Python: Functions

Computer Science & Engineering Department

The Chinese University of Hong Kong

Content

- Import module
- Define & Invoking functions
- Parameters passing
- Return value
- List Comprehension
- Scope
- Lambda function

Importing Modules

- Remember those print(), input() and len() that we used before?
- They belong to *built-in functions* provided by Python environment
- To support more complicated usages, Python provide different modules to be incorporated into our own program
- A few common modules:
    1. math – mathematics functions
    2. sys – system utilities such as file open, read/write etc
    3. random – random number generator for simulations
- All we need is to use *import* statement to use them

import Statement (Example #1)

```python
import math

while (1):
    str1 = input("x? ")
    x = float(str1)
    if (x <= 0):
        break;
    print("square root = ", math.sqrt(x))
print("Bye!");
```

```
x? 15
square root =  3.872983346207417
x? 16
square root =  4.0
x? -3
Bye!
```

Some of functions in math module

Functions	Description	Examples

`ceil( x )`	rounds *x* to the smallest integer not less than *x*
	$\lceil x \rceil$
		`ceil(9.2) is 10.0`
		`ceil(-9.8) is -9.0`

`floor ( x )`	rounds *x* to the largest integer not greater than *x*
	$\lfloor x \rfloor$
		`floor(9.2) is 9.0`
		`floor(-9.8) is -10.0`

`exp( x )`	exponential function e^x	`exp(1.0) is 2.71828`

`fabs( x )`	absolute value of *x*
	$\lvert x \rvert$
		`fabs(5.1) is 5.1`
		`fabs(0.0) is 0.0`
		`fabs(-8.76) is 8.76`

`pow( x, y )`	*x* raised to power *y*
	x^y
		`pow(2, 7) is 128.0`
		`pow(9, .5) is 3.0`

`sqrt( x )`	square root of *x*
	$\sqrt{x}$
		`sqrt(900.0) is 30.0`
		`sqrt(9.0) is 3.0`

Some of functions in math module

Functions	Description	Examples

`log ( x )`	natural logarithm of *x* (base *e*)
	$\log_e$ x or **ln x**
	`ln e = 1`
	ln e^x = x * ln e = x
		`log(2.718282) ≈ 1.0`
		`log(exp(3.0)) is 3.0`

`log10 ( x )`	logarithm of *x* (base 10)
	$\log_{10}$ x
		`log(10.0) is 1.0`
		`log(100.0) is 2.0`

`sin( x )`
`cos( x )`
`tan( x )`
	trigonometric sine, cosine and tangent of *x* (*x* in radians)
	sin x
	cos x
	tan x
	90° = π/ 2
	180° = π
	270° = 3 * π / 4
		`sin(0.0) is 0.0`
		`cos(0.0) is 1.0`
		`tan(0.0) is 0.0`
		`Let pi = 3.141592654`
		`sin(pi / 2) ≈ 1.0`
		`cos(pi / 2) ≈ 0.0`
		`tan(pi / 2) ≈ a large #`

import Statement (Example #2)

- random module provide a random number generation
- randint() provide a random integer ranged between input parameters

```python
import random
for i in range(5):
    print(random.randint(1, 6))
```

```
2
4
2
1
6
```

import Statement (Example #3)

- sys.exit() provide an early ending program option i.e. terminate

```python
import sys

while True:
    print('Type exit to exit.')
    response = input()
    if response == 'exit':
        sys.exit()
    print('You typed ' + response + '.')
```

```
Type exit to exit.
hello
You typed hello.
Type exit to exit.
exit
>>>
```

From import statement

- When we used those functions, we always need to provide the prefix of the module name eg. math.sqrt()

- Alternative way is to use *from*

- Not recommended in big project using *

```
from random import *
for i in range(5):
    print(randint(1, 6))

from random import randint, random

import tensorflow
```

Functions

- What we have used in previous slides are "functions"
- Besides using those provided, we can write our own!
- Writing our function is defining it
- In terms of Python operation, we first define a function to Python, then used it

- A function is a block of organized, reusable code to perform a single, related action

Function Syntax

- *functionName* : a valid identifier
- *Parameters* : information passing to function
- *'''function docstring'''*: an optional statement - the documentation string
- *return* : exit a function, optionally passing back an expression to caller

```
def functionName( parameters ):
    '''function_docstring'''
    statements
    return [expression]
```

Functions

- Functions must be defined before use
- Can be called more than once

```
def printBar():
    print("*****************")

printBar()
print("  Hello World!")
printBar()
```

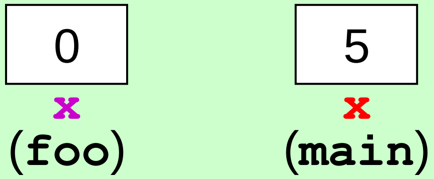
```
*****************
  Hello World!
*****************
>>>
```

Functions

- Every function can have its own variables
- Variables declared in a function are said to be local to that function

```
def foo():
    x = 0;
    print("In foo(): x = ", x)

x = 5
print("Before: In main program: x = ", x);
foo();
print("After: In main program: x = ", x);
```

x in **foo()** and **x** in **main()** are two different variables.



```
Before: In main program: x =  5
In foo(): x =  0
After: In main program: x =  5
>>>
```

Variables defined in one function are not directly accessible in another function.

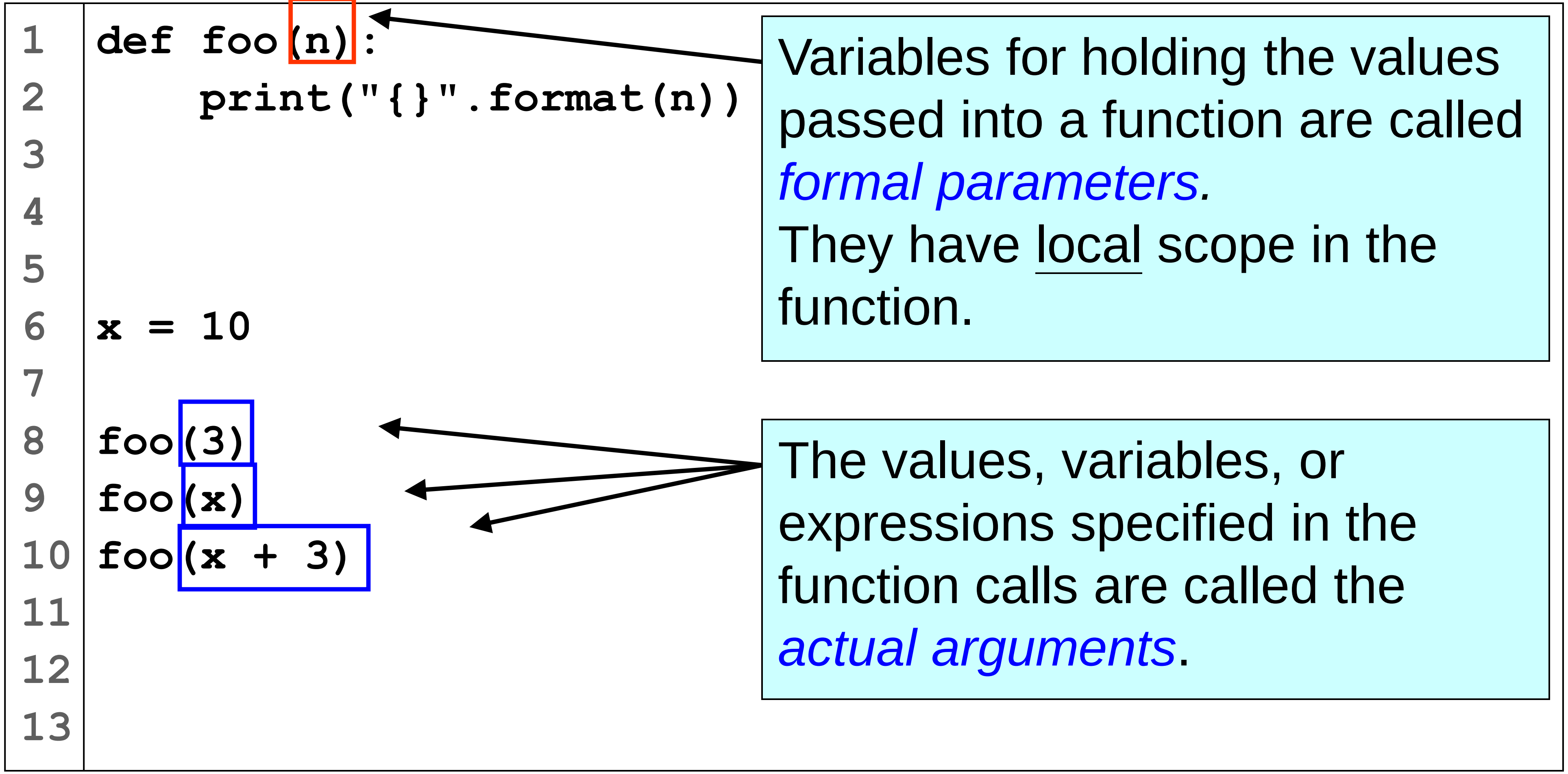
```
1   def bar():
2       y = 0;
3       print("In bar(): y = {}" .format(y))
4
5
6   bar()
7   print("y = {}" .format(y))  # error !
8
9
10
```

Formatted output, y will replace placeholder {}

```
Traceback (most recent call last):
  File "python/function6.py", line 7, in
<module>
    print("y = {}" .format(y)) # error !
NameError: name 'y' is not defined
```

Variables declared in a function are *local variables* and are only accessible in that function.

y, being declared in **bar()**, is not accessible in main program.



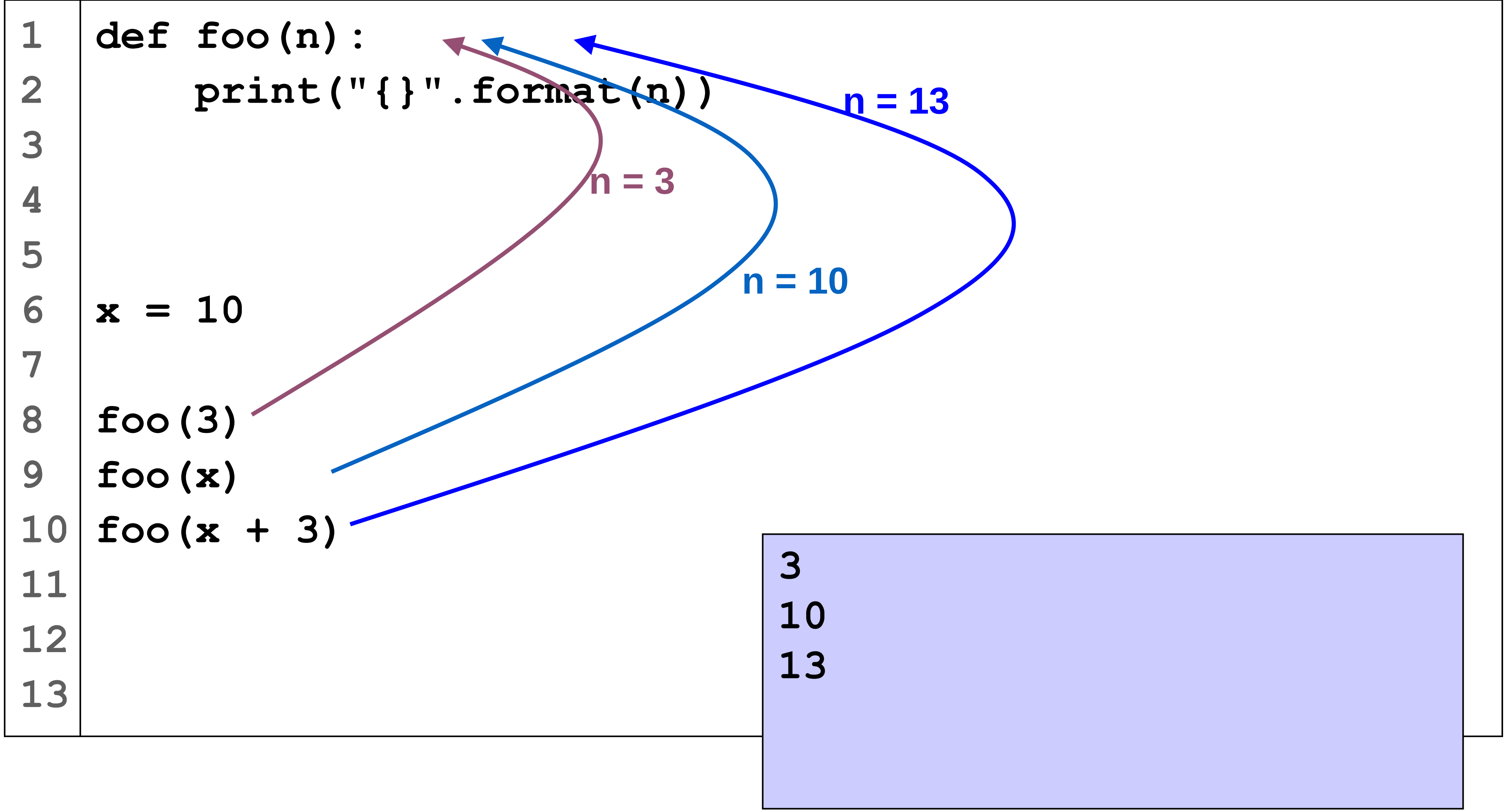
```
def foo(n):
    print("{}".format(n))



x = 10

foo(3)
foo(x)
foo(x + 3)
```

Variables for holding the values passed into a function are called *formal parameters.*
They have local scope in the function.

The values, variables, or expressions specified in the function calls are called the *actual arguments*.

2. Parameters

- Allows a function to accept data from its caller
- Allows programmers to reuse code for different values

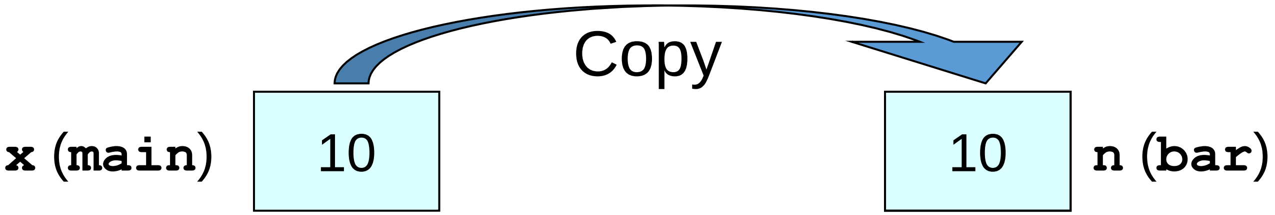


```python
def foo(n):
    print("{}".format(n))



x = 10

foo(3)
foo(x)
foo(x + 3)



```

```
3
10
13
```

- Values of the actual arguments are copied to the corresponding formal parameters.

Passing Parameter



```
x = 10
bar(x)  → bar(10)
print("x is {}" .format(x))
```

```
def bar(n):
    print("n is {}" .format(n))
    n = n + 1
```

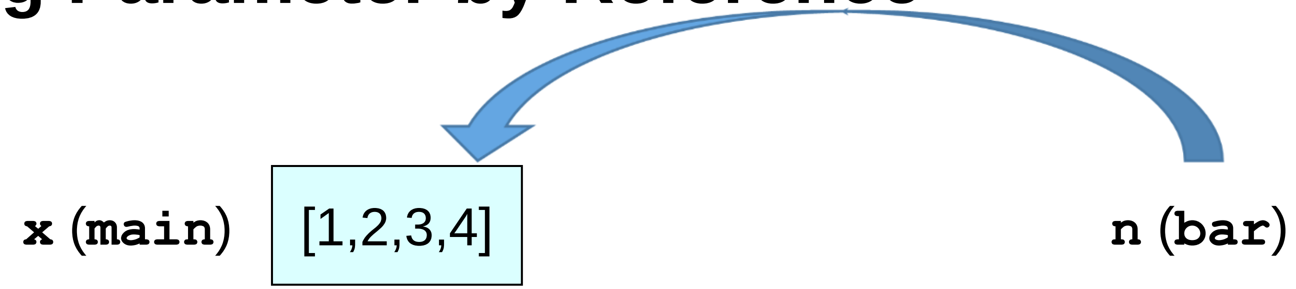
Output

```
n is 10
x is 10
```

For numeric values, only value is passed.

During the function call, only the value of **`x`** is copied to **`n`**. Changing **`n`** does not affect **`x`**.

Passing Parameter by Reference



x (main) [1,2,3,4]

n (bar)

```
x = [1, 2, 3, 4]
bar(x)
print(x)
```

```
def bar(n):
    print(n)
    n.append( [5, 6, 7])
```

Output

```
[1, 2, 3, 4]
[1, 2, 3, 4, [5, 6, 7]]
```

n keep the reference (location) of x (a list)

During the function call, any changes affecting n therefore also take effect on **x**.

Pass By Reference

- You can make the following assumption for parameter passing in Python

1. For simple data such as integer and floating point numbers, pass by value

2. For **all other** data types such as list, …, pass by reference

Defining Functions with Parameters

```
def function_name( parameter_list ):
    doc string
    statements
```

- parameter_list
  - Zero or more parameters separated by commas in the form

    param$_1$, param$_2$, …, param$_N$

```
def printBar(n, ch):
    "A function that prints ch n times"
    while (n > 0):
        print(ch, end="")
        n = n - 1
    print("")

printBar(17, '#')
print("  Hello World!")
printBar(17, '*')
```

Documentation string

```
#################
  Hellow World!
*****************
```

Example:

An improved version of **`printBar()`**

```python
def foo(x, y):
    print("{} {}" .format(x, y))

x = 3
y = 2

foo(x, y)

foo(y, x)

```

Example: Defining and calling a function with multiple parameters

- What is the output produced in the above example?

- Arguments and parameters are matched by positions (not by names !).

```python
def foo(x, y):
    print("{} {}" .format(x, y))

foo(x=3, y=2)

foo(y=2, x=3)

foo(3, y=2)
```

```
3 2
3 2
3 2
```

Keyword arguments

- Arguments in function call can also be in the form of keyword (match by names)
- Must use keyword argument for all others once used for the first of them

```python
def foo(x, y=2):
    print("{} {}" .format(x, y))

foo(x=3)

foo(3)

foo(x=2, y=3)

foo(2, 3)
```

```
3 2
3 2
2 3
2 3
```

Default arguments

- an argument that assumes a default value if a value is not provided in the function call for that argument
- Default argument must appear later than those without default value i.e. (x=3, y) is not allowed

3. Returning a Value From a Function

```
def cube(x) :
    return x * x * x






print( "Cube of 3 is {}" .format(cube(3)) )
print( "Cube of 8 is {}" .format(cube(8)) )
```

A function can return a value to its caller.

We need to explicitly specify what value (of the proper type) to be returned using the keyword `return` .

```
Cube of 3 is 27
Cube of 8 is 512
```

Defining Functions That Returns a Value

```
def function_name( parameter_list ):
    …
    return expression
```

- `return expression`
  - `return` is a keyword, it returns the value of `expression` to the caller.
  - `expression` is evaluated first before `return` is executed.

Evaluating functions that return a value

```
def cube(x) :
    return x * x * x
```

cube(3) is called first

```
print( "Cube of 3 is {}" .format(cube(3)) )
```

When **cube(3)** finishes, the value it returns becomes the value of the expression represented by "**cube(3)**", which is 27.

```
print( "Cube of 3 is 27" )
```

Evaluating functions that return a value

In general, functions are called first if they are part of an expression.

```
x = cube(1) + cube(2) * cube(3);
```

```
x = 1 + cube(2) * cube(3);
```

```
x = 1 + 8 * cube(3);
```

```
x = 1 + 8 * 27;
```

```
x = 1 + 216;
```

```
x = 217;
```

Interrupting Control Flow with `return`

A **`return`** statement can also force execution to leave a function and return to its caller immediately.

```
def smaller(x, y):
    if (x > y):
        return y
    return x
```

When "`return y`" is executed, execution immediately stops in **`smaller()`** and resumes at its caller.
So in this example, if (**`x > y)`** is true, "**`return x`**" will not be executed.

Example (with multiple **`return`**'s)

```
def daysPerMonth(m, y):
    " Returns number of days in a particular month "
    if (m == 1 or m == 3 or m == 5 or
       m == 7 or m == 8 or m == 10 or m == 12):
        return 31

    if (m == 4 or m == 6 or m == 9 or m == 11):
        return 30;

    # if y is a leap year
    if (y % 4 == 0 and y % 400 == 0):
        return 29

    return 28
```

Only one of the "return" statements will be executed.

Example (with only one **return**)

```
def daysPerMonth(m, y):
    " Returns number of days in a particular month "
    if (m == 1 or m == 3 or m == 5 or
       m == 7 or m == 8 or m == 10 or m == 12):
        days = 31

    elif (m == 4 or m == 6 or m == 9 or m == 11):
        days = 30;
    else:
        # if y is a leap year
        if (y % 4 == 0 and y % 400 == 0):
            days = 29
        else:
            days = 28
    return days
```

A function is easier to debug if there is only one `return` statement because we know exactly where an execution leaves the function.

The `return` keyword

- If there is no data to be returned, write

  `return`

```
def askSomething( code ):
    if (code != 7):
        print("Who are you?")
        return   # Leave the function immediately
    print("How are you today, James?");
    return;   # This return statement is optional
```

If nothing to return, placing a return as the last statement is optional (it is implied).

None Value

- In Python, there is a value called None, which means absence of value
- In other language , it is called Nil, null , etc.
- For function return nothing, None is being returned

```
def answerNothing( ):
    print("do something")
    return

>>> code = answerNothing()
do something
>>> code == None
True
```

None is being returned

```python
def outerFun(a, b):
    def innerFun(c, d):
        return c + d
    return innerFun(a, b)

res = outerFun(5, 10)
print(res)
```

```
15
```

Nested Function

- Allows function defined within a function
- innerFun only available within outerFun

Return List and Tuple

- Used as body of a function

```
def ListofSquares( a ):
    b = []
    for x in a: b.append(x*x)
    return b
>>> ListofSquares([1,2,3])
[1, 4, 9]
```

- Return multiple values as a tuple, the braces are optional

```
def TupleofGP( a ):
    return (a, a*a, a*a*a)
>>> TupleofGP(3)
(3, 9, 27)
```

List Comprehensions

- Used as body of a function

```
def ListofSquares( a ):
    return [x*x for x in a]
>>> ListofSquares([1,2,3])
[1, 4, 9]
```

- Operations on dictionaries (to be discussed) performed by selecting values from range of keys, then returning items with selected keys

```
d = {1:'fred', 7:'sam', 8:'alice', 22:'helen'}
>>>[d[i] for i in d.keys() if i%2==0]
['alice', 'helen']
```

Name Scope

- Names defined outside functions have *global* scope
- Any local names will shadow the global (same name)
- All values & names destroyed after return

```
>>> x=4
>>> def scopetest(a):
...    return x + a
...
>>> print (scopetest(3))
 7
>>>
```

```
>>> x=4
>>> def scopeTest(a):
...    x=7
...    return x + a
...
>>> print (scopeTest(3))
10
```

Recursive Function

- Python also accepts function recursion, which means a defined function can call itself.

```
def f( n ):
  if( n==1):
    result = 1
  else:
    result = n * f(n-1)
  return result
>>> print (f(3))
6
```

Using globals

- To have assignment access on global variables, use global statement

```
>>> def scopeTest (a):
...   global b
...   b = 4
...   print ('inside func, b is ', b)
...
>>> a = 1
>>> b = 2
>>> scopeTest(a)
inside func, b is  4
>>> print ('after func, b is ', b)
after func, b is  4
```

Raise exception

- When a program is running at a point that the current scope/function cannot solve the problem, we can raise an exception
- Raise exception will cause program execution halted, thus programmer can check for the error using information provided

```
def compoundYear( balance, rate, numYears):
    if rate < 0:
          raise RuntimeError("-ve interest rate")
      if numYears < 0:
          raise RuntimeError("-ve number of years")
      for year in range(9, numYears):
          balance = compound(balance, rate)
      return balance
  print ('after 10 yrs,', compoundYear(1000, -5, 3))
```

```
Traceback (most recent call last):
    File "COURSE/python/function20.py", line 10, in <module>
  print ('after 10 yrs,', compoundYear(1000, -5, 3))
    File "COURSE/python/function20.py", line 3, in compoundYear
      raise RuntimeError("-ve interest rate")
  RuntimeError: -ve interest rate
```

Lambda function

- Passing function to another function, say filter, we may do it this way

```
def even(x):
    return x % 2 == 0
a = [1,2,3,4,5]
print (list(filter(even, a)))
>>> [2, 4]
```

- We may just want to define a very simple function
- using *def* function becomes quite cumbersome
- lambda is used to pass simple function

```
print (list(filter( lambda x : x % 2 == 0, a)))
[2, 4]
```