Fun Functions

What are functions?

There are two parts to a function:

1. Its name
2. Sequence of statements that will be executed when it is called

```
def print_lyrics():
  print("I'm a lumberjack, and I'm okay.")
  print("I sleep all night and I work all day.")
```

What is "wrong" with this function?

```
def print_numbers():
  print("I'm a lumberjack, and I'm okay.")
  print("I sleep all night and I work all day.")
```

Function calls

To execute a function, you must **call** it.

```
>>> type(42)
<class 'int'>
```

You call a function by naming it–`type`–and following the name with parentheses, `()`. If you put an expression in the parentheses–`(42)`, the expression is called the **argument** to the function. The result of the function–`<class 'int'>`–is referred to as its **return value**.

It is common to say that a function "takes" an argument and "returns" a result.

Python provides a set of functions to convert values from one type to another:

- `int` converts strings (if it can) and floats to integers
- `float` converts integers and strings to floating-point numbers
- `str` converts its argument to a string

Function calls: `int`

`int` can only convert values that "make sense"

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

`int` will **truncate** floating-point values:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

Function calls: `float` and `str`

`float`

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

`str`

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

Math functions

Python has a `math` **module** that contains functions for common mathematical computations. A module is a file that contains a set of related functions.

To use a module, import it with an **import statement**:

```
>>> import math
```

This creates a **module object** called `math`.

```
>>> math
<module 'math' (built-in)>
```

Math functions

This `math` module object *contains* the functions and variables defined in the module. You can access one of these functions or variables using **dot notation**.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)

>>> degrees = 45
>>> radians = degrees / 180.0 * math.pi
>>> math.sin(radians)
0.707106781187
```

How is accessing a *function* (`math.sin`) different from accessing a *variable* (`math.pi`)?

Composition

So far, we have looked at the elements of a program—variables, expressions, and statements—in isolation, without talking about how to *combine* them.

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them.

For example, the argument to a function can be any kind of expression, including arithmetic operations:

```
height = math.sin(0.7)
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

And even function calls:

```
x = math.exp(math.log(x+1))
```

Composition

Almost anywhere you can put a value, you can put an expression, with *one* exception:
The left side of an assignment statement has to be a variable name.

```
>>> minutes = hours * 60                # right
>>> hours * 60 = minutes                # wrong!
SyntaxError: can't assign to operator
```