

Lecture 2-2

Functions in Python

Week 2 Wednesday

Miles Chen, PhD

Adapted from *Think Python* by Allen B. Downey and *A Whirlwind Tour of Python* by Jake VanderPlas

All Programs can reduced to the following instructions

- *input* - get input from keyboard, a file, network, or some device
- *output* - display data to the screen, save to a file, send over network, etc.
- *math* - perform a mathematical operation
- *conditional execution* - check for certain conditions and run the appropriate code
- *repetition* - perform some action repeatedly, usually with some variation

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**

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
```

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.

Function calls

We call functions by writing the function name and parenthesis.

```
In [2]: print # does not call the function. This is the object of the function itself
```

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

Function calls

We call functions by writing the function name and parenthesis.

```
In [2]: print # does not call the function. This is the object of the function itself
```

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

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

```
hello
```


Function calls

We call functions by writing the function name and parenthesis.

```
In [2]: print # does not call the function. This is the object of the function itself
```

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

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

```
hello
```

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

```
1 2 3
```

Function calls

We call functions by writing the function name and parenthesis.

```
In [2]: print # does not call the function. This is the object of the function itself
```

```
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)`

In Jupyter Lab, you can also hit **Ctrl + I** or choose "Show Contextual Help" from the Help Menu. This will open another tab in Jupyter that displays help. Like any other Jupyter tab, it can be dragged to a more convenient location for viewing.

Getting Help

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

In Jupyter Lab, you can also hit **Ctrl + I** or choose "Show Contextual Help" from the Help Menu. This will open another tab in Jupyter that displays help. Like any other Jupyter tab, it can be dragged to a more convenient location for viewing.

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 `\`

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 [7]:

```
print("I can't believe it!")
```

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

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 [7]:

```
print("I can't believe it!")
```

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

In [8]:

```
print('I can't believe it!')
```

```
File "<ipython-input-8-40258c6dceef>", line 1
```

```
    print('I can't believe it!')
```

```
    ^
```

```
SyntaxError: invalid syntax
```

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

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

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

```
File "<ipython-input-8-40258c6dceef>", line 1
    print('I can't believe it!')
```

```
      ^
SyntaxError: invalid syntax
```

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

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


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

I can't believe it!

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

```
File "<ipython-input-8-40258c6dceef>", line 1
    print('I can't believe it!')
```

```
      ^
SyntaxError: invalid syntax
```

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

I can't believe it!

```
In [10]: 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):
```

If you want the function to return an object, you must use the `return` statement.

Defining a function

To define a new function, use the statement

```
def functionname(arguments):
```

If you want the function to return an object, you must use the `return` statement.

```
In [11]: def shouting(phrase):  
         shout = phrase.upper() + '!!!'  
         return shout
```

Defining a function

To define a new function, use the statement

```
def functionname(arguments):
```

If you want the function to return an object, you must use the `return` statement.

```
In [11]: def shouting(phrase):  
         shout = phrase.upper() + '!!!'  
         return shout
```

```
In [12]: shouting('hi my name is miles')
```

```
Out[12]: 'HI MY NAME IS MILES!!!'
```

In [13]: shouting(5)

```
-----  
AttributeError                                Traceback (most recent call last)  
<ipython-input-13-32f05294ee9d> in <module>  
----> 1 shouting(5)  
  
<ipython-input-11-ca319fd14cc7> in shouting(phrase)  
      1 def shouting(phrase):  
----> 2     shout = phrase.upper() + '!!!'  
      3     return shout  
  
AttributeError: 'int' object has no attribute 'upper'
```

```
In [13]: shouting(5)
```

```
-----  
AttributeError                                Traceback (most recent call last)  
<ipython-input-13-32f05294ee9d> in <module>  
----> 1 shouting(5)  
  
<ipython-input-11-ca319fd14cc7> in shouting(phrase)  
      1 def shouting(phrase):  
----> 2     shout = phrase.upper() + '!!!'  
      3     return shout  
  
AttributeError: 'int' object has no attribute 'upper'
```

```
In [14]: def shouting(phrase):  
         # attempt to convert the input object to a string  
         shout = str(phrase).upper() + '!!!'  
         return shout
```

```
In [13]: shouting(5)
```

```
-----  
AttributeError                                Traceback (most recent call last)  
<ipython-input-13-32f05294ee9d> in <module>  
----> 1 shouting(5)  
  
<ipython-input-11-ca319fd14cc7> in shouting(phrase)  
      1 def shouting(phrase):  
----> 2     shout = phrase.upper() + '!!!'  
      3     return shout  
  
AttributeError: 'int' object has no attribute 'upper'
```

```
In [14]: def shouting(phrase):  
        # attempt to convert the input object to a string  
        shout = str(phrase).upper() + '!!!'  
        return shout
```

```
In [15]: shouting(5)
```

```
Out[15]: '5!!!'
```

Returning a value

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

Returning a value

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

In [16]:

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

Returning a value

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

In [16]:

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

In [17]:

```
greeting = shouting("hi")
```

Returning a value

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

```
In [16]: def shouting(phrase):  
         # attempt to convert the input object to a string  
         shout = str(phrase).upper() + '!!!'  
         return shout
```

```
In [17]: greeting = shouting("hi")
```

```
In [18]: greeting
```

```
Out[18]: 'HI!!!'
```

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

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

In [19]:

```
def quiet(phrase):  
    shh = str(phrase).lower()  
    shh
```

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

```
In [19]: def quiet(phrase):  
         shh = str(phrase).lower()  
         shh
```

```
In [20]: whisper = quiet("HELLO")
```

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

```
In [19]: def quiet(phrase):  
         shh = str(phrase).lower()  
         shh
```

```
In [20]: whisper = quiet("HELLO")
```

```
In [21]: whisper
```

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

```
In [19]: def quiet(phrase):  
         shh = str(phrase).lower()  
         shh
```

```
In [20]: whisper = quiet("HELLO")
```

```
In [21]: whisper
```

```
In [22]: print(whisper)
```

None

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

```
In [19]: def quiet(phrase):  
         shh = str(phrase).lower()  
         shh
```

```
In [20]: whisper = quiet("HELLO")
```

```
In [21]: whisper
```

```
In [22]: print(whisper)
```

None

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

```
Out[23]: NoneType
```

Returning multiple values

A function can return multiple values as a tuple. We will explore tuples in a future lecture.

Returning multiple values

A function can return multiple values as a tuple. We will explore tuples in a future lecture.

In [24]:

```
def powersof(number):  
    square = number ** 2  
    cube = number ** 3  
    return number, square, cube
```

Returning multiple values

A function can return multiple values as a tuple. We will explore tuples in a future lecture.

```
In [24]: def powersof(number):  
         square = number ** 2  
         cube = number ** 3  
         return number, square, cube
```

```
In [25]: powersof(3)
```

```
Out[25]: (3, 9, 27)
```

tuple unpacking

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

tuple unpacking

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

```
In [26]: x, y, z = powersof(3)
```

tuple unpacking

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

```
In [26]: x, y, z = powersof(3)
```

```
In [27]: print(x)
```

3

tuple unpacking

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

```
In [26]: x, y, z = powersof(3)
```

```
In [27]: print(x)
```

3

```
In [28]: print(y) # all of the values are stored separately  
         print(z)
```

9
27

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

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

```
In [29]: j = powersof(4)
```

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

```
In [29]: j = powersof(4)
```

```
In [30]: print(j)
```

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

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

```
In [29]: j = powersof(4)
```

```
In [30]: 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]`.

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

```
In [29]: j = powersof(4)
```

```
In [30]: 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 [31]: j[0]
```

```
Out[31]: 4
```

```
In [32]: j[2]
```

```
Out[32]: 64
```

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.

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 = powerof(5)
```

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

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)
```

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).

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
```

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

Spam Spam
Spam Spam

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

Spam
Spam
None
None

In [37]:

```
print_twice("Spam " * 2)
```

```
Spam Spam  
Spam Spam
```

In [38]:

```
print_twice(print_twice("Spam"))
```

```
Spam  
Spam  
None  
None
```

What happened here?

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

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)
```

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

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  
or  
  
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 [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  
or  
  
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 [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 [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 [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 arguments.
```

```
-----  
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 o  
f the arguments.  
  
TypeError: junk() got multiple values for argument 'a'
```

```
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 arguments.
```

```
-----  
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 o  
f the arguments.  
  
TypeError: junk() got multiple values for argument 'a'
```

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

```
10 0 5
```

Function Variables and Parameters are Local

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

Function Variables and Parameters are Local

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

In [49]:

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

Function Variables and Parameters are Local

When you create a variable inside a function, it is local, which means that it only exists inside the scope of 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 = 'bidi bidi'  
line2 = 'bom bom'  
cat_twice(line1, line2)
```

```
bidi bidi bom bom  
bidi bidi bom bom
```

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

When `cat_twice` terminates, the variable `cat` is destroyed. If we try to refer to `cat` in the global environment, 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
```


When `cat_twice` terminates, the variable `cat` is destroyed. If we try to refer to `cat` in the global environment, 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
```

Error Tracebacks

If an error occurs during a function call, Python prints the offending line. If the offending line is a function, it prints out the contents of that function and the offending line there. It continues this until it reaches the top-most *frame*.

Values that are not defined inside a function are defined in the frame `__main__`.
`__main__` is the top-level script environment.

Error Tracebacks

If an error occurs during a function call, Python prints the offending line. If the offending line is a function, it prints out the contents of that function and the offending line there. It continues this until it reaches the top-most *frame*.

Values that are not defined inside a function are defined in the frame `__main__`.

`__main__` is the top-level script environment.

For example, I modified the function `print_twice()`. It tries to access the variable `cat` which is not defined inside `print_twice()`.

Error Tracebacks

If an error occurs during a function call, Python prints the offending line. If the offending line is a function, it prints out the contents of that function and the offending line there. It continues this until it reaches the top-most *frame*.

Values that are not defined inside a function are defined in the frame `__main__`. `__main__` is the top-level script environment.

For example, I modified the function `print_twice()`. It tries to access the variable `cat` which is not defined inside `print_twice()`.

In [53]:

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

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

In [54]:

```
line1 = 'bidi bidi'
line2 = 'bom bom'
cat_twice(line1, line2)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-54-fdce103e5d5e> in <module>
      1 line1 = 'bidi bidi'
      2 line2 = 'bom bom'
----> 3 cat_twice(line1, line2)

<ipython-input-53-fd2c2f843dda> in cat_twice(part1, part2)
      5 def cat_twice(part1, part2):
      6     cat = part1 + " " + part2
----> 7     print_twice(cat)

<ipython-input-53-fd2c2f843dda> in print_twice(bruce)
      1 def print_twice(bruce):
----> 2     print(cat)
      3     print(cat)
      4
      5 def cat_twice(part1, part2):

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

```
<ipython-input-53-fdce103e5d5e> in <module>
      1 line1 = 'bidi bidi'
      2 line2 = 'bom bom'
----> 3 cat_twice(line1, line2)
```

The traceback starts with the lines we just executed. There are no problems with lines 1 and 2 where we simply assign some lyrics to variable names. Python tells us the offending line is line 3 when we called `cat_twice()`

```
<ipython-input-52-fd2c2f843dda> in cat_twice(part1, part2)
      5 def cat_twice(part1, part2):
      6     cat = part1 + " " + part2
----> 7     print_twice(cat)
```

The next part of the traceback enters the function `cat_twice()`. It tells us that the offending line is line 7: when we made a call to `print_twice()`

```
<ipython-input-52-fd2c2f843dda> in print_twice(bruce)
      1 def print_twice(bruce):
----> 2     print(cat)
      3     print(cat)
      4
      5 def cat_twice(part1, part2):
```

Finally, the traceback shows us the contents of `print_twice()` and says the offending line is line 2: when we try to print the variable `cat`.

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

It gives us a `NameError` and states that the name `cat` is not defined.

Global Scope

In the following cell, I run the same code but define `cat` in the global scope. Even though `cat` is not found inside the local scope of the function `print_twice()`, it is defined in the global scope. When `print_twice()` is called from within `cat_twice()`, the variable `cat` is found in the global environment and printed.

In [55]:

```
def print_twice(bruce):  
    print(cat)  
    print(cat)  
  
def cat_twice(part1, part2):  
    cat = part1 + " " + part2  
    print_twice(cat)  
  
line1 = 'bidi bidi'  
line2 = 'bom bom'  
  
cat = "something else entirely"  
  
cat_twice(line1, line2)
```

```
something else entirely  
something else entirely
```

%who, %whos, and %who_ls

iPython has a few magic commands that list the objects defined in the global environment
`%who` prints the names, `%whos` prints the names and details of each object, and `%who_ls` returns a list with object names as strings.

In [56]:

```
%who
```

```
cat      cat_twice      greeting      j      junk      line1      line2      math
powersof
print_twice      quiet      shouting      stuff      whisper      x      y
z
```

In [57]:

```
%whos
```

Variable	Type	Data/Info
-----	-----	-----
cat	str	something else entirely
cat_twice	function	<function cat_twice at 0x00000247892CEB88>
greeting	str	HI!!!
j	tuple	n=3
junk	function	<function junk at 0x00000247891E5828>
line1	str	bididi
line2	str	bom bom
math	module	<module 'math' (built-in)>
powersof	function	<function powersof at 0x00000247892ACB88>
print_twice	function	<function print_twice at 0x00000247892CEDC8>
quiet	function	<function quiet at 0x00000247892A4B88>
shouting	function	<function shouting at 0x00000247892A4318>
stuff	function	<function stuff at 0x00000247891D0DC8>

whisper	NoneType	None
x	int	3
y	int	9
z	int	27

In [58]: `%who_ls`

Out[58]: ['cat',
'cat_twice',
'greeting',
'j',
'junk',
'line1',
'line2',
'math',
'powersof',
'print_twice',
'quiet',
'shouting',
'stuff',
'whisper',
'x',
'y',
'z']

Scoping rules

Scoping rules

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

Scoping rules

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

In [59]:

```
x = 5
```

Scoping rules

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

```
In [59]: x = 5
```

```
In [60]: x
```

```
Out[60]: 5
```

Scoping rules

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

```
In [59]: x = 5
```

```
In [60]: x
```

```
Out[60]: 5
```

```
In [61]: def alter_x(x):  
         x = x + 1  
         return x
```

Scoping rules

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

```
In [59]: x = 5
```

```
In [60]: x
```

```
Out[60]: 5
```

```
In [61]: def alter_x(x):  
         x = x + 1  
         return x
```

```
In [62]: alter_x(x)
```

```
Out[62]: 6
```


Scoping rules

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

```
In [59]: x = 5
```

```
In [60]: x
```

```
Out[60]: 5
```

```
In [61]: def alter_x(x):  
         x = x + 1  
         return x
```

```
In [62]: alter_x(x)
```

```
Out[62]: 6
```

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

```
Out[63]: 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.

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 [64]:

```
def alter_global_x():  
    global x  
    x = x + 1  
    return x
```

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

```
In [65]: x = 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 [64]: def alter_global_x():  
         global x  
         x = x + 1  
         return x
```

```
In [65]: x = 5
```

```
In [66]: alter_global_x()
```

```
Out[66]: 6
```

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

```
In [65]: x = 5
```

```
In [66]: alter_global_x()
```

```
Out[66]: 6
```

```
In [67]: x
```

```
Out[67]: 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.

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 [68]:

```
# 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
```


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 [68]: # 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 [69]: search_for_x()
```

6

```
Out[69]: 6
```

Scope Order in Python

Taken from: <https://realpython.com/python-scope-legb-rule/>

Python will search scopes in the following order:

- Local (or function) scope is the code block or body of any Python function. This Python scope contains the names that you define inside the function. These names will only be visible from the code of the function.
- Enclosing (or nonlocal) scope is a special scope that only exists for functions nested inside other functions. If the local scope is an inner or nested function, then the enclosing scope is the scope of the outer or enclosing function. This scope contains the names that you define in the enclosing function. The names in the enclosing scope are visible from the code of the inner and enclosing functions.
- Global scope is the top-most scope in a Python program, script, or module. This Python scope contains all of the names that you define at the top level of a program or a module. Names in this Python scope are visible from everywhere in your code.
- Built-in scope is a special Python scope that's created whenever you run a script or open an interactive session. This scope contains names such as keywords, functions, exceptions, and other attributes that are built into Python.