Lecture 2-1

Functions in Python

Week 2 Monday

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**

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. This is the object of the function itself
```

Out[2]:
```
<function print(*args, sep=' ', end='\n', file=None, flush=False)>
```

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.

In [6]:
```
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.
```

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!')
```

```
  Cell In[8], line 1
    print('I can't believe it!')
                               ^
SyntaxError: unterminated string literal (detected at line 1)
```

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.

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)
Cell In[13], line 1
----> 1 shouting(5)

Cell In[11], line 2, 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.

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`.

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.

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.

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

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.

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

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[33], line 1
----> 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. 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.

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 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

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)
Cell In[41], line 1
----> 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 arguments.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[47], line 1
----> 1 junk(5, a = 10, b = 0) # python will get confused if you name only some of 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.

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.

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

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[51], line 1
----> 1 cat

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

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

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[52], line 1
----> 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.

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)
Cell In[54], line 3
      1 line1 = 'bidi bidi'
      2 line2 = 'bom bom'
----> 3 cat_twice(line1, line2)

Cell In[53], line 7, in cat_twice(part1, part2)
      5 def cat_twice(part1, part2):
      6     cat = part1 + " " + part2
----> 7     print_twice(cat)

Cell In[53], line 2, in print_twice(bruce)
      1 def print_twice(bruce):
----> 2     print(cat)
      3     print(cat)

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 exectued. There are no problems with lines 1 and 2 where we simply assign some lyrics to variable names. Python tell 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 0x0000019F7A163060>
greeting      str         HI!!!
j             tuple       n=3
junk          function    <function junk at 0x0000019F7A1611C0>
line1         str         bidi bidi
line2         str         bom bom
math          module      <module 'math' (built-in)>
powersof      function    <function powersof at 0x0000019F79C37D80>
print_twice   function    <function print_twice at 0x0000019F7A161080>
quiet         function    <function quiet at 0x0000019F79C37B00>
shouting      function    <function shouting at 0x0000019F79C37740>
stuff         function    <function stuff at 0x0000019F7A160FE0>
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

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.

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.

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.

In [70]:
```
x, y, z = 1, 1, 1

def f():
    y = 2  # changing y to 2, only affects the value inside the function
    return x, y, z   # it does not find x or z in the local environment, so it searches the higher scope

print(f())
print(x, y, z)
```

```
(1, 2, 1)
1 1 1
```

In [71]:
```
x, y, z = 1, 1, 1
```

```
def f():
    y = 2
    def g():
        z = 3
        return x, y, z
    return g()

print(f())
print(x, y, z)
```

```
(1, 2, 3)
1 1 1
```

`g()` is defined inside `f()`

When we call the function `f()`, the final line of `f()` calls `g()` and returns the value of `g()`.

When `g()` runs, it sets `z = 3`. Inside `g()`, `x` and `y` are not defined. To find those values, it searches the higher scope `f()` for x and y. It finds the value of `y = 2` defined inside `f()`. It finds `x = 1` in the top level scope.

When `f()` runs, it returns `x = 1`, `y = 2`, `z = 3` while x, y, z are all equal to 1 in the top-level environment.

In [72]:
```
x, y, z = 1, 1, 1

def g():
    z = 3
    return x, y, z

def f():
    y = 2
    return g()

print(f())
print(x, y, z)
```

```
(1, 1, 3)
1 1 1
```

`g()` and `f()` are both defined in the global environment.

The function `f()` returns the value of `g()`

When `g()` runs, it sets `z = 3`. Inside `g()`, `x` and `y` are not defined. To find those values, it searches the higher scope which is the global environment because `g()` is defined inside the global environment. It uses the values in the global environment `x = 1` and `y = 1`.

It does not matter that `g()` was called from inside `f()`. When `g()` needs to search a higher scope, it searches the environment in which the function is defined.

In [73]:
```
# keyword global gives the function access to the value in the global environment
x, y, z = 1, 1, 1

def f():
    y = 2
    def g():
        global z  # calling global, gives g access to the global value of z
        z = 3     # will assign 3 to the global variable z
        return x, y, z
    return g()

print(f())
print(x, y, z)
```

```
(1, 2, 3)
1 1 3
```

`g()` is defined inside `f()`

When we call the function `f()`, the final line of `f()` calls `g()` and returns the value of `g()`.

When `g()` runs, it accesses the global variable `z`. It sets `z = 3` in the global environment. Inside `g()`, `x` and `y` are not defined. To find those values, it searches the higher scope `f()` for x and y. It finds the value of `y = 2` defined inside `f()`. It finds `x = 1` in the top level scope.

When `f()` runs, it returns `x = 1`, `y = 2`, `z = 3`.

Because `g()` has access to `z` in the global environment, the value of z is now 3 after the function runs.

In [74]:
```
x, y, z = 1, 1, 1

def g():
```

```
    z = 3
    return x, y, z

def f():
    global y
    y = 2
    return g()

print(g()) # when we first run g(), it uses the global values of x and y, but the local value of z. Local value of z
print(x, y, z)
```

```
(1, 1, 3)
1 1 1
```

`g()` and `f()` are both defined in the global environment.

When `g()` runs, it sets `z = 3`. Inside `g()`, `x` and `y` are not defined. To find those values, it searches the higher scope which is the global environment because `g()` is defined inside the global environment. It uses the values in the global environment `x = 1` and `y = 1`.

In [75]:
```
print(f())  # when we run f(), the global value of y is changed.
print(x, y, z)
```

```
(1, 2, 3)
1 2 1
```

When we call the function `f()`, it modifies the value of `y` in the global environment. The final line of `f()` calls and returns the value of `g()`. This time, when `g()` looks for a value of `y`, it finds the value of `y` in the global environment which is now 2.

In [76]:
```
p, q = 1, 1

def f():
    global s   # will create s in the global
    s = 2
    return p, q, s
f()
```

Out[76]:
```
(1, 1, 2)
```

In [77]:
```
s
```

Out[77]: 2

If you use the keyword `global` inside a function it will create the variable in the global environment if necessary.

In [78]:
```
x, y, z = 1, 1, 1

def f():
    global y
    print("current value of y is " + str(y))
    y = 4
    def g():
        global y
        print("current value of y is now " + str(y))
        y = 10
        print("current value of y is finally " + str(y))
        global z
        z = 3
        return x, y, z
    return g()

print(f())
print(x, y, z)
```

```
current value of y is 1
current value of y is now 4
current value of y is finally 10
(1, 10, 3)
1 10 3
```

Both the function `g()` and `f()` access the global variable `y`. Each time we assign a new value to `y`, it updates the value in the global environment.

In [79]:
```
x, y, z = 1, 1, 1

def f():
    y = 4
    def g():
        nonlocal y
        y = 10  # affects the y defined inside f
        global z
        z = 3
```

```
        return x, y, z
    print(x, y, z)  # this line is run before g() is called
    return g()  # when g() is called, y will be modified

print(f())
print(x, y, z)
```

```
1 4 1
(1, 10, 3)
1 1 3
```

When we call the function `f()`, it sets a local variable `y = 4`. It defines a function `g()` inside `f()`. It prints the values `x, y, z`. At this time, `y = 4`.

The final line of `f()` calls `g()` and returns the value of `g()`. When `g()` is called, it accesses the nonlocal variable `y`. The nonlocal keyword tells the function to search the higher scope, in this case, the scope of `f()`. It sets nonlocal `y = 10` and global `z = 3`. It returns `x = 1` global, `y = 10` nonlocal, `z = 3` global.

Because `g()` has access to `z` in the global environment, the value of z is now 3 after the function runs. However, the value `y` in the global environment remains 1 because it only modified the nonlocal variable `y`.

In [80]:
```
p, q = 1, 1

def f():
    nonlocal r   # will return an error because r does not exist in the nonlocal environment
    r = 2
    return p, q, r

f()
```

```
  Cell In[80], line 4
    nonlocal r   # will return an error because r does not exist in the nonlocal environment
    ^
SyntaxError: no binding for nonlocal 'r' found
```

If you ask for a nonlocal variable but there is no higher scope (other than the global environment), Python will return an error.