Functions

Functions

A function is a reusable block of code. Functions

- have names (usually),
- contain a sequence of statements, and
- return values, either explicitly or implicitly.

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

Georgia Tech

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 expressions and statements.

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 arguments to the function: values that are bound to parameter names.
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!
```

Georgia Tech

Variable Scope

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
```

Georgia Tech

Muliple Parameters

A function can take any number of parameters.

```
>>> 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
```

Georgia Tech

Positional and 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 you provide an argument for a parameter with a default value, the parameter takes the argument value passed in the call instead of the default value.

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

Georgia Tech

Variable Argument Lists

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'}
```

Georgia Tech

Return Values

Functions return values.

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

If you don't explicitly return a value, None is returned implicitly.

```
>>> def g():
...     print("man")
...
>>> fbi = g()
man
>>> fbi
>>> type(fbi)
<class 'NoneType'>
```

Functions are expressions like any other.

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

Georgia Tech

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. Any recursive computation can be formulated as an imperative computation.