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 on built-in function round in module builtins:

round(number, ndigits=None)
    Round a number to a given precision in decimal digits.

    The return value is an integer if ndigits is omitted or None.  Otherwise
    the return value has the same type as the number.  ndigits may be negati
ve.
```

`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))
```

Loading [MathJax]/extensions/Safe.js

```
Help on int object:

class int(object)
 |  int([x]) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer litera
l.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Built-in subclasses:
 |      bool
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __and__(self, value, /)
 |      Return self&value.
 |
 |  __bool__(self, /)
 |      True if self else False
 |
 |  __ceil__(...)
 |      Ceiling of an Integral returns itself.
 |
 |  __divmod__(self, value, /)
 |      Return divmod(self, value).
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __float__(self, /)
 |      float(self)
 |
 |  __floor__(...)
 |      Flooring an Integral returns itself.
 |
 |  __floordiv__(self, value, /)
 |      Return self//value.
 |
 |  __format__(self, format_spec, /)
 |      Default object formatter.
```

Loading [MathJax]/extensions/Safe.js

```
|
|   __ge__(self, value, /)
|       Return self>=value.
|
|   __getattribute__(self, name, /)
|       Return getattr(self, name).
|
|   __getnewargs__(self, /)
|
|   __gt__(self, value, /)
|       Return self>value.
|
|   __hash__(self, /)
|       Return hash(self).
|
|   __index__(self, /)
|       Return self converted to an integer, if self is suitable for use as
an index into a list.
|
|   __int__(self, /)
|       int(self)
|
|   __invert__(self, /)
|       ~self
|
|   __le__(self, value, /)
|       Return self<=value.
|
|   __lshift__(self, value, /)
|       Return self<<value.
|
|   __lt__(self, value, /)
|       Return self<value.
|
|   __mod__(self, value, /)
|       Return self%value.
|
|   __mul__(self, value, /)
|       Return self*value.
|
|   __ne__(self, value, /)
|       Return self!=value.
|
|   __neg__(self, /)
|       -self
|
|   __or__(self, value, /)
|       Return self|value.
|
|   __pos__(self, /)
|       +self
|
|   __pow__(self, value, mod=None, /)
|       Return pow(self, value, mod).
|
|   __radd__(self, value, /)
```

```
 |          Return value+self.
 |
 |  __rand__(self, value, /)
 |          Return value&self.
 |
 |  __rdivmod__(self, value, /)
 |          Return divmod(value, self).
 |
 |  __repr__(self, /)
 |          Return repr(self).
 |
 |  __rfloordiv__(self, value, /)
 |          Return value//self.
 |
 |  __rlshift__(self, value, /)
 |          Return value<<self.
 |
 |  __rmod__(self, value, /)
 |          Return value%self.
 |
 |  __rmul__(self, value, /)
 |          Return value*self.
 |
 |  __ror__(self, value, /)
 |          Return value|self.
 |
 |  __round__(...)
 |          Rounding an Integral returns itself.
 |
 |          Rounding with an ndigits argument also returns an integer.
 |
 |  __rpow__(self, value, mod=None, /)
 |          Return pow(value, self, mod).
 |
 |  __rrshift__(self, value, /)
 |          Return value>>self.
 |
 |  __rshift__(self, value, /)
 |          Return self>>value.
 |
 |  __rsub__(self, value, /)
 |          Return value-self.
 |
 |  __rtruediv__(self, value, /)
 |          Return value/self.
 |
 |  __rxor__(self, value, /)
 |          Return value^self.
 |
 |  __sizeof__(self, /)
 |          Returns size in memory, in bytes.
 |
 |  __sub__(self, value, /)
 |          Return self-value.
 |
 |  __truediv__(self, value, /)
```

Loading [MathJax]/extensions/Safe.js

```
 |          Return self/value.
 |
 |  __trunc__(...)
 |          Truncating an Integral returns itself.
 |
 |  __xor__(self, value, /)
 |          Return self^value.
 |
 |  as_integer_ratio(self, /)
 |          Return integer ratio.
 |
 |          Return a pair of integers, whose ratio is exactly equal to the origi
nal int
 |          and with a positive denominator.
 |
 |          >>> (10).as_integer_ratio()
 |          (10, 1)
 |          >>> (-10).as_integer_ratio()
 |          (-10, 1)
 |          >>> (0).as_integer_ratio()
 |          (0, 1)
 |
 |  bit_count(self, /)
 |          Number of ones in the binary representation of the absolute value of
self.
 |
 |          Also known as the population count.
 |
 |          >>> bin(13)
 |          '0b1101'
 |          >>> (13).bit_count()
 |          3
 |
 |  bit_length(self, /)
 |          Number of bits necessary to represent self in binary.
 |
 |          >>> bin(37)
 |          '0b100101'
 |          >>> (37).bit_length()
 |          6
 |
 |  conjugate(...)
 |          Returns self, the complex conjugate of any int.
 |
 |  to_bytes(self, /, length=1, byteorder='big', *, signed=False)
 |          Return an array of bytes representing an integer.
 |
 |          length
 |            Length of bytes object to use.  An OverflowError is raised if the
 |            integer is not representable with the given number of bytes.  Defa
ult
 |            is length 1.
 |          byteorder
 |            The byte order used to represent the integer.  If byteorder is 'bi
 |            the most significant byte is at the beginning of the byte array.
```

```
If
 |          byteorder is 'little', the most significant byte is at the end of
the
 |          byte array.  To request the native byte order of the host system,
use
 |          `sys.byteorder' as the byte order value.  Default is to use 'big'.
 |        signed
 |          Determines whether two's complement is used to represent the integ
er.
 |          If signed is False and a negative integer is given, an OverflowErr
or
 |          is raised.
 |
 |    ----------------------------------------------------------------------
 |    Class methods defined here:
 |
 |    from_bytes(bytes, byteorder='big', *, signed=False) from builtins.type
 |        Return the integer represented by the given array of bytes.
 |
 |        bytes
 |          Holds the array of bytes to convert.  The argument must either
 |          support the buffer protocol or be an iterable object producing byt
es.
 |          Bytes and bytearray are examples of built-in objects that support
the
 |          buffer protocol.
 |        byteorder
 |          The byte order used to represent the integer.  If byteorder is 'bi
g',
 |          the most significant byte is at the beginning of the byte array.
If
 |          byteorder is 'little', the most significant byte is at the end of
the
 |          byte array.  To request the native byte order of the host system,
use
 |          `sys.byteorder' as the byte order value.  Default is to use 'big'.
 |        signed
 |          Indicates whether two's complement is used to represent the intege
r.
 |
 |    ----------------------------------------------------------------------
 |    Static methods defined here:
 |
 |    __new__(*args, **kwargs) from builtins.type
 |        Create and return a new object.  See help(type) for accurate signatu
re.
 |
 |    ----------------------------------------------------------------------
 |    Data descriptors defined here:
 |
 |    denominator
 |        the denominator of a rational number in lowest terms
 |
 |    imag
 |        the imaginary part of a complex number
 |
```

Loading [MathJax]/extensions/Safe.js

```
|  numerator
|      the numerator of a rational number in lowest terms
|
|  real
|      the real part of a complex number
```

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)`

```
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
    flush
      whether to forcibly flush the stream.
```

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

Loading [MathJax]/extensions/Safe.js

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:

```python
In [5]: print(
            least_difference(1, 10, 100),
            least_difference(1, 10, 10),
            least_difference(5, 6, 7), # Python allows trailing commas in argument l
        )
```

```
9 0 1
```

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

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

```
Help on function least_difference in module __main__:

least_difference(a, b, c)
```

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

```python
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)
```

Loading [MathJax]/extensions/Safe.js

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)
```

```
Help on function least_difference in module __main__:

least_difference(a, b, c)
    Return the smallest difference between any two numbers
    among a, b and c.

    >>> least_difference(1, 5, -5)
    4
```

> **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?

```python
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),
)
```

```
None None None
```

Loading [MathJax]/extensions/Safe.js

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)
```

```
None
```

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=' < ')
```

```
1 < 2 < 3
```

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)
```

```
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
         greet("world")
```

```
Hello, Colin
Hello, Kaggle
Hello, 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:

Loading [MathJax]/extensions/Safe.js

```
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
          )
```

```
5
25
```

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',
          )
```

```
Which number is biggest?
100
Which number is the biggest modulo 5?
14
```

Your Turn

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

Loading [MathJax]/extensions/Safe.js

Loading [MathJax]/extensions/Safe.js