Math 246 Unit 4. Defining and using Python functions

Professor Brenton LeMesurier

September 17, 2015; revised on September 22 to clarify the exercise

Introduction

Our main objective in this Unit is to learn how to define our own Python **functions,** and see how these can help us to deal with sub-tasks done repeatedly (with variation) within a larger task.

Typically, core mathematical tasks (like evaluating a function $f(x)$ or solving an equation $f(x) = 0$) will be done within Python functions designed to communicate with other parts of the Python code rather than getting their input interacively or returning results by displaying them on the screen.

Then, interactive and top-level organization of testing and using such functions will be done by a so-called **script**: a Python file intended to be used directly, such as by running it within Spyder or iPython. Alternatively, the top-level file that coordinates the activities of other parts of the code could be an iPython notebook. That makes it easier to combine detailed explananation of what is going on, Python code, and computed results in a single document, making for a more "literate" understandable presentation.

A bit later, we will look at how to organize collections of functions and such into your own modules, akin to modules such as `math`. This will be useful when you have functions that you wish to use repeatedly within different tasks.

Example 1: A very simple function for computing the mean of two numbers

To illustrate the syntax, let's start with a very simple function: computing the mean of two numbers.

In [202]:

```
def mean(a, b):
    mean_of_a_and_b = (a + b)/2
    return mean_of_a_and_b
```

This can be used as follows:

In [203]:

```
mean0 = mean(2, 6)
print('The mean of 2 and 6 is', mean0)
```

```
The mean of 2 and 6 is 4.0
```

Note a few things:

1. The definition of a function begins with the command `def`
2. This is followed by:

   - a name for the function,
   - a parenthesized list of varable names, which are called the input *arguments*,
   - and finally a colon, which as always introduces an indented list of statements, in this case the statement th at describe the actions perfomed by the function.
3. When using the function, it is given a list of expressions (variable names or values or more elaborate formulas) and the values of these are *copied* in order to the internal variables given by the list of input arguments mentioned in the function's `def` line.
4. As soon as the function gets to a statement beginning with `return`, it evaluates the expression on that line, and ends, sending back this as the value of the function.
5. The name use for the variable into which the value of a function is assigned (here, `mean0`) does not have to be the same as the name ued internalt inth retrn statement (here, `mean_of_a_and_b`). In fact, the return statement can in genral use an expression (formula) that evaluates to the value that it sends back rather than just listing the name of an internal variable.

Variables "created" inside a function definition are local to it

A more subtle point: all the variables appearing in the function's `def` line (here `a` and `b`) and any created inside with an assignment statement (here just `mean_of_a_and_b`) are purely *local* to the function; they do not exist outside the function. For that reason, when you call a function, you have to do something with the return value, like assign it to a variable (as done here with mean0) or use it as input to another function (see below).

Aside: we will see later a way that a variable can be shared between a function and the rest of the file in which the function definition appears; so-called *global variables*.

To illustrate this point about local variables, let as look for the values of variables `a` and `mean_of_a_and_b` in the code after the function is called:

In [204]:

```
mean1 = mean(3,10)
```

In [205]:

```
print('After using function mean:')
print('a =', a)
```

```
After using function mean:
a = 1
```

In [206]:

```
print('mean_of_a_and_b =', mean_of_a_and_b)
```

```
---------------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-206-c105a3bab17b> in <module>()
----> 1 print('mean_of_a_and_b =', mean_of_a_and_b)

NameError: name 'mean_of_a_and_b' is not defined
```

In [207]:

```
print('mean1 =', mean1)
```

```
mean1 = 6.5
```

The same name can even be used locally in a function and also outside it in the same file; they are different objects with independent values:

In [208]:

```
def double(a):
    a = 2 * a
    return a

a = 1
print('Before calling function mean:')
print('a =', a)
b = double(a)
print('After calling function mean:')
print('b =', b)
print('but still a =', a)
```

```
Before calling function mean:
a = 1
After calling function mean:
b = 2
but still a = 1
```

Example 2: multiple output values, using tuples

Often, a function computes several quantities; one example is a function version of our quadratic equation solver, which takes three input parameters and computes a pair of roots. Here is a very basic function for this, ignoring for now possible problems like division by zero:

In [209]:

```
from math import sqrt
```

In [210]:

```
def solve_quadratic(c_2, c_1, c_0):
    '''
    Note: c_i is the coefficient of x^i
    '''
    discriminant = c_1**2 - 4 * c_0 * c_2
    root0 = (-c_1 + sqrt(discriminant))/(2 * c_2)
    root1 = (-c_1 - sqrt(discriminant))/(2 * c_2)
    return (root0, root1)
```

In [211]:

```
(rootA, rootB) = solve_quadratic(2, -10, 8)
print('One root is', rootA, 'and the other is', rootB)
```

```
One root is 4.0 and the other is 1.0
```

Tuples for output

Note that the multiple output values are delivered as a *tuple*; a parenthesized list of values separated by commas, as seen in Unit 2A. (Aside: actually you can omit the parentheses: a tuple can be specified by giving a comma-separated list of values -- but there must always be a comma after the first element, even if it is the only one!)

You can think of the collection of input arguments as a tuple too. The input to and output from functions is probably the main place that we will use tuples in this course.

The ability to return the collection of output values from a function in a single tuple can be convenient if you wish to store that collection for later use, or when using the multiple values provided by one function as input to another.

Here I illustrate the strategy of dividing a task into several smaller, simpler pieces, each of which can then be reused independently. Step 1 is to compute the roots; step 2 is print them, sorted into increasing order.

In [212]:

```
def printtworootsinorder(roots):
    '''
    Print two numbers in increasing order.
    Warning: this asumes real input values, and is not very styiah whenthe two n
umbers are equal.
    '''
    if roots[0] <= roots[1]:
        # Easy; they are already in order
        print('The roots are %g and %g' % roots)
    else:
        # swap them:
        print('The roots are %g and %g' % (roots[1], roots[0]))
```

In [213]:

```
bothroots = solve_quadratic(2, -10, 8)
print('The unsorted tuple of roots is', bothroots)
printtworootsinorder(bothroots)
```

```
The unsorted tuple of roots is (4.0, 1.0)
The roots are 1 and 4
```

We can compose functions, and so do these two steps in one:

In [214]:

```
printtworootsinorder(solve_quadratic(1, 5, -6))
```

```
The roots are -6 and 1
```

Keyword arguments: specifying input values by name

Sometimes a function has multiple input arguments, and it might be hard to keep track of what each means, so that specifying them in order makes for less than easily understandable code.

Python has a nice optional feature of specifying input arguments *by name*; they are then called *keyword arguments*.

For example:

In [215]:

```
moreroots = solve_quadratic(c_2=1, c_1=3, c_0=-10)
printtworootsinorder(moreroots)
```

```
The roots are -5 and 2
```

When you are specifying the parameters by name, there is no need to have them in any particular order. For example, if you like to write polynomials "from the bottom up", as with $-10 + 3x + x^2$, which is $c_0 + c_1 x + c_2 x^2$, you could do this:

In [216]:

```
sameroots = solve_quadratic(c_0=-10, c_1=3, c_2=1)
printtworootsinorder(sameroots)
```

```
The roots are -5 and 2
```

Functions as input to other functions

In mathematical computing, we often wish to define a (Python) function that does something with a (mathematical) function. A simple example is implementing the basic difference quotient approximation of the derivative

$$f'(x) = Df(x) \approx \frac{f(x+h) - f(x)}{h}$$

with a function `Df_approximation`, whose input will include the function f as well as the two numbers x and h.

Python makes this fairly easy, since Python functions, like numbers, can be the values of variables, and given as input to other funcitons in the same way: in fact the statement

```
def f(...): ...
```

creates variable `f` with this function as its value.

So here is a suitable definition:

In [217]:

```
def Df_approximation(f, x, h):
    return (f(x+h) - f(x))/h
```

and one way to use it is as follows:

In [218]:

```
def p(x):
    return 2*x**2 - 10*x + 8
x0 = 1
h = 1e-8
Df_x0_h = Df_approximation(p, x0, h)
print('Df(%g) is approximately %g' % (x0, Df_x0_h))
```

```
Df(1) is approximately -6
```

Optional input arguments and default values

Sometimes it makes sense for a funciot to have default values for arguments, so that not al argument valeus need to be specified. For example, we will see in Math 245 that the above value $h = 10^{-8}$ is in some sense close to "ideal", so let us make it the default, by giving it a "suggested" value as part of the functions's (new, improved) definition:

In [219]:

```
def Df_approximation(f, x, h=1e-8):
    return (f(x+h) - f(x))/h
```

The value for input argument `h` can now optionally be omitted when the function is used, getting the same result as before:

In [220]:

```
Df_x0_h = Df_approximation(p, x0)
print('Df(%g) is approximately %g' % (x0, Df_x0_h))
```

```
Df(1) is approximately -6
```

or we can specify a value when we want to use a different one:

In [221]:

```
big_h = 0.01
Df_x0_h = Df_approximation(p, x0, big_h)
print('Using h=%g, Df(%g) is approximately %g' % (big_h, x0, Df_x0_h))
```

```
Using h=0.01, Df(1) is approximately -5.98
```

Arguments with default values come after all others in the `def`

When default values are given for some arguments but not all, these must appear in the function definition after all the arguments without default values, as is the case with `h=1e-8` above.

Optional topic: anonymous functions, a.k.a. lambda functions

Note: *Several student have asked about "lambda functions", so I am adding this brief introduction. However, this topic is not needed for this course; it is only a convenience, and if you are new to computer programming, I suggest that you skip this section for now.*

One inconvenience in the above example with `Df_approximation` is that we had to first put the values of each input argument into three variables. Sometimes we would rather skip that step, and indeed we have seen that we could put the numerical argument values in directly:

In [222]:

```
Df_x_h = Df_approximation(p, 4, 1e-4)
print('The derivative is approximately %g' % Df_x_h)
```

```
The derivative is approximately 6.0002
```

However, we still needed to define the function first, and give it a name, `p`.

If the function is only ever used this one time, we can avoid this, specifying the function directly as an input argument value to the function `Df_approximation`, without first naming it. This is done with what is called an *anonymous function*, or for mysterious historical reasons, a *lambda function*.

For the example above, we can do this:

In [223]:

```
Df_x_h = Df_approximation(lambda x: 2*x**2 - 10*x + 8, 4, 1e-4)
print('The derivative is approximately %g' % Df_x_h)
```

```
The derivative is approximately 6.0002
```

We can even do it all in a single line by composing two functions, `print` and `Df_approximation`:

In [224]:

```
print('The derivative is approximately %g' % Df_approximation(lambda x: 2*x**2 -
10*x + 8, 4, 1e-4))
```

```
The derivative is approximately 6.0002
```

Here, the expression

```
lambda x: 2*x**2 - 10*x + 8
```

creates a function that is mathematically the same as function `p` above; it just does not give it a name.

In general the form is a single-line expression with four elements:

- It starts with `lambda`
- next is a list of input argument names, separated by commas if there are more than one (but no parentheses!?)
- then a colon
- and finally, a formula involving the input variables.

We can if we want assign a lambda function to a variable, so we could have defined `p` as

In [225]:

```
p = lambda x: 2*x**2 - 10*x + 8
```

though I am not sure if that has any advantage over doing this with `def`.

As an example of that, and also of having a lambda function return multilpe values, here is yet another quadratic equation solver.

In [226]:

```
solve_quadratic = lambda a, b, c: ( (-b + sqrt(b**2 - 4*a*c))/(2 * a), (-b - sqr
t(b**2 - 4*a*c))/(2 * a) )
```

In [227]:

```
print('The roots of 2*x**2 - 10*x + 8 are', solve_quadratic(2, -10, 8))
```

```
The roots of 2*x**2 - 10*x + 8 are (4.0, 1.0)
```

Anonymous functions have most of the fancy features of functions created with `def`, with the big exception that they must be defined on a single line. For example, they also allow the use of keyword arguments, allowing the input argument values to be specified by keyword in any order. It is also possible to give default values to some arguments at the end of the argument list.

To show off a few of these refinements:

In [228]:

```
printtworootsinorder(solve_quadratic(b=-10, a=2, c=8))
```

```
The roots are 1 and 4
```

Exercise

1. Refine the above function for solving quadratics, as follows:
   A. Its input arguments are three real (floating point) numbers, giving the coefficients a, b, and c of the quadratic equation $ax^2 + bx + c = 0$.
   B. It always **returns** a pair of numerical values for the roots for any "genuine" quadratic equation, meaning one with $a \neq 0$. That is, these the function gives this pair as its value, in a tuple specified by a `return` statement -- no print statement in the function.
   C. In the "prohibited" case $a = 0$, have it produce a custom error message, by "raising a ValueError exception". We will learn more about handling "exceptional" situations later, but for now just use the command:

      ```
      raise ZeroDivisionError("The coefficient 'a' of x^2 cannot h
      ave the value zero.")
      ```

2. Put this function definition into a Python "script" file, which then tests and demonstrates it with a list of cases similar to the ones discussed last week:
   A. $2x^2 - 10x + 8 = 0$
   B. $x^2 - 2x + 1 = 0$
   C. $x^2 + 2 = 0$
   D. $x^2 + 6x + 25 = 0$
   E. $4x - 10 = 0$ Do that forbidden case last: you will see why! (Later, we will learn how to clean up the ugly output, and how to have a Python file handle and continue after problems like a ZeroDivisionError.)
3. As practice with `for` loops, run this series of test cases using a `for` loop, where each value of the loop variable is a tuple of the a, b,and c values.