You've already seen and used functions such as `print` and `abs`. But Python has many more functions, and defining your own functions is a big part of python programming.

In this lesson, you will learn more about using and defining functions.

Getting Help

You saw the `abs` function in the previous tutorial, but what if you've forgotten what it does?

The `help()` function is possibly the most important Python function you can learn. If you can remember how to use `help()`, you hold the key to understanding most other functions.

Here is an example:

```
In [1]:
help(round)
```

`help()` displays two things:

1. the header of that function `round(number, ndigits=None)`. In this case, this tells us that `round()` takes an argument we can describe as `number`. Additionally, we can optionally give a separate argument which could be described as `ndigits`.
2. A brief English description of what the function does.

Common pitfall: when you're looking up a function, remember to pass in the name of the function itself, and not the result of calling that function.

What happens if we invoke help on a *call* to the function `round()`? Unhide the output of the cell below to see.

```
In [2]:
help(round(-2.01))
```

Python evaluates an expression like this from the inside out. First it calculates the value of `round(-2.01)`, then it provides help on the output of that expression.

(And it turns out to have a lot to say about integers! After we talk later about objects, methods, and attributes in Python, the help output above will make more sense.)

`round` is a very simple function with a short docstring. `help` shines even more when dealing with more complex, configurable functions like `print`. Don't worry if the following output looks inscrutable... for now, just see if you can pick anything new out from this help.

```
In [3]:
help(print)
```

If you were looking for it, you might learn that print can take an argument called `sep`, and that this describes what we put between all the other arguments when we print them.

Defining functions

Builtin functions are great, but we can only get so far with them before we need to start defining our own functions. Below is a simple example.

```
In [4]:
def least_difference(a, b, c):
    diff1 = abs(a - b)
    diff2 = abs(b - c)
    diff3 = abs(a - c)
    return min(diff1, diff2, diff3)
```

This creates a function called `least_difference`, which takes three arguments, `a`, `b`, and `c`.

Functions start with a header introduced by the `def` keyword. The indented block of code following the `:` is run when the function is called.

`return` is another keyword uniquely associated with functions. When Python encounters a `return` statement, it exits the function immediately, and passes the value on the right hand side to the calling context.

Is it clear what `least_difference()` does from the source code? If we're not sure, we can always try it out on a few examples:

```
In [5]:
print(
    least_difference(1, 10, 100),
    least_difference(1, 10, 10),
    least_difference(5, 6, 7), # Python allows trailing commas in argument lists. How
nice is that?
)
```

Or maybe the `help()` function can tell us something about it.

```
In [6]:
help(least_difference)
```

Python isn't smart enough to read my code and turn it into a nice English description. However, when I write a function, I can provide a description in what's called the **docstring**.

Docstrings

```
In [7]:
def least_difference(a, b, c):
    """Return the smallest difference between any two numbers
    among a, b and c.

    >>> least_difference(1, 5, -5)
    4
    """
    diff1 = abs(a - b)
    diff2 = abs(b - c)
    diff3 = abs(a - c)
    return min(diff1, diff2, diff3)
```

The docstring is a triple-quoted string (which may span multiple lines) that comes immediately after the header of a function. When we call `help()` on a function, it shows the docstring.

```
In [8]:
help(least_difference)
```

Aside: The last two lines of the docstring are an example function call and result. (The `>>>` is a reference to the command prompt used in Python interactive shells.) Python doesn't run the example call - it's just there for the benefit of the reader. The convention of including 1 or more example calls in a function's docstring is far from universally observed, but it can be very effective at helping someone understand your function. For a real-world example, see this docstring for the numpy function `np.eye`.

Good programmers use docstrings unless they expect to throw away the code soon after it's used (which is rare). So, you should start writing docstrings, too!

Functions that don't return

What would happen if we didn't include the `return` keyword in our function?

```
In [9]:
def least_difference(a, b, c):
    """Return the smallest difference between any two numbers
    among a, b and c.
    """
    diff1 = abs(a - b)
    diff2 = abs(b - c)
    diff3 = abs(a - c)
    min(diff1, diff2, diff3)

print(
    least_difference(1, 10, 100),
    least_difference(1, 10, 10),
    least_difference(5, 6, 7),
)
```

Python allows us to define such functions. The result of calling them is the special value `None`. (This is similar to the concept of "null" in other languages.)

Without a `return` statement, `least_difference` is completely pointless, but a function with side effects may do something useful without returning anything. We've already seen two examples of this: `print()` and `help()` don't return anything. We only call them for their side effects (putting some text on the screen). Other examples of useful side effects include writing to a file, or modifying an input.

```
In [10]:
mystery = print()
print(mystery)
```

Default arguments

When we called `help(print)`, we saw that the `print` function has several optional arguments. For example, we can specify a value for `sep` to put some special string in between our printed arguments:

```
In [11]:
print(1, 2, 3, sep=' < ')
```

But if we don't specify a value, `sep` is treated as having a default value of `' '` (a single space).

```
In [12]:
print(1, 2, 3)
```

Adding optional arguments with default values to the functions we define turns out to be pretty easy:

```
In [13]:
def greet(who="Colin"):
    print("Hello,", who)

greet()
greet(who="Kaggle")
# (In this case, we don't need to specify the name of the argument, because it's unam
biguous.)
greet("world")
```

Functions Applied to Functions

Here's something that's powerful, though it can feel very abstract at first. You can supply functions as arguments to other functions. Some example may make this clearer:

```
In [14]:
def mult_by_five(x):
    return 5 * x

def call(fn, arg):
    """Call fn on arg"""
    return fn(arg)

def squared_call(fn, arg):
    """Call fn on the result of calling fn on arg"""
    return fn(fn(arg))

print(
    call(mult_by_five, 1),
    squared_call(mult_by_five, 1),
    sep='\n', # '\n' is the newline character - it starts a new line
)
```

Functions that operate on other functions are called "higher-order functions." You probably won't write your own for a little while. But there are higher-order functions built into Python that you might find useful to call.

Here's an interesting example using the `max` function.

By default, `max` returns the largest of its arguments. But if we pass in a function using the optional `key` argument, it returns the argument `x` that maximizes `key(x)` (aka the 'argmax').

```
In [15]:
def mod_5(x):
    """Return the remainder of x after dividing by 5"""
    return x % 5

print(
    'Which number is biggest?',
    max(100, 51, 14),
    'Which number is the biggest modulo 5?',
    max(100, 51, 14, key=mod_5),
    sep='\n',
)
```

Your Turn

Functions open up a whole new world in Python programming. **Try using them yourself**.
