**What Good are Functions?**

You might have considered the situation where you would like to reuse a piece of code, just with a few different values. Instead of rewriting the whole code, it's much cleaner to define a *function*, which can then be used repeatedly.

**Function Junction**

Functions are defined with three components:

1. The *header*, which includes the defkeyword, the name of the function, and any *parameters* the function requires. Here's an example:

def hello\_world(): # There are no parameters

1. An optional *comment* that explains what the function does.

"""Prints 'Hello World!' to the console."""

1. The *body*, which describes the procedures the function carries out. The body is *indented*, just like conditional statements.

print "Hello World!"

Here's the full function pieced together:

def hello\_world(): """Prints 'Hello World!' to the console.""" print "Hello World!"

# Call and Response

After defining a function, it must be calledto be implemented. In the previous exercise, spam() in the last line told the program to look for the function called spam and execute the code inside it.

**Parameters and Arguments**

Let's take another look at the definition of the function square from the previous exercise:

def square(n):

Here, n is a *parameter* of square. A parameter is a variable that is an input to a function. It says, "Later, when square is used, you'll be able to input any value you want, but for now we'll call that future value n." A function can have any number of parameters.

The values of the parameters passed into a function are known as the *arguments*. Recall in the previous example, we called:

square(10)

Here, the function square was called with the parameter n set to the argument 10.

Typically, when you call a function, you should pass in the same number of arguments as there are parameters.

To summarize:

* When defining a function, placeholder variables are called parameters.
* When using, or calling, a function, inputs into the function are called arguments.

# Functions Calling Functions

* We've seen functions that can print text or do simple arithmetic, but functions can be much more powerful than that. For example, a function can call another function:
* def fun\_one(n): return n \* 5 def fun\_two(m): return fun\_one(m) + 7

# Practice Makes Perfect

Let's create a few more functions just for good measure.

def shout(phrase): if phrase == phrase.upper(): return "YOU'RE SHOUTING!" else: return "Can you speak up?" shout("I'M INTERESTED IN SHOUTING")

The example above is just there to help you remember how functions are structured.

Don't forget the colon at the end of your function definition!

# I Know Kung Fu

Remember import this from the first exercise in this course? That was an example of importing a module. A module is a file that contains definitions—including variables and functions—that you can use once it is imported.

# Generic Imports

Did you see that? Python said: NameError: name 'sqrt' is not defined. Python doesn't know what square roots are—yet.

There is a Python module named maththat includes a number of useful variables and functions, and sqrt() is one of those functions. In order to access math, all you need is the import keyword. When you simply import a module this way, it's called a **generic import**.

# Function Imports

Nice work! Now Python knows how to take the square root of a number.

However, we only really needed the sqrtfunction, and it can be frustrating to have to keep typing math.sqrt().

It's possible to import only certain variables or functions from a given module. Pulling in just a single function from a module is called a function import, and it's done with the from keyword:

from module import function

Now you can just type sqrt() to get the square root of a number—no more math.sqrt()!

 (You don't need the () after sqrt in the from math import sqrt bit.)

# Universal Imports

Great! We've found a way to handpick the variables and functions we want from modules.

What if we still want all of the variables and functions in a module but don't want to have to constantly type math.?

Universal import can handle this for you. The syntax for this is:

from module import \*

# Here Be Dragons

Universal imports may look great on the surface, but they're not a good idea for one very important reason: they fill your program with a ton of variable and function names without the safety of those names still being associated with the module(s) they came from.

If you have a function of your very own named sqrt and you import math, your function is safe: there is your sqrt and there is math.sqrt. If you do from math import \*, however, you have a problem: namely, two different functions with the exact same name.

Even if your own definitions don't directly conflict with names from imported modules, if you import \* from several modules at once, you won't be able to figure out which variable or function came from where.

For these reasons, it's best to stick with either import module and type module.name or just import specific variables and functions from various modules as needed.

|  |
| --- |
| import math # Imports the math module  everything = dir(math) # Sets everything to a list of things from math  print everything # Prints 'em all! |

|  |
| --- |
| ['\_\_doc\_\_', '\_\_name\_\_', '\_\_package\_\_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc'] |

# On Beyond Strings

Now that you understand what functions are and how to import modules, let's look at some of the functions that are built in to Python (no modules required!).

You already know about some of the built-in functions we've used with strings, such as .upper(), .lower(), str(), and len(). These are great for doing work with strings, but what about something a little more analytic?

|  |
| --- |
| def biggest\_number(\*args):  print max(args)  return max(args)    def smallest\_number(\*args):  print min(args)  return min(args)  def distance\_from\_zero(arg):  print abs(arg)  return abs(arg)  biggest\_number(-10, -5, 5, 10)  smallest\_number(-10, -5, 5, 10)  distance\_from\_zero(-10) |

|  |
| --- |
| 10  -10  10 |

# max()

The max() function takes any number of arguments and returns the largest one. ("Largest" can have odd definitions here, so it's best to use max() on integers and floats, where the results are straightforward, and not on other objects, like strings.)

For example, max(1,2,3) will return 3(the largest number in the set of arguments).

# min()

min() then returns the smallest of a given series of arguments.

# abs()

The abs() function returns the **absolute value** of the number it takes as an argument—that is, that number's distance from 0 on an imagined number line. For instance, 3 and -3 both have the same absolute value: 3. The abs() function always returns a positive value, and unlike max() and min(), it only takes a single number.

# type()

Finally, the type() function returns the type of the data it receives as an argument. If you ask Python to do the following:

print type(42) print type(4.2) print type('spam')

Python will output:

<type 'int'> <type 'float'> <type 'str'>