Chapter 8— Reuse of Code

Introduction

Imagine for a moment how difficult it would be if you had to write the functionality of the print statement or the random number generator, every time you wanted to use it. In the earliest days of computer programming, this was often the case.

It became apparent, quickly, that programmers needed to create blocks of code that do common functions and then re-use them. We call these blocks of re-usable code functions. In Python, we will use the `def` statement to define them.

Objectives

Upon completion of this chapter's exercises, you should be able to:

- Explain the need for reuse of code in a complex system.
- Create a function that accepts zero or more values being passed into it.
- Use default values in the function definition.
- Use return values from functions in the calling context.
- Apply the concept of variable scoping when creating and using functions,
- Apply passed values to a function into a tuple.

Prerequisites

This Chapter requires...

Simple Function

The simplest function is one that does not need information sent to it, and does not return a value. In some languages this is called a subroutine and is a simple block of code that performs a specific action.[5] In the example below we create a def called line to print a short line, It is used twice in the rest of the program.

```
def line():
    print("-------")
```

5 NEED REFERENCE

```

print (1)
print("+", 9)
line()
print(1+9)

print()
print(9)
print("*",3)
line()
print(9*3)
```

```
1
+ 9
-------
10

9
* 3
-------
27
```

In the definition of line we use the `def` statement followed by the routine's name, a set of parentheses, and a colon. Like in other blocks of code (`if`, `for`) the indented code following the colon is part of the block.

To "call" the function, simply use its name followed by empty parenthesis. In future sections, we will learn how to pass values to the function and how to have the function return values to us.

Def *function_name*(*arguments...*):
    suite
	Statement

Define a function... https://docs.python.org/3/reference/compound_stmts.html#function-definitions	

Creating a Main Def

Commonly in complex Python programs the main or initial part of a program is defines in a function named main() and a single function call to main() is made at the bottom of the code to start the program. This insures that all the variables are local and not globally visible, we will discuss this in more detail further in the chapter.

```python
def count_to_ten():
    for x in range(1,11):
        print(x)

def main():
    count_to_ten()

main()
```

```
1
2
3
4
5
6
7
8
9
10
```

From this point on in this book, we will be using a function main() in most of the examples. While this is not strictly required, and often debated, it is required in languages like C. It is introduced here so that when you transition to another programming language you are familiar with this constrict.

Returning a Value from a Function

In addition to a `def` allowing you to define a block of reusable code, it can be used to create and return a value. The `return` statement will cause a def to stop executing and to send a value back to the part of the program that "called" it.

```python
import random

def coin():
    if random.randint(0,1) == 0:
        return "head"
    return "tail"

def main():
    for t in range(5):
        c = coin()
        print(c)

main()
```

```
head
head
tail
head
head
```

The returned value from a `def` may be any type of value (integer, float, string, list,…) and may be stored into a variable or used directly.

return
return *expression*
	Statement

The `return` statement is used to terminate the function and return control to the code that called the function. An expression may be returned from the function, if the function call was used in an expression the returned value will be used. https://docs.python.org/3/reference/compound_stmts.html#index-13	

Passing Values to a Function

Functions become even more powerful when we can send values to them to use. These values are passed to a special type of variables called "free variables".[6] The free variables are defined in the `def` statement and are bound to the value where the function was called.

Th the example below, you see how four values are sent into the def and the def formats and prints them out in a pretty format.

```
def showmath(a, operation, b, answer):
    print(str(a).rjust(7))
    print(operation, str(b).rjust(5))
    print("-------")
    print(str(answer).rjust(7))
```

6 https://en.wikipedia.org/wiki/Free_variables_and_bound_variables

```

def main():
    showmath(1, "+", 9, 1+9)
    print()
    showmath(9, "*", 3, 9*3)

main()
```

```
      1
+     9
-------
     10

      9
*     3
-------
     27
```

You may pass any type into a def.

```
def listinfo(x):
    print("the list", x, "contains", len(x), "items")

def main():
    numbers = [1,2,3]
    listinfo(numbers)
    fruit = ["apple","berry","cherry"]
    listinfo(fruit)

main()
```

```
the list [1, 2, 3] contains 3 items
the list ['apple', 'berry', 'cherry'] contains 3 items
```

Passing Optional Arguments to a Function

Python allows you to define default values when we define the free variables (in the `def`). You follow the variable name with an equal sign and the value. When a definition is called without the corresponding argument, the free variable is set to the default value.

In the first `def` below, the variable b is set to a default of `None` (a special value meaning that there is not a value). In the body of the function an `if` statement is used to see if b was passed. In the second def, you can see that the tax rate was set to a default of 6%.

```
def show(a, b=None):
    if(b):
        print("Two values", a, "and", b)
    else:
        print("One value", a)

def tax(price, taxrate=.06):
    # calculate sales tax (default rate is 6%)
    return round(price * taxrate, 2)

show("XZ")
show(99, "Foo")
print("The default tax on 100.00 is", tax(100))
print("The local (7.25%) tax on 1000 is", tax(1000, .0725))
```

```
One value XYZ
Two values 99 and Foo
The default tax on 100.00 is 6.0
The local (7.25%) tax on 1000 is 72.5
```

Functions All the Way

In the previous topics, you have seen that we can pass values into a function and return a value from a function. In many uses of functions we will do both.

```
def area(w, d=None):
    # Return area of a square or a rectangle
    if(d):
        return(w*d)
    else:
        return(w*w)

print("The area of a square that is 9x9 is",area(9))
print("The area of a rectangle that is 8x5 is",area(8,5))
```

```
The area of a square that is 9x9 is 81
The area of a rectangle that is 8x5 is 40
```

In another example, lets create a function that will input an integer. If the user enters a value that can't be converted to an integer then keep asking until they enter a value

```
def getint(prompt="Enter an integer?"):
    ## the getint function asks the user for a required integer
    while True:
        try:
            i = int(input(prompt))
            return I
        except:
            pass

a = getint("How old are you?")
b = getint()
print("You are",a,"years old.")
print("You entered",b)
```

Scope and Binding of Variables

Python handles variables in a way that many other languages don't.[7] It uses a process of binding the names of variables to values in the computer's memory.

With the statement `SomeVar = 9` what is happening is that the name of the variable is bound (connected) to the value. The variable is not a static storage location in memory, but a symbol attached to the actual value.

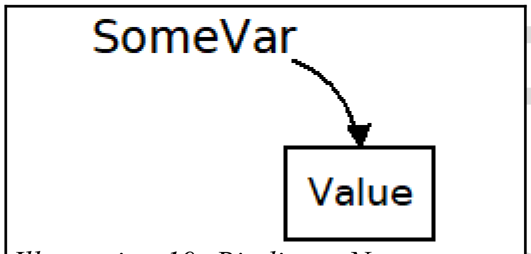

Illustration 19: Binding a Name

In the following program we will see how a name that is bound in the code outside a def is usable inside the def, as long as it is never re-bound. When it is re-bound inside the def, the original binding is still un-changed.

```
def showa():
```

7 https://docs.python.org/3/reference/executionmodel.html

```
    print("inside def showa - a=", a)

def differenta():
    a = "crap" #### Bind a to a new value
    print("inside def differenta - after binding a=", a)

a = "stuff"
print("before calls a=", a)
showa()
differenta()
print("after calls a=", a)
```

```
before calls a= stuff
inside def showa - a= stuff
inside def differenta - after binding a= crap
after calls a= stuff
```

In the code, above, you will see that in the def showa, the variable is not re-bound and is bound to the value from the variable that was created outside. In the def differenta, we see that the name was re-bound to a new value, but that the original value was not changed.

It can be taken from this discussion that new names that are bound inside a def exist only for the duration of the def and the value are not available to the program outside.

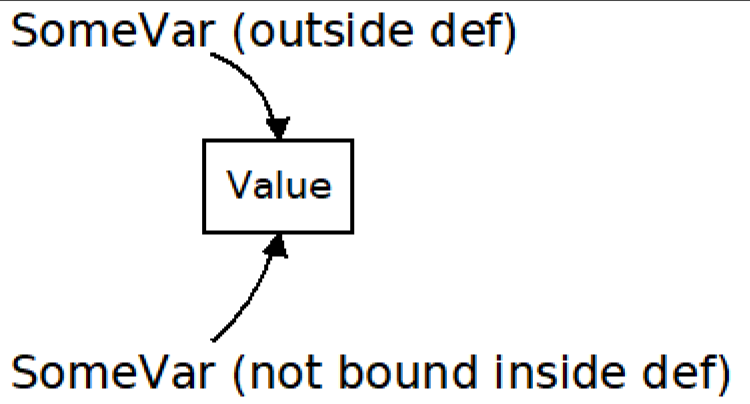

Illustration 20: Bound Name in a Def Without Re-Binding

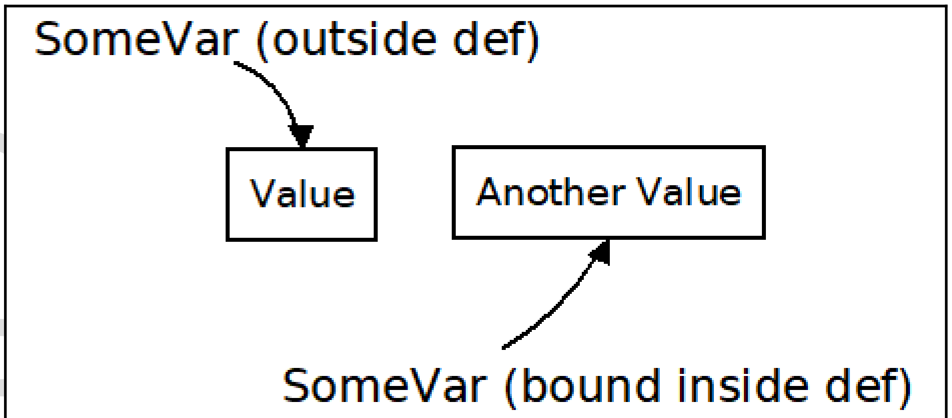

Illustration 21: Bound Name in a Def When it is Re-Bound

Getting Arguments as a List

```
def what(*a):
    if len(a) == 0:
        print("no arguments")
    else:
        for i in range(len(a)):
            print("argument", i, "is", a[i])

what()
what("Hola")
what("Bon", "Jour")
what("foo", [1,2,3], "bar")
```

```
no arguments
argument 0 is Hola
argument 0 is Bon
argument 1 is Jour
argument 0 is foo
argument 1 is [1, 2, 3]
argument 2 is bar
```

Summary

Goes here

Important Terms

- argument
- binding
- def
- default
- function
- indent
- main
- main
- parenthesis
- passing
- return
- suite

Exercises

Here

Word Search

```
j f a k f p p a w f c i
z h r d d b a r r k i n
o c m e e i r g e p j d
m z m f f n e u t a f e
a w l w a d n m u s u n
i j p h u i t e r s n t
n k f j l n h n n i c t
a · · m t g e t h n t ·
p u w a o h s k a g i q
f y z i · f i y w m o o
r y k n k c s z r v n a
q m s u i t e n j j j v
```

argument, binding, def, default, function, indent, main, main, parenthesis, passing, return, suite

References