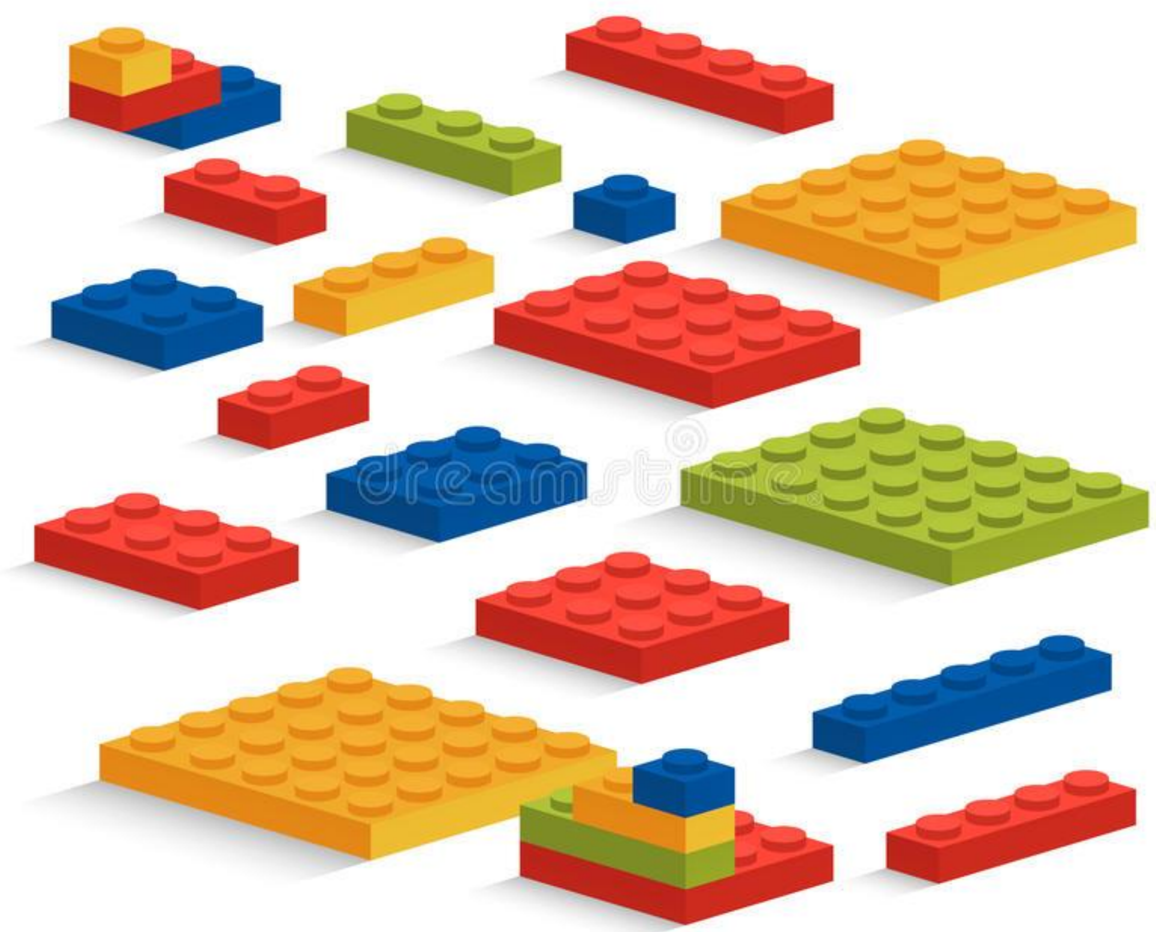
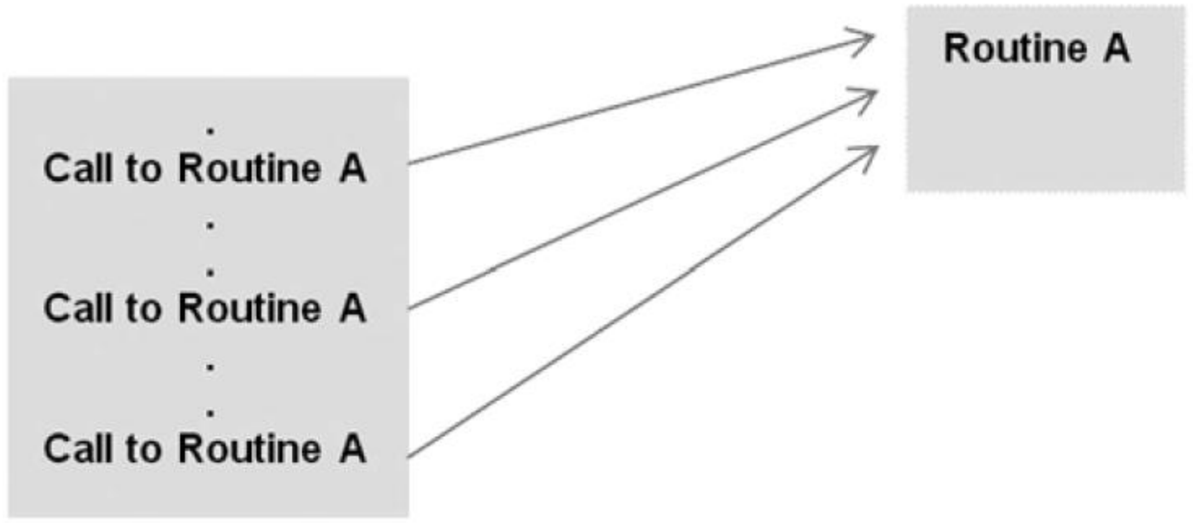
Functions

Functions are building blocks

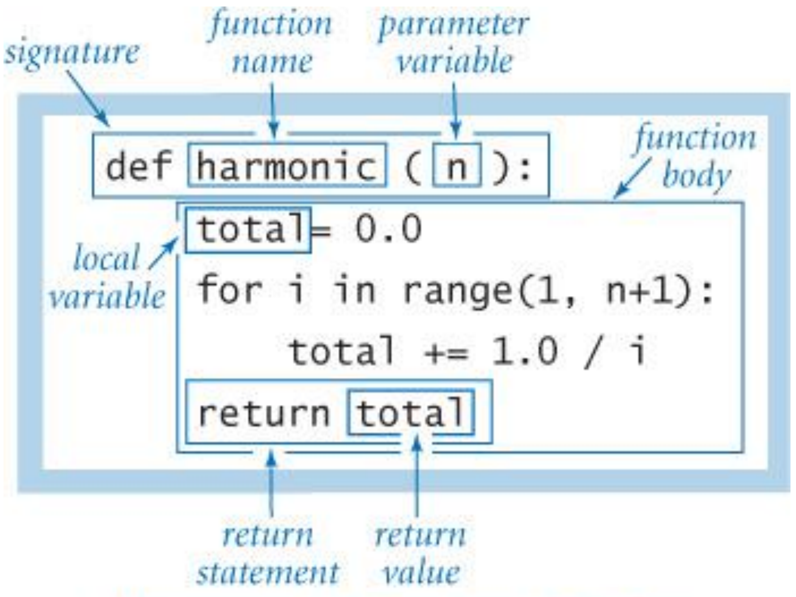


Routine

- A program **routine** is a named group of instructions that accomplishes some task.
- A routine maybe **invoked (called)** as many times as needed in a given program.
- A function is Python's version of a program routine.



Defining functions



```
def harmonic ( n ):
    total = 0.0
    for i in range(1, n+1):
        total += 1.0 / i
    return total
```

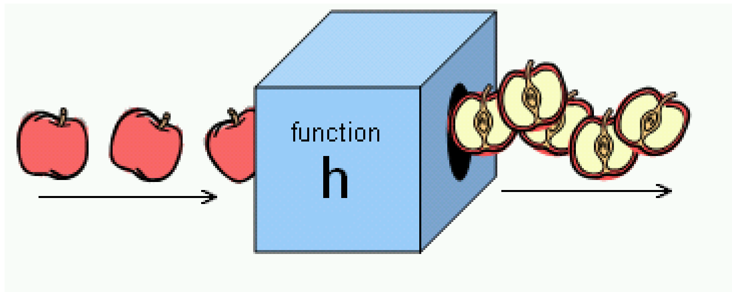
Anatomy of a function definition

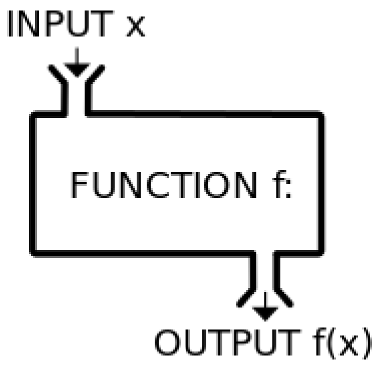
Actual arguments, or simply “arguments,” are the values passed to functions to be operated on.
Formal parameters, or simply “parameters,” are the “placeholder” names for the arguments passed.

"EVERY FUNCTION MUST BE DEFINED BEFORE IT IS CALLED"

Value returning functions

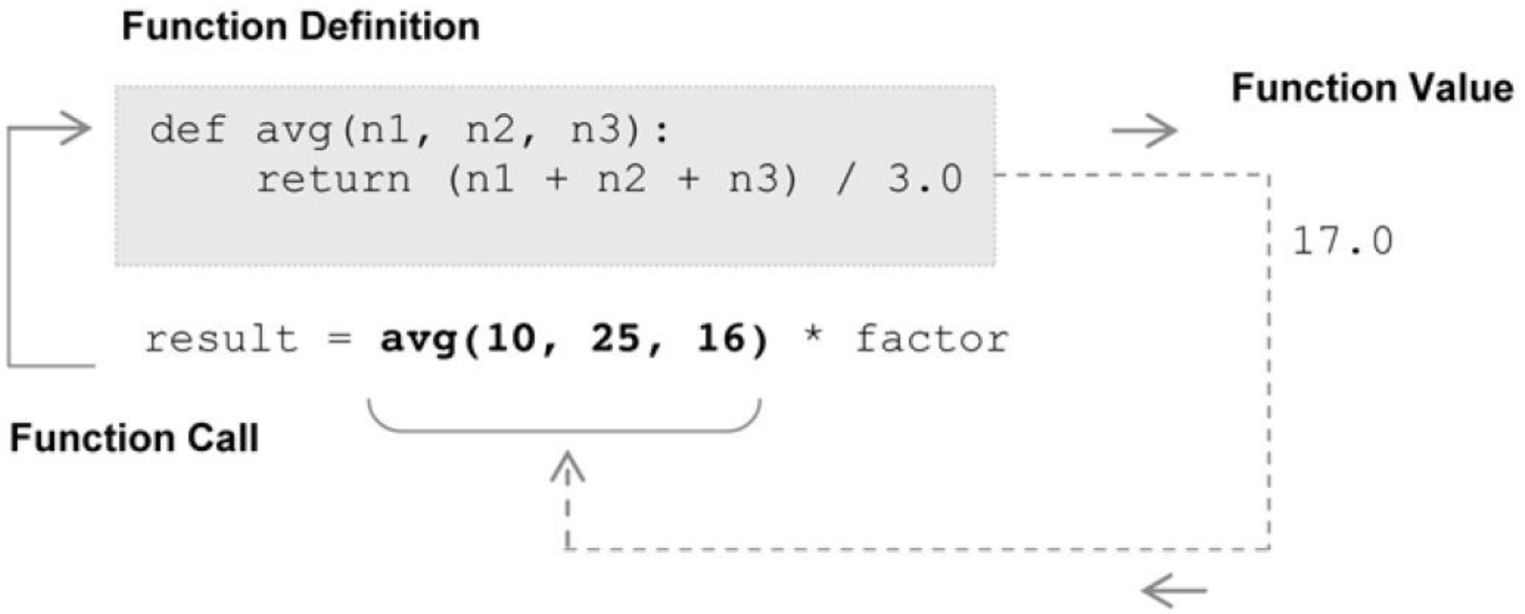
- A **value-returning function** in Python is a program routine called for its return value, and is therefore similar to a mathematical function.





return *expr*

Value returning functions



```
def avg(n1, n2, n3):
    return (n1 + n2 + n3) / 3.0

result = avg(10, 25, 16) * factor
```

LET'S TRY IT

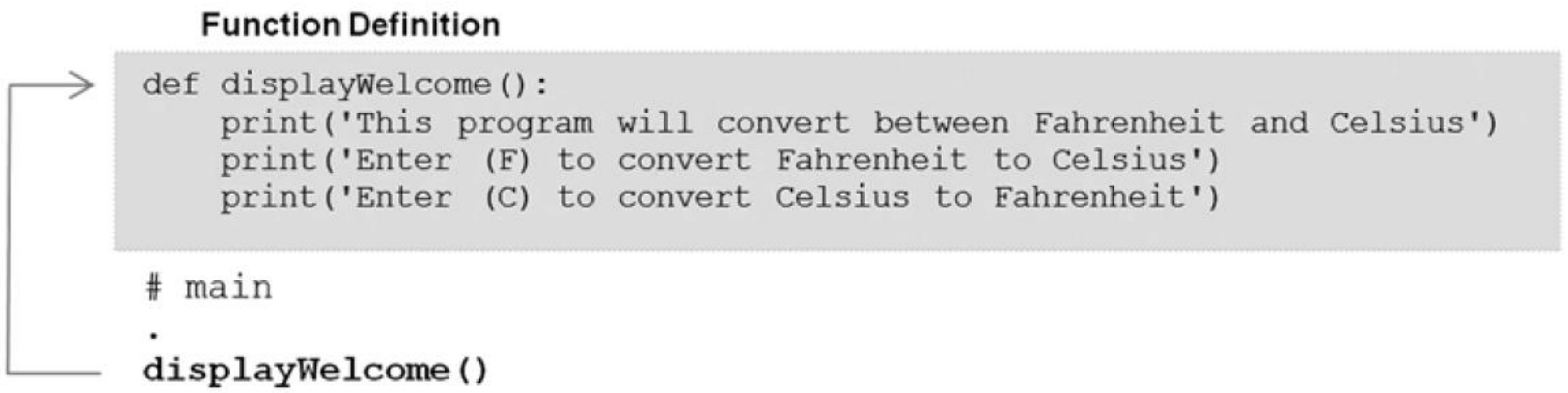
From the Python Shell, first enter the following function, making sure to indent the code as given. Hit return twice after the last line of the function is entered. Then enter the following function calls and observe the results.

```
>>> def avg(n1, n2, n3):
        return (n1 + n2 + n3) / 3.0

>>> avg(10, 25, 40)
???
```

```
>>> avg(40, 10, 25)
???

>>> avg(40, 25, 10)
???
```

Non-value returning functions

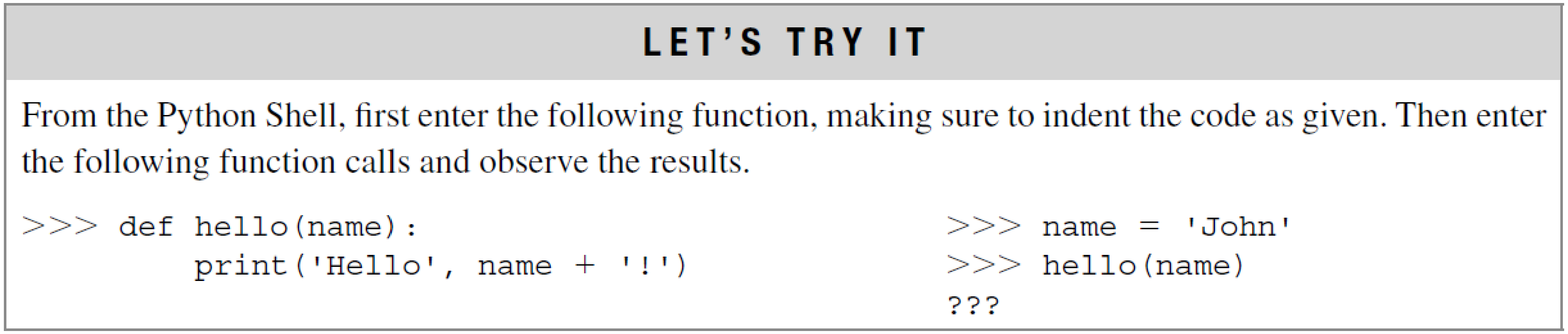
- A **non-value-returning function** is a function called for its *side effects*, and not for a returned function value.



```
def displayWelcome():
    print('This program will convert between Fahrenheit and Celsius')
    print('Enter (F) to convert Fahrenheit to Celsius')
    print('Enter (C) to convert Celsius to Fahrenheit')

# main
.
displayWelcome()
```

ANY FUNCTION THAT DOES NOT EXPLICITLY RETURN A FUNCTION VALUE (VIA A RETURN STATEMENT) AUTOMATICALLY RETURNS THE SPECIAL VALUE NONE

Non-value returning functions



LET'S TRY IT

From the Python Shell, first enter the following function, making sure to indent the code as given. Then enter the following function calls and observe the results.

```
>>> def hello(name):
        print('Hello', name + '!')
```

```
>>> name = 'John'
>>> hello(name)
???
```

Calling value returning functions

- Function calls to value-returning functions can be used anywhere that a function’s return value is appropriate.

```
result = max(num_list) * 100
```

```
(a) result = max(num_list1) * max(num_list2)
(b) result = abs(max(num_list))
(c) if max(num_list) < 10:...
(d) print('Largest value in num_list is ', max(num_list))
```

Calling value returning functions

- For returning more than one value a single tuple can be used

function definition

```
def maxmin(num_list):
    return (max(num_list), min(num_list))
```

function use

```
weekly_temps = [45, 30, 52, 58, 62, 48, 49]
```

```
(a) highlow_temps = maxmin(weekly_temps)
(b) high, low = maxmin(weekly_temps)
```

In (a) above, the returned tuple is assigned to a single variable, `highlow_temps`. Thus, `highlow_temps[0]` contains the maximum temperature, and `highlow_temps[1]` contains the minimum temperature. In (b), however, a *tuple assignment* is used. In this case, variables `high` and `low` are each assigned a value of the tuple based on the order that they appear. Thus, `high` is assigned to the tuple value at index 0, and `low` the tuple value at index 1 of the returned tuple.

LET'S TRY IT

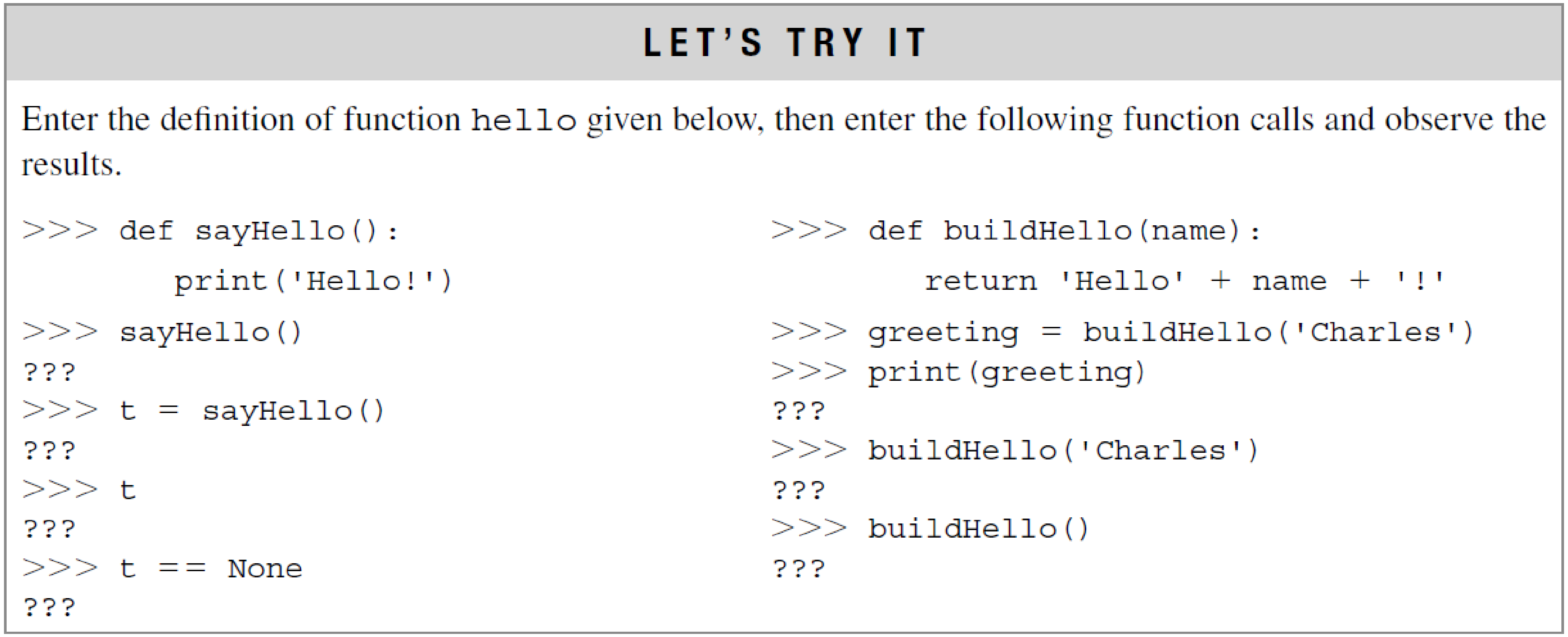
Enter the definitions of functions `avg` (from section 5.1.2) and `minmax` given above. Then enter the following function calls and observe the results.

```
>>> avg(10,25,40)
???

>>> avg(10,25,40) + 10
???

>>> if avg(10,25,-40) < 0:
        print 'Invalid avg'
???

>>> avg(avg(2,4,6),8,12)
???

>>> avg(1,2,3) * avg(4,5,6)
???
```

```
>>> num_list = [10,20,30]

>>> max_min = maxmin(num_list)
>>> max_min[0]
???

>>> max_min[1]
???

>>> max, min = maxmin(num_list)
>>> max
???

>>> min
???
```

Calling Non-Value-Returning Functions

- Function calls to non-value-returning functions can be used anywhere that an executable statement is allowed.



LET'S TRY IT

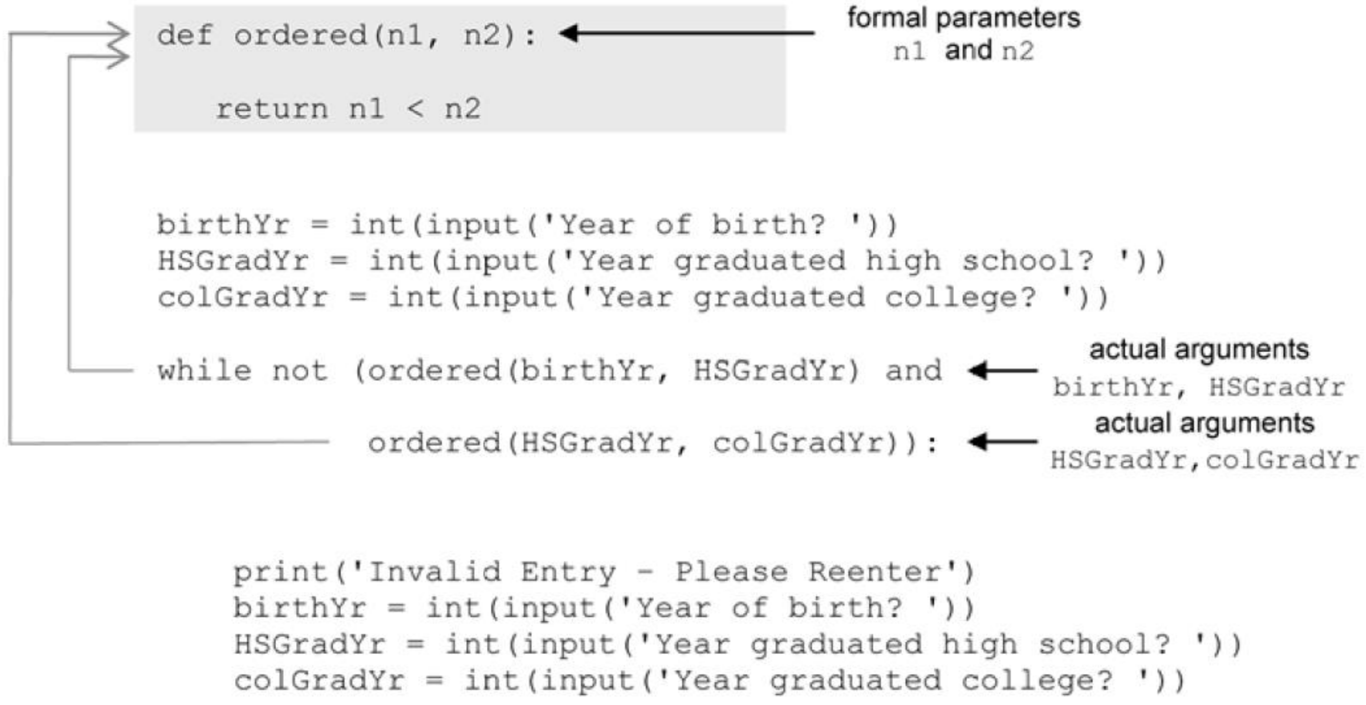
Enter the definition of function `hello` given below, then enter the following function calls and observe the results.

```
>>> def sayHello():
         print('Hello!')
>>> sayHello()
???
>>> t = sayHello()
???
>>> t
???
>>> t == None
???
```

```
>>> def buildHello(name):
         return 'Hello' + name + '!'
>>> greeting = buildHello('Charles')
>>> print(greeting)
???
>>> buildHello('Charles')
???
>>> buildHello()
???
```

Parameter passing

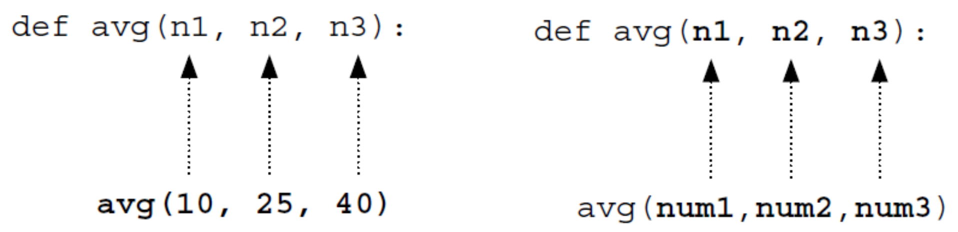
- The correspondence of actual arguments and formal parameters is determined by the *order* of the arguments passed, and not their names.



```
def ordered(n1, n2):

   return n1 < n2


birthYr = int(input('Year of birth? '))
HSGradYr = int(input('Year graduated high school? '))
colGradYr = int(input('Year graduated college? '))

while not (ordered(birthYr, HSGradYr) and
           ordered(HSGradYr, colGradYr)):

    print('Invalid Entry - Please Reenter')
    birthYr = int(input('Year of birth? '))
    HSGradYr = int(input('Year graduated high school? '))
    colGradYr = int(input('Year graduated college? '))
```

formal parameters n1 and n2
actual arguments birthYr, HSGradYr
actual arguments HSGradYr,colGradYr

Mutable and Inmutable



```
def countDown(n):
    while n >= 0:
        if (n != 0):
            print(n, '..', end='')
        else:
            print(n)
        n = n - 1
```

```
>>> num_tics = 10
>>> countDown(num_tics)
>>> num_tics
???
```

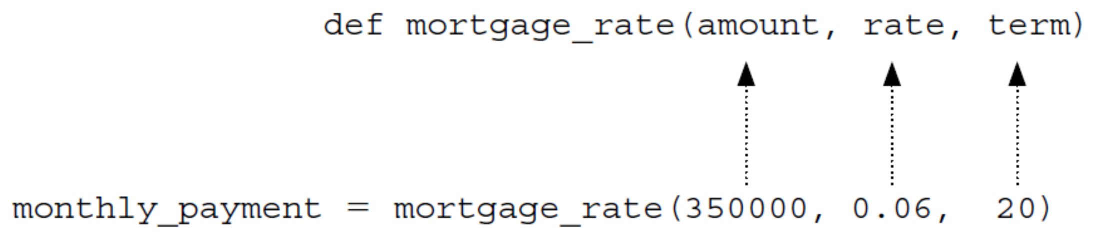
```
def sumPos(nums):
    for k in range(0, len(nums)):
        if nums[k] < 0:
            nums[k] = 0

    return sum(nums)
```

```
>>> nums_1 = [5, -2, 9, 4, -6, 1]
>>> total = sumPos(nums_1)
>>> total
19
>>> nums_1
[5,0,9,4,0,1]
```

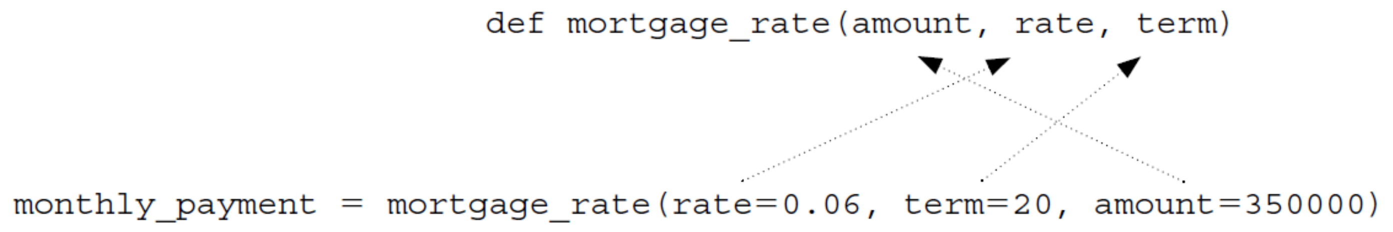
- Only arguments of mutable type can be altered when passed as an argument to a function. In general, function results should be through a function’s return value, and not through altered parameters.

Keyword Arguments in Python

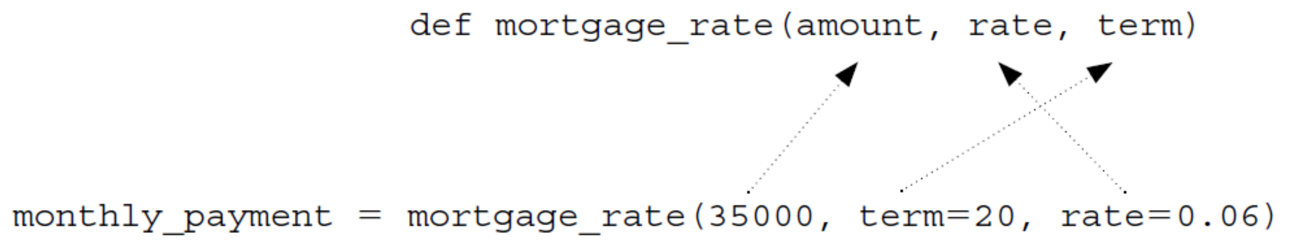
- A **positional argument** is an argument that is assigned to a particular parameter based on its position in the argument list.
- A **keyword argument** is an argument that is specified by parameter name.



```
def mortgage_rate(amount, rate, term)

monthly_payment = mortgage_rate(350000, 0.06,  20)
```



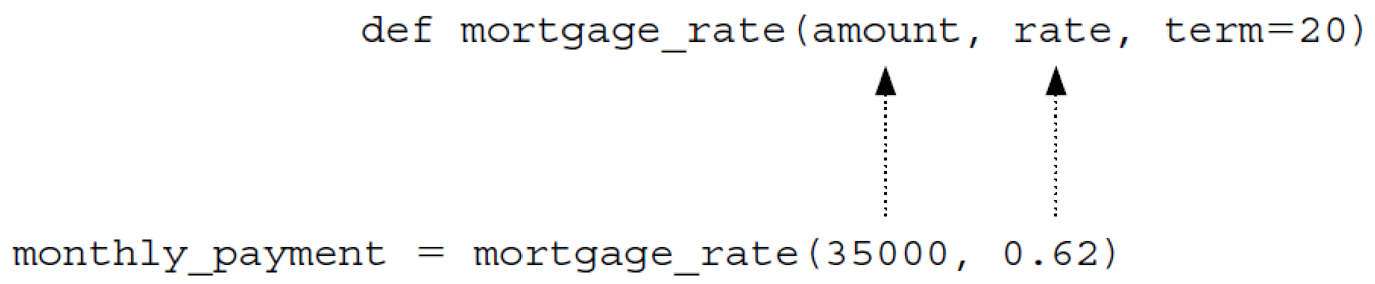
```
def mortgage_rate(amount, rate, term)

monthly_payment = mortgage_rate(rate=0.06, term=20, amount=350000)
```



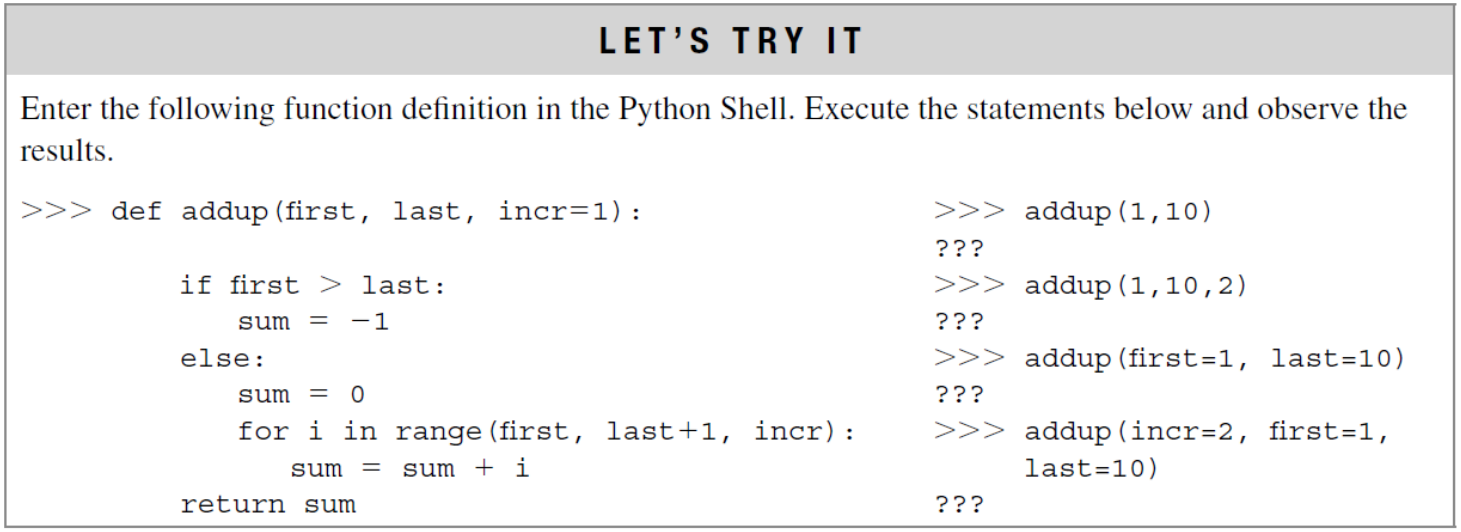
```
def mortgage_rate(amount, rate, term)

monthly_payment = mortgage_rate(35000, term=20, rate=0.06)
```

Default arguments

- A **default argument** is an argument that can be optionally provided in a given function call. When not provided, the corresponding parameter provides a default value.



```
def mortgage_rate(amount, rate, term=20)

monthly_payment = mortgage_rate(35000, 0.62)
```



LET'S TRY IT

Enter the following function definition in the Python Shell. Execute the statements below and observe the results.

```
>>> def addup(first, last, incr=1):

        if first > last:
            sum = -1
        else:
            sum = 0
            for i in range(first, last+1, incr):
                sum = sum + i
        return sum
```

```
>>> addup(1,10)
???
>>> addup(1,10,2)
???
>>> addup(first=1, last=10)
???
>>> addup(incr=2, first=1,
    last=10)
???
```