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
```

Adding new functions

A **function definition** specifies the name of a new function and the sequence of statements that run when the function is called.

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

Things to notice:

- `def` is a keyword that indicates that this is a function definition.
- Functions follow the same naming rules as variables.
- The empty parentheses indicate that this function does not take any arguments.

What is the **header** of this function? What is the **body**?

Parameters and arguments

Some functions, like `math.sin`, take an argument, `math.sin(0.7)`. Some even take multiple, like `math.pow(6, 2)`.

Within the function, the arguments are assigned to variables called **parameters**.

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

Whatever you pass to `print_twice` as an argument will become the value of the variable `bruce` in the function.

```
>>> print_twice('Spam')
Spam
Spam
```

Parameters and arguments: Composition

Remember that with composition you can use any expression as the argument to a function.

```
>>> print_twice('Spam ' * 4)
Spam Spam Spam Spam
Spam Spam Spam Spam
```

```
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

Variable scope

Variables that are created in a function are **local** to the function, meaning they only exist within that function. Where a variable exists is its **scope**.

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)

cat_twice('Hello, ', 'World!')
print(cat)
```

What will `print(cat)` do?

Variable scope: Parameters

Parameters, which are also variables created in a function, are local to their functions.

```
def print_twice(bruce):
    print(bruce)
    print(bruce)

print_twice('Potato')
print(bruce)
```

What will `print(bruce)` do?

Fruitful and void functions

Some functions, like `math` functions, return results. Let's call them **fruitful functions**.

```
>>> result = math.sqrt(49)
>>> print(result)
7.0
```

Other functions, like `print_twice`, perform an action, but do not return a value. They are called **void functions**.

```
>>> result = print_twice('Potato')
Potato
Potato
>>> print(result)
None
```

This `None` is *not* `'None'`, the string. It is a special value with its own type:

```
>>> type(result)
<class 'NoneType'>
```

Fruitful functions

When looking at a function's definition, we can identify that it is fruitful if it contains a **return statement**, which defines what value will be returned.

```
def area(radius):
    a = math.pi * radius**2
    return a

circle_area = area(2)
print(circle_area)
```

The return statement says "Return from this function (i.e. go back to where you called it from) and use the value of the expression as the return value."

Fruitful functions

When you call a fruitful function in interactive mode, Python will display the result:

```
>>> area(2)
12.566370614359172
```

However, if you call a fruitful function in script mode and do not store the return value in a variable, the return value is lost forever.

```
area(2)
```

It is not very useful if you do not store or display the return value of a fruitful function.

Fruitful functions

Let's look at an example:

```
an_integer = '100'  # Call the appropriate function to change the data type of this value
print(type(an_integer))
```

`int` is a fruitful function—it returns the value of its argument as an integer. However, if we don't do anything with the return value of `int`, it is not useful.

```
an_integer = '100'  # Call the appropriate function to change the data type of this value
int(an_integer)
print(type(an_integer)) # Still a string
```

So, what do we do?

Why functions?

Grouping statements and giving them a name makes your program easier to read and debug.

```
circumference = 9
diameter = circumference / math.pi
radius = diameter / 2
area = math.pi * radius**2
print(f'Area of circle with circumference {circumference} is {area}')
```

```
def area(circumference):
  diameter = circumference / math.pi
  radius = diameter / 2
  return math.pi * radius**2

print(f'Area of circle with circumference {circumference} is {area(9)}')
```

Why functions?

Functions help eliminate repetitive code. Later, if you need to make a change, you only need to make it in one place.

```
circumference = 9
diameter = circumference / math.pi
radius = diameter / 2
area = math.pi * radius**2
print(f'Area of circle with circumference {circumference} is {area}')
circumference = 5
diameter = circumference / math.pi
radius = diameter / 2
area = math.pi * radius**2
print(f'Area of circle with circumference {circumference} is {area}')
```

```
def area(circumference):
  diameter = circumference / math.pi
  radius = diameter / 2
  return math.pi * radius**2

print(f'Area of circle with circumference {circumference} is {area(9)}')
print(f'Area of circle with circumference {circumference} is {area(5)}')
```

Why functions?

Dividing a long program into functions allows you to debug parts one at a time.

```
def area(circumference):
  diameter = circumference / math.pi
  radius = diameter / 2
  return math.pi * radius**2

print(area(9))
print(area(5))
```

```
def area(circumference):
  diameter = circumference / math.pi
  radius = diameter / 2
  return math.pi * radius**2

print(f'Area of circle with circumference {circumference} is {area(9)}')
print(f'Area of circle with circumference {circumference} is {area(5)}')
```

Why functions?

Well-designed functions can be reused across programs.

```
circumference = 5
diameter = circumference / math.pi
radius = diameter / 2
area = math.pi * radius**2
```

```
def area(circumference):
  diameter = circumference / math.pi
  radius = diameter / 2
  return math.pi * radius**2
```

Flow of execution

A program's **flow of execution** is the order in which statements in the program are executed.

- Execution always begin at the first statement in the program
- Function definitions do not change the flow; but, their statements are not executed until the function is called
- When a function is called, the flow jumps to the body of the function, executes the statements in the funtion, then jumps back to pick up where it left off

Flow of execution

```
a = 2
b = 7
c = 13


def mean(a, b, c):
    return (a + b + c) / 3


def print_mean(a, b, c, mean):
    print(f'The average of {a}, {b}, and {c} is {mean}.')


print(f'Today, we will print the average of {a}, {b}, and {c}!')
average = mean(a, b, c)
print_mean(a, b, c, average)
print('Thanks for using this average calculator! Goodbye!')
```