BCSE101E

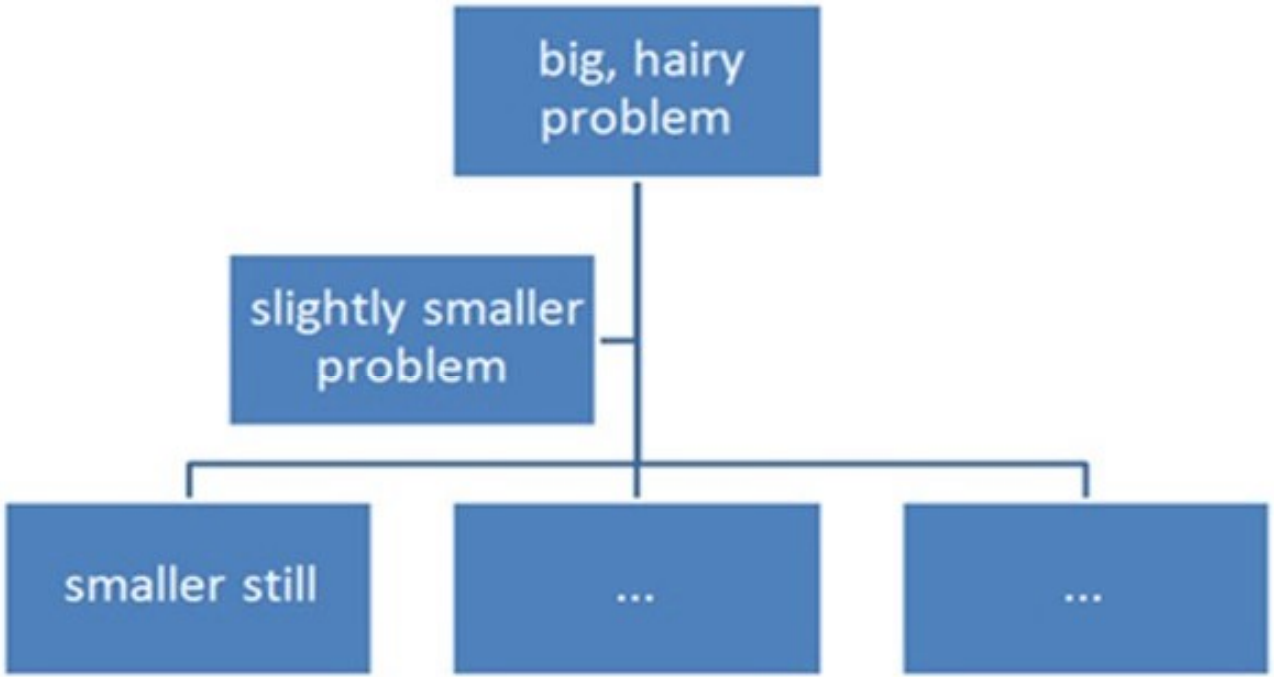
Computer Programming: Python

Course Faculty: Rajesh M – SCOPE, VIT Chennai

Module – 6
Function and Files in Python

Functions in Python

Functions



Python Functions

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

- Functions are sub-programs which perform tasks which may need to be repeated
- Some functions are "bundled" in standard libraries which are part of any language's core package. We've already used many built-in functions, such as input(), eval(), etc.
- Functions are similar to methods, but may not be connected with objects
- Programmers can write their own functions
- A function is a block of organized, reusable code that is used to perform a single, related action. Functions provides better modularity for your application and a high degree of code reusing.
- As you already know, Python gives you many built-in functions like print() etc. but you can also create your own functions. These functions are called user-defined functions.

Python Functions

A *function* is **a piece of code that performs a ta**sk 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.

Why Write Functions?

- Reusability
- Fewer errors introduced when code isn't rewritten
- Reduces complexity of code
- Programs are easier to maintain
- Programs are easier to understand

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."

Function Elements

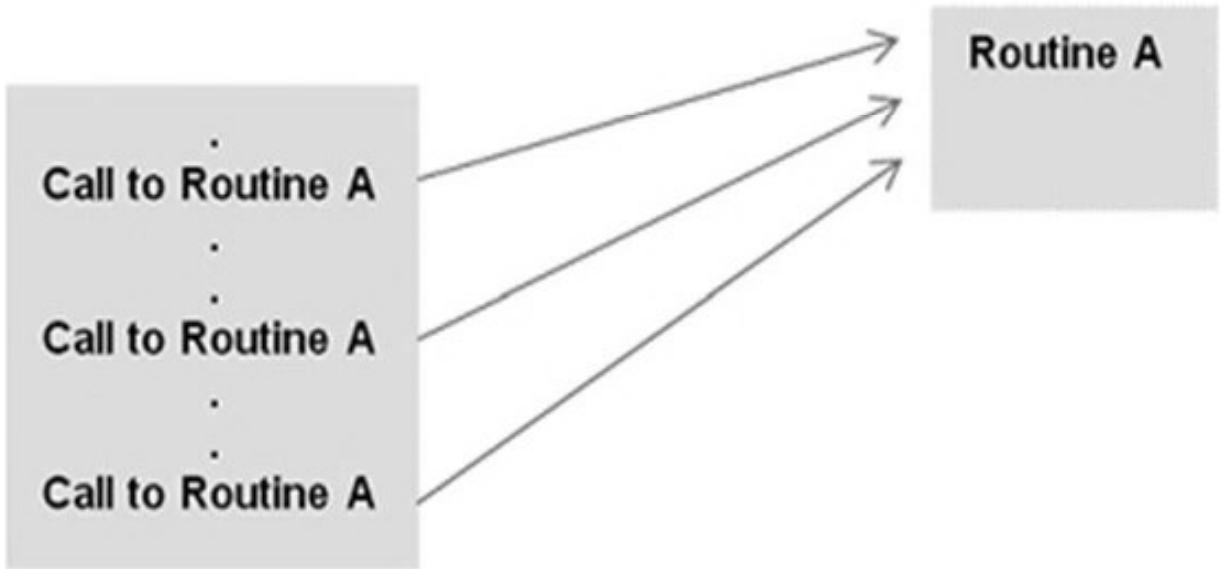
- Before we can use functions we have to define them. So there are two main elements to functions:

1. **Define the function**. The function definition can appear at the beginning or end of the program file.
2. **Invoke or call the function**. This usually happens in the body of the main() function, but subfunctions can call other subfunctions too.

What is a Function Routine?

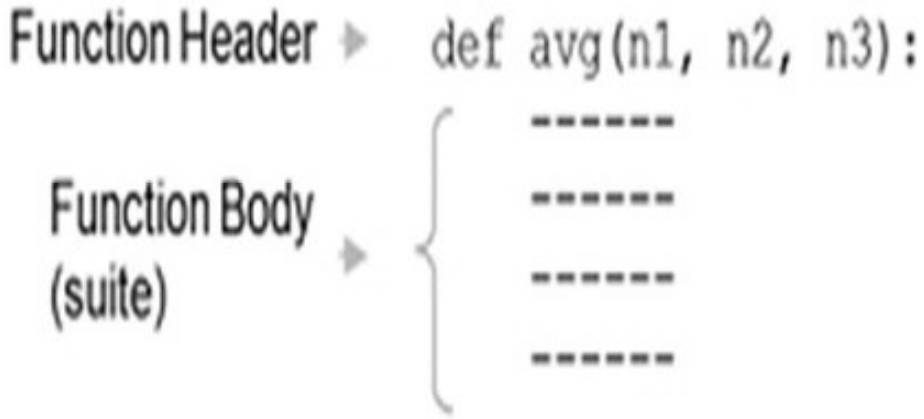
- A **routine** is a named group of instructions performing some task. A routine can be **invoked** (*called*) as many times as needed in a given program.

- **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 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



Defining a Function

Here are simple rules to define a function in Python:

- Function blocks begin with the keyword def followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or docstring.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.
- **Syntax:**

```
def functionname( parameters ):
Function body return [expression]
```

Defining a Function

- Syntax:

```
def functionname( parameters ):
        #func. body
        return (expression)
```

- By default, parameters have a positional behavior, and you need to inform them in the same order that they were defined.

- Example:

```
def printme( str ):
        "This prints a passed string function"
        print(str)
        return
```

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.

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`)

Assignment Statements Recap

```
# Assign tuple of values to list of names

>>> [a, b, c] = (1, 2, 3)

>>> a, c

(1, 3)

>>> (a, b, c) = "ABC"

# Assign string of characters to tuple

>>> a, c

('A', 'C')
```

Assignment Statements Recap

```
>>> seq = [1, 2, 3, 4]

>>> a, b, c, d = seq

>>> print(a, b, c, d)

1 2 3 4

>>> a, b = seq

ValueError: too many values to unpack (expected 2)
```

Assignment Statements Recap

```
>>> a, *b = seq

>>> a

1

>>> b

[2, 3, 4]
```

Assignment Statements Recap

```
>>> *a, b = seq

>>> a

[1, 2, 3]

>>> b

4
```

Assignment Statements Recap

```
>>> a, *b, c = seq
>>> a
1
>>> b
[2, 3]
>>> c
4
```

Assignment Statements Recap

```
>>> a, *b = 'spam'
>>> a, b
('s', ['p', 'a', 'm'])
>>> a, *b, c = 'spam'
>>> a, b, c
('s', ['p', 'a'], 'm')
>>> a, *b, c = range(4)
>>> a, b, c
(0, [1, 2], 3)
```

Example Functions

```
def printer(messge):
        print('Hello ' + message)

def adder(a, b=1, *c):
        return a + b + c[0]
```

Example Functions

```
>>> def times(x, y):
        # Create and assign function ...
        return x * y
        # Body executed when called ...
```

➢When Python reaches and runs this def, it creates a new function object that packages the function's code and assigns the object to the name times.

Calls

```
>>> times(2, 4)
# Arguments in parentheses
8
>>> x = times(3.14, 4)     # Save the result object >>> x
12.56

>>> times('Ni', 4)         # Functions are "typeless" 'NiNiNiNi'
```

Calls

```
>>> times(2, 4)
# Arguments in parentheses
8
>>> x = times(3.14, 4)     # Save the result object >>> x
12.56

>>> times('Ni', 4)          # Functions are "typeless" 'NiNiNiNi'
```

Calling a Function

- Following is the example to call printme() function:
- # "This is a print function"

```
def printme( str ):
    print (str)
    return
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

- This would produce following result:
  - I'm first call to user defined function!
  - Again second call to the same function

Pass by reference vs value

- All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example:

```
def changeme( mylist ):  #"This changes a passed list"
        mylist.append([1,2,3,4])
        print ("Values inside the function: ", mylist)
        return
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

- So this would produce following result:
- Values inside the function: [10, 20, 30, [1, 2, 3, 4]]
- Values outside the function: [10, 20, 30, [1, 2, 3, 4]]

There is one more example where argument is being passed by reference but inside the function, but the reference is being over-written.

```
def changeme( mylist ):   #"This changes a passed list"
    mylist = [1,2,3,4]
    print ("Values inside the function: ", mylist )
    return
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

- The parameter mylist is local to the function changeme. Changing mylist within the function does not affect mylist. The function accomplishes nothing and finally this would produce following result:

```
Values inside the function: [1, 2, 3, 4]
Values outside the function: [10, 20, 30]
```

Function Arguments

A function by using the following types of formal arguments::

- **Required arguments**
- **Keyword arguments**
- **Default arguments**
- **Variable-length arguments**

Required arguments:

- Required arguments are the arguments passed to a function in correct positional order.

```
def printme( str ):  #"This prints a passed string"
    print (str)
    return
printme()
```

- This would produce following result:

```
Traceback (most recent call last):
File "test.py", line 11, in <module> printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

Keyword arguments

- Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.
- This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

```
def printme( str ):  #"This prints a passed string"
    print (str)
    return
printme( str = "My string")
```

- This would produce following result:

```
My string
```

- Following example gives more clear picture.
- Note, here order of the parameter does not matter:

```
def printinfo( name, age ):
    print ("Name: ", name)
    print ("Age ", age)
    return

printinfo( age=50, name="miki" )
```

- This would produce following result:

  Name: miki Age 50

Default arguments

- A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.
- Following example gives idea on default arguments, it would print default age if it is not passed:

```
def printinfo( name, age = 35 ):
    print ("Name: ", name)
    print ("Age ", age)
    return
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

- This would produce following result:

```
Name: miki Age 50 Name: miki Age 35
```

Variable-length arguments

- You may need to process a function for more arguments than you specified while defining the function.
- These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.
- The general syntax for a function with non-keyword variable arguments is this:

```
def functionname([formal_args,] *var_args_tuple ):
    # func_body
    return (expression)
```

- An asterisk (*) is placed before the variable name that will hold the values of all non-keyword variable arguments.
- This tuple remains empty if no additional arguments are specified during the function call. For example:

```
def printinfo( arg1, *vartuple ):
    print ("Output is: " )
    print (arg1)
    for var in vartuple:
                print (var)
    return
printinfo( 10 )
printinfo( 70, 60, 50 )
```

This would produce following result:

```
Output is:
10
Output is:
70
60
50
```

▪ The *Anonymous* Functions (lambda)

- You can use the *lambda* keyword to create small anonymous functions. These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword.
- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression.
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to *inline* statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.
- **Syntax:**

  lambda [arg1 [,arg2,.....argn]]:expression

Example

- Following is the example to show how *lembda* form of function works:

```
sum = lambda arg1, arg2: arg1 + arg2
print ("Value of total : ", sum( 10, 20 ) )
print ("Value of total : ", sum( 20, 20 ) )
```

- This would produce following result:

```
Value of total : 30
Value of total : 40
```

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']
```

Equivalent Comprehension

```
>>> [x for x in s1 if x in s2]
['S', 'A', 'M']
```

Works for list also:

```
>>> x = intersect([1, 2, 3], (1, 4))      # Mixed types
>>> x                                     # Saved result object [1]
```

Def Statements

- **creates a function object** and assigns it to a name.

- def is a true executable statement: when it runs, it creates a new function object and assigns it to a name.

- Because it's a statement, a def can appear anywhere a statement can—even nested in other statements

Def Statements

```
if test:
        def func():         # Define func this way      else:
        def func():         # Or else this way          ... .

 func()
# Call the version selected and built
```

Function definition in selection statement Example

```
a = 4
if a%2==0:
        def func():
                print ('even')
else:
        def func():
                print('odd')
 func()
```

Output:

even

Function definition in selection statement Example

```
a = 4
if a%2==0:
        def func():
                print ('even')
else:
        def func():
                printl('odd')
# error no function printl is defined
 func()
```

Error in only condition satisfied item is found.

Otherwise code execute normal

Output:

```
even
```

Function definition in selection statement Example

```
a = 5
if a%2==0:
        def func():

                print ('even')
else:
        def func():
                printl('odd') # error no function printl is defined
 func()
```

Output:

error

Function Call through variable

```
def one():
    print('one')

def two():
    print('two')

def three():
    print('three')
```

Function Call through variable

```
a = 3

if a == 1:
    call_Func=one
elif a == 2:
    call_Func=two
else:
    call_Func=three

call_Func()
```

Scope of Variables

- All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.
- The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python:
- Global variables
- Local variables
- Global vs. Local variables:
- Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.
- This means that local variables can be accessed only inside the function in which they are declared whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope.

Example

```
total = 0 # This is global variable.
def sum( arg1, arg2 ):
    #"Add both the parameters"
    total = arg1 + arg2
    print ("Inside the function local total : ", total )
    return total
# Now you can call sum function
sum( 10, 20 )
print ("Outside the function global total : ", total)
```

- This would produce following result:

```
Inside the function local total : 30
Outside the function global total : 0
```

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

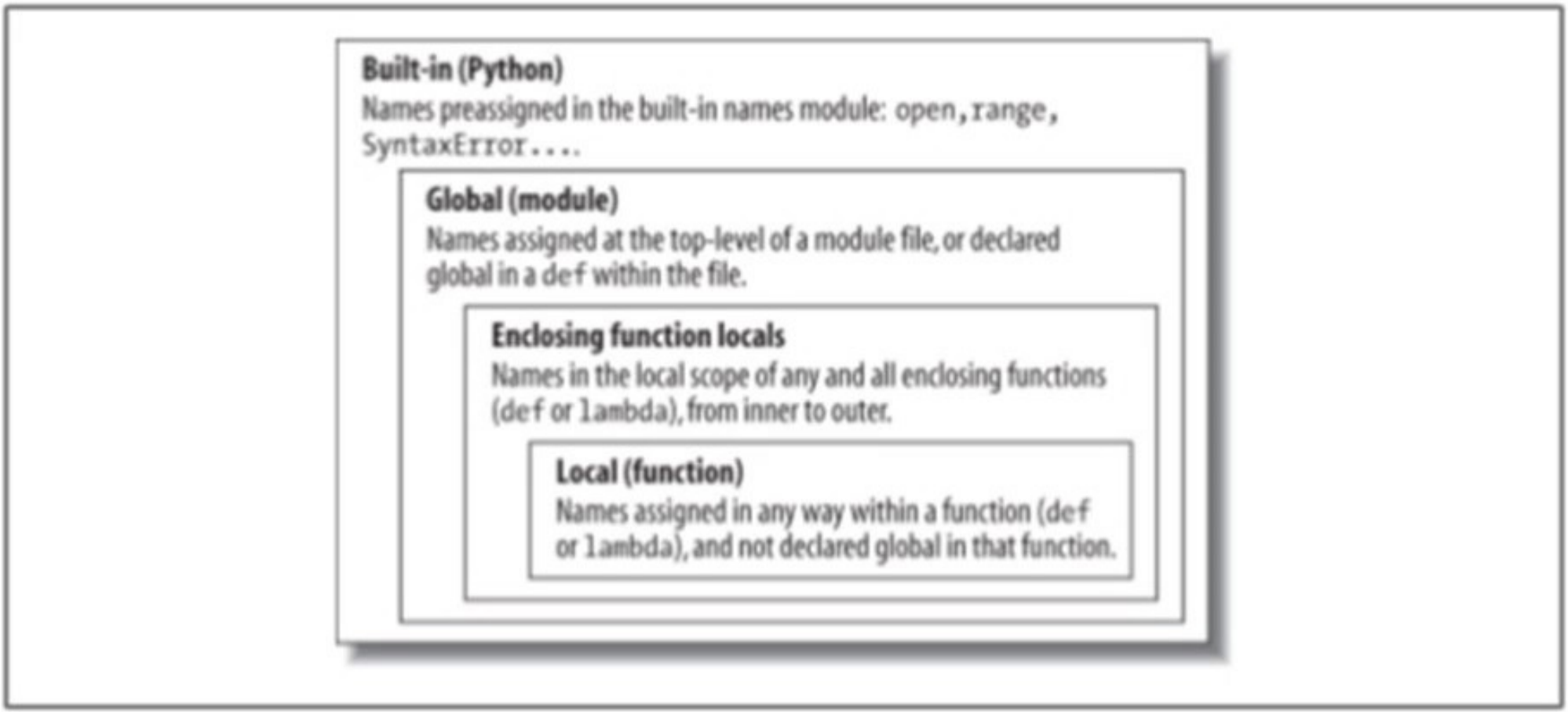


Figure 17-1. The LEGB scope lookup rule. When a variable is referenced, Python searches for it in this order: in the local scope, in any enclosing functions' local scopes, in the global scope, and finally in the built-in scope. The first occurrence wins. The place in your code where a variable is assigned 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
```

Accessing Global Variables

```
y, z = 1, 2                    # Global variables in module
def all_global():
    global x                   # Declare globals assigned    x = y + z
# No need to declare y, z: LEGB rule
```

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.

Nested Functions

- X = 99 **# Global scope** name: not used

```
def f1():
        X = 88          # Enclosing def local
        def f2():
                print(X)
                # Reference made in nested def
        f2()
f1()                    # Prints 88: enclosing def local
```

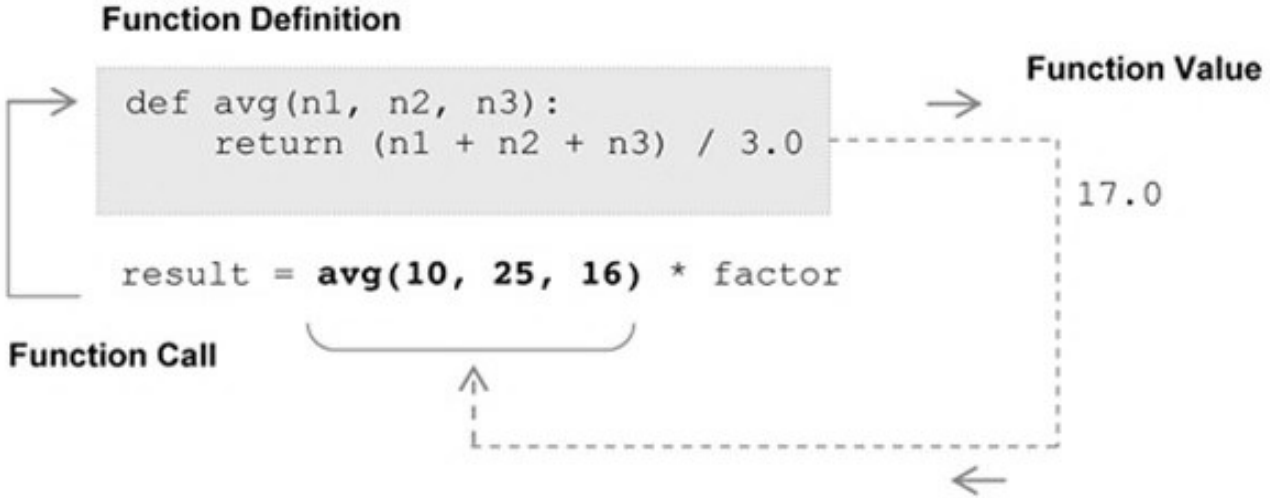
Return Functions

- Following code defines **a function that makes and returns another function**
- def f1():

```
    X = 88
    def f2():
        print(X)  # Remembers X in enclosing def scope
    return f2         # Return f2 but don't call it
action = f1()         # Make, return function
action()              # Call it now: prints 88
```

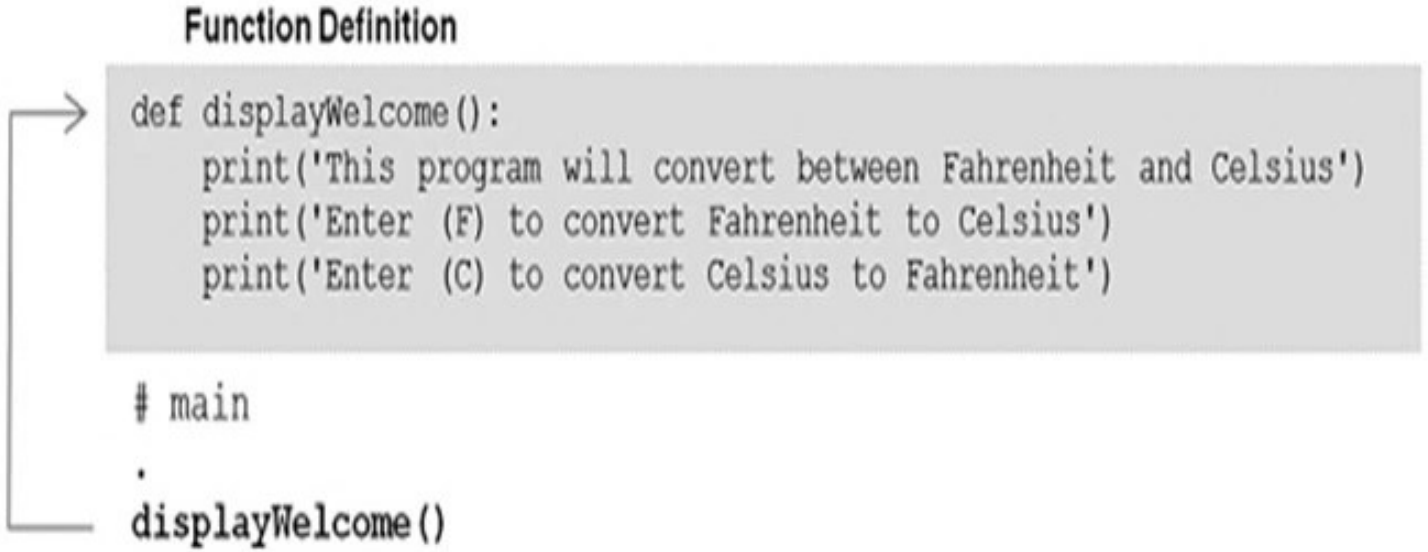
Value-Returning Functions

- 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])
```

Keyword Arguments in Python

- The **functions we have looked at so far** were called with a **fixed 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.

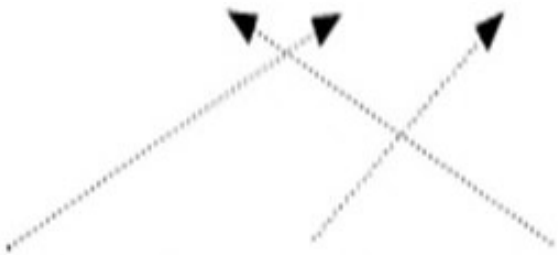
```
def mortgage_rate(amount, rate, term)

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

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

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

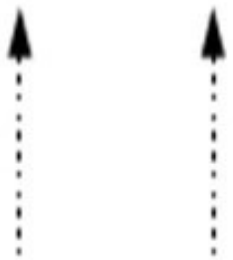


```
monthly_payment = mortgage_rate(rate=0.06, term=20, amount=350000)
```

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
```

Defaults

```
>>> f(1, c=6)          # Choose defaults
1 2 6
```

Combining keywords and defaults

```
def func(spam, eggs, toast=0, ham=0):   # First 2 required
                print((spam, eggs, toast, ham))
func(1, 2)                          # Output: (1, 2, 0, 0)
func(1, ham=1, eggs=0)                  # Output: (1, 0, 0, 1)
func(spam=1, eggs=0)                    # Output: (1, 0, 0, 0)
func(toast=1, eggs=2, spam=3)   # Output: (3, 2, 1, 0)
func(1, 2, 3, 4)                    # Output: (1, 2, 3, 4)
```

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}
```

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)

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
```

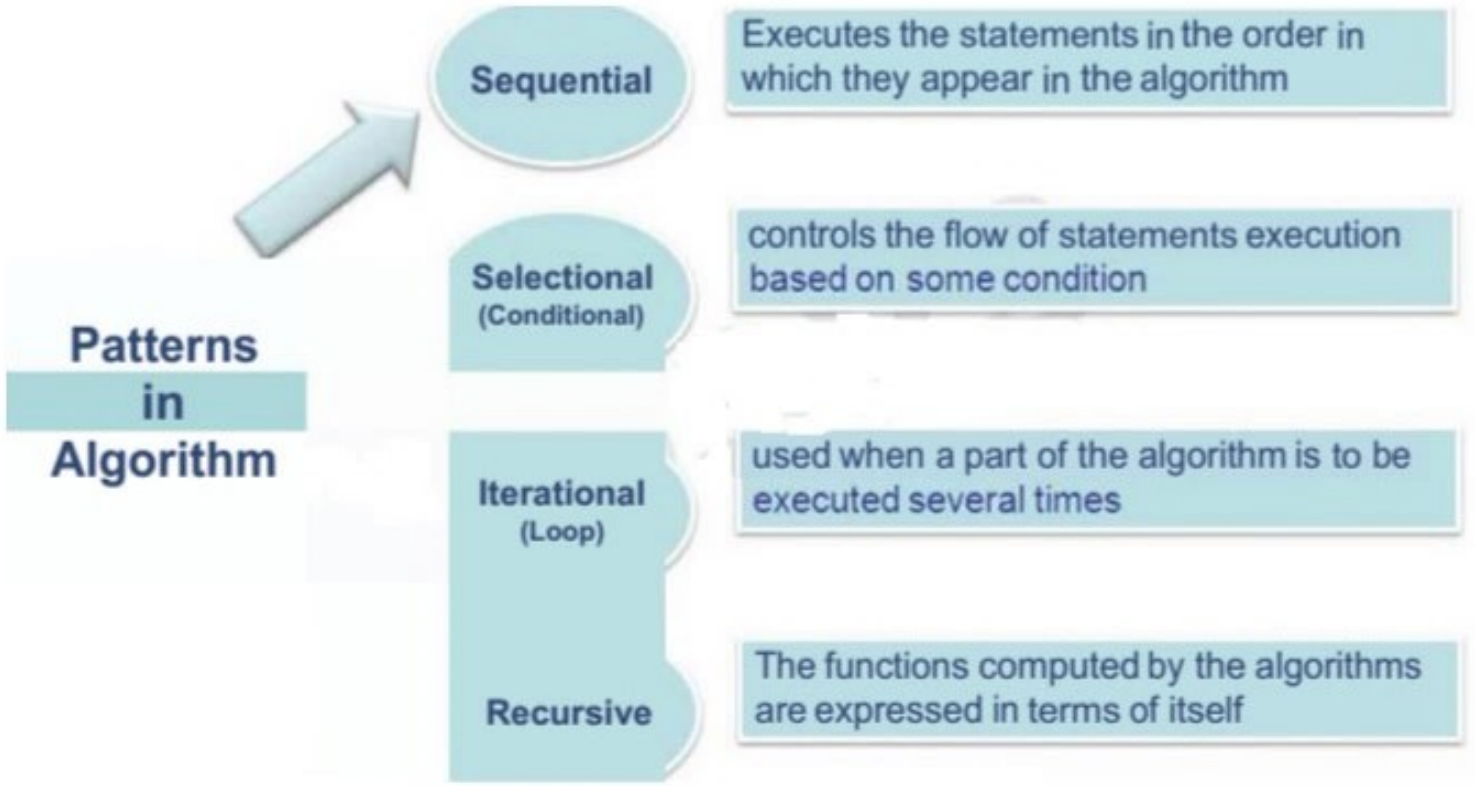
```
>>> addup(1,10)
???
>>> addup(1,10,2)
???
>>> addup(first=1, last=10)
???
>>> addup(incr=2, first=1,
        last=10)
???
```

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

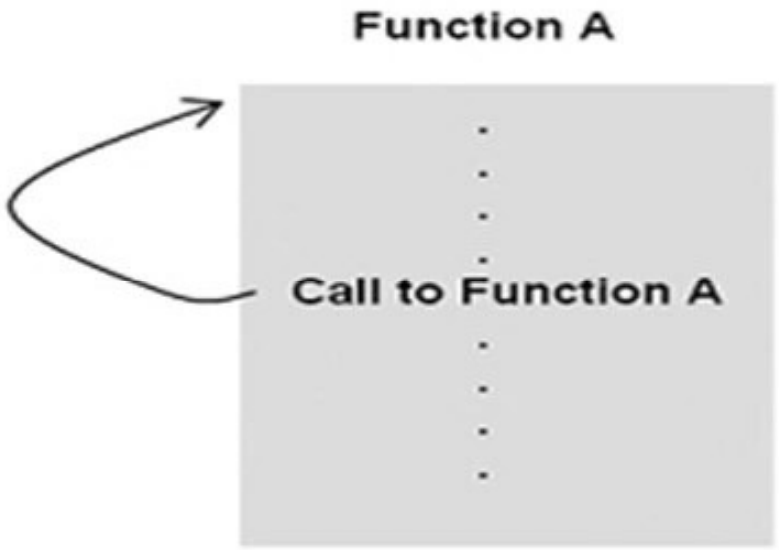
  Algorithm:

  Algorithm Factorial(n)

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

What Is a Recursive Function?

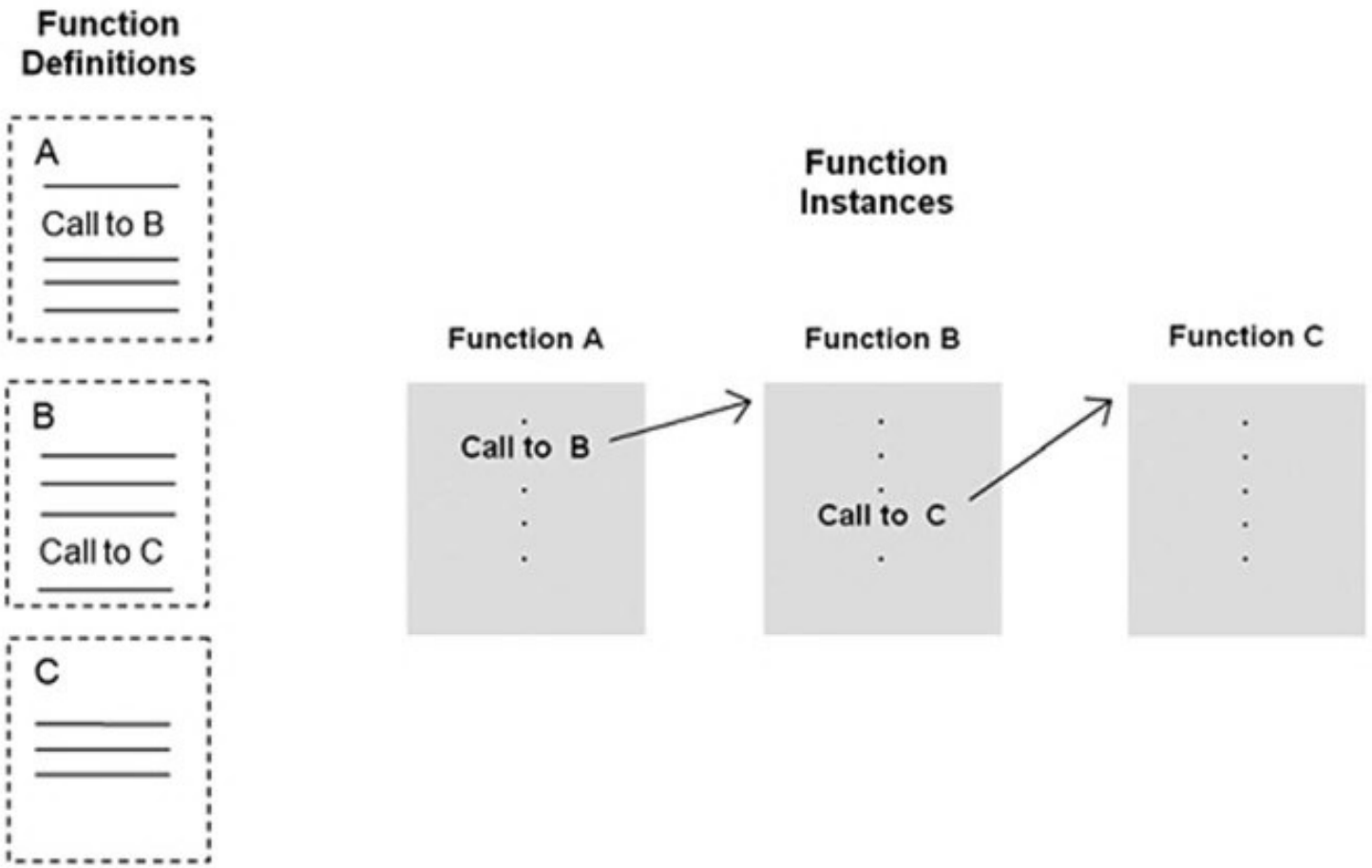
- A **recursive function** is often defined as "a function that calls itself."



Function Definition and execution instances

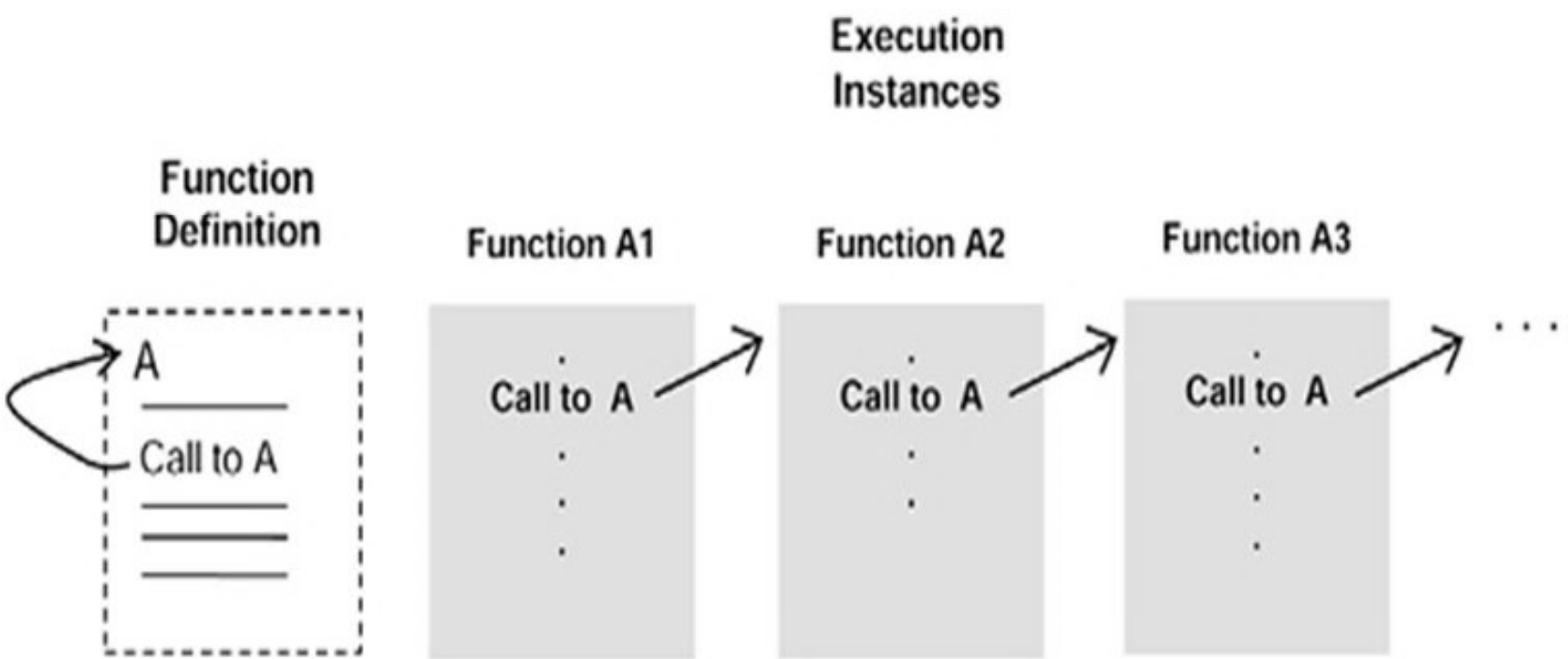
- The illustration in the figure depicts a function, A, that is defined at some point to call function A (itself). The notion of a self-referential function is inherently confusing.
- There are two types of entities related to any function however—the ***function definition***, and any **current *execution 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.



LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> def rfunc(n):
        print(n)
        if n > 0:
            rfunc(n - 1)

>>> rfunc(4)
???

>>> rfunc(0)
???

>>> rfunc(100)
???
```

```
>>> def rfunc(n):
        if n == 1:
            return 1
        else:
            return n + rfunc(n - 1)

>>> rfunc(1)
???

>>> rfunc(3)
???

>>> rfunc(100)
???
```

Example: Factorial

- Problem:

The factorial function is an often-used example of the use of recursion. The computation of the factorial of 4 is given as,

factorial(4) = 4 * 3 * 2* 1= 24

In general, the computation of the factorial of any (positive, nonzero) integer n is,

factorial(n) = n . (n-1). (n-2) . . . 1

The one exception is the factorial of 0, defined to be 1.

logic

- The factorial of n can be defined as n times the factorial of n - 1,

$$\text{factorial}(n) = n \cdot \underbrace{(n-1) \cdot (n-2) \cdot \cdots \cdot 1}_{\text{factorial}(n-1)}$$

Thus, the complete definition of the factorial function is,

$$\begin{aligned} \text{factorial}(n) &= 1, && \text{if } n = 0 \\ &= n \cdot \text{factorial}(n-1), && \text{otherwise} \end{aligned}$$

A Recursive Factorial Function Implementation

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

Input
Factorial(4)

Factorial Recursive Instance Calls

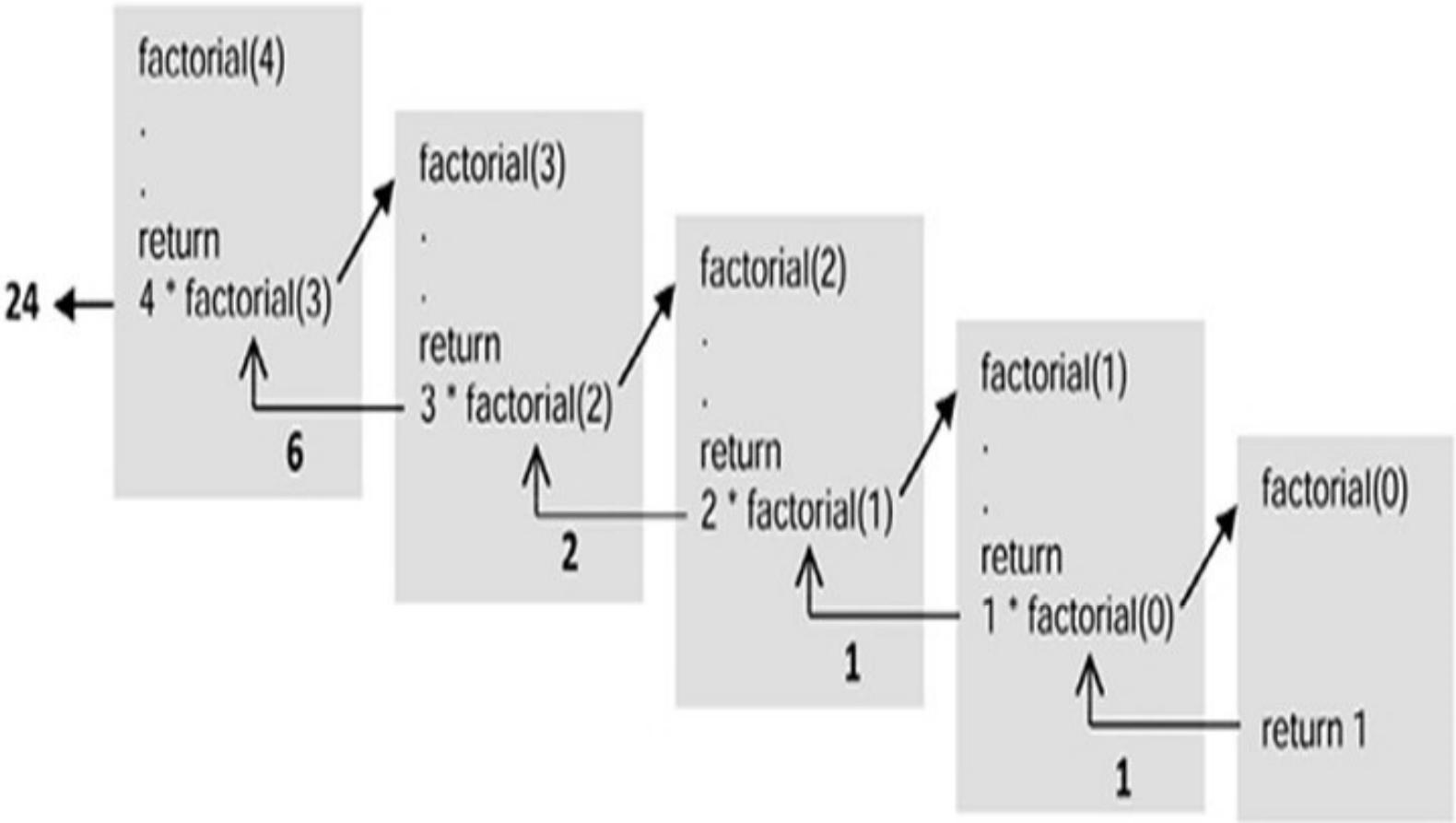


FIGURE 11-6 Factorial Recursive Instance Calls

LET'S TRY IT

From the Python Shell, enter the following and observe the results.

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


>>> factorial(4)
???
>>> factorial(0)
???
>>> factorial(100)
???
>>> factorial(10000)
???
```

```
>>> def ifactorial(n):
        result = 1
        if n == 0:
            return result
        for k in range(n, 0, -1):
            result = result * k
        return result
>>> ifactorial(0)
???
>>> ifactorial(100)
???
>>> ifactorial(10000)
???
```

Problem

It is challenge to place four queens in a 4 X 4 board such that they do not attack each other. Given a list of positions of four queens, write a program to determine if the board positions are solution to the problem. If it is a solution then print 'NO ATTACK' and print 'ATTACK' otherwise.
One of the solution for the problem is [(0,1), (1,3), (2,0), (3,2)].

To reduce the number of lines of code, write a routine to check for collide of queens.

PSEUDOCODE

```
READ num_of_board_pos
FOR j = 0 to num_of_board_pos
  READ position_of_queens
     FOR i = 0 to 4
           FOR  j = i+1 to  4
                    attack = CALL check_Attack(position_of_queens[i],
                                position_of_queens[j])
                    IF attack == true THEN
                                PRINT 'ATTACK'
           END FOR
                    PRINT 'NO ATTACK'
   END FOR
END FOR
```

Pseudocode for check_Attack

```
input : board_pos1, board_pos2
IF (board_pos1.x == board_pos2.x) THEN
        RETURN true
ELIF (board_pos1.y == board_pos2.y) THEN
        RETURN true
ELIF    ((abs(board_pos1.x - board_pos2.x) == 1) and
        (abs(board_pos1.y - board_pos2.y) == 1)) THEN
        RETURN TRUE
ELSE
        RETURN false
```

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.

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? Write a recursive function to find the LCM of a number using recursion.