User-Defined Functions

Python 3

home (../handouts.html)

Introduction: User-Defined Functions

A **user-defined function** is a *named code block* -- very simply, a block of Python code that we can call by name. These functions are used and behave very much like built-in functions, except that we define them in our own code.

```
def addthese(val1, val2):  # function definition; argument signature
    valsum = val1 + val2
    return valsum          # return value

x = addthese(5, 10)        # function call; 2 arguments passed;
                           # return value assigned to x
print(x)                    # 15
```

There are two primary reasons functions are useful: to *reduce code duplication* and to *organize our code*:

Reduce code duplication: a named block of code can be called numerous times in a program, which means the same series of statements can be executed repeatedly, without having to type them out multiple times in the code.

Organize code: large programs can be difficult to read, even with helpful comments. Dividing code into named blocks allows us to identify the major steps our code can take, and see at a glance what steps are being taken and the order in which they are taken.

We have learned about using simple functions for sorting; in this unit we will learn about:

1) different ways to define function arguments
2) the "scoping" of variables within functions
3) the four "naming" scopes within Python

Objectives for the Unit: User-Defined Functions

- define functions that take *arguments* and return *return values*
- define functions that take *positional* and *keyword* arguments
- define functions that can take an *arbitrary number* of arguments
- learn about the four *variable scopes* and how scopes interact

Review: functions are *named code blocks*

The block is executed every time the function is called.

```
def print_hello():
    print("Hello, World!")

print_hello()             # prints 'Hello, World!'
print_hello()             # prints 'Hello, World!'
print_hello()             # prints 'Hello, World!'
```

When we run this program, we see the greeting printed three times.

Review: function argument(s)

Any argument(s) passed to a function are *aliased* to variable names inside the function definition.

```
def print_hello(greeting, person):    # 2 strings aliased to objects
                                      # passed in the call

    full_greeting = "{}, {}!".format(greeting, person)
    print(full_greeting)

print_hello('Hello', 'World')         # pass 2 strings:  prints "Hello, World!"
print_hello('Bonjour', 'Python')      # pass 2 strings:  prints "Bonjour, Python!"
print_hello('squawk', 'parrot')       # pass 2 strings:  prints "squawk, parrot!"
```

Review: the *return* statement returns a value

Object(s) are returned from a function using the **return** statement.

```
def print_hello(greeting, person):
    full_greeting = greeting + ", " + person + "!"
    return full_greeting

msg = print_hello('Bonjour', 'parrot')   # full_greeting
                                         # aliased to msg

print(msg)                                # 'Bonjour, parrot!'
```

Summary argument types: *positional* and *keyword*

Your choice of type depends on whether they are required.

positional: args are required and in particular order

```
def sayname(firstname, lastname):
    print("Your name is {} {}".format(firstname, lastname))

sayname('Joe', 'Wilson')    # passed two arguments:  correct

sayname('Joe')              # TypeError:  sayname() takes exactly 2 arguments (1 given)
```

keyword: args are not required, can be in any order, and the function specifies a default value

```
def sayname(lastname, firstname="Citizen"):
    print("Your name is {} {}".format(firstname, lastname))


sayname('Wilson', firstname='Joe')  # Your name is Joe Wilson

sayname('Wilson')                   # Your name is Citizen Wilson
```

Variable name *scoping* inside functions

Variable names initialized inside a function are *local* to the function, and not available outside the function.

```
def myfunc():
    a = 10
    return a

var = myfunc()     # var is now 10
print(a)           # NameError ('a' does not exist here)
```

Note that although the object associated with **a** is returned and assigned to **var**, the *name* **a** is not available outside the function. Scoping is based on names.

global variables (i.e., ones defined outside a function) are available both inside and outside functions:

```
var = 'hello global'

def myfunc():
    print(var)

myfunc()              # hello global
```

The four variable scopes: (L)ocal, (E)nclosing, (G)lobal and (B)uiltin

Variable scopes "overlay" one another; a variable can be "hidden" by a same-named variable in a "higher" scope.

From top to bottom:

- **Local**: local to (defined in) a function
- **Enclosing**: local to a function that may have other functions in it
- **Global**: available anywhere in the script (also called *file scope*)
- **Built-in**: a built-in name (usually a function like **len()** or **str()**) A variable in a given scope can be "hidden" by a same-named variable in a scope above it (see example below):

```
def myfunc():
    len = 'inside myfunc'  # local scope:  len is initialized in the function
    print(len)

print(len)                  # built-in scope:  prints '<built-in function len>'

len = 'in global scope'    # assigned in global scope:  a global variable
print(len)                  # global scope:  prints 'in global scope'

myfunc()                   # prints 'inside myfunc' (i.e. the function executes)

print(len)                  # prints 'in global scope' (the local len is gone, so we see the global

del len                    # 'deletes' the global len
print(len)                  # prints '<built-in function len>'
```

Summary exception: UnboundLocalError

An **UnboundLocalError** exception signifies a local variable that is "read" before it is defined.

```
x = 99
def selector():
    x = x + 1       # "read" the value of x; then assign to x
selector()

# Traceback (most recent call last):
#   File "test.py", line 1, in
#   File "test.py", line 2, in selector
# UnboundLocalError: local variable 'x' referenced before assignment
```

Remember that a *local* variable is one that is initialized or assigned inside a function. In the above example, **x** is a local variable. So Python sees **x** not as the global variable (with value **99**) but as a local variable. However, in the process of initializing **x** Python attempts to *read* **x**, and realizes that is hasn't been initialized yet -- the code has attempted to *reference* (i.e., read the value of) **x** before it has been assigned.

Since we want Python to treat **x** as the global **x**, we need to tell it to do so. We can do this with the **global** keyword:

```
x = 99
def selector():
    global x
    x = x + 1
selector()
print(x)            # 100
```