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
- Accessing documentation for functions and modules

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: https://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 (https://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: fractions and statistics

Python has standard library support for fractions and statistical operations. Summing fractions:

[8]:
```
from fractions import Fraction

f0 = Fraction(2, 3)
f1 = Fraction(3, 7)
result = f0 + f1
print("Fraction:", result)
print("Float:", float(result))
```

```
Fraction: 23/21
Float: 1.0952380952380953
```

We can use the `statistics` module perform statistical operations on a list of numbers. In this example we will represent the numbers using `fractions.Fraction`.

[9]:
```
import statistics
```

```
data = [Fraction(1, 2), Fraction(2, 1), Fraction(1, 3), Fraction(1, 17),␣
 ↪Fraction(-1, 13)]
print("numbers:", data)

print("mean:", statistics.mean(data))
print("mode:", statistics.mode(data))
var = statistics.variance(data)
print("variance: {} ({})".format(var, float(var)))
```

```
numbers: [Fraction(1, 2), Fraction(2, 1), Fraction(1, 3), Fraction(1, 17),
Fraction(-1, 13)]
mean: 3733/6630
mode: 1/2
variance: 3060917/4395690 (0.6963450561800308)
```

7 Tab completion

Tab completion is useful for seeing which functions a module provides. For example, after importing the `math` module type `math.` and press `tab`. This will show all the available functions in `math`. Typing `math.c` and pressing `tab` will show all the availale functions that start with 'c'.

8 Accessing function documentation

The documentation for a function can be accessed using the `help` function, e.g.:

[10]:
```
help(math.factorial)
```

```
Help on built-in function factorial in module math:

factorial(x, /)
    Find x!.

    Raise a ValueError if x is negative or non-integral.
```

The `help` function can also be used to access the documentation for a module:

[11]:
```
help(math)
```

```
Help on module math:

NAME
    math

MODULE REFERENCE
    https://docs.python.org/3.8/library/math

    The following documentation is automatically generated from the Python
```

```
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.

    asinh(x, /)
        Return the inverse hyperbolic sine of x.

    atan(x, /)
        Return the arc tangent (measured in radians) of x.

    atan2(y, x, /)
        Return the arc tangent (measured in radians) of y/x.

        Unlike atan(y/x), the signs of both x and y are considered.

    atanh(x, /)
        Return the inverse hyperbolic tangent of x.

    ceil(x, /)
        Return the ceiling of x as an Integral.

        This is the smallest integer >= x.

    comb(n, k, /)
        Number of ways to choose k items from n items without repetition and
without order.

        Evaluates to n! / (k! * (n - k)!) when k <= n and evaluates
        to zero when k > n.

        Also called the binomial coefficient because it is equivalent
        to the coefficient of k-th term in polynomial expansion of the
        expression (1 + x)**n.
```

```
        Raises TypeError if either of the arguments are not integers.
        Raises ValueError if either of the arguments are negative.

    copysign(x, y, /)
        Return a float with the magnitude (absolute value) of x but the sign of
y.

        On platforms that support signed zeros, copysign(1.0, -0.0)
        returns -1.0.

    cos(x, /)
        Return the cosine of x (measured in radians).

    cosh(x, /)
        Return the hyperbolic cosine of x.

    degrees(x, /)
        Convert angle x from radians to degrees.

    dist(p, q, /)
        Return the Euclidean distance between two points p and q.

        The points should be specified as sequences (or iterables) of
        coordinates.  Both inputs must have the same dimension.

        Roughly equivalent to:
            sqrt(sum((px - qx) ** 2.0 for px, qx in zip(p, q)))

    erf(x, /)
        Error function at x.

    erfc(x, /)
        Complementary error function at x.

    exp(x, /)
        Return e raised to the power of x.

    expm1(x, /)
        Return exp(x)-1.

        This function avoids the loss of precision involved in the direct
evaluation of exp(x)-1 for small x.

    fabs(x, /)
        Return the absolute value of the float x.

    factorial(x, /)
```

```
    Find x!.

    Raise a ValueError if x is negative or non-integral.

floor(x, /)
    Return the floor of x as an Integral.

    This is the largest integer <= x.

fmod(x, y, /)
    Return fmod(x, y), according to platform C.

    x % y may differ.

frexp(x, /)
    Return the mantissa and exponent of x, as pair (m, e).

    m is a float and e is an int, such that x = m * 2.**e.
    If x is 0, m and e are both 0.  Else 0.5 <= abs(m) < 1.0.

fsum(seq, /)
    Return an accurate floating point sum of values in the iterable seq.

    Assumes IEEE-754 floating point arithmetic.

gamma(x, /)
    Gamma function at x.

gcd(x, y, /)
    greatest common divisor of x and y

hypot(…)
    hypot(*coordinates) -> value

    Multidimensional Euclidean distance from the origin to a point.

    Roughly equivalent to:
        sqrt(sum(x**2 for x in coordinates))

    For a two dimensional point (x, y), gives the hypotenuse
    using the Pythagorean theorem:  sqrt(x*x + y*y).

    For example, the hypotenuse of a 3/4/5 right triangle is:

        >>> hypot(3.0, 4.0)
        5.0

isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)
```

```
        Determine whether two floating point numbers are close in value.

          rel_tol
            maximum difference for being considered "close", relative to the
            magnitude of the input values
          abs_tol
            maximum difference for being considered "close", regardless of the
            magnitude of the input values

        Return True if a is close in value to b, and False otherwise.

        For the values to be considered close, the difference between them
        must be smaller than at least one of the tolerances.

        -inf, inf and NaN behave similarly to the IEEE 754 Standard.  That
        is, NaN is not close to anything, even itself.  inf and -inf are
        only close to themselves.

    isfinite(x, /)
        Return True if x is neither an infinity nor a NaN, and False otherwise.

    isinf(x, /)
        Return True if x is a positive or negative infinity, and False
otherwise.

    isnan(x, /)
        Return True if x is a NaN (not a number), and False otherwise.

    isqrt(n, /)
        Return the integer part of the square root of the input.

    ldexp(x, i, /)
        Return x * (2**i).

        This is essentially the inverse of frexp().

    lgamma(x, /)
        Natural logarithm of absolute value of Gamma function at x.

    log(…)
        log(x, [base=math.e])
        Return the logarithm of x to the given base.

        If the base not specified, returns the natural logarithm (base e) of x.

    log10(x, /)
        Return the base 10 logarithm of x.
```

```
    log1p(x, /)
        Return the natural logarithm of 1+x (base e).

        The result is computed in a way which is accurate for x near zero.

    log2(x, /)
        Return the base 2 logarithm of x.

    modf(x, /)
        Return the fractional and integer parts of x.

        Both results carry the sign of x and are floats.

    perm(n, k=None, /)
        Number of ways to choose k items from n items without repetition and
with order.

        Evaluates to n! / (n - k)! when k <= n and evaluates
        to zero when k > n.

        If k is not specified or is None, then k defaults to n
        and the function returns n!.

        Raises TypeError if either of the arguments are not integers.
        Raises ValueError if either of the arguments are negative.

    pow(x, y, /)
        Return x**y (x to the power of y).

    prod(iterable, /, *, start=1)
        Calculate the product of all the elements in the input iterable.

        The default start value for the product is 1.

        When the iterable is empty, return the start value.  This function is
        intended specifically for use with numeric values and may reject
        non-numeric types.

    radians(x, /)
        Convert angle x from degrees to radians.

    remainder(x, y, /)
        Difference between x and the closest integer multiple of y.

        Return x - n*y where n*y is the closest integer multiple of y.
        In the case where x is exactly halfway between two multiples of
        y, the nearest even value of n is used. The result is always exact.
```

```
    sin(x, /)
        Return the sine of x (measured in radians).

    sinh(x, /)
        Return the hyperbolic sine of x.

    sqrt(x, /)
        Return the square root of x.

    tan(x, /)
        Return the tangent of x (measured in radians).

    tanh(x, /)
        Return the hyperbolic tangent of x.

    trunc(x, /)
        Truncates the Real x to the nearest Integral toward 0.

        Uses the __trunc__ magic method.

DATA
    e = 2.718281828459045
    inf = inf
    nan = nan
    pi = 3.141592653589793
    tau = 6.283185307179586

FILE
    /usr/local/Cellar/python@3.8/3.8.5/Frameworks/Python.framework/Versions/3.8/
lib/python3.8/lib-dynload/math.cpython-38-darwin.so
```

In a notebook it is also possible to use `?` to access the documentation, which will be displayed at the bottom of the notebook:

[12]:
```
import random
random.gauss?
```

9 Exercises

Complete now the 05 Exercises notebook.