

x = 2.0
x = math.sqrt(x)
print(x)
```

```
1.4142135623730951
```

Dissecting the above code block, the line

```
import math
```

makes the math module available in our program. It is good style to put all `import` statements at the top of a file (or at the top of a cell when using a Jupyter notebook).

The function call

```
x = math.sqrt(x)
```

says 'use the `sqrt` function from the `math` module to compute the square root'.

By prefixing `sqrt` with `math`, we are using a *namespace* (which in this case is `math`). This makes clear precisely which `sqrt` function we want to use - there could be more than one `sqrt` function available.

> *Namespaces:* The prefix '`math`' indicates which '`sqrt`' function we want to use. This might seem pedantic, but in practice there are often different algorithms for performing the same or similar operation. They might vary in speed and accuracy. In some applications we might need an accurate (but slow) method for computing the square root, while for other applications we might need speed with a compromise on accuracy. But, if two functions have the same name and are not distinguished by a name space, we have a *name clash*.
>
> In a large program, two developers might choose the same name for two functions that perform similar but slightly different tasks. If these functions are in different modules, there will be no name clash since the module name provides a 'namespace' - a prefix that provides a distinction between the two functions. Namespaces are extremely helpful

for multi-author programs. Older languages, like C and Fortran, might not support namespaces. Most modern languages do support namespaces.

We can import specific functions from a module, e.g. importing only the `sqrt` function:

```
[2]: from math import sqrt

     x = 2.0
     x = sqrt(x)
     print(x)
```

```
1.4142135623730951
```

This way, we are importing (making available) only the `sqrt` function from the `math` module (the `math` module has a large number of functions).

We can even choose to re-name functions that we import:

```
[3]: from math import sqrt as some_math_function

     x = 2.0
     x = some_math_function(x)
     print(x)
```

```
1.4142135623730951
```

Renaming functions at import can be helpful to keep code short, and we will see below it can be useful for switching between different functions. However the above choice of name is very poor practice - the name '`some_math_function`' is not descriptive. Below is a more sensible example.

Say we program a function that computes the roots of a quadratic function using the quadratic formula:

```
[4]: from math import sqrt as square_root

     def compute_roots(a, b, c):
         "Compute roots of the polynomial f(x) = ax^2 + bx + c"
         root0 = (-b + square_root(b*b - 4*a*c))/(2*a)
         root1 = (-b - square_root(b*b - 4*a*c))/(2*a)
         return root0, root1

     # Compute roots of f = 4x^2 + 10x + 1
     root0, root1 = compute_roots(4, 10, 1)
     print(root0, root1)
```

```
-0.10435607626104004 -2.3956439237389597
```

The above is fine as long as the polynomial has real roots. However, the function `math.sqrt` will give an error (technically, it will 'raise an exception') if a negative argument is passed to it. This is to stop naive mistakes.

We do know about complex numbers, so we want to compute complex roots. The Python module cmath provides functions for complex numbers. If we were to use cmath.sqrt to compute the square root, our function would support complex roots. We do this by importing the cmath.sqrt functions as square_root:

[5]:
```
# Use the function from cmath as square_root to compute the square root
# (this will replace the previously imported sqrt function)
from cmath import sqrt as square_root

# Compute roots (roots will be complex in this case)
root0, root1 = compute_roots(40, 10, 1)
print(root0, root1)

# Compute roots (roots will be real in this case, but cmath.sqrt always returns␣
↪a complex type)
root0, root1 = compute_roots(4, 10, 1)
print(root0, root1)
```

```
(-0.125+0.09682458365518543j) (-0.125-0.09682458365518543j)
(-0.10435607626104004+0j) (-2.3956439237389597+0j)
```

The function now works for all cases because square_root is now using cmath.sqrt. Note that cmath.sqrt always returns a complex number type, even when the complex part is zero.

5 String functions and string formatting

A standard function that we have used from the start is 'print'. This function turns arguments into a string and displays the string to the screen. So far, we have only printed simple variables and relied mostly on the default conversions to a string for printing to the screen (the exception was printing the floating point representation of 0.1, where we needed to specify the number of significant digits to see the inexact representation in binary).

5.1 Formatting

We can control how strings are formatted and displayed. Below is an example of inserting a string variable and a number variable into a string of characters:

[6]:
```
# Format a string with name and age
name = "Amber"
age = 19
text_string = "My name is {} and I am {} years old.".format(name, age)

# Print to screen
print(text_string)

# Short-cut for printing without assignment
name = "Ashley"
age = 21
print("My name is {} and I am {} years old.".format(name, age))
```

```
My name is Amber and I am 19 years old.
My name is Ashley and I am 21 years old.
```

For floating-point numbers we often want to control the formatting, and in particular the number of significant figures displayed. Using the display of π as an example:

[7]:
```
# Import math module to get access to math.pi
import math

# Default formatting
print("The value of   using the default formatting is: {}".format(math.pi))

# Control number of significant figures in formatting
print("The value of   to 5 significant figures is: {:.5}".format(math.pi))
print("The value of   to 8 significant figures is: {:.8}".format(math.pi))
print("The value of   to 20 significant figures and using scientific notation␣
 ↪is: {:.20e}".format(math.pi))
```

```
The value of   using the default formatting is: 3.141592653589793
The value of   to 5 significant figures is: 3.1416
The value of   to 8 significant figures is: 3.1415927
The value of   to 20 significant figures and using scientific notation is:
3.14159265358979311600e+00
```

There are many more ways in which float formatting can be controlled - search online if you want to format a float in a particular way.

6 Module example: parallel processing

Standard modules can make very technical problems simpler. An example is parallel processing.

The majority of CPUs - from phones to supercomputers - now have CPUs with multiple cores, with each core performing computations. To benefit from the multiple cores, we need to compute in *parallel*. A ‘standard’ program performs tasks in order, and usually only one core will be utilised and the rest will remain idle. To get the best performance from the hardware, we need to compute in parallel. That is, we perform multiple tasks at the same time.

Parallel processing (or *concurrency*) is an enormous and highly technical topic on its own, but we can touch upon it here because we have standard libraries that make it easy to use. Managing parallel tasks at a low-level can be extremely technical, but standard libraries can simplify parallel operations. We will use the `multiprocessing` module, and use it to sort lists of numbers concurrently.

We start by looking at how to generate a list of random integers using the `random` module. The following code creates a list (more on lists in the following notebook) of 10 random integers in the range 0 to 100 (not including 100):

[8]:
```
import random
x = random.sample(range(0, 100), 10)
print(x)
```

```
[67, 75, 92, 90, 27, 13, 73, 29, 42, 9]
```

To create a sorted list, we used the built-in function `sorted`:

[9]:
```
y = sorted(x)
print(y)
```

```
[9, 13, 27, 29, 42, 67, 73, 75, 90, 92]
```

Now, if we need to sort multiple different lists, we could sort the lists one after the other, or we could sort several lists at the same time (in parallel). Our operating system will then manage the dispatch of the sorting task to different processor cores. Before seeing how to do this, we implement a function to perform the sorting:

[10]:
```
import multiprocessing
import random

def mysort(N):
    "Create a list of random numbers of length N, and return a sorted list"

    # Create random list
    x = random.sample(range(0, N), N)

    # Print process identifier (just out of interest)
    print("Process id: {}".format(multiprocessing.current_process()))

    # Return sorted list of numbers
    return sorted(x)
```

To create the sorted lists, making available three processes (threads), we use:

[11]:
```
N = 20000
with multiprocessing.Pool(processes=3) as p:
    p.map(mysort, [N, N, N])  # Call function mysort three times
```

```
Process id: <ForkProcess(ForkPoolWorker-3, started daemon)>
Process id: <ForkProcess(ForkPoolWorker-2, started daemon)>
Process id: <ForkProcess(ForkPoolWorker-1, started daemon)>
```

We see from the output that three different processes have worked on our problem - one for each sorting task.

We use parallel processing the make computations faster. Let's time our computation using different numbers of processes to see how it affects performance. To perform the timing, we first encapsulate our problem in a function:

[12]:
```
def parallel_sort(N, num_proc):
    "Create three lists of random numbers (each of length N) using num_proc␣
 ↪processes"
    with multiprocessing.Pool(processes=num_proc) as p:
        p.map(mysort, [N, N, N])
```

Using the magic command '%time', we time the sorting using just one process (the one process sorts the lists one after the other):

[13]:
```
N = 500000
%time parallel_sort(N, 1)
```

```
Process id: <ForkProcess(ForkPoolWorker-4, started daemon)>
Process id: <ForkProcess(ForkPoolWorker-4, started daemon)>
Process id: <ForkProcess(ForkPoolWorker-4, started daemon)>
CPU times: user 54.6 ms, sys: 27.9 ms, total: 82.5 ms
Wall time: 1.67 s
```

We see from '`Process id`' that the same process worked on all three lists.

We now try with up to 4 processes (there are only three lists to sort, so only three will be used):

[14]:
```
%time parallel_sort(N, 4)
```

```
Process id: <ForkProcess(ForkPoolWorker-6, started daemon)>
Process id: <ForkProcess(ForkPoolWorker-7, started daemon)>
Process id: <ForkProcess(ForkPoolWorker-5, started daemon)>
CPU times: user 58.5 ms, sys: 37.2 ms, total: 95.6 ms
Wall time: 753 ms
```

We see from `Process id` that three different processes have worked on the lists. The parallel execution should be faster, but this can depends heavily on the CPU type.

7 Exercises

Complete now the 05 Exercises notebook.