Python #8
Creating new functions to do your work

Location: https://people.sc.fsu.edu/~jburkardt/classes/python_2022/python08/python08.pdf

Freely adapted from the Python lessons at https://software-carpentry.org/

Functions

- *Python has built-in functions, with more available from libraries like* `numpy`*;*
- *A function produces output by operating on input;*
- *Users can define new functions that carry out useful operations;*
- *Functions begin* `def output = function_name ( input ):`
- *A* `return` *statement transfers the output to the user;*
- *Functions can be defined interactively, but are better saved in a file;*
- *A user function can be stored in a file and accessed with an* `import` *statement.*

1 What a function looks like

In Python, a *function* is an object which has a name, accepts input, carries out a calculation that uses that input, and returns a result as output. The classic way of using a function is something like this:

```
output = function_name ( input )
```

A function might have no inputs, in which case you just see a pair of empty parentheses after the function name. One of the few Python functions of this kind is the rarely-used `globals()`:

```
dictionary = globals ( )
```

However, there are many reasons why a user might want to create functions with no input.

A function might take several inputs separated by commas:

```
biggest = max ( input1, input2, input3 )
```

A function might not have any output variable

```
print ( 'x = ', x )
```

A function can have multiple outputs. If so, when the user makes a call, the outputs should be separated by commas. As an example, the `divmod()` function is given numbers `a` and `b`, usually integers, and returns the quotient `q` and remainder `r` of integer division `a/b`:

```
q, r = divmod ( a, b )
```

2 Accessing functions that aren't built-in

As you have probably already seen in other examples, there are many libraries of Python functions which can be useful in particular calculations. MATLAB users may be reminded of the corresponding Toolbox feature in that language.

In order to access a function from such a library, several things must happen:

- The library must already be installed. If you are using Anaconda, this will usually not be a problem.
- The user must issue an `import` command to gain access to the function;
- Depending on the `import` command, the user may need to specify the library name as well.

There are several versions of the `import` command, depending on the user's preference. Since `numpy` is one of the most commonly used libraries, we will suppose that the user wants to access the function `std()` from that library, which computes the standard deviation of a vector. Here are four possible `import` statements:

```
import numpy
value = numpy.std ( v )      # Include library name in function call

import numpy as np           # Specify a short "nickname" for library
value = np.std ( v )         # Include nickname in function call

from numpy import std        # Only get std() from numpy
value = std ( v )            # No need to specify library name

from numpy import *          # Get everything from numpy
value = std ( v )            # No need to specify library name
```

Most users prefer the second option, always using `np` as a nickname for `numpy` (although actually, you could use any nickname, such as "fred", instead!).

Two other commonly used libraries are the scientific computing library `scipy`, whose common nickname is `sp`, and the graphics library with the awful name `matplotlib.pyplot`, nicknamed `plt`.

If you want to know the names of all the functions in a library, you can use the `dir()` command:

```
dir ( numpy )
```

and if you want more information on one of the library functions, you can use `help()`, specifying the name of the function, as well as the library name or nickname, if required:

```
help ( numpy.who )  # because you used "import numpy"
help ( np.who )     # because you used "import numpy as np"
help ( who )        # because you used "from numpy import who" or else "from numpy import *"
```

3 Form of a user-defined function

A user can write new functions, which follow the form and use of the Python built-in functions. We will see that, usually, a function is stored in a file, but it's certainly possible to (carefully and sequentially) enter a short function during an interactive session.

For starters, here's a three-line function that evaluates the n-th triangular number:

```
def triangular_number ( n ):
  t = ( n * ( n + 1 ) ) // 2
  return t
```

Let's analyze this little code to death:

1. This line begin with `def` because it *defines* a function. There follows the function name (choose one you like!), a list of input quantities in parentheses, and a final colon.
2. This function only uses a single computational line; here we apply the formula for triangular numbers to our input value `n`, storing the result as `t`.
3. A `return` statement exits the function, and returns the named value as the output or result. (Some functions have no output, in which case they can have a bare `return` statement, or skip the `return` completely.)

Notice that the usual indenting rules apply. Every line that is to be included in the function must be indented in a consistent way. As soon as Python sees an unindented line, it knows the function definition is complete.

Once we have defined `triangular_number()`, we can test it by

```
  t = triangular_number ( 1 )    # 1
  t = triangular_number ( 10 )   # 55
  t = triangular_number ( 100 )  # 5050
```

The function we have defined remains available as long as our Python session continues, but when we `quit()` Python, it will vanish, and no longer be accessible. This may not be a tragedy for a short function, but we will surely want a way to be able to construct more elaborate functions and then not lose them so quickly!

4 The BMI function

Hospitals often combine the patient's height and weight in a formula known as the body-mass-index (BMI). In metric units, this is defined by the weight in kilograms, divided by the square of the height in meters. Mathematically, we might write:

$$\text{bmi} = \frac{\text{kg}}{\text{m}^2}$$

Unfortunately, using English units, the process is not so simple, as we have to convert to the metric system. In order to hide this extra work, we can write our own `bmi()` function. The input will be the patient's weight in pounds, and height in inches, while the output will be the BMI.

Here is what such a function would look like:

```
def bmi ( weight_lb, height_in ):
  value = ( weight_lb / 2.204 ) * ( 39.370 / height_in )**2
  return value
```

You can demonstrate this function by computing

```
  bmi ( 100, 50 ) = 28.13
  bmi ( 150, 60 ) = 29.30
  bmi ( 200, 70 ) = 28.70
```

Using this `bmi()` function and a `while()` statement, determine the maximum weight for which the BMI is 25, given that a person is 50 inches tall. You can assume that the weight will be an integer value.

5 Documenting your function

You may discover that the `help()` function can be very useful in trying to get some information about a particular built-in or library function. If you write a function yourself, you can also add information to it that `help()` can report. Students doing homework exercises often don't have the patience to do this, and regard each function they write as a sort of throwaway or disposable item. If, however, you end up doing research programming or working on a software team, you will very likely have to share the code you work on, and in such a case, documentation can be required. So even if you are not prepared to go to the extra trouble right now, you should be aware of how to add and access such documentation.

In any programming language, a function ought to include the following information:

- **Purpose:** one line description of the function.
- **Discussion:** if appropriate, a more extensive explanation of the algorithm.
- **Example:** at least one example of how the function might be used, with results.
- **Modified:** the date when the function was created, or most recently changed.
- **Author:** the name of the person who wrote the code.
- **Reference:** the name of a reference book or paper, if appropriate.
- **Input:** the type and meaning of each input variable.
- **Output:** the type and meaning of each output variable.

Here is a second version of our BMI function, which now looks enormous because it started out essentially being one line. In order to make the information visible to `help()`, it must start just after the function statement, and begin and end with triple single quotes, which are indented.

```
def bmi ( weight_lb, height_in ):
  '''
  Purpose: bmi() computes the BMI (body mass index).

  Discussion: the BMI has a simple formula when the input is defined in metric units
  This program accepts data in English units (pounds and inches) and applies the
  correct unit conversions.  An "ideal" BMI is between 18.5 and 30.  Above 30 begins
  to count as "obese" and below 18.5 is considered "underweight".

  Example: bmi ( 150, 60 ) returns the value 29.30

  Modified: 12 May 2022

  Author: John Burkardt

  Reference:  https://en.wikipedia.org/wiki/Body_mass_index

  Input:
    weight_lb, the weight in pounds.
    height_in, the height in inchdes.

  Output:
    value, the body mass index.
  '''
```

```
  value = ( weight_lb / 2.204 ) * ( 39.370 / height_in )**2
  return value
```

As this example suggests, proper documentation of a function can sometimes require more work than writing the actual formulas. But whenever you have written code that you may want to use again later, or share with others, it is a valuable way to ensure that the code is usable.

6 Putting your function in a file

Obviously, if we are going to document the `bmi()` function so carefully, it's really ridiculous if, the minute we `quit()` our Python session, all that information disappears forever! Although it's easy to experiement with short pieces of code interactively, you will find that for any serious work, you need to focus on preparing work in files beforehand, so that when you go interactive, you are invoking things that are already done.

Let's look at how this might be done for our `bmi()` function. To start with, we need to choose a name for the file that is to contain our function. We will actually plan to eventually put *several* functions into this file, so we'll give the file the general name *my_library.py*

Using an editor, or applying cut and paste to this document, we assemble the text for the `bmi()` function and put it into the file *my_library.py*.

To access our function, we need an `import` statement that specifies the function name and the library containing it. We will also assume that when we start Python, we are working in the same directory that contains our library file, so we don't have to worry about how to describe a file in another directory. Assuming all that, here is a very short Python session to demonstrate our function:

```
python3
>>> import bmi from my_library
>>> value = bmi ( 150, 60 )
>>> print ( 'bmi(150,60) = ', value )
>>> quit()
```

7 Adding another function to our library, and accessing it

Since we are on an English units streak, let's write another function, which converts temperatures in Fahrenheit to Celsius. The mathematical formula is simply

$$C^\circ = \frac{5}{9}\left(F^\circ - 32\right)$$

and the reverse is

$$F^\circ = \frac{9}{5}C^\circ + 32$$

Write functions `f_to_c(f)` and `c_to_f(c)` which convert between the two temperature scales. Add these two new functions to your file `my_library()`. Use them in a Python calculation to verify the following interesting facts in which the Celsius temperature is approximately equal to the Fahrenheit temperature with the digits reversed:

```
 F   C
40   04
61   16
82   28
```

Your code might begin:

```
import f_to_c from my_library
for f in [ 40, 61, 82 ]:
  ... more stuff
```

Verify that one function is the inverse of the other by taking temperature values and converting them back and forth.

```
  C ->   F ->   C
-40 -> -40 -> -40
  0 ->  32 ->   0
100 -> 212 -> 100
```

Note that you can write expressions like

```
fnew = c_to_f ( f_to_c ( f ) )
```

to do both steps of the conversion in one line.

8 One function can call another

Aside from the Celsius and Fahrenheit temperature scales, physicists work with the Kelvin temperature. The temperature in degrees Kelvin, $K°$, is related to the Celsius temperature by

$$K = C - 273.15$$

You should be able to quickly write two new functions, `c_to_k(c)` and `k_to_c(k)`, which convert back and forth between these two systems. Add them to *my_library.py*.

We shall also write two more functions, `f_to_k(f)` and `k_to_f(k)`, which convert between Fahrenheit and Kelvin. However, we can take a shortcut, since we already know how to get from F to C, and then from C to K. So the function `f_to_k(f)` could be written something like this:

```
def f_to_k ( f ):
  c = convert f to c  # Replace by the correct function call
  k = convert c to k  # Replace by the correct function call
  return k
```

and the function `k_to_f(k)` should be just as easy.

Using these new functions, verify the following:

```
  F ->   K ->   F
-40 -> 233 -> -40
 32 -> 273 ->  32
212 -> 373 -> 212
```

9 A function that returns an array

A polynomial of degree n can be described by an $n+1$ vector of coefficients that multiply successive powers of a parameter x:

$$p(x) = c_0 + c_1 x + c_2 x^2 + ... + c_n x^n$$

Suppose we know the coefficients of p(x), and we wish to compute the coefficients d of the derivative of the polynomial, $p'(x) = \frac{dp(x)}{dx}$, whose full expression is

$$\begin{aligned} p'(x) &= d_0 + d_1 x + d_2 x^2 + d_{n-1} x^{n-1} \\ &= c_1 + 2c_2 x + 3c_3 x^2 + n * c_n x^{n-1} \end{aligned}$$

In other words, we want to compute a vector d of length n with values:

$$\begin{aligned} d_0 &= c_1 \\ d_1 &= 2 * c_2 \\ &... \\ d_{n-1} &= n * c_n \end{aligned}$$

Can we write a Python function of the form `poly_dif ( c )` which computes and returns this vector?

Consider the tasks you must carry out:

- Perhaps you need to import a certain library to work with arrays;
- Given only the array `c`, determine `n`, the degree of the original polynomial;
- Create a vector `d` of the appropriate size;
- Set up a loop (perhaps a `for` loop?) which computes each value `d[i]`;
- return d

Create such a function, perhaps put it in a library called *poly.py*, and test it on the following polynomial:

$$p(x) = 100 + 50x + 25x^2 + 10 * x^3 + 5 * x^4 + 2 * x^5 + 6 * x^6$$

A second useful function that you could add to your `poly` library might be called `value = poly_value(c,x)`. This function would take as input the vector `c` of polynomial coefficients, and a point `x`. It would return in `value` the value of the polynomial at `x`.

A third useful function might be called `value = poly_integrate(c,a,b)`. It would determine the integral of the polynomial over the interval `[a,b]`. This takes a little bit of work to get right. You could compute the coefficients of the antiderivative polynomial, and then call `poly_val()` twice to determine the integral.