Chapter 4: Functions I

Last Update: March 15, 2023

Program 1: This program demonstrates various ways to use a function, with or without parameters, with or without return value, etc.

04 11-square.py

```python
# A function may have 0 or more input parameters
# A function can return one value, but the value can be a structure that consists of multiple values
# If you do not use return, None is returned
# Here are all the combinations
def next():
    input("> ")

# No parameter
# No return
def sq1():
    print(f"Case 1. Print inside the function: {2**2}, no return")

print(f"Case 1. Calling sq1() and print out the result: {sq1()}")
print("Case 1. None is returned because there is no explicit RETURN.")
next()

# No parameter
# with return
def sq2():
    return (2**2)

print(f"Case 2. Calling sq2() and returning the result: {sq2()}")
next()

# With parameters
# No return
def sq3(arg1):
    print(f"Case 3. Print inside the function: {arg1**2}, no return")

print(f"Case 3. Calling sq3() with an argument (3): {sq3(3)}")
next()

# With parameters
# With return
def sq4(arg1):
    return(arg1**2)

x = sq4(4)
print(f"sq4(4): Saving the return value {x} so it can be reused.")
next()

# With parameters
# No return
def sq5(arg1, arg2):
    return arg1**2 + arg2**2

print(f"Case 5: sq5(1.5,2): Multiple arguments with different types: {sq5(1.5, 2)}.")
print(f"Case 5: sq5(10**2, x+1): Using expressions as arguments: {sq5(10**2, x+1)}.")
```

Program 2: This program demonstrates calling a function with different parameters. Notice that the types of parameters do not have to be the same. You can pass even an expression.

```
04  12-para.py
def next():
    input("\nNext> ")

def test(x):
    print(f"\t\t\tBefore: {id(x)}, {x}")
    x = x*2
    print(f"\t\t\tAfter:  {id(x)}, {x}")
    return x

print("Case 1: float")
y = 3.14159
print(f"\tBefore: {id(y)}, {y}")
test(y)
print(f"\tAfter:  {id(y)}, {y}")
next()

print("Case 2: string")
y = "Hello world"
print(f"\tBefore: {id(y)}, {y}")
test(y)
print(f"\tAfter:  {id(y)}, {y}")
next()

print("Case 3: int")
y = 9
print(f"\tBefore: {id(y)}, {y}")
test(y)
print(f"\tAfter:  {id(y)}, {y}")
next()

print("Case 4: int")
y = 9
print(f"\tBefore: {id(y)}, {y}")
y = test(y)
print(f"\tAfter:  {id(y)}, {y}")
next()

print("Case 5: int expression")
y = 9
print(f"\tBefore: {id(y)}, {y}")
test(y*y)
print(f"\tAfter:  {id(y)}, {y}")
```

Program 3: Your function may have a return statement or not. In the last example, we place a return inside an if-statement. That's a bad practice. Why?

04 13-return.py

```python
#  Keyword: return value
#      with or without return
#
def test1():
    greeting = "Have a Nice Day"
    print(greeting)
    return greeting

def test2():
    greeting = "Good Morning"
    print(greeting)

def test3(morning):
    if morning:
        greeting = "Good Morning"
        print(greeting)
        return greeting    # try moving outside of it

print("Case 1 returns:", test1())
print("Case 2 returns:", test2())
morning = True    # try False
print("Case 3 returns:", test3(morning))
```

Program 4: Functions can be placed anywhere in the program with the restriction that a function must be defined before it is used. Functions are typically placed at the beginning of the program.

```
04  14-placement.py
```

```python
#  Keyword: placement of functions & calls
#
def test1():
    return("111")
print(test1())  # try calling test2
print()

def test2():
    return("222")
print(test2())
print()

def test3():
    return("333")
print(test3())

print("\n A better arrangement") # Probably use different names
def test1():
    return("111")

def test2():
    return("222")

def test3():
    return("333")

# Main function
print(test1())
print()
print(test2())
print()
print(test3())
```

Program 5: If you want to use a value from the main program, pass it as a parameter. You may have as many parameters as you want. You may access the "global" variables directly, but it is a bad practice (read "Global variable considered harmful," https://dl.acm.org/doi/10.1145/953353.953355). Please don't do it in your assignments.

```
04 15-global.py
```

```python
def show_list1():  # no parameter
    print("Case 1:")
    for item in shopping_list: # Global
        print(item, end=' ')
    print()

def show_list2(list):  # pass through parameter
    print("Case 2:")
    for item in list:
        print(item, end=' ')
    print()

shopping_list = ["Bread", "Milk", "Butter", "Apple"]
show_list1()
print()
show_list2(shopping_list)
```

Program 6: A function may have some or no parameters and returns some or no values, as shown in this example.

```
04 20-definitions.py
```

```python
def next():
    input("Next> ")

def test1():
    print("\t Calling test1")
    print("\t Do somthing here.")

def test2(name):
    print("\t Calling test2")
    print(f"\t {name}, do something here.")

def test3():
    print("\t Calling test3")
    age = input("\t Enter your age: ")
    return age

def test4(name):
    print("\t Calling test4")
    age = input("\t Enter your age: ")
    grade = input("\t Enter your grade: ")
    return age, grade

def test5(name, age, grade):
    print("\t Calling test5")
    print(f"\t Name = {name}, Age = {age}, Grade = {grade}.")

print("Case 1")
test1()
next()

print("Case 2")
test2("Stephen")
next()

print("Case 3")
age = test3()
print(f"Age = {age}.")
next()

print("Case 4")
name = "Stephen"
age, grade = test4(name)
print(f"Name = {name}, Age = {age}, Grade = {grade}.")
next()

print("Case 5")
result = test5(name, age, grade)
print(result)
```

Program 7: This program shows how to pass parameters to a function. The function definition specified three parameters. To call the function, we must pass (exactly) three **arguments** to it. The arguments can be a variable, a constant, or an expression, as shown. The association of the parameters and the arguments is by the position, i. e., the i-th parameter receives the i-th arguments.

04 21-para.py

```python
# Parameter passing
def func(para1, para2, para3):
    local1, local2, local3 = para1*10, para2*20, para3*30
    print(f"\t {local1}, {local2}, {local3}.")

arg1, arg2, arg3 = 1, 2, 3
func(arg1, arg2, arg3)
func(10, 20, 30)
func(100+arg1, 200+arg2, 300+arg3)
```

Program 8: This program shows three functions being called separately, not chained.

04 22-call level.py

```python
# Function calls are independent of each other
def test1(level):
    print("    "*level, "Entering test() at level ", level)

def test2(level):
    print("    "*level, "Entering foo() at level ", level)

def test3(level):
    print("    "*level, "Entering bar() at level ", level)

test1(1)
test2(1)
test3(1)
```

Program 9: The three function calls in this program are chained (test > foo > bar). The indentations show when a function starts and ends. In the test(), the parameter "level" was given a 1, to begin with. The level in function foo is 2. When the control returns to function test(), the level is 1 (never changed).

```
04 23-call chain.py
def test(level):
    print("    "*level, "Entering test() at level ", level)
    foo(level+1)
    print("    "*level, "Leaving  test()", level)

def foo(level):
    print("    "*level, "Entering foo() at level ", level)
    bar(level+1)
    print("    "*level, "Leaving  foo() at level ", level)

def bar(level):
    print("    "*level, "Entering bar() at level ", level)
    print("    "*level, "Leaving  bar() at level ", level)

test(1)
```

Program 10: In each of the three functions, there is a parameter (level) and one local variable "a". So, there are four variables called "a" in the example. Is the "a" in the main code changed after returning from the function calls? What if I change the parameter level inside the function bar()? Will the value of para be changed?

```
04 24-local var.py
def test(level):
    print("    "*level, "Entering test() at level ", level)
    a = 100
    print("    "*level, "a = ", a)
    print("    "*level, "Leaving  test() at level ", level)

def foo(level):
    print("    "*level, "Entering foo()  at level ", level)
    a = 200
    print("    "*level, "a = ", a)
    print("    "*level, "Leaving  foo()  at level ", level)

def bar(level):
    print("    "*level, "Entering bar()  at level ", level)
    a = 300
    print("    "*level, "a = ", a)
    print("    "*level, "Leaving  bar()  at level ", level)

a, para = 0, 1
test(para)
foo(para)
bar(para)
print ("a = ", a)
```

Program 11: Composition of function calls.

```
04 25-composition.py
```

```python
def double(num):
    return 2*num

num = 9
num = double(num)       # value changed
print(num)
print(double(num))      # value *not* changed
num = double(double(num))     # value changed
print(num)
```

Program 12: A common question about return statements is: "Can I return more than one value?". Technically, you can return only one value, but it can be a value composed of several values. When the value is received, we can unpack it into several values. The first case shows how it works, and the second one fails.

```
04 26-return.py
```

```python
#  keyword: return value
#  Technically, a function returns one value, BUT
#  it can be a value that is made up of several values.
#
def next():
    input("Next> ")

def get_value():
    a = input("Enter a name: ")
    b = int(input("Enter an age: "))
    return a, b

print(f"Store the return value into ONE variable. {get_value()} as a
tuple")
name, age = get_value()   # unpack into two variables
print(f"\nUnpack the return value into two variables.  name = {name},
age = {age}")
next()

# The following code will cause an error.  Try it.
def test():
    return 1, 2, 3, 4

x, y = test()   # unpacking 4 values into two?
print(f"x = {x},   y = {y}.")
```

Program 13: This program is the only example using a triple-quote string. It shows two ways to get the information (docstring) from a function. The second way is easier to remember.

04 27-docstring.py

```
def test_function():
    """    This is a test function to demonstrate
    The use of docstring. :)
    We usually keep some vital information
    here for the user of this function. """
    print("test_function() called\n")

test_function()
print(test_function.__doc__)
help(test_function)
```

Program 14: This is one of a few recursive programs we are testing.

04 28-recursion.py

```
def test(level):
    print(" :    "*level, "Entering test() at level ", level)
    if level<5:
        test(level+1)
    print(" :    "*level, "Leaving  test() at level ", level)
    return 1

test(0)
```

Program 15: The association of the arguments with the parameters may also be done by specifying the parameter name instead of by position.

```
04 31-keyword argument.py
def print_info(name, height, weight):
    print(name, "is", height, "cm tall and weights", weight,
"pounds.")

print_info("Bob", 170, 135)
print_info("Bob", height=170, weight=135)
print_info("Bob", weight=135, height=170)
print_info(name="Bob", weight=135, height=170)

print("test1", "test2", sep='/')
print("test1", "test2", "test3", sep='/')
```

Program 16: Function composition combines two or more function calls such that the output of one function becomes the input of the following function.

```
04 32-composition.py
# Function composition is the way of combining two or more
# functions in such a way that the output of one function
# becomes the input of the second function.
#
print(abs(eval(input("Enter a value: "))))
# Inner-most first
```

Program 17: BMI calculation without using functions.

```
04 33-bmi.py
# Not using functions
weight = float(input("Enter weight [pounds]: "))
height = float(input("Enter height [inches]: "))
bmi = 703*weight/height**2
print("Your body mass index is:", bmi)
```

Program 18: Putting everything in a function. You can call the function using any name.

```
04 34-bmi.py
def main():
    weight = float(input("Enter weight [pounds]: "))
    height = float(input("Enter height [inches]: "))
    bmi = 703*weight/height**2
    print(f"Your body mass index is: {bmi:6.2f}")

main()
```

Program 19: Using one function to calculate the BMI.

04 35-bmi.py

```
def bmi(w, h):
    return(703*w/h**2)

weight = float(input("Enter weight [pounds]: "))
height = float(input("Enter height [inches]: "))
print(f"Your body mass index is: {bmi(weight, height):6.2f}")
```

Program 20: Using two functions.

04 36-bmi.py

```
def getNum(valueType, unit):
    return(eval(input("Enter "+valueType+" in "+unit+": ")))
# note that we are using eval()

def bmi():
    weight = getNum("weight", "pounds")
    height = getNum("height", "inches")
    bmi = 703*weight/height**2
    print(f"Your body mass index is: {bmi:6.2f}")

bmi()
```

Program 21: Using two functions in a different arrangement.

04 37-bmi.py

```
def cal_bmi(w, h):
    return 703*w/h**2

def getNum(valueType, unit):
    return(eval(input("Enter "+valueType+" in "+unit+": ")))

weight = getNum("weight", "pounds")
height = getNum("height", "inches")
bmi = cal_bmi(weight, height)
print(f"Your body mass index is: {bmi:6.2f}")
```

Program 22:

04 38-main.py

```python
def cal_bmi(w, h):
    return 703*w/h**2

def getNum(valueType, unit):
    return(eval(input("Enter "+valueType+" in "+unit+": ")))

def main():
    weight = getNum("weight", "pounds")
    height = getNum("height", "inches")
    yourBmi = cal_bmi(weight, height)
    print(f"Your body mass index is: {yourBmi:6.2f}")

main()
```

Program 23: Importing a library.

04 41-library.py

```python
import random

primes = [2, 3, 5, 7, 11, 13, 17, 19]
print(f"A randomly selected element: {random.choice(primes)}")
print(f"A randomly selected element: {random.choice(primes)}")
print(f"A random number between 0 and 1:  {random.random()}")
print(f"A random number between 0 and 1:  {random.random()}")
```

Program 24: Importing a library as.

04 42-import as.py

```python
# keyword: import, as, from
import random as ra

ra.seed(123)
primes = [2, 3, 5, 7, 11, 13, 17, 19]
for i in range(5):
    print(ra.choice(primes))

for i in range(5):
    print(ra.random())
```

Program 25: Importing selected functions of a library.

```
01 43-from import.py
```

```python
# from ... import ...
from random import choice, random

primes = [2, 3, 5, 7, 11, 13, 17, 19]
print(choice(primes))
print(random())
```