STRUCTURING PROGRAMS: FUNCTIONS

ES 112

Brief Recap

- Iteration patterns
- Structuring Programs: abstraction and decomposition

Menu for Today!

- Understanding Functions and Function Invocations

Getting Started on Functions

- Definition
  - *name*
  - *parameters (0 or more)*
  - *docstring (optional but recommended)*
  - *body*
  - *returns something (an object or* None*)*
- Invocation (or call)
- Only one definition; multiple invocations possible

Example Function

```
def power(x,y):
    '''Computes the yth power of x
       Assumes y is an integer, but does not do the type checking'''
    result = 1
    for i in range(0,y):
        result = result * x
    return result

power(19, 3)
power(10, 4)
```

Functions

- Function Defintion:

```
def <name of Function> (<list of formal parameters>):
    <body of the function>
```

- Function call

```
<name of function> (<list of actual parameters>)
```

What Happens When You Call A Function

- Evaluate actual parameters and bind formal parameters to the resulting values
- Move point of execution to the first statement of the function.
- Execute the code in the body until
  - *Either a return statement is encountered. In this case, the value of expression following* `return` *becomes the value of the function invocation*
  - *Or there are no more statements to execute. In this case, the value of the function invocation will be* None
- Point of execution is transferred back to the code immediately following the invocation

Execute by Hand!

```
def add(x, y):
    result = x+y
    return result
def mult(x, y):
    result = x * y
    print(result)
```

```
add(1,2 + 1)
print(add(2,3))
mult(3,4)
print(mult(4,5))
```

Invocation:
Binding Arguments to Parameters

```
def count_to_n(n):
    for i in range(1, n+1):
        print(i)

count_to_n(10)
count_to_n()
count_to_n(3,5)
count_to_n(3.5)
```

A function invocation binds the formal parameter to the value of the actual argument being passed at the time of invocation

Checking if a Number is a Prime

```
def isPrime(number):
    if number == 1:
        return(False)
    result = True
    for  factor in range(2, number // 2 + 1):
        if (number % factor == 0):
            result = False
            break
    return(result)
```

Twin Primes Revisited

```
a = int(input('Give me a number'))
b = int(input('Give me another number'))
diffABis2 = (a - b) == 2 or (b - a) == 2
if diffABis2 and isPrime(a) and isPrime(b):
    print(f'{a} and {b} are twin primes')
else:
    print(f'{a} and {b} are not twin primes')
```

Next Prime

```
a = int(input(‘Give me a number’))
b = a + 1
while not isPrime(b):
    b =  b + 1
print(f’The first prime after {a} is {b}’)
```

Functions help with Abstraction

```
def promptForIntegerInput():
    '''This function prompts the user for an input,
       casts the input as an integer, and
       returns the integer value'''
    data = input("Give me an integer")
    return int(data)

value1 = promptForIntegerInput()
value2 = promptForIntegerInput()
sum = value1 + value2
print("The sum of the two numbers is ", sum)
```

Suppose I wanted value1 and value2 to be between 0 and 100 only.
How would my input function change so that

- The prompt is clear
- We check for out of range values and report

Functions help with Abstraction

```
def promptForIntegerInput():
    '''This function prompts the user for an input between 0 and 100,
       casts the input as an integer, and
       returns the integer value'''
    data = int(input("Give me an integer between 0 and 100"))
    if (data < 0 or data > 100):
        print(f'{data} is outside the range')
        return(None)
    return data
value1 = promptForIntegerInput()
value2 = promptForIntegerInput()
if (value1 != None and value2 != None):
    sum = value1 + value2
    print("The sum of the two numbers is ", sum)
```

`return` vs `print`

- You can only do a `return` only from inside a function
- While there may be many `return` statements in a function, only one `return` will be executed
- The code inside the function but after `return` statement will not executed (like `break`)
- `return` has a value associated with it, which is given to function caller. If no value is given, None is returned

- You can call print anywhere in your program
- You can execute many print statements inside a function
- The code inside function after a print statement can be executed
- The value associated with the print is outputted to the console

Scope of a Variable

- The binding of an value to a formal parameter is limited to the body of the function

```
def myFunc(x):
    y = 1
    print(f'Inside the function, x = {x} and y = {y}')
    x = x + 1
    return(x)
x = 2
y = 6
print("Before the function call, x = ", x)
z = myFunc(y)
print("After the function call, x = ", x)
print("After the function call, y = ", y)
print("After the function call, z = ", z)
```

How Do Variables Get Their Value Inside Functions

- Bindings are maintained in a space in memory called a “stack”
- Whenever a new binding is created, it is stored at the top of the stack
- When a function is called, a marker is written at the top of the stack to indicate that we are entering a function call
  - *All variable binding used in the function are written on top of this marker when the function is called. This is called a stack frame*
  - *Initially all variables in a function are marked as unbound*
  - *Inside a function, we look for the variable value in the stack frame of that function*
  - *When we written from a call, the stack frame is removed or “popped”*
- Each stack frame is called a “scope”

How Do Variables Get Their Value Inside Functions