Creating new functions
Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm

https://people.sc.fsu.edu/~jburkardt/classes/python_2025/functions/functions.pdf

---

Functions

- *A function produces output by operating on input;*
- *Users can define new functions that carry out useful operations;*
- *Functions begin* `def function_name ( input ):`
- *A* `return output` *statement transfers output values to the user;*
- *A function has a separate memory space; only input and output are seen elsewhere;*
- *A user function in a separate file and accessed with an* `import` *statement.*
- *Input parameters can be given default values;*
- *Input parameters can be associated with keywords;*

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 )
```

We have already used built-in functions from Python:

```
  from math import factorial, max, sin, sqrt
  height = s * sin ( angle )
  longest = max ( a, b, c )
  area = sqrt ( s * ( s - a ) * ( s - b ) * ( s - c ) )
  print ( 'Area = ', area )
  nchoosek = factorial ( n ) / factorial ( n - k ) / factorial ( k )
```

A function is a convenient tool for carrying out an operation that we need to access many times. We need to know how to make new tools for our own special needs.

2 Form of a user-defined function

To create a new function, you essentially have to fill in the missing information in the following simplified function template:

```
def name ( input ):
  indented statements
  return output
```

That is, you choose a *name* for your function, you give names to the *input* you expect to receive, you write out the operations you will perform, and then you return the `output`.

Here's a three-line function that evaluates the n-th triangular number:

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

and here's one that evaluates the area of a triangle with sides of length `a, b, c`:

```
def triangle_area ( a, b, c ):
# triangle_area() uses Heron's formula
  from math import sqrt
  s = ( a + b + c ) / 2
  area = sqrt ( s * ( s - a ) * ( s - b ) * ( s - c ) )
  return area
```

Notice the indenting rules. After the function `def` statement, the lines forming the body of the function are indented. If you are defining the function interactively, then as soon as you enter an unindented line, Python knows your 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
```

If we have only defined a function during an interactive session, it will vanish when the session is over. Since functions can be tricky to type correctly, and are probably useful later, we would prefer to have a way to keep such functions available.

We could create the function in advance, using an editor, storing it in a separate file called *triangular_number.py*, or *triangle_area.py*, or we could store both these functions in a common file *my_library.py*. In that case, when we are doing our programming and need to get one of these functions, we have to use the `import` statement, just as we have done to access functions from Python's `math` library.

```
  from my_library import triangular_number, triangle_area
  t = triangular_number ( 10 )
  area = triangle_area ( 1.0, 3.0, 2.0 )
```

3 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_engish()` 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_english ( weight_lb, height_in ):
  value = ( weight_lb / 2.204 ) * ( 39.370 / height_in )**2
  return value
```

You can demonstrate this function by computing

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

Using this `bmi_english()` 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.

4 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.

5 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
```

6 The double factorial function

You are familiar with the factorial function of a nonnegative integer n, written $n!$, defined mathematically by

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ 1 * 2 * ... * n & \text{otherwise} \end{cases}$$

and available in Python by

```
from math import factorial
```

A related function is the *double factorial*, symbolized by $n!!$, and defined for nonnegative integers n by:

$$n!! = \begin{cases} 1 & \text{if } n = 0 \\ 1 * 3 * 5 * ... * n - 2 * n & \text{if } n \text{ is odd} \\ 2 * 4 * 6 * ... * n - 2 * n & \text{if } n \text{ is even and greater than } 0 \end{cases}$$

and available in Python by

```
from scipy.special import factorial2
```

For instance, $5!! = 1 * 3 * 5 = 15$ and $8!! = 2 * 4 * 6 * 8 = 384$.

We would like to try to write our own Python version of this function. The definition might seem messy, but think about it this way. Start with `value=1`. Now we are going to repeatedly multiply `value` by `n`, `n-2`, ..., continuing to decrease the factors by 2 until we reach 1 or 0. Thus, to compute `8!!` we initialize `value` to 1, multiply it by 8, then 6, then 4, then 2, then...stop! A `while()` statement will let us express this process.

Let's assume our version of the function is called `double_factorial(n)`, and is added to our library file *my_library.py*. We will shortly need to get it from there.

7 Integrating a product of cosines and sines

Suppose your work involves evaluating definite integrals of a certain form, and that you have a formula for the value:

$$\int_0^{\frac{\pi}{2}} \cos^m \theta \sin^n \theta d\theta = \begin{cases} \frac{(m-1)!!(n-1)!!}{(m+n)!!} \frac{\pi}{2} & m \text{ and } n \text{ are both even} \\ \frac{(m-1)!!(n-1)!!}{(m+n)!!} & \text{otherwise} \end{cases}$$

Here, we require $1 < m$, and $1 < n$.

Suppose you need to evaluate this integral for a variety of pairs of values m, n. We can write a function `cosm_sinn()` to do this, but of course we will want to call on our `double_factorial()` function for help.

```
def cosm_sinn ( m, n ):

  from math import pi
  from my_library import double_factorial

# backslash lets us split the following long line

  value = double_factorial ( m - 1 ) * double_factorial ( n - 1 ) \
    / double_factorial ( m + n )

  if ( ( m % 2 == 0 ) and ( n % 2 == 0 ) ):
    value = value * pi / 2.0

  return value
```

We can test our two functions by constructing a table of integral values:

```
from my_library import cosm_sinn

for m in range ( 2, 5 ):
  print ( '' )
  for n in range ( 2, 5 ):
    print ( m, n, cosm_sinn ( m, n ) )
```

Because we mention it in our code, we have to import `cosm_sinn()`. However, even though this means we will also need `double_factorial()`, we don't have to import that, because we didn't use it directly in our code; Python will find it for us.

8 A function that returns an array

T 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 vector of coefficients c of $p(x)$, and we wish to compute the vector of coefficients d of the derivative of the polynomial, $p'(x) = \frac{dp(x)}{dx}$. We can read off the values of d from the formula for $p'(x)$:

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

In other words, if we are given a vector c of length $n+1$, 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? This is an unfair question, since we still don't know how to handle vectors or arrays. You might have some idea from some hints we have seen already, or else from your experience with other languages. So, as a preview of how our functions can deal with arrays, let't give it a try.

Consider the tasks you must carry out:

- The number of entries in a numpy array `c` is `len(c)`;
- We create a new slightly shorter array `d`
- A `for()` loop computes `d[i]` from `c[i+1]`;
- return d

```
def poly_dif ( c ):
  import numpy as np          # This abbreviates numpy to "np"
  n = len ( c )               # If c is a numpy array, len(c) is its length.
  d = np.zeros ( n - 1 )      # np.zeros(n-1) creates a zero array of length n-1.
  for i in range ( 0, n - 1 ):
    d[i] = ( i + 1 ) * c[i+1]
  return d
```

Consider the following polynomial:

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

for which the coefficient vector c is defined as:

```
c = np.array ( [ 100, 50, 25, 10, 5, 2, 6 ] )
```

Use the `poly_dif()` function to determine the coefficients of $p'(x)$.

Now compute the coefficients of $p''(x)$.