Lecture 3 - Function Basics

Week 2 Friday

Miles Chen, PhD

Adapted from Think Python by Allen B Downey

Functions

Functions calls are how functions are executed.

Function calls consist of the **name** of the function and **parenthesis** with any **arguments** inside the parenthesis.

Some functions produce a **return value**

In [1]:
```
type(42)
```

Out[1]:
```
int
```

the name is `type` , the argument is `42` , the return value is `int`

Function calls

We call functions by writing the function name and parenthesis.

In [2]:
```
print  # does not call the function
```

Out[2]:
```
<function print>
```

In [3]:
```
print('hello')  # calls the function
```

```
hello
```

In [4]:
```
print(1, 2, 3)
```

```
1 2 3
```

In [5]:
```
print(1, 2, 3, sep = '-')
```

```
1-2-3
```

Getting Help

You can view the reference by using `help(functionname)`

or `?functionname` which will call the pager

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

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

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

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

Side note about single and double quotes.

Both single and double quotes can be used to denote a string. Use double quotes if there will be an apostrophe ' . Or if you want to use single quotes with an apostrophe, the apostrophe must be escaped with a backslash \

In [8]:
```
print("I can't believe it!")
```

```
I can't believe it!
```

In [9]:
```
print('I can't believe it!')
```

```
  File "<ipython-input-9-40258c6dceef>", line 1
    print('I can't believe it!')
                  ^
SyntaxError: invalid syntax
```

In [10]:
```
print('I can\'t believe it!')
```

```
I can't believe it!
```

In [11]:
```
print('I can"t believe it!')
```

```
I can"t believe it!
```

Defining a function

To define a new function, use the statement

`def functionname(arguments):`

The function needs to use `return` to return an object

In [12]:
```
def shouting(phrase):
    shout = phrase.upper() + '!!!'
    return shout
```

In [13]:
```
shouting('hi my name is miles')
```

Out[13]:
```
'HI MY NAME IS MILES!!!'
```

In [14]:
```
shouting(5)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-14-32f05294ee9d> in <module>
----> 1 shouting(5)

<ipython-input-12-ca319fd14cc7> in shouting(phrase)
      1 def shouting(phrase):
----> 2     shout = phrase.upper() + '!!!'
      3     return shout

AttributeError: 'int' object has no attribute 'upper'
```

In [15]:
```
def shouting(phrase):
    # attempt to convert the input object to a string
    shout = str(phrase).upper() + '!!!'
    return shout
```

In [16]:
```
shouting(5)
```

Out[16]:
```
'5!!!'
```

Returning a value

If a function returns a value, the result of the function can be assigned to an object.

In [17]:
```
def shouting(phrase):
    # attempt to convert the input object to a string
    shout = str(phrase).upper() + '!!!'
    return shout
```

In [18]:
```
greeting = shouting("hi")
```

In [19]:
```
greeting
```

Out[19]:
```
'HI!!!'
```

If a function does not use `return` to return a value, the result of the function will be `None`.

In [20]:
```
def quiet(phrase):
    shh = str(phrase).lower()
    shh
```

In [21]:
```
whisper = quiet("HELLO")
```

In [22]:
```
whisper
```

In [23]:
```
print(whisper)
```

```
None
```

In [24]:
```
type(whisper)
```

Out[24]:
```
NoneType
```

Returning multiple values

A function can return multiple values as a tuple. I'll fully explain a tuple in a future lecture.

In [25]:
```
def powersof(number):
    square = number ** 2
    cube = number ** 3
    return number, square, cube
```

In [26]:
```
powersof(3)
```

Out[26]:
```
(3, 9, 27)
```

tuple unpacking

If the function returns a tuple, it can be unpacked into separate elements.

In [27]:
```
x, y, z = powersof(3)
```

In [28]:
```
print(x)
```

```
3
```

In [29]:
```
print(y)  # all of the values are stored separately
print(z)
```

```
9
27
```

Conversely, you can just capture the tuple as a single object

In [30]:
```
j = powersof(4)
```

In [31]:
```
print(j)
```

```
(4, 16, 64)
```

Python uses 0-indexing, so you can access the first element of a tuple by using square brackets with a 0 inside: `[0]` .

In [32]:
```
j[0]
```

Out[32]: 4

To perform tuple unpacking, the number of elements to be unpacked must match the number of values being assigned.

The following is not allowed because `powerof()` returns a tuple with three elements and we are trying to assign it to two names.

In [33]:
```
g, h = powersof(5)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-33-798cf3e7a2ea> in <module>
----> 1 g, h = powersof(5)

ValueError: too many values to unpack (expected 2)
```

Flow of Execution

Execution always begins at the first statement of the program. Statements are run one at a time, in order from top to bottom.

Function **definitions** do not alter the flow of execution of the program, but remember that *statements inside the function don't run until the function is called.*

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, runs the statements there, and then comes back to pick up where it left off.

Parameters and Arguments

Inside a function, the arguments of a function are assigned to variables called parameters.

In [34]:
```
# a silly function
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

The function assigns the argument to a parameter named `bruce`. When the function is called, it prints the value of the parameter (whatever it is).

In [35]:
```
print_twice("spam")
```

```
spam
spam
```

In [36]:
```
import math
print_twice(math.sin(math.pi / 2))
```

```
1.0
1.0
```

In [37]:
```
print_twice("Spam " * 2)
```

```
Spam Spam
Spam Spam
```

In [38]:
```
print_twice(print_twice("Spam"))
```

```
Spam
Spam
None
None
```

What happened above?

The inner `print_twice()` ran first. It printed "Spam" on one line and printed "Spam" again on the next line.

However, the function `print_twice()` has no return value. It returns `None`. So the outer call of `print_twice()` prints `None` two times.

Default arguments

you can also specify default arguments that will be used if they are not explicitly provided

In [39]:
```
# example without defaults
def stuff(a, b, c):
    print(a, b, c)
```

In [40]:
```
stuff(1, 2, 3)
```

```
1 2 3
```

In [41]:
```
stuff(1, 2) # if you do not provide the correct arguments, you get an error
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-41-f434fb9eb065> in <module>
----> 1 stuff(1, 2) # if you do not provide the correct arguments, you get an error

TypeError: stuff() missing 1 required positional argument: 'c'
```

In [42]:
```
# example with defaults
def junk(a=1, b=2, c=3):
    print(a, b, c)
```

In [43]:
```
junk()
```

```
1 2 3
```

In [44]:
```
junk(4) # specifying only one will put it in the first argument
```

```
4 2 3
```

In [45]:
```
junk(b = 4)
```

```
1 4 3
```

In [46]:
```
junk(5, 10, 0)
```

```
5 10 0
```

In [47]:
```
junk(5, a = 10, b = 0) # python will get confused if you name only some of the argument
s.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-47-60c03f9dcb05> in <module>
----> 1 junk(5, a = 10, b = 0) # python will get confused if you name only some of th
e arguments.

TypeError: junk() got multiple values for argument 'a'
```

In [48]:
```
junk(c=5, a=10, b = 0)
```

```
10 0 5
```

Variables and Parameters are Local

When you create a variable inside a function, it is local, which means that it only exists inside the function.

In [49]:
```
def print_twice(bruce):
    print(bruce)
    print(bruce)

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

In [50]:
```
line1 = 'Bing tiddle '
line2 = 'tiddle bang.'
cat_twice(line1, line2)
```

```
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

When cat_twice terminates, the variable cat is destroyed. If we try to print it, we get an error. Parameters are also local. For example, outside print_twice, there is no such thing as bruce.

In [51]:
```
cat
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-51-8f6abfbac8c8> in <module>
----> 1 cat

NameError: name 'cat' is not defined
```

In [52]:
```
bruce
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-52-5b060e0da5b6> in <module>
----> 1 bruce

NameError: name 'bruce' is not defined
```

Stack diagrams

To keep track of which variables can be used where, it is sometimes useful to draw a stack diagram. Like state diagrams, stack diagrams show the value of each variable, but they also show the function each variable belongs to.

Each function is represented by a frame. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example is shown in Figure 3.1 in the text.

The frames are arranged in a stack that indicates which function called which, and so on. In this example, `print_twice` was called by `cat_twice`, and `cat_twice` was called by `__main__`, which is a special name for the topmost frame. When you create a variable outside of any function, it belongs to `__main__`.

Each parameter refers to the same value as its corresponding argument. So, `part1` has the same value as `line1`, `part2` has the same value as `line2`, and `bruce` has the same value as `cat`.

If an error occurs during a function call, Python prints the name of the function, the name of the function that called it, and the name of the function that called that, all the way back to `__main__`.

For example, if you try to access `cat` from within `print_twice`, you get a `NameError`:

In [53]:
```
def print_twice(bruce):
    print(cat)
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

In [54]:
```
line1 = 'Bing tiddle '
line2 = 'tiddle bang.'
cat_twice(line1, line2)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-54-27fa4ab8fc74> in <module>
      1 line1 = 'Bing tiddle '
      2 line2 = 'tiddle bang.'
----> 3 cat_twice(line1, line2)

<ipython-input-53-f90c12bae8f1> in cat_twice(part1, part2)
      3 def cat_twice(part1, part2):
      4     cat = part1 + part2
----> 5     print_twice(cat)

<ipython-input-53-f90c12bae8f1> in print_twice(bruce)
      1 def print_twice(bruce):
----> 2     print(cat)
      3 def cat_twice(part1, part2):
      4     cat = part1 + part2
      5     print_twice(cat)

NameError: name 'cat' is not defined
```

Assignment operations only affect values inside the function and do not interact with values outside the function.

In [55]:
```
x = 5
```

In [56]:
```
x
```

Out[56]: 5

In [57]:
```
def alter_x(x):
    x = x + 1
    return x
```

In [58]:
```
alter_x(x)
```

Out[58]: 6

In [59]:
```
x
```

Out[59]: 5

Global variables'

If you want your function to alter variables outside of its own scope, you can use the keyword `global`

Be careful with this keyword.

In [60]:
```
def alter_global_x():
    global x
    x = x + 1
    return x
```

In [61]:
```
x = 5
```

In [62]:
```
alter_global_x()
```

Out[62]: 6

In [63]:
```
x
```

Out[63]: 6

If a function calls for a value that is not provided in the arguments or is not defined inside the function, the Python will search for the value in the higher scopes.

In [64]:
```
# in this function, we ask Python to print the value of x
# even though we do not define its value. Python finds x
# in the global environment

def search_for_x():
    print(x)
    return x
```

In [65]:
```
search_for_x()
```

```
6
```

Out[65]:
```
6
```