Functions

Python Functions

A *function* is a piece of code that performs a task of some kind.

- A function has a name that is used when we need for the task to be executed. Asking that the task be executed is referred to as "calling" the function.

- Some functions need one or more pieces of input when they are called. Others do not.

- Some functions give back a value; others do not. If a function gives back a value, this is referred to as "returning" the value.

Functions

- To write programs that use functions to reduce code duplication and increase program modularity.
- Imagine the effort needed to develop and debug software of that size. It certainly cannot be implemented by any one person, it takes a team of programmers to develop such a project.

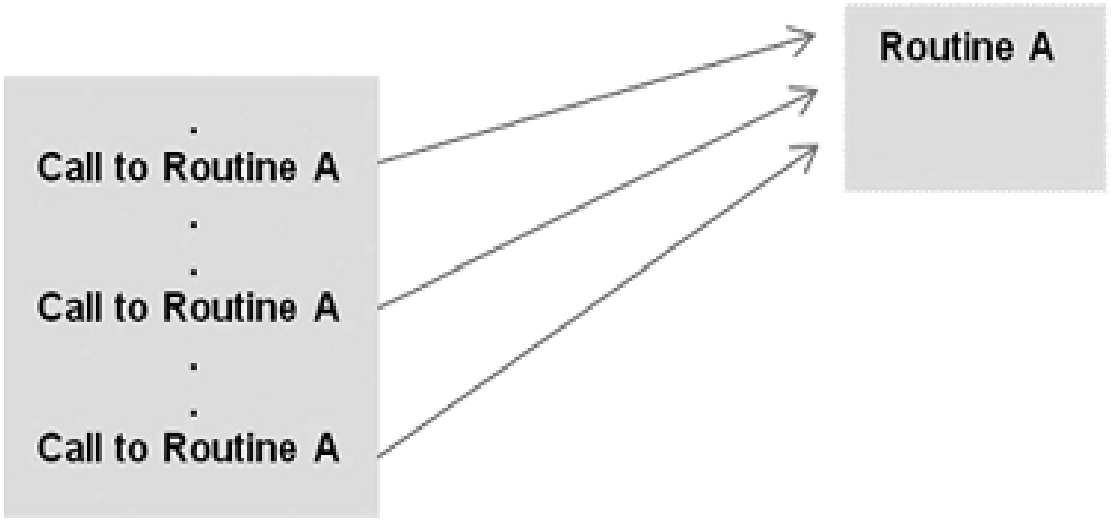
Term	Number of Lines of Code (LOC)	Equivalent Storage
KLOC	1,000	Application programs
MLOC	1,000,000	Operating systems / smart phones
GLOC	1,000,000,000	Number of lines of code in existence for various programming languages

Functions Contd…

- In order to manage the complexity of a large problem, it is broken down into smaller sub problems. Then, each sub problem can be focused on and solved separately.
- In programming, we do the same thing. Programs are divided into manageable pieces called *program routines* (or simply *routines*).
- In addition, program routines provide the opportunity for code reuse, so that systems do not have to be created from “scratch.”

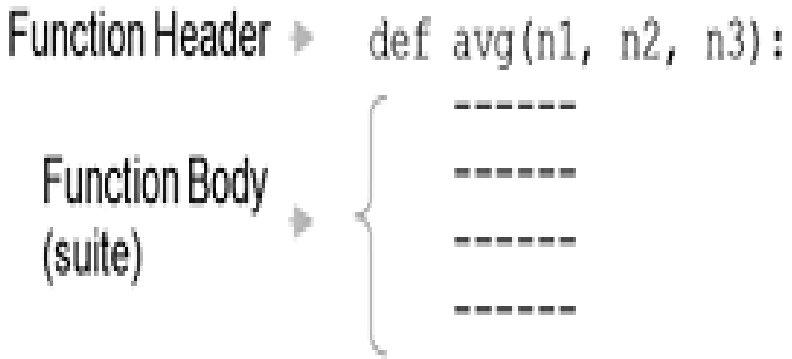
What Is a Function Routine?

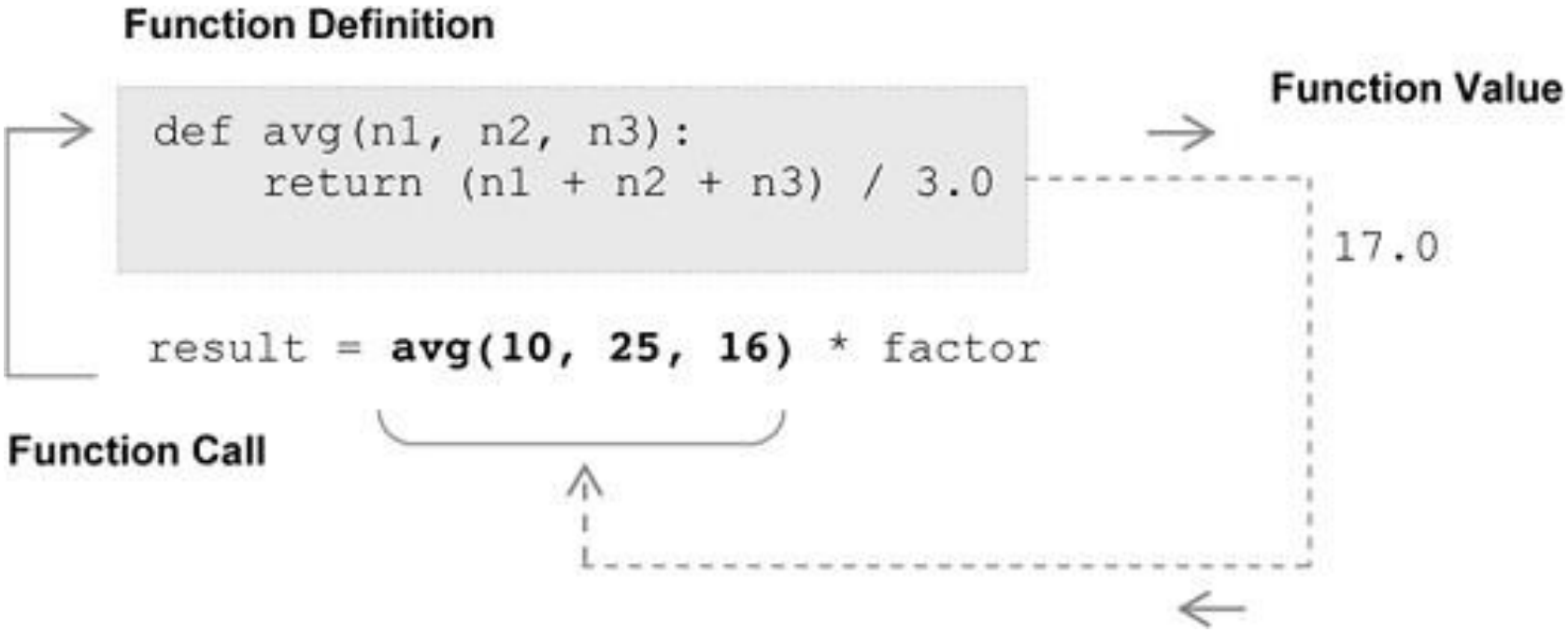
- A **routine** is a named group of instructions performing some task. A routine can be **invoked** (*called*) as many times as needed in a given program
- When a routine terminates, execution automatically returns to the point from which it was called. Such routines may be predefined in the programming language, or designed and implemented by the programmer.



Defining Functions

- In addition to the built-in functions of Python, there is the capability to define new functions. Such functions may be generally useful, or specific to a particular program. The elements of a function definition are given




Function Definition
def avg(n1, n2, n3):
return (n1 + n2 + n3) / 3.0
Function Value
17.0
result = avg(10, 25, 16) * factor
Function Call

Defining Functions Contd…

- The number of items in a parameter list indicates the number of values that must be passed to the function, called **actual arguments** (or simply “arguments”), such as the variables num1, num2,and num3 below.

```
>>> num1 = 10
>>> num2 = 25
>>> num3 = 16

>>> avg(num1,num2,num3)
```

- Functions are generally defined at the top of a program. However, *every function must be defined before it is called*.

Parameters

- **Actual parameters**, or simply “arguments,” are the values passed to functions to be operated on.

- **Formal parameters**, or simply the “placeholder” names for the arguments passed.

Factorial

```
def factorial(n):
    f=1
    for i in range(1,n+1):
        f=f*i
    return f


print(factorial(4))

print(factorial(12))
print(factorial(14))
print(factorial(24))
```

Check for prime number

```
def isPrime(n):
    for i in range(2,int(n**0.5)+1):
        if n%i==0:
            return 0

    return 1
```

Assignment Statements Recap

Table 11-1. Assignment statement forms

Operation	Interpretation
`spam = 'Spam'`	Basic form
`spam, ham = 'yum', 'YUM'`	Tuple assignment (positional)
`[spam, ham] = ['yum', 'YUM']`	List assignment (positional)
`a, b, c, d = 'spam'`	Sequence assignment, generalized
`a, *b = 'spam'`	Extended sequence unpacking (Python 3.X)
`spam = ham = 'lunch'`	Multiple-target assignment
`spams += 42`	Augmented assignment (equivalent to `spams = spams + 42`)

Example function

```
def intersect(seq1, seq2):
        res = []                # Start empty
        for x in seq1:          # Scan seq1
                if x in seq2:       # Common item?
                res.append(x)       # Add to end
return res

>>> s1 = "SPAM"
>>> s2 = "SCAM"
>>> intersect(s1, s2)           # Strings
['S', 'A', 'M']
```

Scope of Variables

- enclosing module is a global scope
- global scope spans a single file only
- Assigned names are local unless declared global or nonlocal
- Each call to a function creates a new local scope

Name Resolution: The LEGB Rule

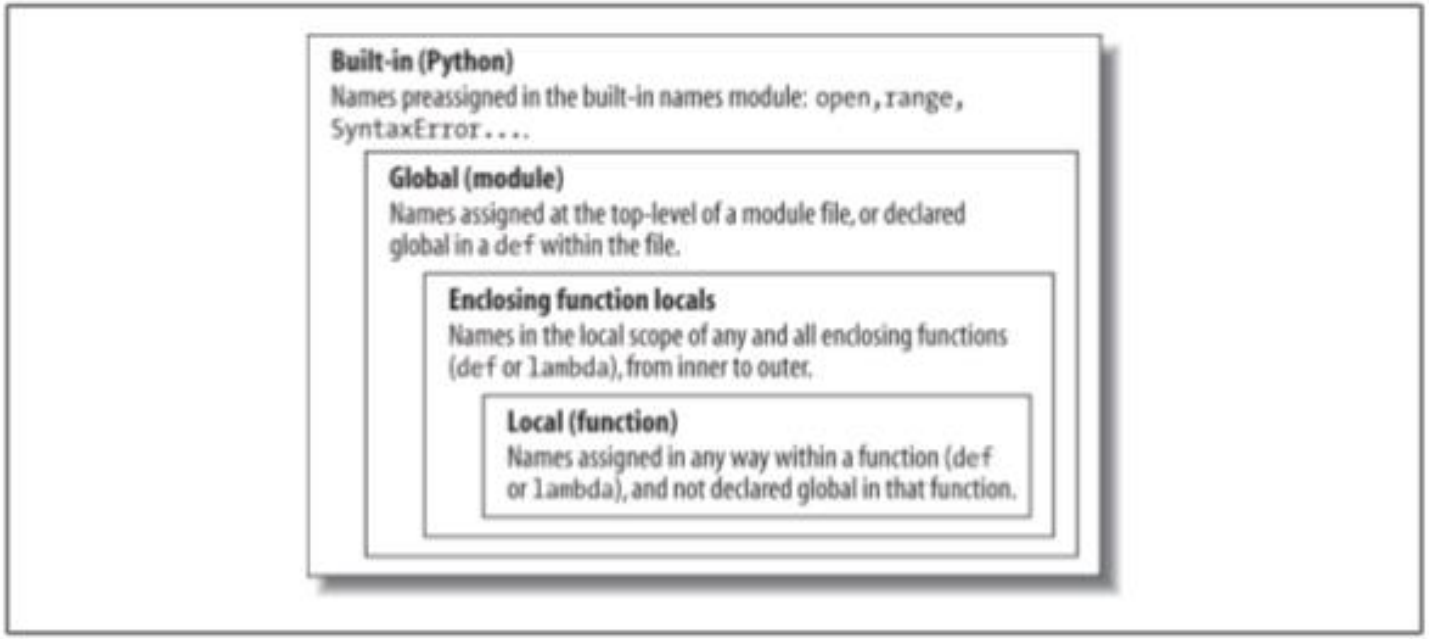


Figure 17-1. The LEGB scope lookup rule. When a variable is referenced, Python searches for it in this order: in the local scope, in any enclosing functions' local scopes, in the global scope, and finally in the built-in scope. The first occurrence wins. The place in your code where a variable is assigned usually determines its scope. In Python 3.X, nonlocal declarations can also force names to be mapped to enclosing function scopes, whether assigned or not.

Scope Example

```
    # Global scope
X = 99
    # X and func assigned in module: global

def func(Y):
        # Y and Z assigned in function: locals
        # Local scope
        Z = X + Y       # X is a global
return Z
func(1)             # func in module: result=100
```

Scope Example

- Global names: X, func
- Local names: Y, Z

Scope Example

```
X = 88                          # Global X
def func():
        X = 99
# Local X: hides global
func()
print(X)                        # Prints 88: unchanged
```

Accessing Global Variables

```
X = 88                      # Global X
def func():
    global X
    X = 99                  # Global X: outside def
func()
print(X)                    # Prints 99
```

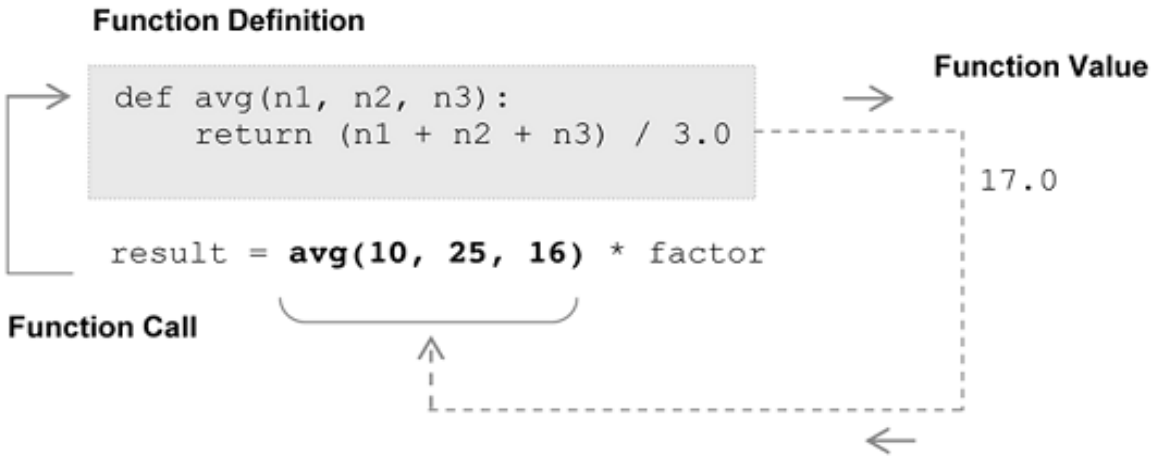
Global Variables and Global Scope

- *The use of global variables is generally considered to be bad programming style*. Although it provides a convenient way to share values among functions, *all* functions within the scope of a global variable can access and alter it. This may include functions that have no need to access the variable, but none-the-less may unintentionally alter it.

- Another reason that the use of global variables is bad practice is related to code reuse. If a function is to be reused in another program, the function will not work properly if it is reliant on the existence of global variables that are nonexistent in the new program. Thus, it is good programming practice to design functions so all data needed for a function (other than its local variables) are explicitly passed as arguments, and not accessed through global variables.

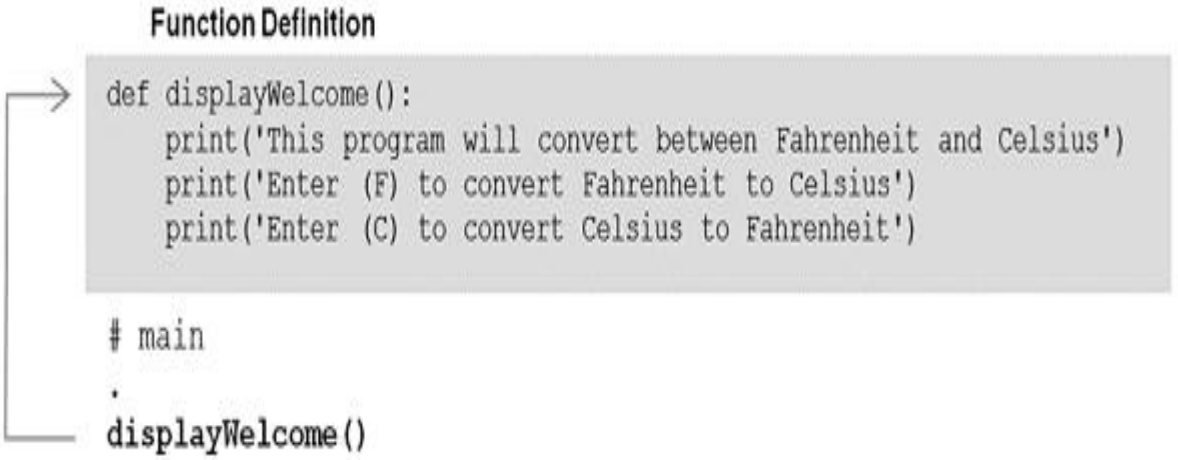
Value-Returning Functions

- A **value-returning function** is a program routine called for its return value, and is therefore similar to a mathematical function.
- Function avg takes three arguments (n1, n2, and n3) and returns the average of the three.
- The *function call* avg(10, 25, 16), therefore, is an expression that evaluates to the returned function value.
- This is indicated in the function's *return statement* of the form return *expr*,where *expr* may be any expression.



Non-Value-Returning Functions

- A **non-value-returning function** is called not for a returned value, but for its *side effects*.
- A **side effect** is an action other than returning a function value, such as displaying output on the screen.



- In this example, function display Welcome is called only for the side-effect of the screen output produced.

```
# Temperature Conversion Program (Celsius-Fahrenheit / Fahrenheit-Celsius)

def displayWelcome():

   print('This program will convert a range of temperatures')
   print('Enter (F) to convert Fahrenheit to Celsius')
   print('Enter (C) to convert Celsius to Fahrenheit\n')

def getConvertTo():

   which = input('Enter selection: ')
   while which != 'F' and which != 'C':
      which = input('Enter selection: ')

   return which

def displayFahrenToCelsius(start, end):

    print('\n  Degrees', ' Degrees')
    print('Fahrenheit', 'Celsius')

    for temp in range(start, end + 1):
       converted_temp = (temp - 32) * 5/9
       print('  ', format(temp, '4.1f'), '    ', format(converted_temp, '4.1f'))

def displayCelsiusToFahren(start, end):

    print('\n  Degrees', ' Degrees')
    print('  Celsius', 'Fahrenheit')

    for temp in range(start, end + 1):
       converted_temp = (9/5 * temp) + 32
       print('  ', format(temp, '4.1f'), '    ', format(converted_temp, '4.1f'))

# ---- main

# Display program welcome
displayWelcome()

# Get which converion from user
which = getConvertTo()

# Get range of temperatures to convert
temp_start = int(input('Enter starting temperature to convert: '))
temp_end = int(input('Enter ending temperature to convert: '))

# Display range of converted temperatures
if which == 'F':
   displayFahrenToCelsius(temp_start, temp_end)
else:
   displayCelsiusToFahren(temp_start, temp_end)
```

Returning Multiple Values

- >>> def multiple(x, y):
        x = 2 # Changes local names only
        y = [3, 4]
    return x, y

Return multiple new values in a tuple

>>> X = 1

>>> L = [1, 2]

>>> X, L = multiple(X, L)

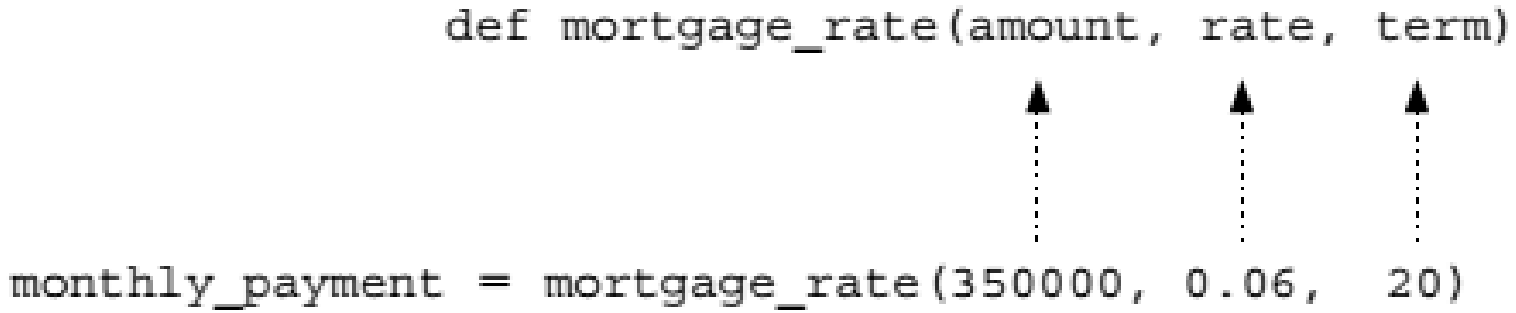
Assign results to caller's names

>>> X, L

(2, [3, 4])

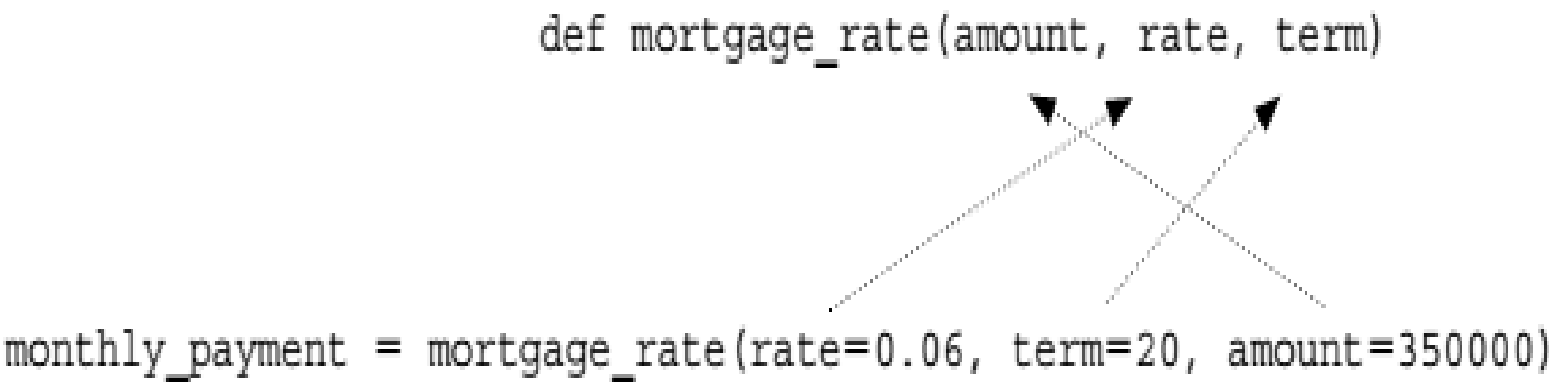
Positional Arguments in Python

- The functions we have looked at so far were called with a fi xed number of positional arguments.
  A **positional argument** is an argument that is assigned to a particular parameter based on its position in the argument list, as illustrated below.



Keyword Arguments in Python Contd..

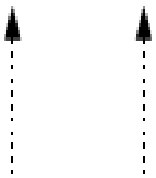
- Python provides the option of calling any function by the use of keyword arguments. A **keyword argument** is an argument that is specified by parameter name, rather than as a positional argument as shown below



Default Arguments in Python

- Python also provides the ability to assign a default value to any function parameter allowing for the use of default arguments.
- A **default argument** is an argument that can be optionally provided,

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



```
monthly_payment = mortgage_rate(35000, 0.62)
```

- In this case, the third argument in calls to function mortgage_rate is optional. If omitted, parameter term will default to the value 20 (years) as shown. If, on the other hand, a third argument is provided, the value passed replaces the default parameter value.

Keyword and Default Examples

- >>> def f(a, b, c):
print(a, b, c)

```
>>> f(1, 2, 3)
1 2 3
>>> f(c=3, b=2, a=1)
1 2 3
>>> f(1, c=3, b=2)
# a gets 1 by position, b and c passed by name 1 2 3
```

Defaults

```
>>> def f(a, b=2, c=3):
                print(a, b, c)
# a required, b and c optional
>>> f(1)                # Use defaults
1 2 3
>>> f(a=1)
1 2 3
>>> f(1, 4)             # Override defaults
1 4 3
>>> f(1, 4, 5)
1 4 5
```

Arbitrary Arguments Examples

- and **, are designed to support functions that take any number of arguments
- Both can appear in either the function definition or a function call, and they have related purposes in the two locations.
- Use of ‘*’
- collects unmatched positional arguments into a tuple:

```
>>> def f(*args):
        print(args)
```

Arbitrary Arguments Examples

- >>> f()

()

- >>> f(1)

(1,)

>>> f(1, 2, 3, 4)

(1, 2, 3, 4)

Arbitrary Arguments Examples

- ** feature is similar, but it only works for keyword arguments—it collects them into a new dictionary

```
>>> def f(**args):
                print(args)
>>> f()
{}
>>> f(a=1, b=2)
{'a': 1, 'b': 2}
```

Arbitrary Arguments Examples

```
>>> def f(a, *pargs, **kargs):
        print(a, pargs, kargs)

>>> f(1, 2, 3, x=1, y=2)
1 (2, 3) {'y': 2, 'x': 1}
```

```
def func(*t):
    print(t)

def func2(**d):
    print(d)

def func3(k):
    print(k)

func(1,2,3,'ABCD',5.7,"Z")

func2(a=1,b=2,c=[4,5,6],f="ABCD",g=89.6)


dictv={121:["ABC",45,46,57],122:["DEF",89,90,98],123:["DEF",79,70,78],124:["DEF",99,92,88]}
func3(dictv)
```

Calls: Unpacking arguments

```
>>> def func(a, b, c, d):
        print(a, b, c, d)
>>> args = (1, 2)
>>>args += (3, 4)
>>> func(*args)
# Same as func(1, 2, 3, 4)
1 2 3 4
```

Calls: Unpacking arguments

```
>>> args = {'a': 1, 'b': 2, 'c': 3}

>>> args['d'] = 4

>>> func(**args)

# Same as func(a=1, b=2, c=3, d=4)

1 2 3 4
```

Table 18-1. Function argument-matching forms

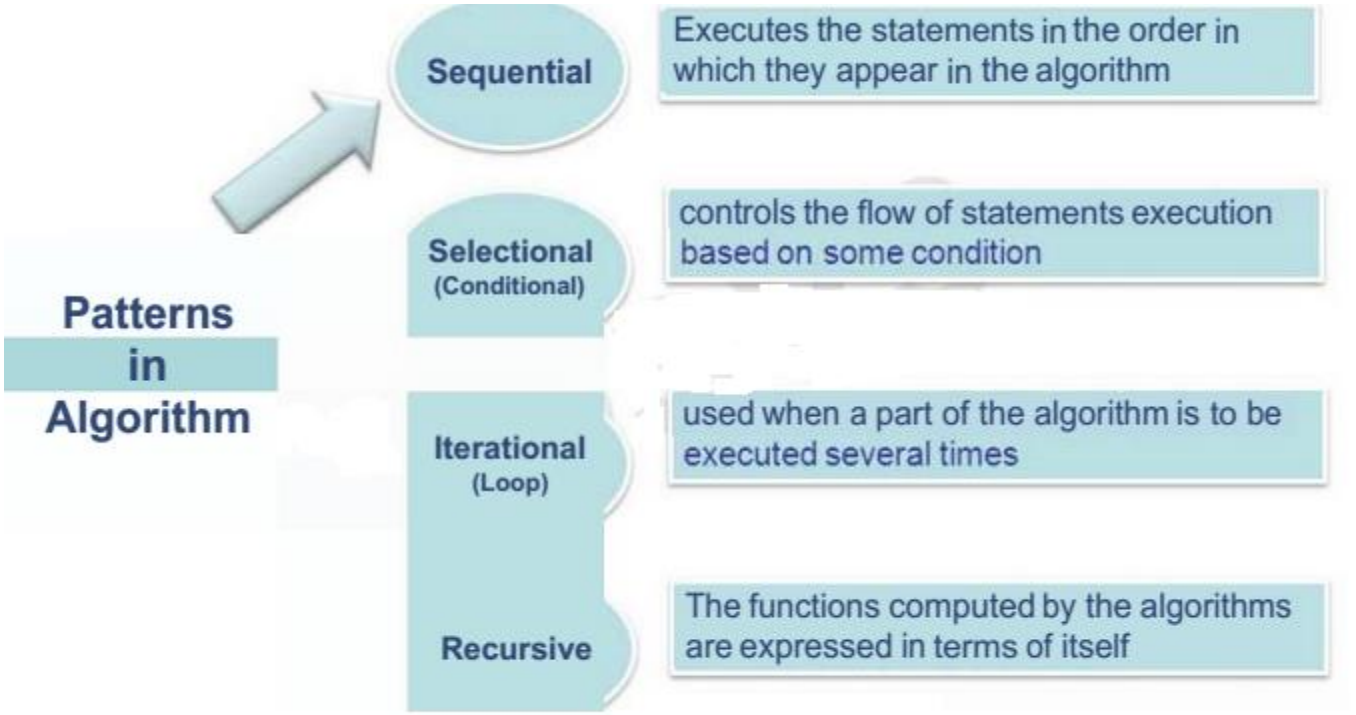
Syntax	Location	Interpretation
`func(value)`	Caller	Normal argument: matched by position
`func(name=value)`	Caller	Keyword argument: matched by name
`func(*iterable)`	Caller	Pass all objects in *`iterable`* as individual positional arguments
`func(**dict)`	Caller	Pass all key/value pairs in *`dict`* as individual keyword arguments
`def func(name)`	Function	Normal argument: matches any passed value by position or name
`def func(name=value)`	Function	Default argument value, if not passed in the call
`def func(*name)`	Function	Matches and collects remaining positional arguments in a tuple
`def func(**name)`	Function	Matches and collects remaining keyword arguments in a dictionary
`def func(*other, name)`	Function	Arguments that must be passed by keyword only in calls (3.X)
`def func(*, name=value)`	Function	Arguments that must be passed by keyword only in calls (3.X)

```
def func(*t):
    print(t)

def func2(**d):
    print(d)


func(1,2,3,'ABCD',5.7,"Z")

func2(a=1,b=2,c=[4,5,6],f="ABCD",g=89.6)
```

Exercises

- Compute area of circle using all possible function prototypes.
- Compute Simple interest for given principle(P), number of years(N) and rate of interest(R). If R value is not given then consider R value as 10.5%. Use keyword arguments for the same.

Different patterns in Algorithm



MOTIVATION-Recursion

- Almost all computation involves the repetition of steps. Iterative control statements, such as the for and while statements, provide one means of controlling the repeated execution of instructions. Another way is by the use of *recursion*.

Recursive algorithms

- In *recursive problem solving*, a problem is repeatedly broken down into similar sub problems, until the sub problems can be directly solved without further breakdown.

Recursive algorithms

- Recursive algorithms
  - The functions computed by the algorithms are expressed in terms of *itself*
  - Example

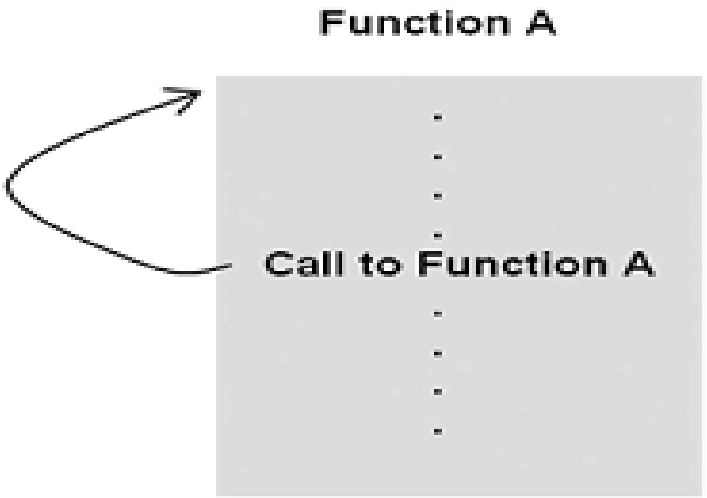
  Task: Find the Factorial of a positive integer
  Algorithm:

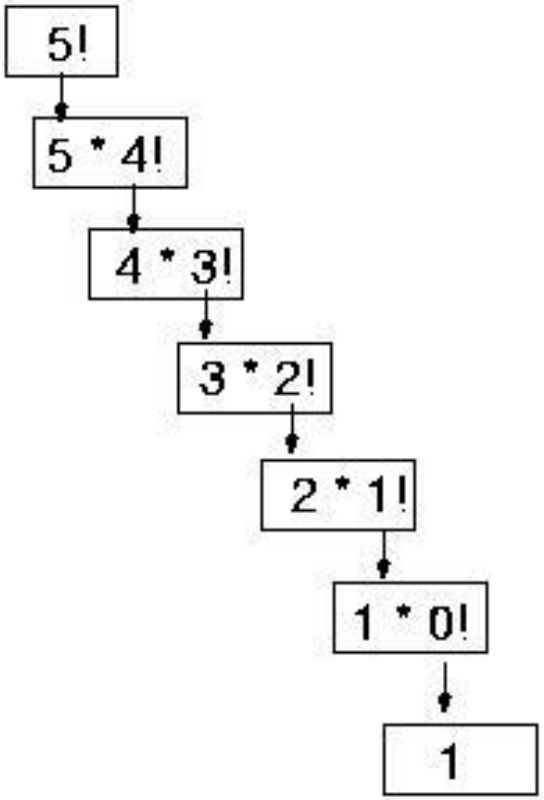
  Algorithm Factorial(n)

Begin
if (n=0) than return 1:
else return(n * Factorial(n-1))
End

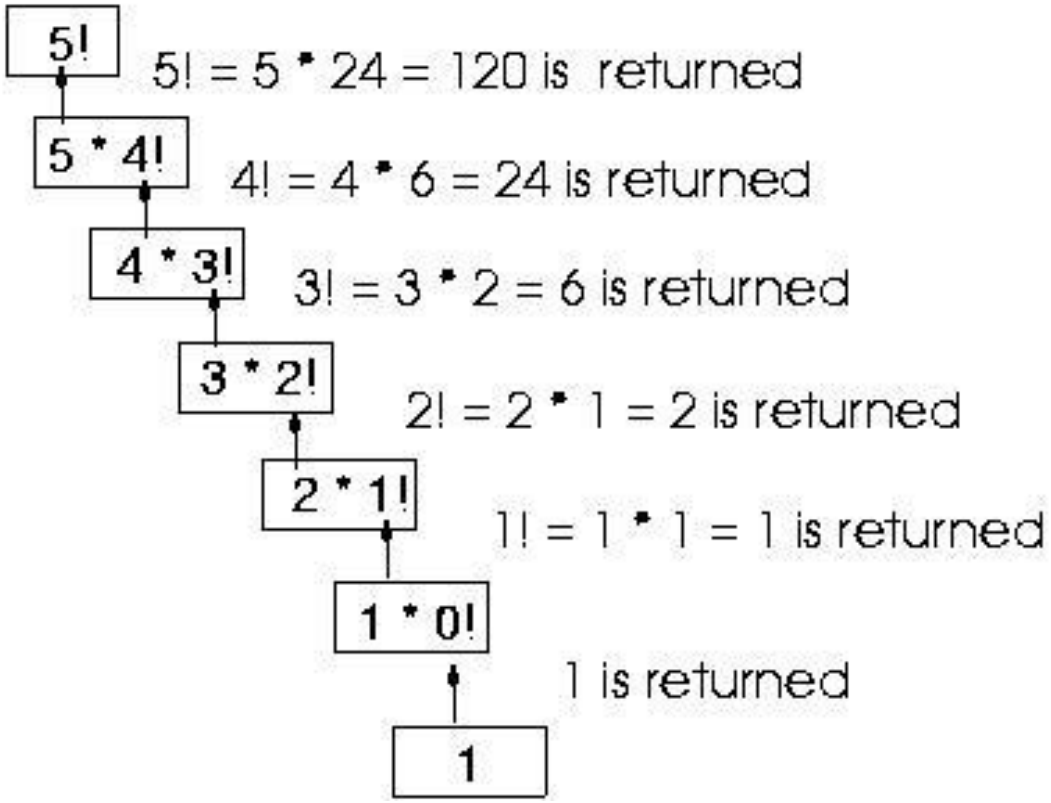
What Is a Recursive Function?

- A **recursive function** is often defined as “a function that calls itself.”





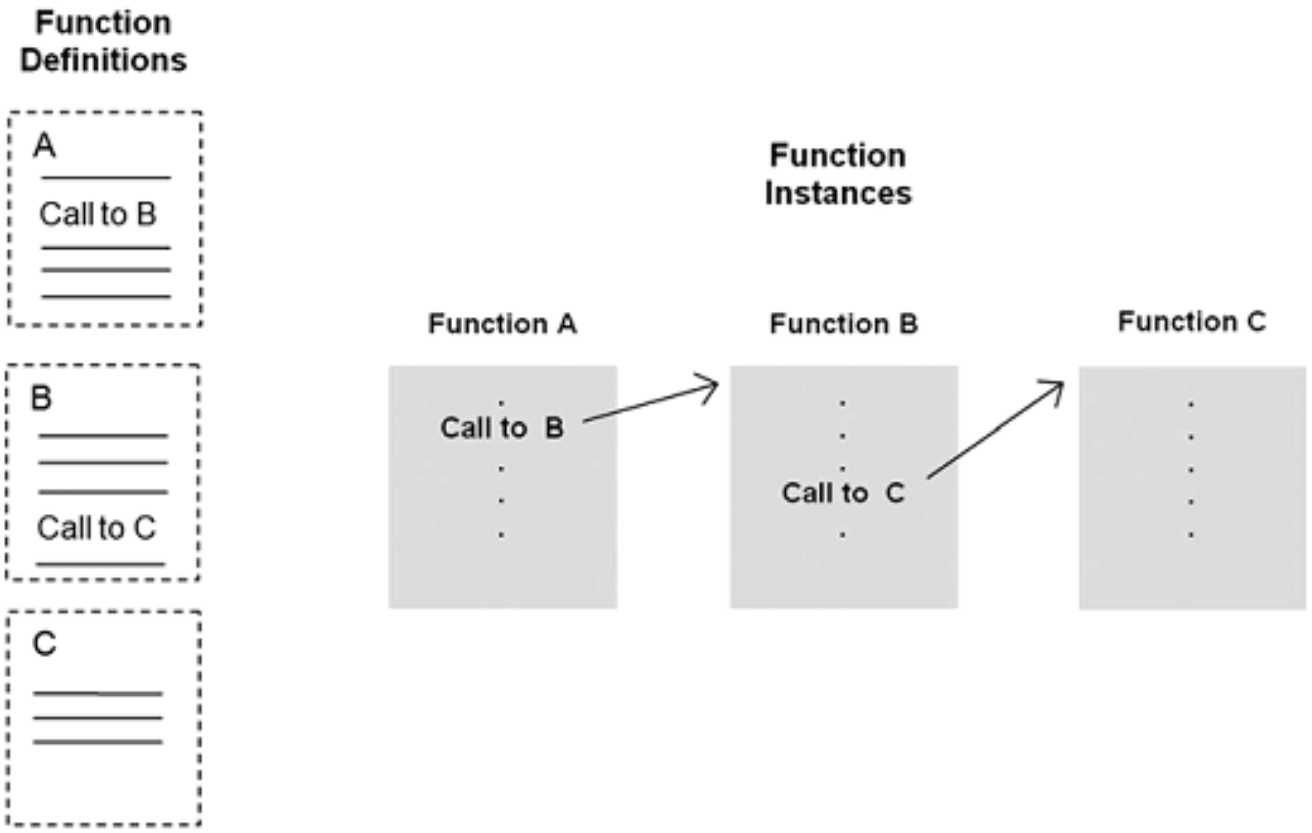
Final value = 120



Function Definition and execution instances

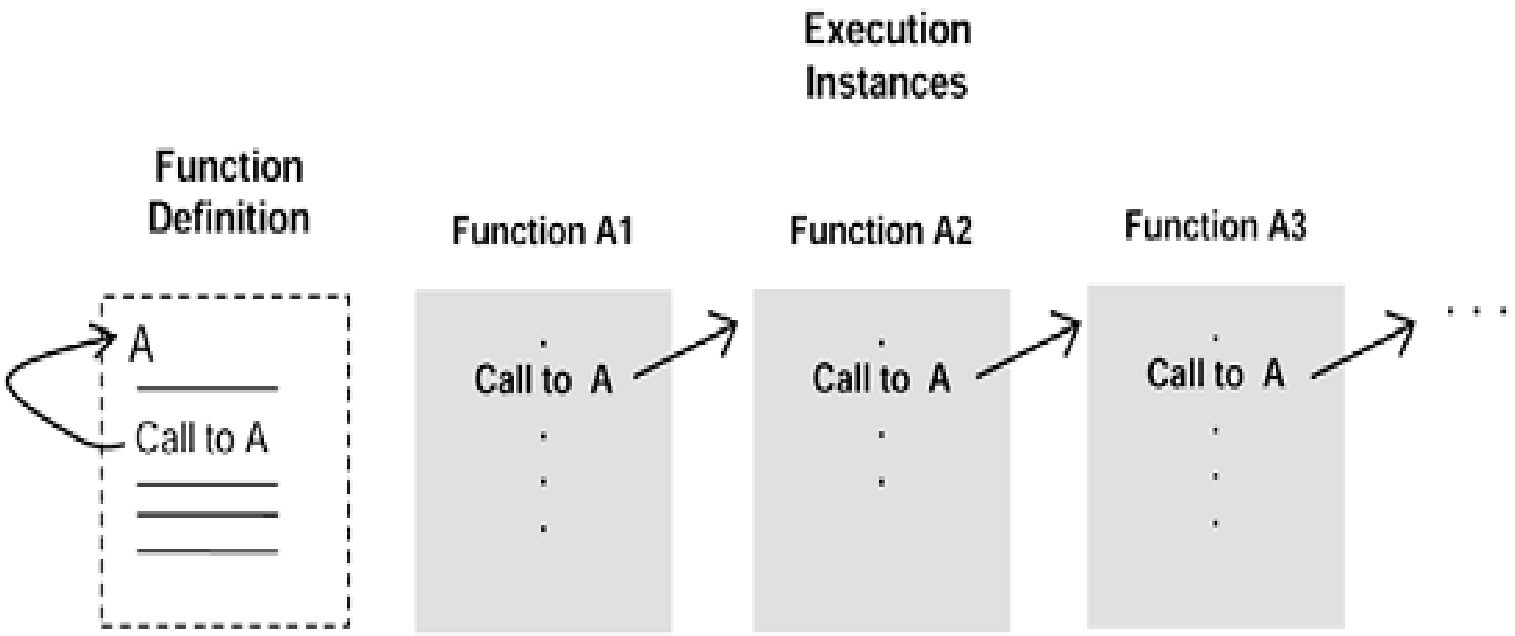
- The illustration in the figure depicts a function, A, that is defined at some point to call function A (itself). The notion of a self-referential function is inherently confusing.
- There are two types of entities related to any function however—the *function definition*, and any current *execution instances*.
- What is meant by the phrase “a function that calls itself ” is a function *execution instance* that calls another *execution instance* of the same function.
- A function definition is a “cookie cutter” from which any number of execution instances can be created. Every time a call to a function is made,another execution instance of the function is created. Thus, while there is only one definition for any function, there can be any number of execution instances.

General mechanism of non-recursive function



Recursive function execution instances

- Note that the execution of a series of recursive function instances is similar to the execution of series of non-recursive instances, except that the execution instances are “clones” of each other (that is, of the same function definition). Thus, since all instances are identical, the function calls occur in exactly the same place in each.



A Recursive Factorial Function Implementation

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Python **Lambda Function**

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.
- Syntax

lambda arguments : expression

The expression is executed and the result is returned

Lambda Function

Add 10 to argument a, and return the result:

```
x = lambda a : a + 10
print(x(5))
```

Output:

15

Lambda Function

- Lambda functions can take any number of arguments:

Example - Multiply argument a with argument b and return the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

Output:
30

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Output:
13

```
# Example of lambda function using if-else
Maxnum = lambda a, b : a if(a > b) else b

print(Maxnum(1, 2))
```

Exercise 1

- Ram is planning to give chocolates to two of his brother. He comes to a shop that has 'n' chocolates of type 1 and 'm' chocolates of type 2, and Ram wants to buy largest but equal number of both. Write a program to determine the number using recursive function.

GCD

```
def gcd(a,b):
    if(b==0):
        return (a)
    else:
        return gcd(b,a%b)
print(gcd(60,100))
```

Fibonacci Series

```
n=int(input("Enter n"))
def fib(n):
    if(n==0):
        return(0)
    elif(n==1):
        return(1)
    else:
        return(fib(n-1)+fib(n-2))


for i in range(0,n):
    print(fib(i))
```

Students' details

INPUT:

Register_number	Name	CSE1001	Maths	Physics
21BCE1001	xyz	99	78	55
21MCE1213	ABC	34	32	89

OUTPUT:

Register_number	Name	CSE1001	Maths	Physics	Average	Result
21BCE1001	xyz	99	78	55	77.33	PASS
21MCE1213	ABC	34	32	89	51.67	FAIL

Write a program using functions to get the details of students' marks in three courses (RegNo, Name, Mark1, Mark2 and Mark3) and calculate their average score to declare the result. If the average is greater than 59 declare the result as pass otherwise declare it as Fail.

Students' details

```
import sys
def result(studdetails):
    for i in range(0,len(studdetails)):
        studdetails[i]+=((studdetails[i][2]+studdetails[i][3]+studdetails[i][4])/3,)
        if(studdetails[i][5]>=50):
            studdetails[i]+=("PASS ",)
        else:
            studdetails[i]+=("FAIL ",)
    return(studdetails)


n=int(input("No of students"))
print("Enter the attributes of each student")
studdetails=[]
for i in range(0,n):
    t=()
    t+=(input("Registration number "),)
    t+=(input("Name "),)
    t+=(int(input("CSE1001 ")),)
    t+=(int(input("English ")),)
    t+=(int(input("Mathematics ")),)
    if((t[2]<0)or(t[3]<0) or (t[4]<0)):
        print("Invalid input")
        sys.exit()
    studdetails.append(t)
p=result(studdetails)
for r in range(0,len(p)):
    print(p[r])
```

Exercise 2

- Asha goes to a grocery store to buy sausages and buns for a hot dog party you're hosting. Unfortunately, sausages come in a pack of 'm1', and buns in a pack of 'm2'.

  What is the least number of sausages and buns Asha need to buy in order to make sure you are not left with a surplus of either sausages or buns?

LCM

```
def lcm(a,b,common):
    if(common%a==0) and (common%b==0):
        return(common)
    else:
        return(lcm(a,b,common+1))

print(lcm(15,40,max(15,40)))
```

Check for prime number

```
def isPrime(n):
    for i in range(2,int(n**0.5)+1):
        if n%i==0:
            return 0

    return 1
```

Given three points, write a program to check if they can form a triangle. Three points can form a triangle, if they do not fall in a straight line and length of a side of triangle is less than the sum of length of other two sides of the triangle. For example, the points (5,10), (20,10) and (15,15) can form a triangle as they do not fall in a straight line and length of any side is less than sum of the length of the other two sides