CS 2316 Data Manipulation for Engineers

Functions

Christopher Simpkins
chris.simpkins@gatech.edu

Functions

A function is a reusable block of code. Functions

- have names (usually),
- contain a sequence of statements, and
- can optionally return values.

We've already used several built-in functions. Today we will learn how to define our own.

Hello, Functions!

We define a function using the `def` keyword:

```
>>> def say_hello():
...     print('Hello')
...
```

(blank line tells Python shell you're finished defining the function)
Once the function is defined, you can *call* it:

```
>>> say_hello()
Hello
```

Defining Functions

The general form of a function definition is

`def` *function_name(parameter_list)*:
    function_body

- The first line is called the header.
- *function_name* is the name you use to call the function.
- *parameter_list* is a list of parameters to the function, which may be empty.
- *function_body* is a sequence of statements.

Our `say_hello()` function has an empty parameter list and a body of only one statement:

```
>>> def say_hello():
...     print('Hello')
...
```

Function Parameters

Provide a list of parameter names inside the parentheses of the function header, which creates local variables in the function.

```
>>> def say_hello(greeting):
...     print(greeting)
...
```

Then call the function by passing an *argument* to the function: a value that is bound to the parameter name. Here we pass the value `'Hello'`, which is bound to `say_hello`'s parameter `greeting` and printed to the console by the code inside `say_hello`.

```
>>> say_hello('Hello')
Hello
```

Here we pass the value 'Guten Tag!':

```
>>> say_hello('Guten Tag!')
Guten Tag!
```

Local Variable and Global Variables

Parameters are *local variables*. They are not visible outside the function:

```
>>> greeting
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'greeting' is not defined
```

Global variables are visible outside the function and inside the function.

```
>>> global_hello = 'Bonjour'
>>> global_hello
'Bonjour'
>>> def say_global_hello():
...     print(global_hello)
...
>>> say_global_hello()
Bonjour
```

Redefining and Shadowing

A function a kind of variable. If you define a function with the same name as a variable, it re-binds the name, and vice-versa.

```
>>> global_hello = 'Bonjour'
>>> def global_hello():
...     print('This is the global_hello() function.')
...
>>> global_hello
<function global_hello at 0x10063b620>
```

A function parameter with the same name as a global variable shadows the global variable.

```
>>> greeting = 'Hi ya!'
>>> def greet(greeting):
...     print(greeting)
...
>>> greeting
'Hi ya!'
>>> greet('Hello')
Hello
```

Multiple Parameters

A function can take any number of paramters.

```
>>> def greet(name, greeting):
...     print(greeting + ', ' + name)
...
>>> greet('Professor Falken', 'Greetings')
Greetings, Professor Falken
```

Parameters can be of multiple types.

```
>>> def greet(name, greeting, number):
...     print(greeting * number + ', ' + name)
...
>>> greet('Professor Falken', 'Greetings', 2)
GreetingsGreetings, Professor Falken
```

Positional Arguments vs. Keyword Arguments

Thus far we've called functions using positional arguments, meaning that argument values are bound to parameters in the order in which they appear in the call.

```
>>> def greet(name, greeting, number):
...     print(greeting * number + ', ' + name)
...
>>> greet('Professor Falken', 'Greetings', 2)
```

We can also call functions with *keyword arguments* in any order.

```
>>> greet(greeting='Hello', number=2, name='Dolly')
HelloHello, Dolly
```

If you call a function with both positional and keyword arguments, the positional ones must come first.

Default Parameter Values

You can specify default parameter values so that you don't have to provide an argument.

```
>>> def greet(name, greeting='Hello'):
...     print(greeting + ', ' + name)
...
>>> greet('Elmo')
Hello, Elmo
```

If provide an argument for a paramter with a default value, the parameter takes the value passed in the call instead of the default value.

```
>>> greet('Elmo', 'Hi')
Hi, Elmo
```

Variable Arguments

You can collect a variable number of positional arguments as a tuple by preprending a parameter name with *

```
>>> def echo(*args):
...     print(args)
...
>>> echo(1, 'fish', 2, 'fish')
(1, 'fish', 2, 'fish')
```

You can collect variable keyword arguements as a dictionary with **

```
>>> def print_dict(**kwargs):
...     print(kwargs)
...
>>> print_dict(a=1, steak='sauce')
{'a': 1, 'steak': 'sauce'}
```

Return Values

Functions can return values.

```
>>> def average(nums):
...     return sum(nums) / len(nums)
...
>>> average([100, 90, 80])
90.0
```

Functions that return values are expressions like any other.

```
>>> grades_line = ['Chris', 100, 90, 80]
>>> grades = {}
>>> grades[grades_line[0]] = average(grades_line[1:])
>>> grades
{'Chris': 90.0}
```

Representing Computation Strategies

Say we want to compute grades using one of two grade-inclusion strategies. One strategy is to drop the lowest grade:

```
>>> grades = [100, 90, 0, 80]
>>> def drop_lowest(grades):
...     return sorted(grades)[1:]
```

The other is to replace the lowest grade with the second-lowest grade:

```
>>> def replace_lowest(grades):
...     result = grades.copy()
...     lowest_two = sorted([(k, i) for i, k in enumerate(grades)])[:2]
...     lowest_index = lowest_two[0][1]
...     second_lowest_index = lowest_two[1][1]
...     result[lowest_index] = result[second_lowest_index]
...     return result
...
>>> replace_lowest(grades)
[100, 90, 80, 80]
>>> grades
[100, 90, 0, 80]
```

Note: these functions return new lists, leaving argument unchanged.

Interlude: enumerate and sorted

Notice that in the list comprehension over the iterator returned by enumerate(grades) we reversed the key and index values:

```
>>> def replace_lowest(grades):
...     result = grades.copy()
...     lowest_two = sorted([(k, i) for i, k in enumerate(grades)])[:2]
...     lowest_index = lowest_two[0][1]
...     second_lowest_index = lowest_two[1][1]
...     result[lowest_index] = result[second_lowest_index]
...     return result
...
```

Why did we do that?

Higher-Order Functions

With the two strategies represented as functions we can write another function that calculates a grade after applying one of the strategies:

```
>>> def calc_grade(grades, strategy):
...     grades_to_use = strategy(grades)
...     return sum(grades_to_use) / len(grades_to_use)
...
>>> sum(grades_to_use) / len(grades_to_use)
67.5
>>> calc_grade(grades, replace_lowest)
87.5
>>> calc_grade(grades, drop_lowest)
90.0
```

- `calc_grade` is a *higher-order function* because it takes a function as a parameter.
- Notice that we refer to a function by name when we pass it as an arument or assign it to a variable, and call it by appending `()` after the function name.

Lambda Functions

Remember when we sorted a list of (grade, index) tuples:

```
sorted([(k, i) for i, k in enumerate(grades)])
```

We reversed the key and index so that `sorted` would sort by grades instead of indexes. Another way to do that is to pass a key function to the `sorted` function. The key function is applied to each element of the sequence before it is sorted. Here we pass an `anonymous function`, or a *lambda function*:

```
>>> sorted([e for e in enumerate(grades)], key=lambda e: e[1])
[(2, 0), (3, 80), (1, 90), (0, 100)]
```

The general form of a `lambda` function is

`lambda <parameters>:  <expression>`

`lambda` function bodies are limited to one expression.

```
>>> (lambda x: x + 1)(1)
2
>>> add_one = lambda x: x + 1
>>> add_one(1)
2
```

Inner Functions

Information hiding is a general principle of software engineering. If you only need a function in one place, inside another function, you can declare it inside that function so that it is visible only in that function.

```
>>> def factorial(n):
...     def fac_iter(n, accum):
...         if n <= 1:
...             return accum
...         return fac_iter(n - 1, n * accum)
...     return fac_iter(n, 1)
...
>>> factorial(5)
120
```

`fac_iter()` is a (tail) recursive function. Recursion is important for computer scientists, but a practically-oriented Python-programming engineer will mostly use iteration, higher-order functions and loops, which are more Pythonic[1]. Any recursive computation can be formulated as an imperative computation.

[1] http://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html

Closing Thoughts

- Functions provide procedural abstraction
- Functions provide code reuse
- Functions are an essential element of any non-trivial program

We've introduced functions today, but there's more to learn.