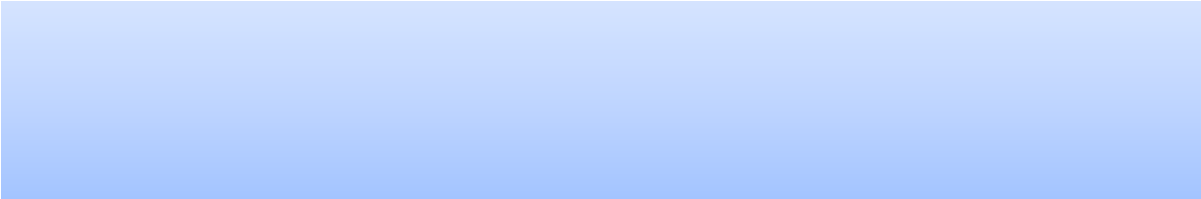
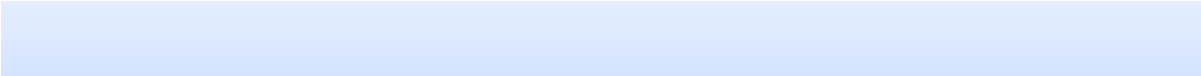


# Functions

* Function is a re-usable block of statements that perform a specific function
* The following is the syntax for writing functions:

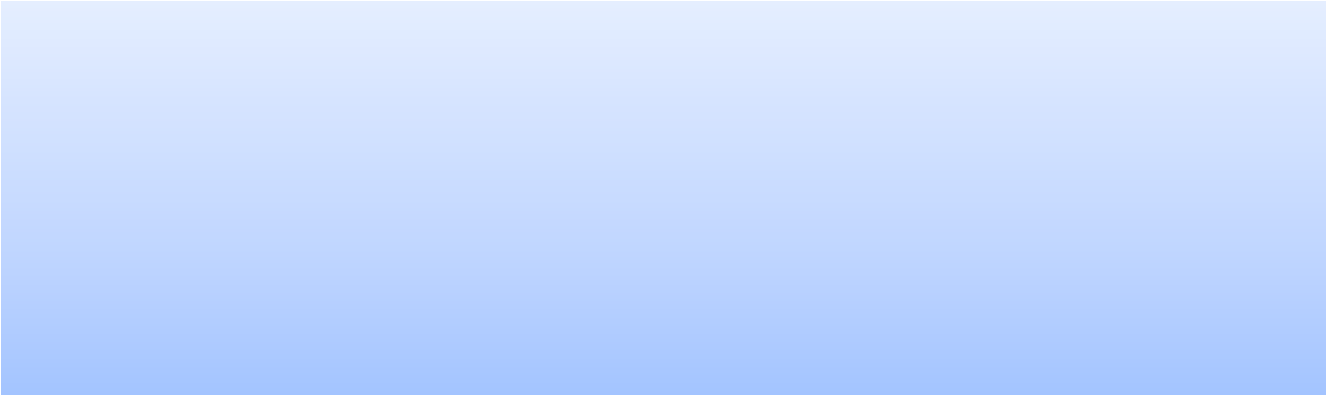
**def functionname( parameters ): "function docstring" function body return [expression]**



* Benefits of using functions:
  + Maximizing code reuse
  + Minimizing redundancy
  + Procedural decomposition

# **def** Statement: Facts

* **def** is executable code
  + Your functions is “not visible” until python reaches and executes **def**
  + It’s legal (and even occasionally useful) to nest **def** statements inside **if** statements, **while** loops and even other **def**s
* **def** creates a new function object and assigns it to a name
  + Function name becomes a reference to a function object
* **def** executes in run time



**if test:**

**def func():**

**# This way**

**else**

**def func():**

**# or this way**

**..**

**func()**

**#call version selected and built**

# **return** Statement

* 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**

# Parameters

* Parameters are like inputs to the function
* They are place in parenthesis
* Also referred to as arguments

# Function Definition: Example



# Function definition

def printme( str):

"This prints a passed string into this function"

print (str)

return

import random

printme("\t\tRandomQuote Generator\t\t")

input('Press any key:')



choice =

random.choice

(

range(1,

6))

# Call printmefunction

if(choice == 1) : printme("Do well today for a better tomorrow")

elif(choice == 2) : printme("The best way to predict the future is to create it")

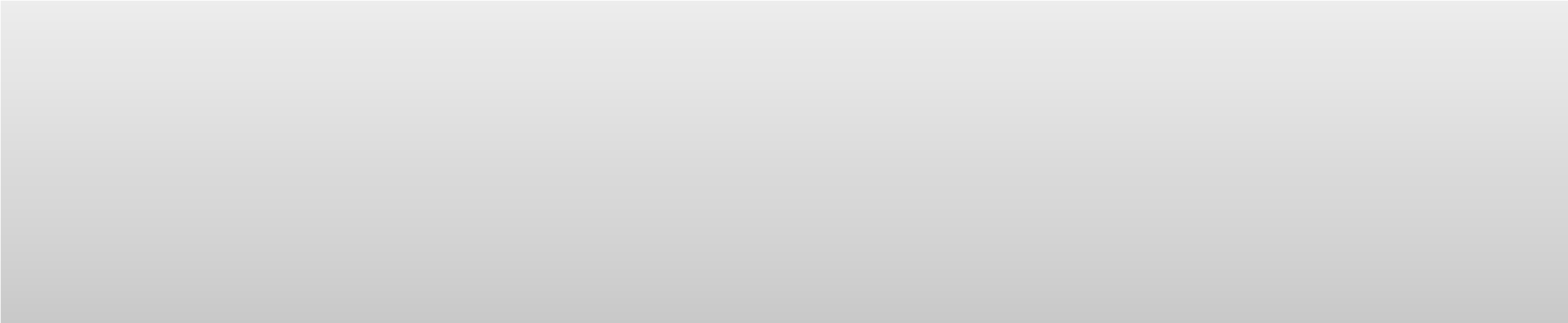
elif(choice == 3) : printme("It's time to reinvent yourself when things are bad")

elif(choice == 4) : printme("The best revenge is overwhelming success")

else : printme("Ametuerslook for inspiration, some of us get up and work!")

printme("Have a nice day")

# Output



>>>

Random Quote Generator

Press any key:

The best revenge is overwhelming success

Have a nice day

>>> ================================ RESTART ================================

>>>

Random Quote Generator

Press any key:



Ametuerslook for inspiration, some of us get up and work!

Have a nice day

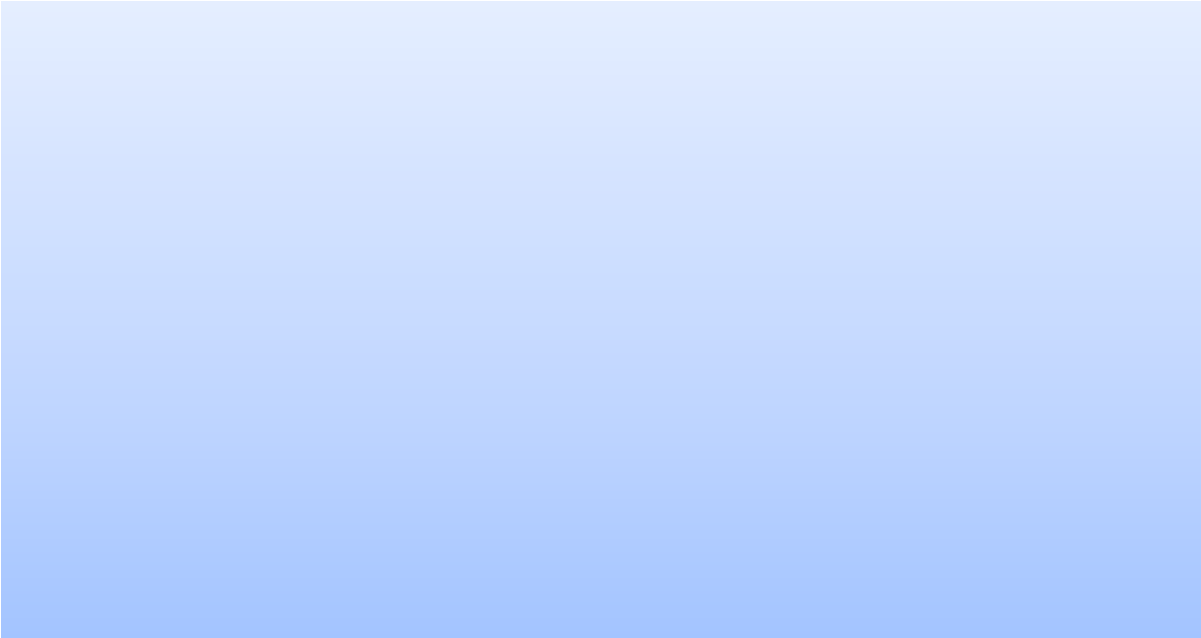
# Scope

* When you use a name in a program, Python creates, changes, or looks up the name in what is known as a **namespace**—a place where names live!
* The term **scope** refers to a namespace
  + The location of a name’s assignment in your source code determines the scope of the name’s visibility to your code
* In a function, names assigned inside a **def** 
  + Can only be seen by that def
  + Do not clash with variables outside def

# Scope

* In general,
  + If a variable is assigned inside a **def**, it is local to that function
  + If a variable is assigned in an enclosing **def**, it is nonlocal to nested functions
  + If a variable is assigned outside all **def**s, it is global to the entire file
* Assigned names are local unless declared gloabal or nonlocal
* Each call to a function creates a new local scope

# Scope Example



N = 10

def print\_star():

N = 5

# local to print\_star()

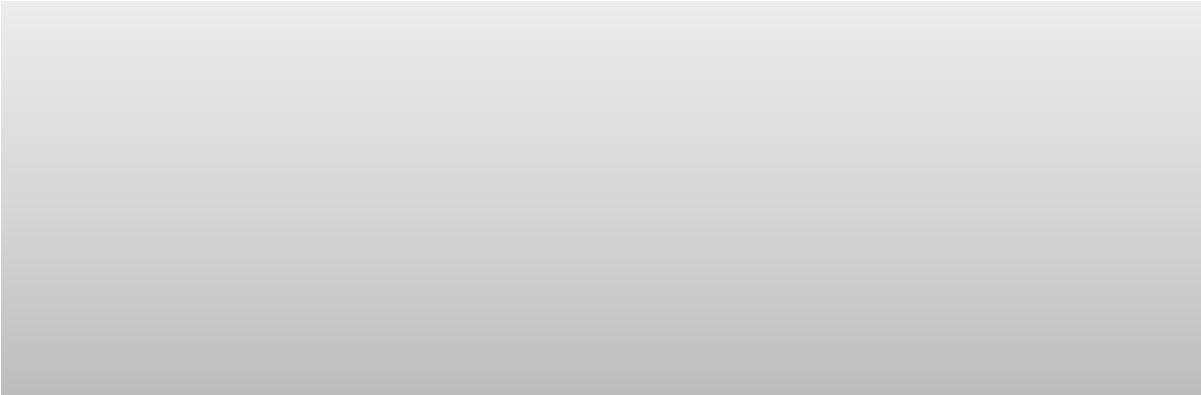
for iin range(1, N + 1):

for j in range (i):

print('\*', end = '')

print()

print\_star()



>>>

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*



N = 10

def print\_star():

global N

# Accessing global variable

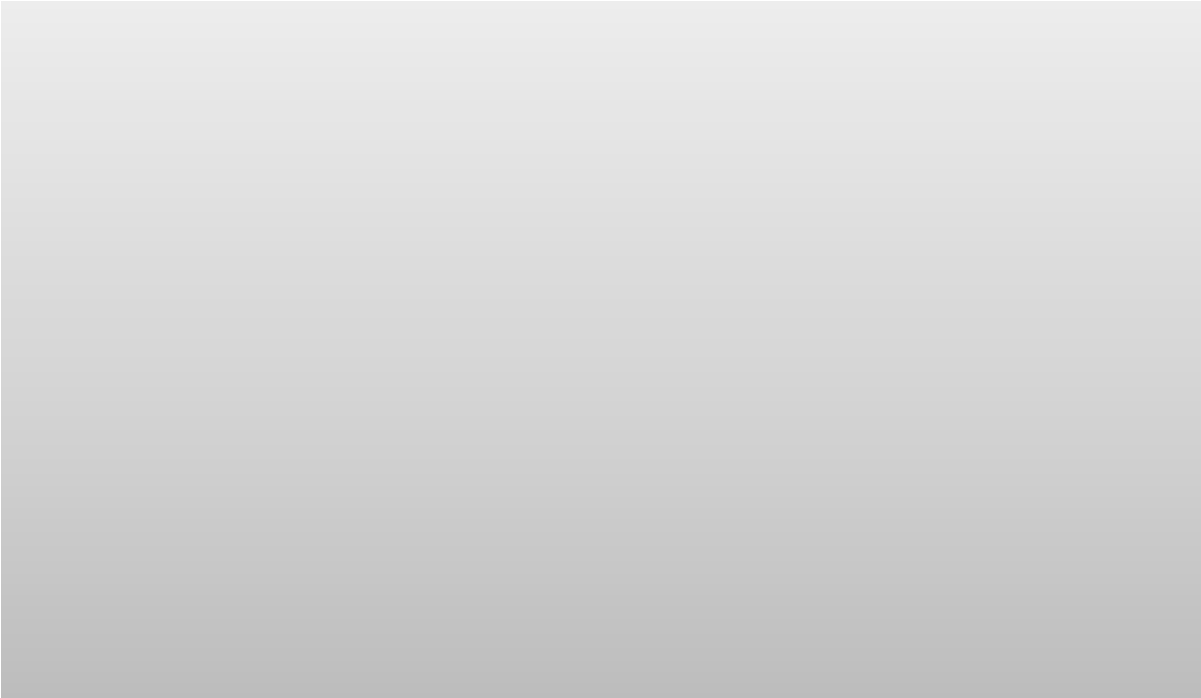
for iin range(1, N + 1):

for j in range (i):

print('\*', end = '')

print()

print\_star()



>>>

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

# Other Ways to Access Globals



var= 99

def local():

var= 0

**# Change local var**

def glob1():

global var

**# Declare global (normal)**

var+= 1

**# Change global var**

def glob2():

var= 0

**# Change local var**

import def\_03\_globals\_otherways

**# Import myself**

def\_03\_globals\_otherways.var += 1

**# Change global var**



def glob3():

var= 0

**# Change local var**

import sys

**# Import system table**

glob = sys.modules['def\_03\_globals\_otherways']

**# Get module object (or use \_\_name\_\_)**

glob.var += 1

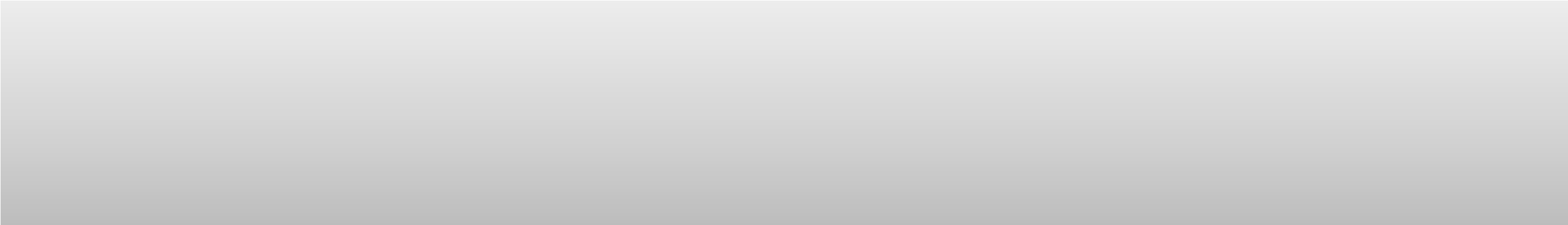
**# Change global var**

def test():

print(var)

local(); glob1(); glob2(); glob3()

print(var)



>>> ================================ RESTART ================================

>>>

>>>

import def\_03\_globals\_otherways

>>>

def\_03\_globals\_otherways.test

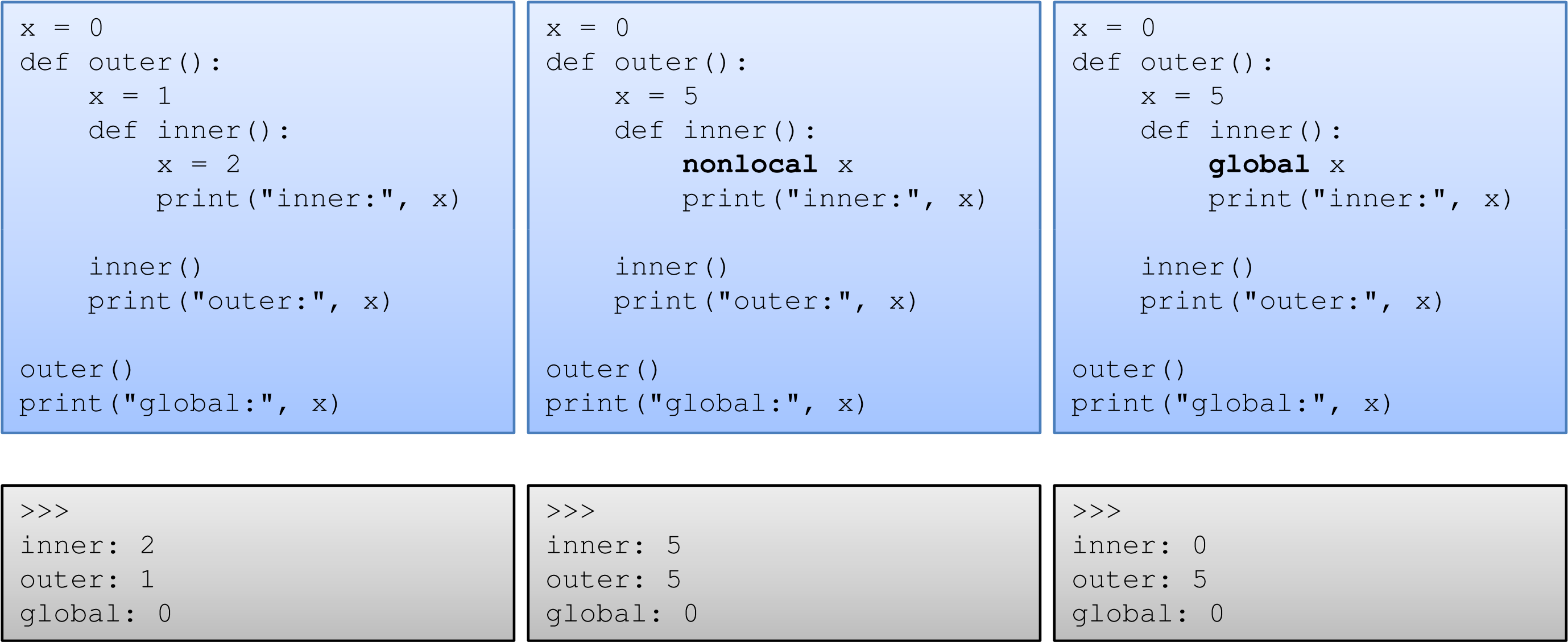
()

99

102

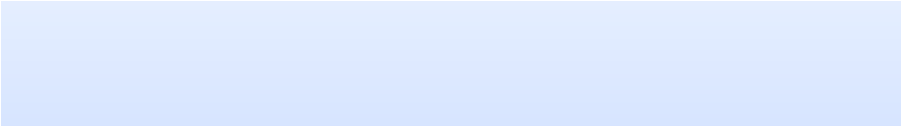
# Scopes and Nested Functions

• Let’s illustrate use of **global** and **nonlocal** with some examples



# Minimize Global Variables

* Though globals are very useful, it is a standard guideline to use it as less in number as possible
* Let’s illustrate this:



X = 99



def func1():

global X

X = 88

def func2():

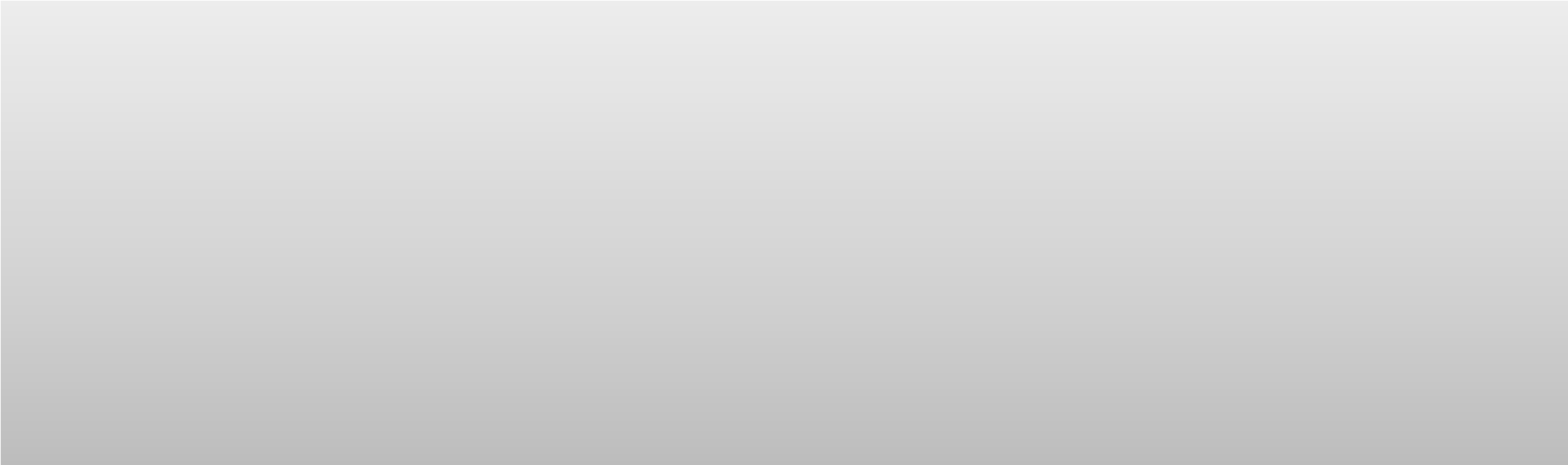
global X

X = 7

* If you were to import this module, how do you predict the value of X here?

# Required Arguments

* Required arguments are the arguments passed to a function in correct positional order
* The number of arguments in the function call should match exactly with the function definition



>>>

def average(a, b, c

):

return (a + b + c)/3

>>>

average(2,

3)

Traceback(most recent call last):

File "<pyshell#1>", line 1, in <module>

average(2, 3)

TypeError: average() takes exactly 3 positional arguments (2 given)

>>>

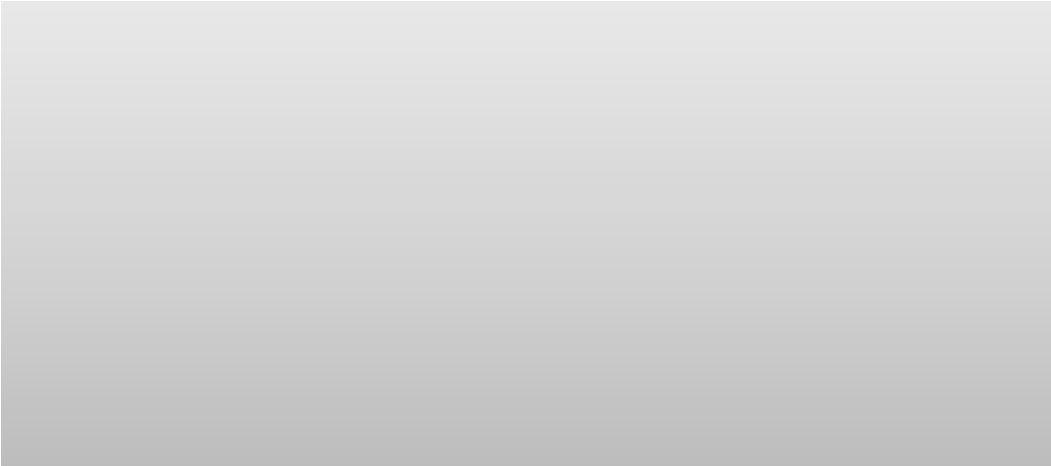
average(2, 3,

4)

3.0

# Passing Different Types

• Visualize every argument as an object and you can pass it to the function as shown in the below examples:



>>>

def

tf

(

T

):

for (x,y) in T:

print(x, y)

tf(((1,2),

>>>

(2,3)))

1

2

2

3



>>>

def

tf

(

N, L, T

):

print(L)

for (x,y) in N:

print(x, y)

print(T.keys())

>>>

N = ((1,2), (2,3))

**# Tuple**

>>>

T = { 'One': 1, 'Two': 2 }

# Dictionary

>>>

L = 56

# Number

>>>

tf(N, L, T

)

56

1

2

2

3

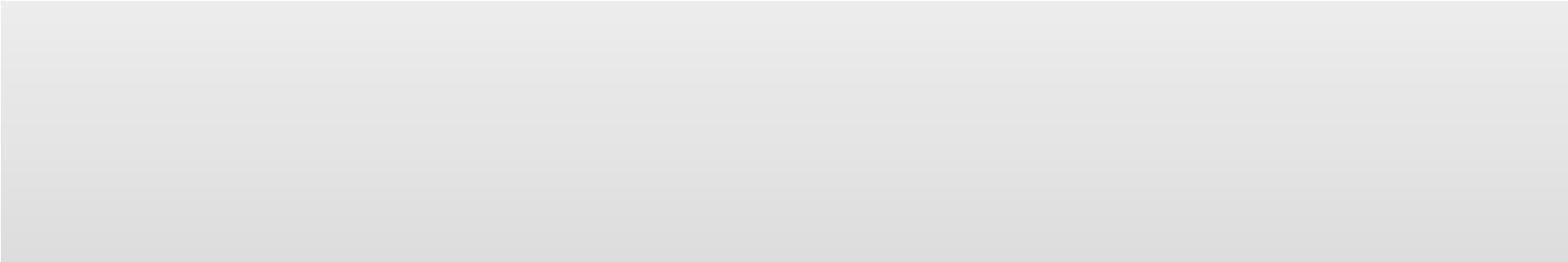
dict\_keys(['Two', 'One'])

# Order of Arbitrary Arguments

* In a function call, arguments must appear in this order:
  + any positional arguments (value);
  + followed by a combination of any keyword arguments (name=value)
  + the \*iterable form
  + followed by the \*\*dict form
* In a function header, arguments must appear in this order:
  + any normal arguments (name);
  + followed by any default arguments (name=value);
  + followed by the \*name (or \* in 3.X) form;
  + followed by any name or name=value keyword-only arguments (in 3.X); – followed by the \*\*name form
* In both the call and header, the \*\*args form must appear last if present
* If you mix arguments in any other order, you will get a syntax error because the combinations can be ambiguous

# Order of Arbitrary Arguments

• The last two matching extensions, \* and \*\*, are designed to support functions that take any number of arguments



>>>

def f(\*args): print(args

)

>>>

f

()

()



>>>

f(1,

2)

(1

,

2)

>>>

f((1,2),

(3,4))

((1

, 2), (3,

4))

>>>

f(1, 2, 3, x=1, y

=2)

Traceback(most recent call last):

File "<pyshell#25>", line 1, in <module>

f(1, 2, 3, x=1, y=2)

TypeError: f() got an unexpected keyword argument 'x'

# Order of Arbitrary Arguments

Study the behavior carefully

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

>>> f(1, 2, 3, x=1, y=2)

1 (2, 3) {'y': 2, 'x': 1}

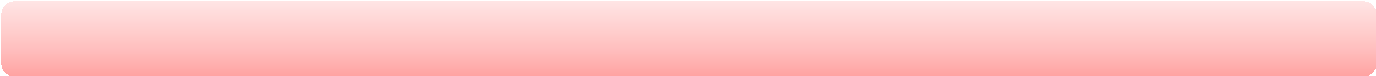
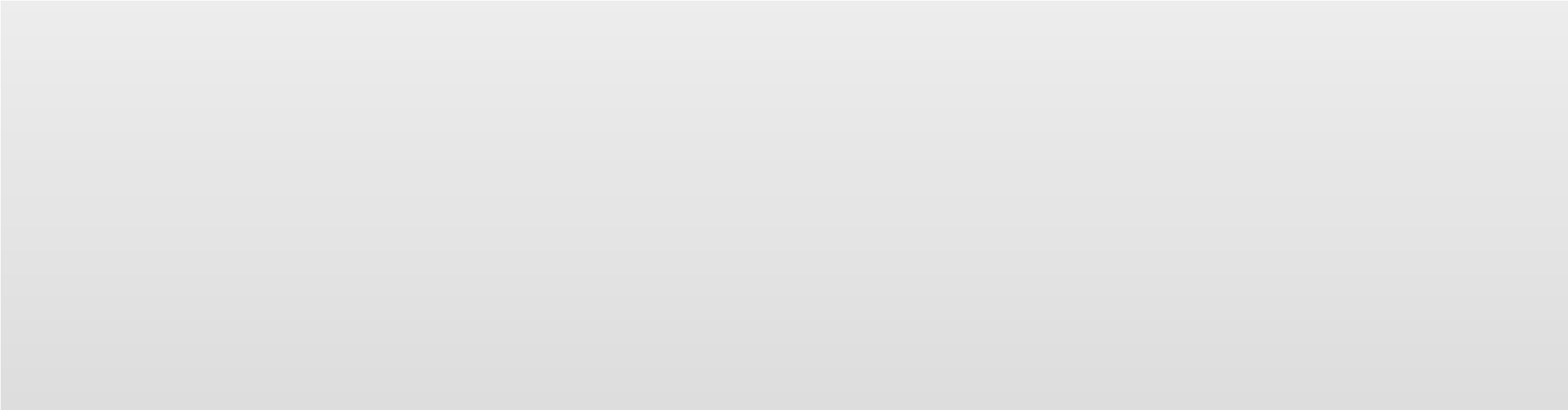
>>> f(1, 2, 3, 4, 5, 6, 7, 'one'=1, 'two'=2, 'three'=3)

SyntaxError: keyword can't be an expression (<pyshell#29>, line 1)

>>> f(1, 2, 3, 4, 5, 6, 7, one=1, two=2, three=3)

1 (2, 3, 4, 5, 6, 7) {'three': 3, 'two': 2, 'one': 1}

>>> def f(a, b, \*\*kargs, \*pargs): print(a, b, kargs, pargs) SyntaxError: invalid syntax (<pyshell#41>, line 1)



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

>>> f(1, 2, 3, 4, 5, 6, 7, one=1, two=2, three=3)

1 2 (3, 4, 5, 6, 7) {'three': 3, 'two': 2, 'one': 1}

>>> f(1, 2, 7, one=1, two=2, three=3)

1 2 (7,) {'three': 3, 'two': 2, 'one': 1}

>>> f(1, 2, one=1, two=2, three=3)

1 2 () {'three': 3, 'two': 2, 'one': 1}

>>> def f(a, b, \*\*kargs): print(a, b, kargs)

>>> f(1, 2, 3, 4, 5, 6, 7, one=1, two=2, three=3) Traceback (most recent call last):

File "<pyshell#38>", line 1, in <module> f(1, 2, 3, 4, 5, 6, 7, one=1, two=2, three=3)

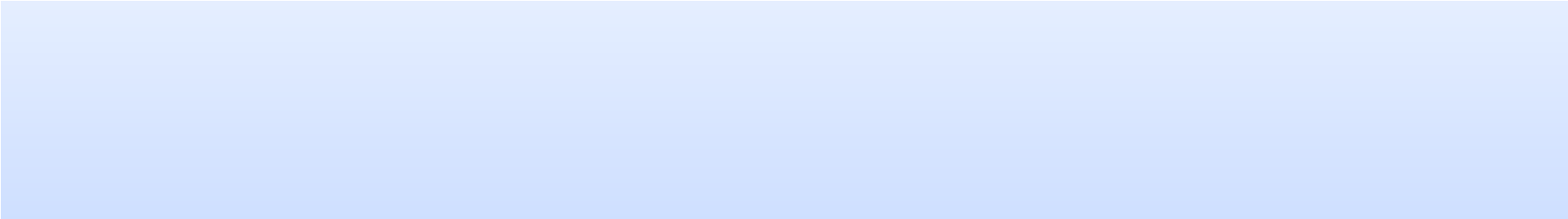
TypeError: f() takes exactly 2 non-keyword positional arguments (7 given)

>>> f(1, 2, one=1, two=2, three=3)

1 2 {'three': 3, 'two': 2, 'one': 1}

# 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

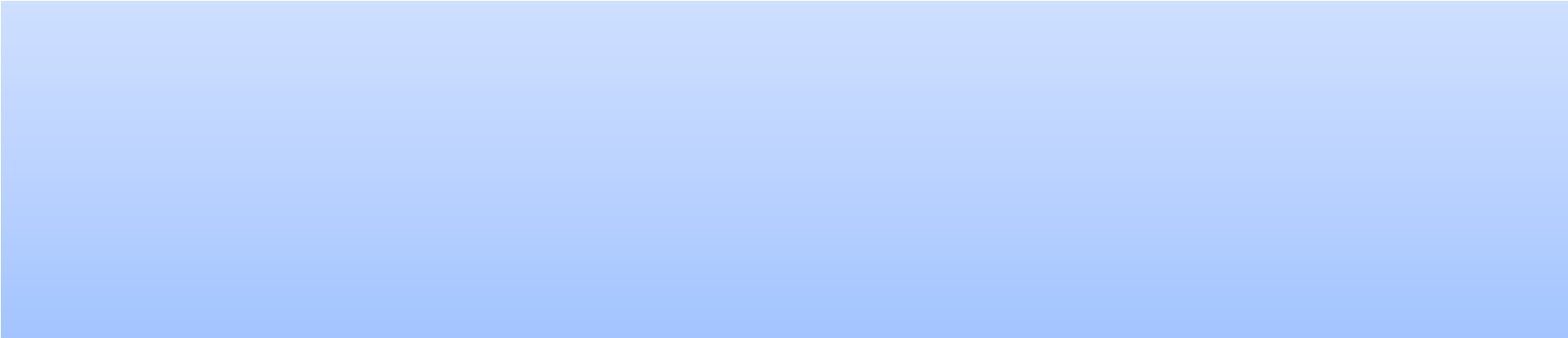


# Function definition

def printinfo( name, age = 35 ):

"This prints a passed info into this function"

print ("Name: ", name)



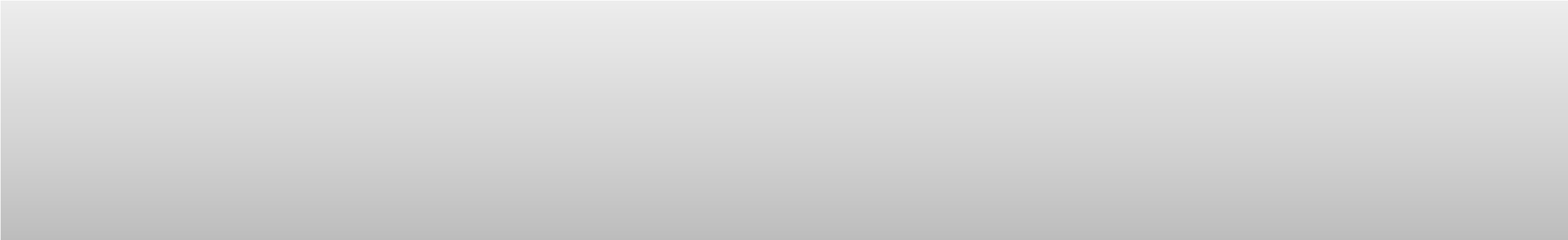
print ("Age ", age)

return;

# Call printinfofunction

printinfo( age=50, name="Tom" )

printinfo( name="Jerry" )



>>> ================================ RESTART ================================

>>>

Name: Tom

Age 50

Name: Jerry

Age 35

# Key-word Arguments

• In this, parameters are passed to the function definition by name as shown in the example



# Function definition

def printinfo( name, age, quote\_of\_the\_day= 'Have a nice day!'):

"This prints a passed info into this function"

print ("Name : ", name)



print ("Age : ", age)

print (quote\_of\_the\_day)

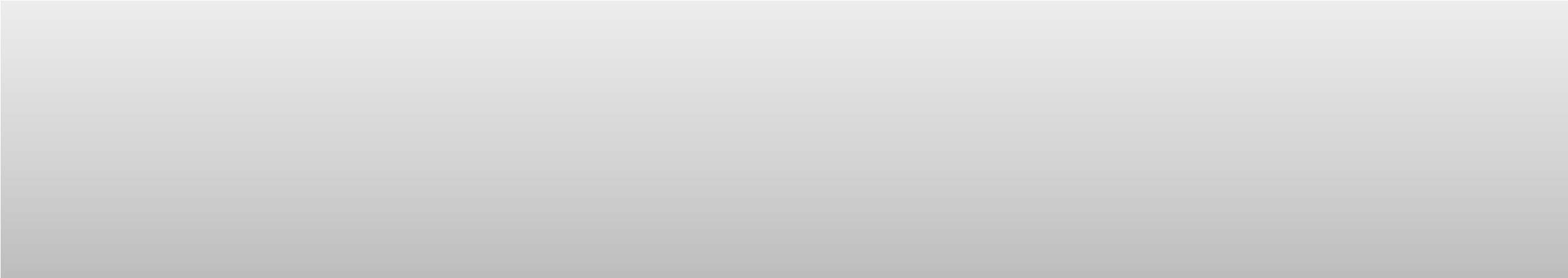
return

# Call printinfofunction

printinfo( age=50, name='Dennis' )

printinfo( name="Calvin", age='15', quote\_of\_the\_day='Be

yourself!' )



>>>

Name : Dennis

Age : 50

Have a nice day!

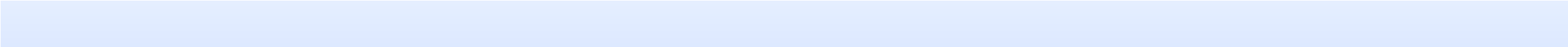
Name : Calvin

Age : 15

Be yourself!

# Returning Values

* The return statement exits a function, optionally passing back an object to the caller.
* A return statement with no arguments is the same as return None.



def tracer(func, \*pargs, \*\*kargs):

# Accept arbitrary arguments



print('calling:', func.\_\_name\_\_)

return func(\*pargs, \*\*kargs)

# Pass along arbitrary arguments

def func(a, b, c, d):

return a + b + c + d

print(tracer(func, 1, 2, c=3, d=4))



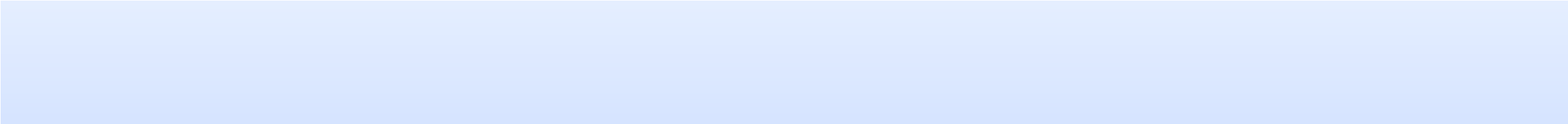
>>>

calling: func

10

# Returning Values

* A tuple, dictionary and even a function object can be returned
* The following example shows how a function object is returned



def make(label):

def echo(message):

print(label + ':' + message)



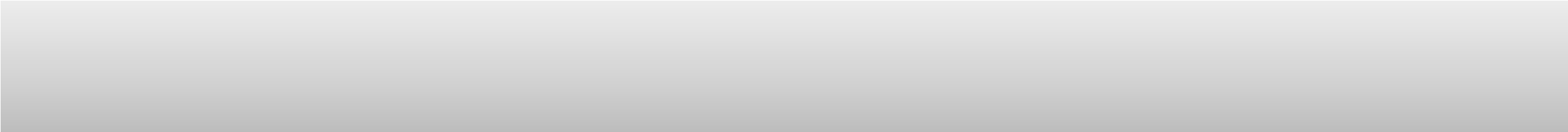
return echo

F = make('Joke of the day:')

# Now F becomes callable

F('Two things are infinite, the universe and human stupidity, and I am not yet

completely sure about the universe. -Albert Einstein')



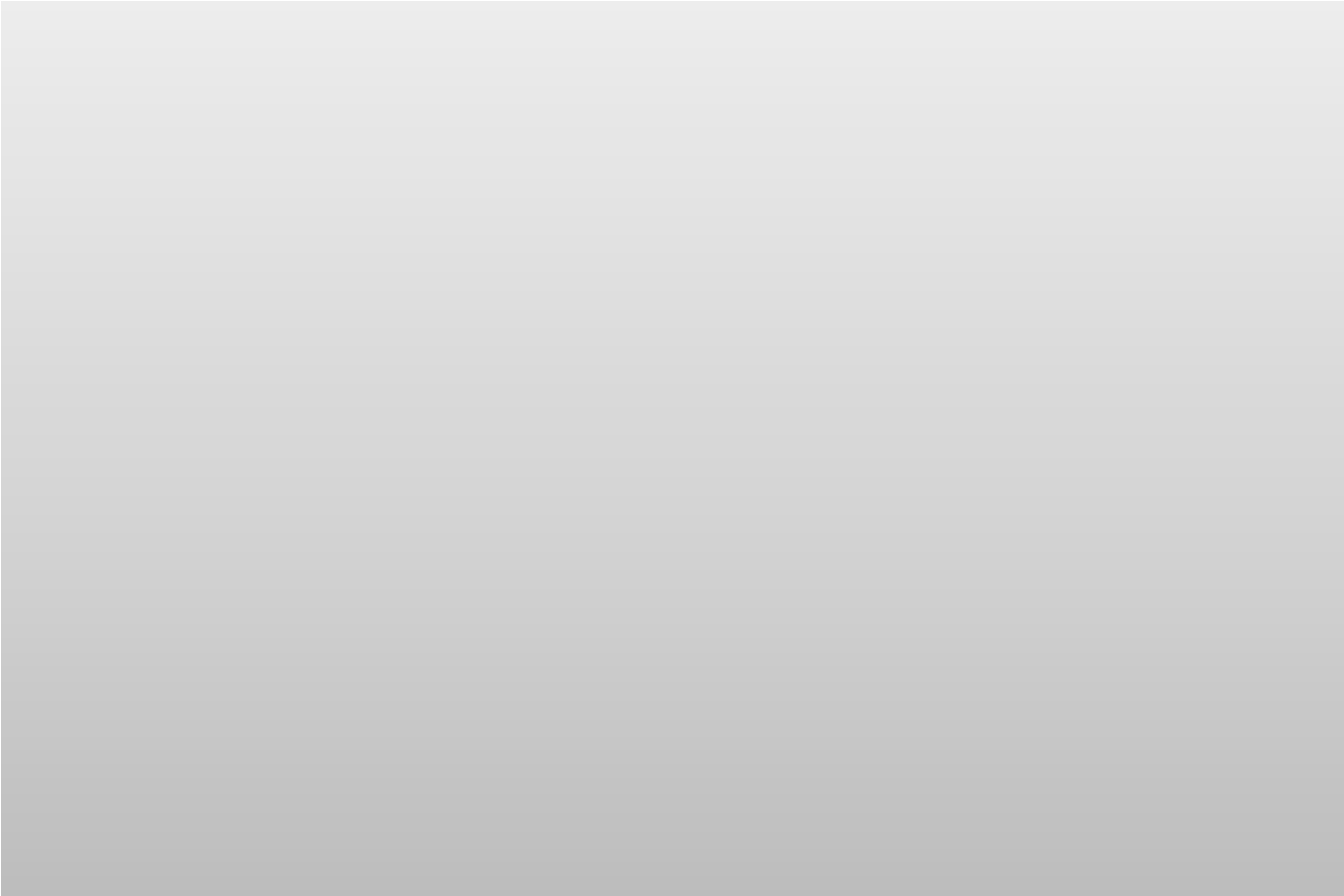
>>>

Joke of the day::Two things are infinite, the universe and human stupidity, and I

am not yet completely sure about the universe. -Albert Einstein

# Recursive Functions

* Recursion functions call themselves repeatedly until a certain condition is met
* It’s a useful technique to know and use in programs that traverse structures that have arbitrary and unpredictable shapes and depths such as
  + Planning travel routes



>>>

def mysum(L

):

print(L) # Trace recursive levels

if not L: # L shorter at each level

return 0

else:

return L[0] + mysum(L[1:])

>>>

mysum([1, 2, 3, 4,

5])

[1

, 2, 3, 4,

5]

[2

, 3, 4,

5]

[3

, 4,

5]

[4

,

5]

[5]

[]

15

* + Analyzing language
  + Crawling links on the web

# Recursion: Pros and Cons

* Pros of recursion
  + Recursive functions make the code look clean and elegant
  + A complex task can be broken down into simpler subproblems using recursion
  + Sequence generation is easier with recursion than using some nested iteration
* Cons of recursion
  + Sometimes the logic behind recursion is hard to follow through
  + Recursive calls are expensive (inefficient) as they take up a lot of memory and time
  + Recursive functions are hard to debug

## A Practical Example: Recursion

• Factorial of a number is the product of all the integers from 1 to that number. Write a python program to print the factorial of a number

# Solution and Output



# An example of a recursive function to

# find the factorial of a number

def factorial(x):

"""This is a recursive function

to find the factorial of an integer"""

if x == 1:

return 1

else:



return (x \* factorial(x

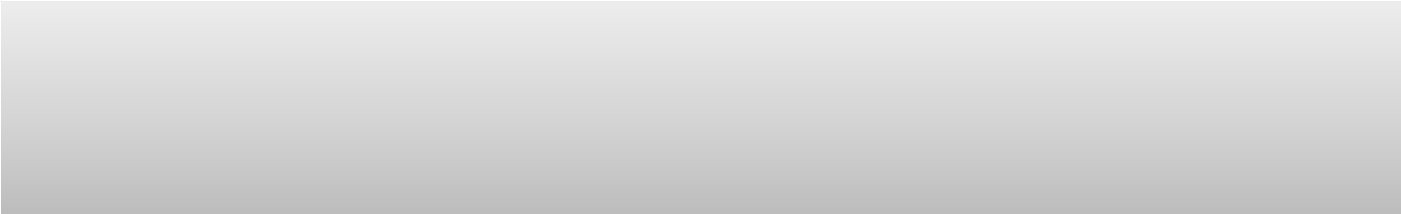
-

1))

num = int(input("Enter a number: "))

if num >= 1:

print("The factorial of", num, "is", factorial(num))



>>>

Enter a number: 6

The factorial of 6 is 720

# Lambda Functions

* The lambda operator or lambda function is a way to create small anonymous functions, i.e. functions without a name
* These functions are throw-away functions, i.e. they are just needed where they have been created
* Lambda functions are mainly used in combination with the functions filter(), map() and reduce()
* The lambda feature was added to Python due to the demand from Lisp programmers



>>>

f = lambda x, y: x + y

>>>

f(10,

20)

30



More information while discussion on functions in p

ython

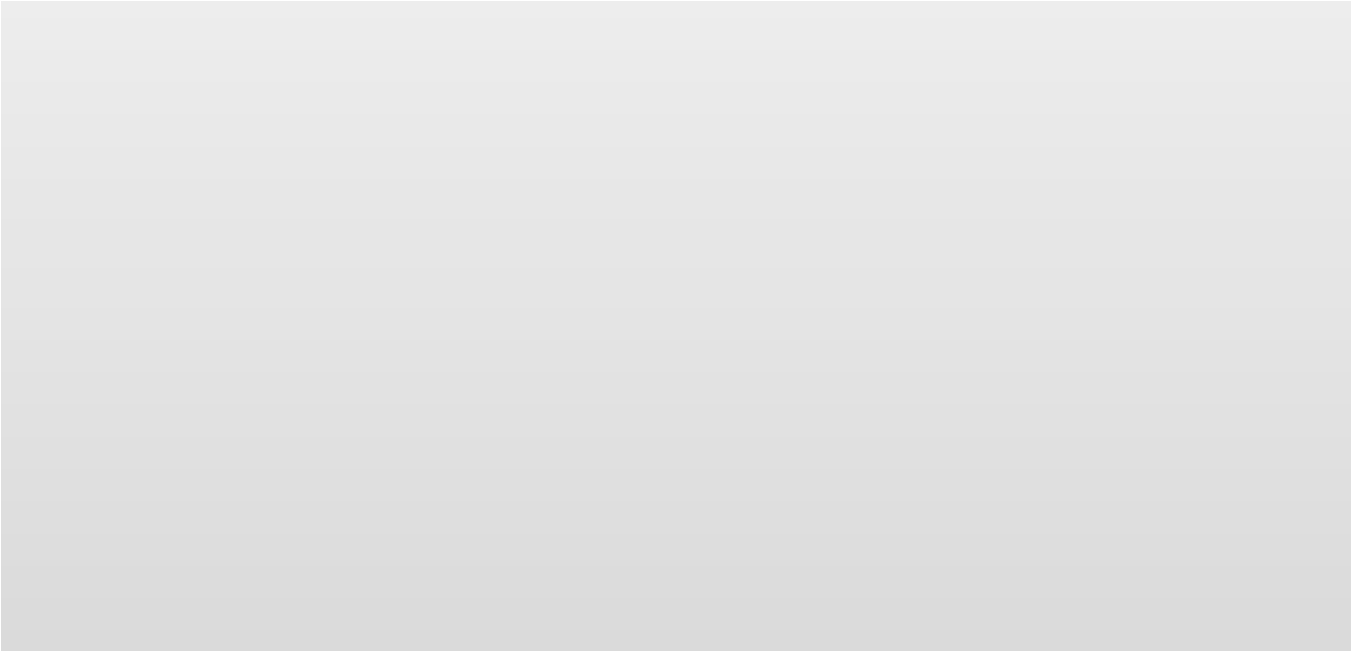
# **map()**

* The advantage of the lambda operator can be seen when it is used in combination with the **map()** function
* **map()** is a function with two arguments:

**r = map(function, sequence)**

* + The first argument *function* is the name of a function
  + The second a sequence (e.g. a list) *sequence*
  + **map()** applies the function *func* to all the elements of the sequence *seq*. It returns a new list with the elements changed by *func*

# **map() :** Usage



):

def fahrenheit(T

>>>

return ((float(9)/5) \* T + 32)

):

def celsius(T

>>>

return (float(5)/9)\*(T-32)

>>>

temp = (36.5, 37,

37.5,39)

)

>>>

F = map(fahrenheit, temp

)

>>>

C = map(celsius, F

C

>>>



>>>

F = map(fahrenheit, temp

)

>>>

fah\_seq= list(F

)

>>>

fah\_seq

99.5, 102.2]

[97.7

, 98.60000000000001,

)

C = map(celsius, fah\_seq

>>>

C

>>>

<

map object at 0x0137DD

10>

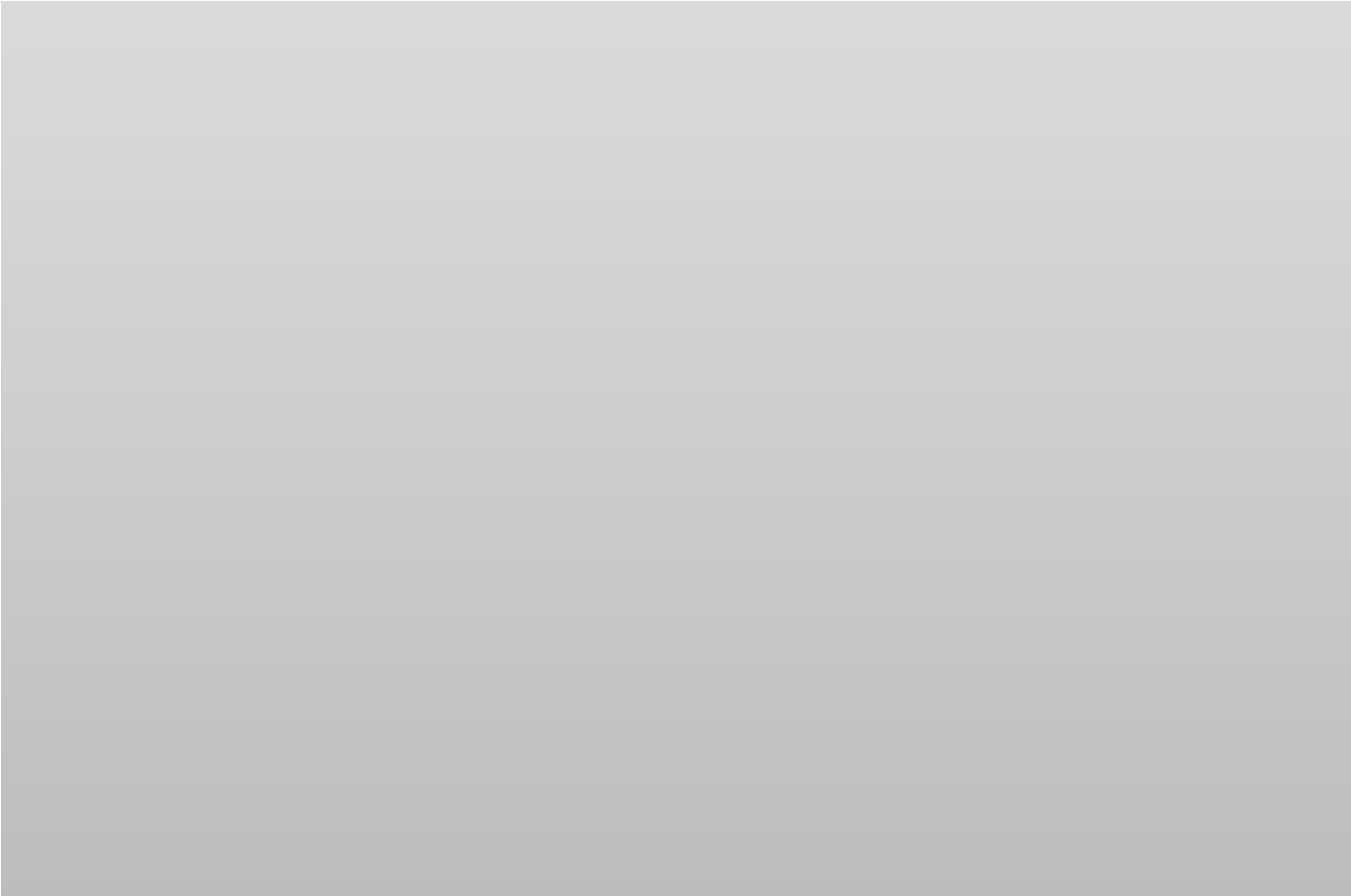
>>>

cel\_seq= list(C

)

>>>

cel\_seq



map object at 0x0137D6D

0>

<

>>>

list(F

)

99.5, 102.2]

, 98.60000000000001,

[97.7

>>>

)

C = map(celsius, F

>>>

C

<

map object at 0x0137DE

10>

list(C

)

>>>

[]

)

fah\_seq= list(F

>>>

fah\_seq

>>>

[]

F

>>>

50>

map object at 0x0137DD

<

>>>

list(F

)

[]



[36.5

, 37.00000000000001, 37.5,

39.0]

>>>



Python 3.x: Creates a map

object which should be listed.

Study the behavior carefully

# **map()** with **lambda**

• By using lambda, we wouldn't have had to define and name the functions fahrenheit() and celsius() in previous example.

You can see this in the following interactive session:



>>>

Celsius = [39.2, 36.5, 37.3,

37.8]

Fahrenheit = map(lambda x: (float(9)/5)\*x + 32, Celsius

>>>

)

>>>

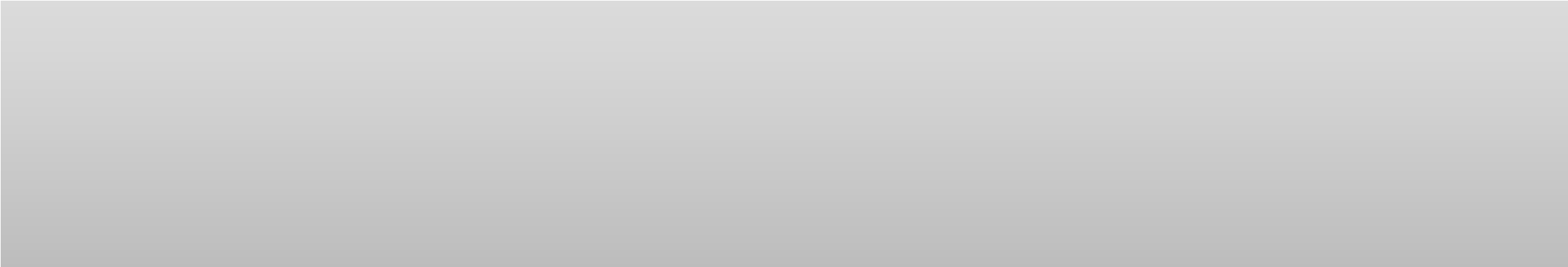
print(Fahrenheit

)

<

map object at 0x0137D1D

0>



>>>

Fahrenheit = list(map(lambda x: (float(9)/5)\*x + 32, Celsius

))

>>>

Fahrenheit

[102.56, 97.7, 99.14, 100.03999999999999]

>>>

C = list(map(lambda x: (float(5)/9)\*(x-32), Fahrenheit

))

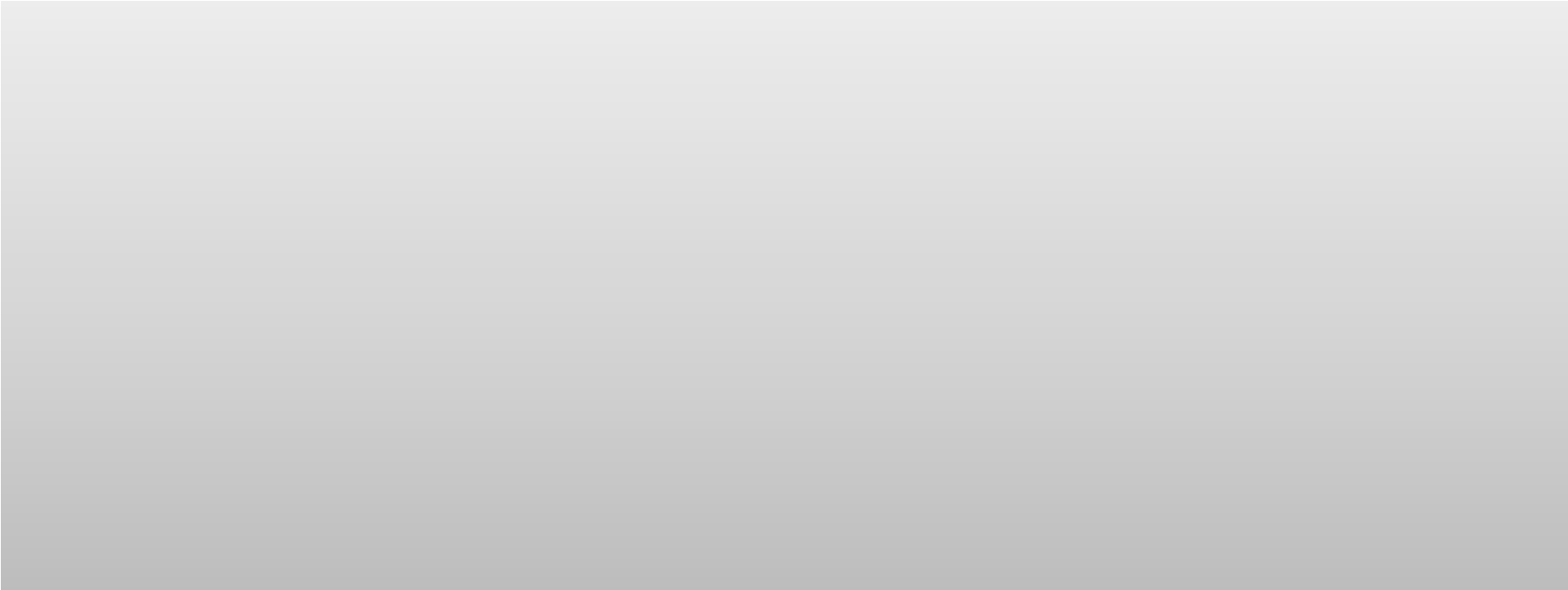
>>>

C

[39.2

, 36.5, 37.300000000000004,

37.8]



>>>

a = [1,2,3,

4]

>>>

b = [17,12,

11,10]

>>>

c = [-1,-4,5,

9]

>>>

map(lambda x,y:x+y, a,b

)

<

map object at 0x0137DCD

0>

>>>

list(map(lambda x,y:x+y, a,b

))

[18

, 14, 14,

14]

>>>

list(map(lambda x,y,z:x+y+z, a,b,c

))

[17

, 10, 19,

23]

>>>

list(map(lambda x,y,z:x+y-z, a,b,c

))

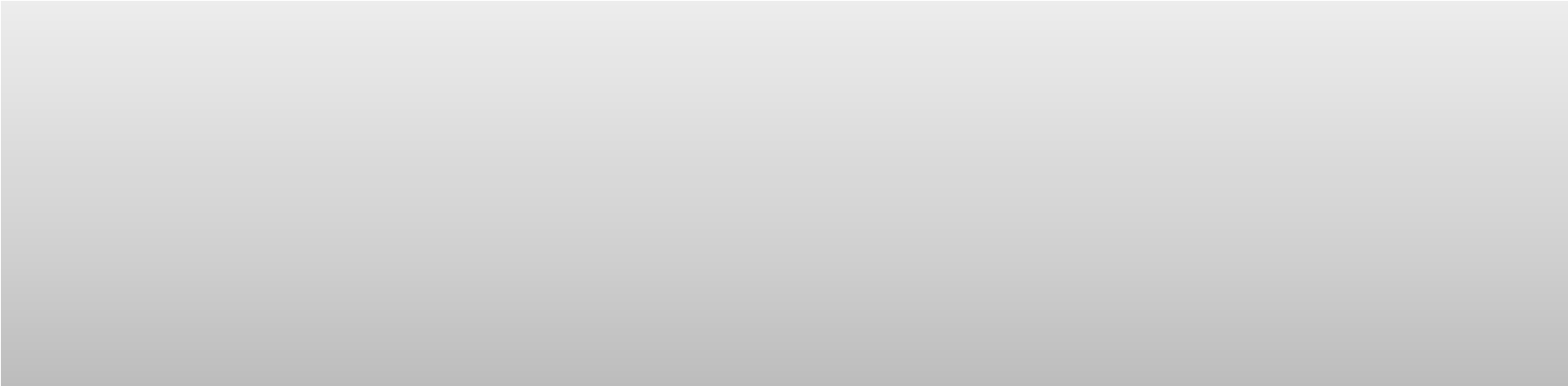
[19

, 18, 9,

5]

## Filtering using **filter()**

* The function **filter(function, list)** offers an elegant way to filter out all the elements of a list, for which the function *function* returns True.
* The function filter() needs a function as its first argument.
  + f returns a Boolean value, i.e. either True or False
  + This function will be applied to every element of the list
  + Only if f returns True will the element of the list be included in the result list.



fib = [0,1,1,2,3,5,8,13,21,

34,55]

>>>

>>>

result = filter(lambda x: x % 2, fib

)

>>>

print(result

)

<

filter object at 0x0137DEB

0>

>>>

result = list(filter(lambda x: x % 2, fib

))

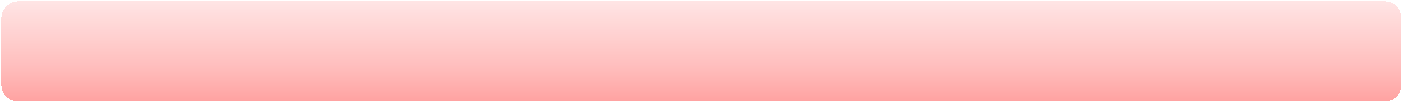
>>>

result

[1

, 1, 3, 5, 13, 21,

55]



What are you trying to do here?

## Reducing a List Using **reduce()**

* The function **reduce(func, seq)** continually applies the function func() to the sequence seq. It returns a single value.
* If seq = [ s1, s2, s3, ... , sn ], calling reduce(func, seq) works like this:
  + At first the first two elements of seq will be applied to func, i.e.

func(s1,s2)

* + The list on which reduce() works looks now like this:

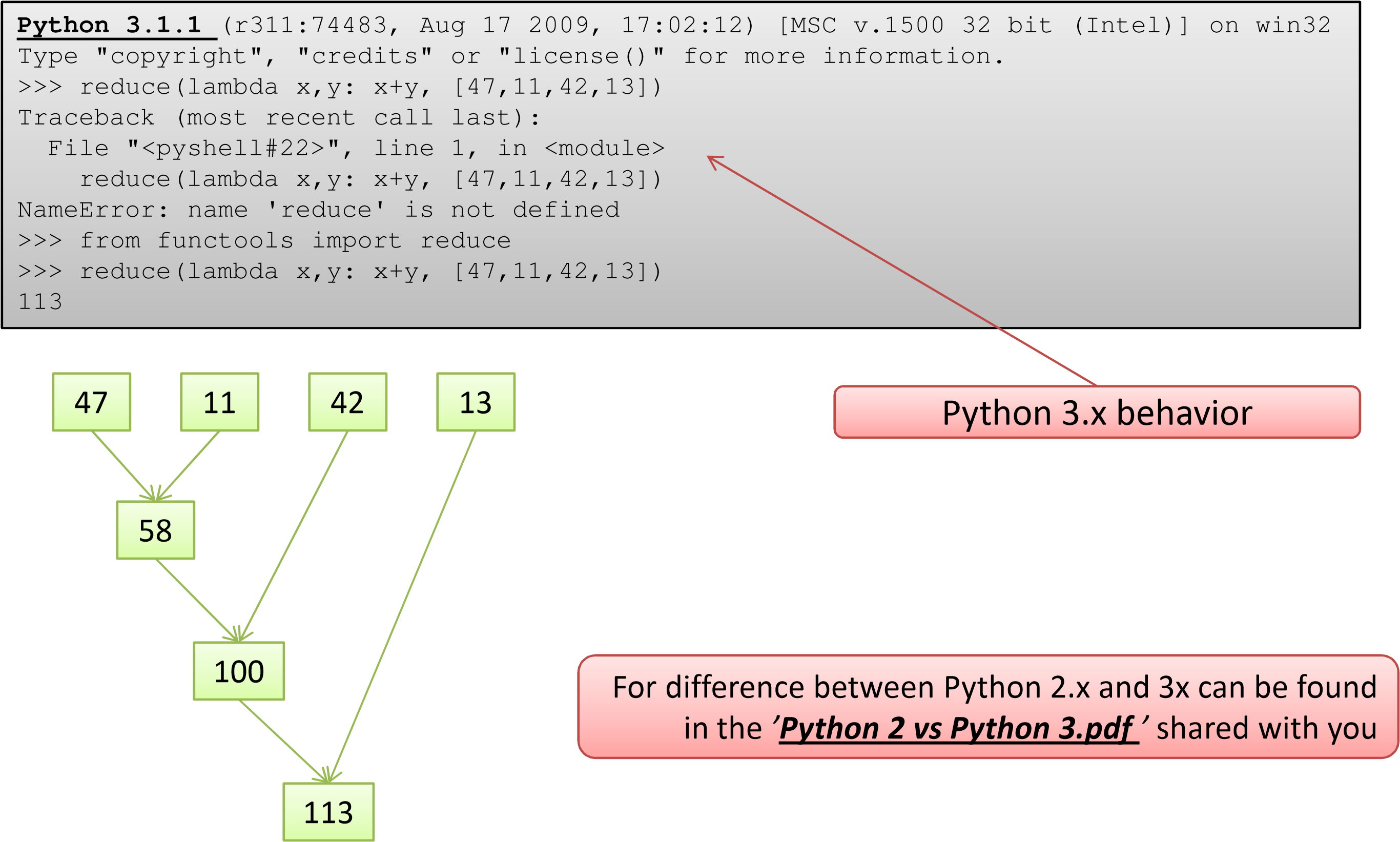
[ func(s1, s2), s3, ... , sn ]

* + In the next step func will be applied on the previous result and the third element of the list, i.e. func(func(s1, s2),s3)

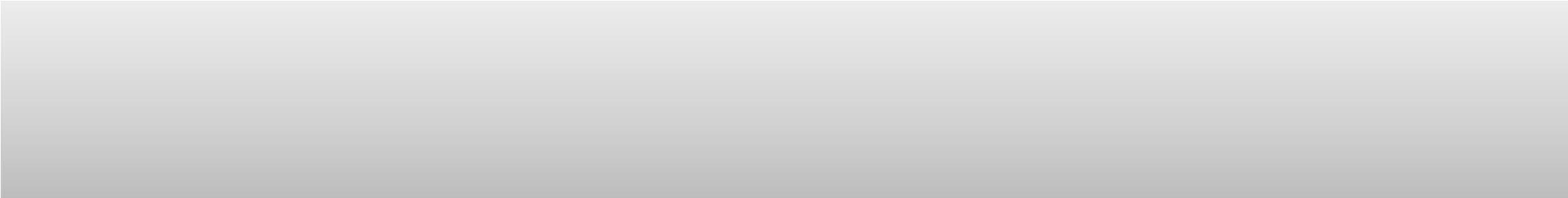
The list looks like this now: [ func(func(s1, s2),s3), ... , sn ]

* + Continue like this until just one element is left and return this element as the result of reduce()

## Reducing a List Using **reduce()**



## Another Example using **reduce()**



>>>

f = lambda a,b: a if (a > b) else b

>>>

reduce(f, [47,

11,42,102,13])

102

>>>

reduce(lambda x, y: x+y, range

(1,101))

5050

For difference between Python 2.x and 3x can be found in the *’****Python 2 vs Python 3.pdf*** *’* shared with you which is an extract of the book **Dive into Python 3** by **Mark Pilgrim**



# **zip()**

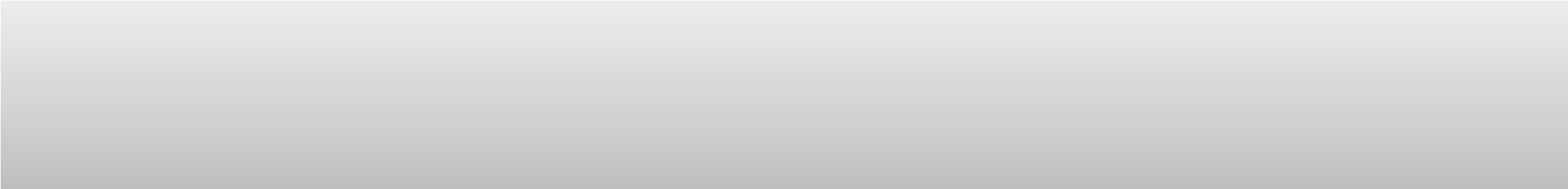
* *zip*() function can take any number of sequences and return a list of tuples.
* The first tuple contained the first item from each sequence; the second tuple contained the second item from each sequence; and so on.

>>> zip((1,2,3,4,5),('one','two','three','four','five'))

<zip object at 0x01381C10>

>>> list(zip((1,2,3,4,5),('one','two','three','four','five')))

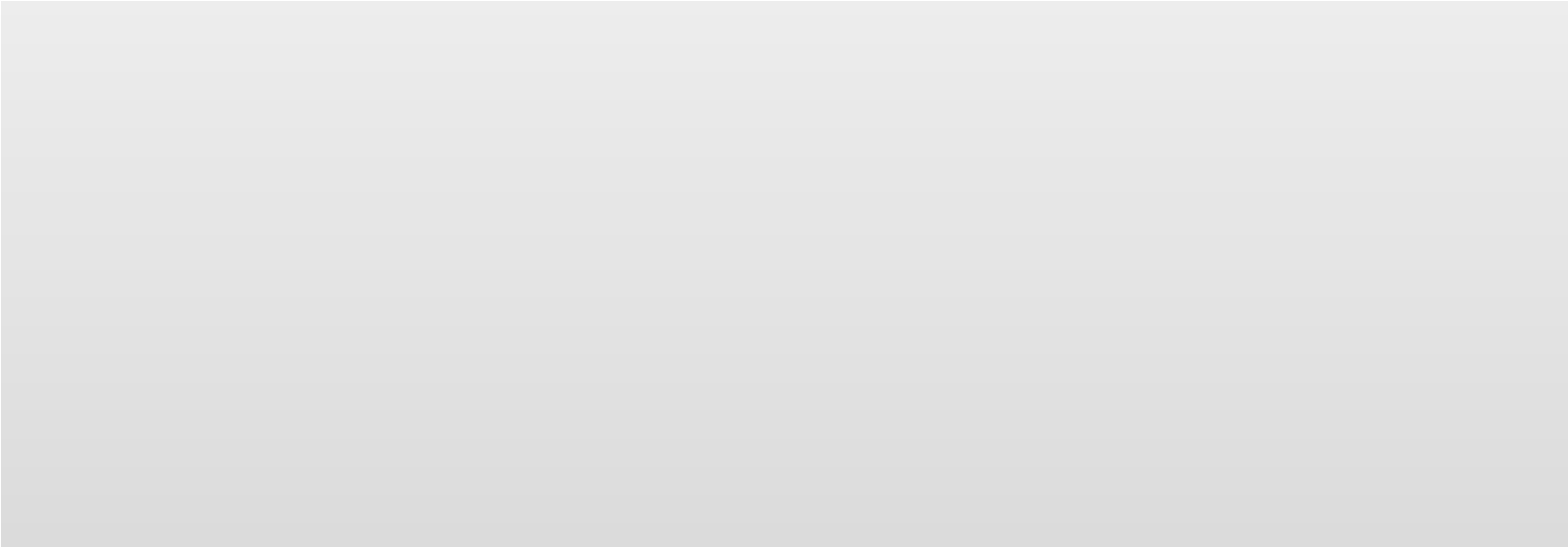
[(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four'), (5, 'five')]



# **yield**

* **yield** is a keyword that is used like return, except the function will return a generator
* Yield is useful when returning large set of values

# **yield**



>>>

def create\_generator(N

):

L = range(N+1)

for iin L:

yield i\*i

generator = create\_generator

(10)

>>>

>>>

generator

8>

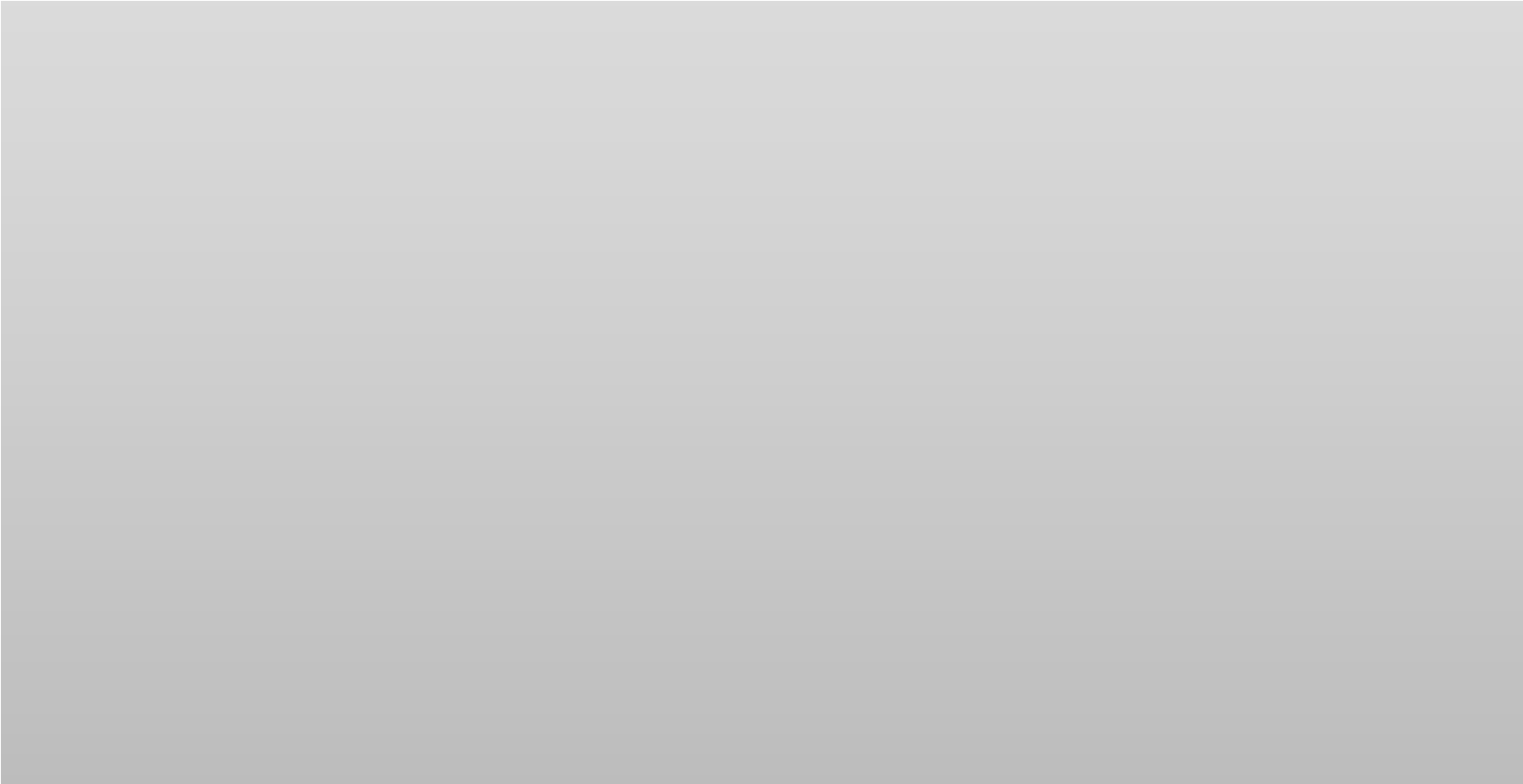
<

generator object create\_generatorat 0x013825F

>>>

for iin generator

:



print(

i

)

0

1

4

9

16

25

36

49

64

81

100

>>>

## Modules

* Modular software design approach helps develop programs that are readable, reliable and maintainable without too much effort
* A module in Python is just a file containing Python definitions and statements.
* The module name is moulded out of the file name by removing the suffix .py.

– For example, if the file name is fibonacci.py, the module name is fibonacci.

## Creating Custom Modules



# File Name: fibonacci.py

def fib(n):

""" This function generates the nth

fibonaccinumber"""

if n == 0:

return 0

elifn == 1:



def gen\_fib(n):

""" This function generates a series

of n fibonaccinumbers"""

a = 0

b = 1

s = []

if n == 0:

print('Not a valid argument')



return 1

else:

return fib(n-1) + fib(n-2)

def ifib(n):

""" This function generates the nth

fibonaccinumber """

a, b = 0, 1

for iin range(n):

a, b = b, a + b

return a



elif

n == 1:

return s.append(a)

elifn == 2:

return s.append([a, b])

else:

s.append([a,b])

for iin range(n-1):

x = a + b

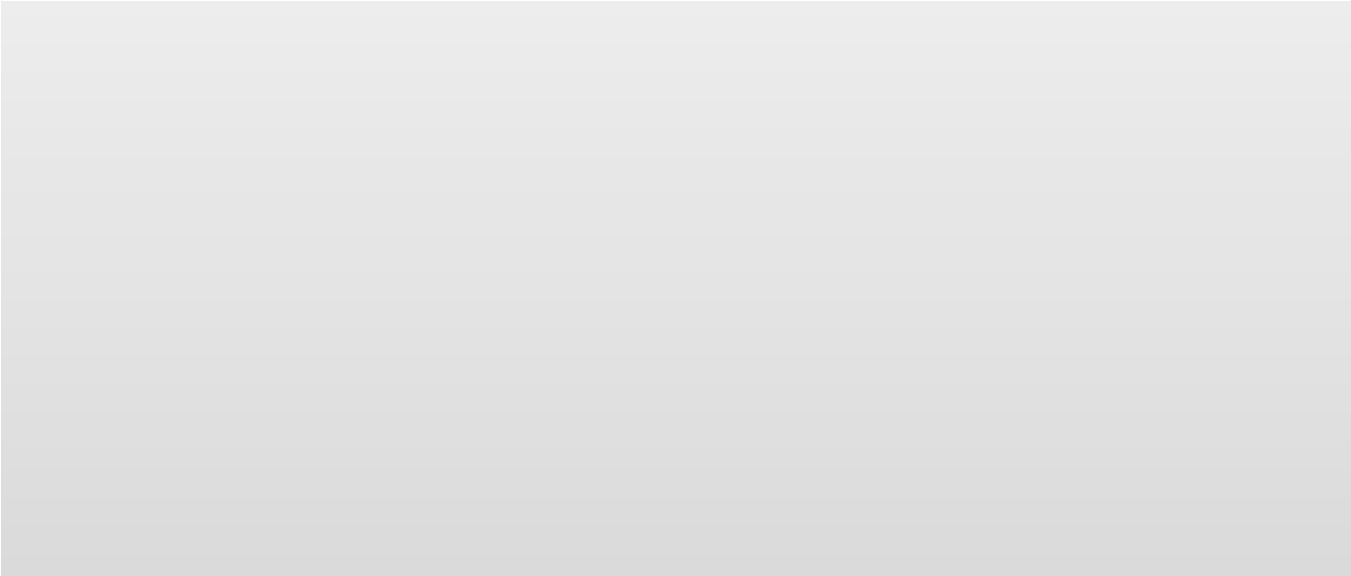
s.append(x)

a = b

b = x

return s

## Using the Module



>>>

import fibonaccias fb

(10)

>>>

fb.fib

55

>>>

fb.ifib

(10)

55

fb.gen\_fib

>>>

(10)

55]

, 1], 1, 2, 3, 5, 8, 13, 21, 34,

[[0

(10)

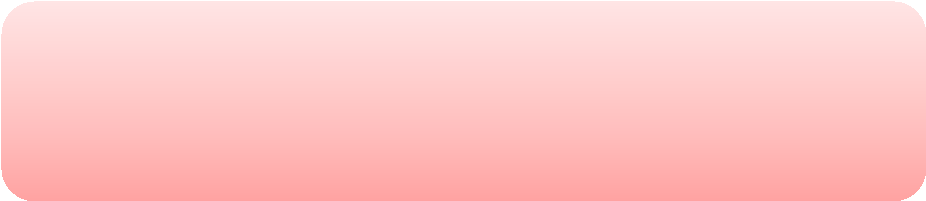
s = fb.gen\_fib

>>>

:

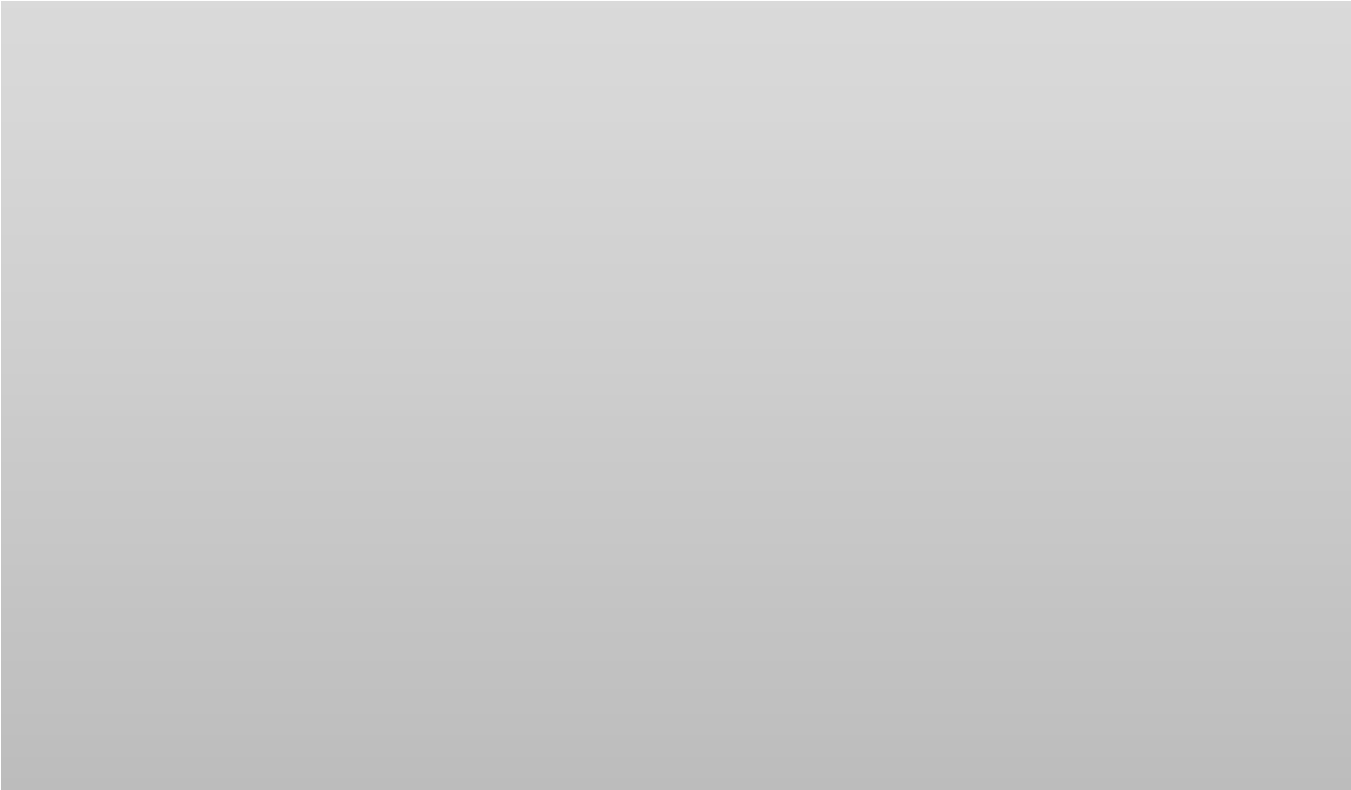
for iin s

>>>



Alias name for the

namespace



print(

i

)

[0

,

1]

1

2

3

5

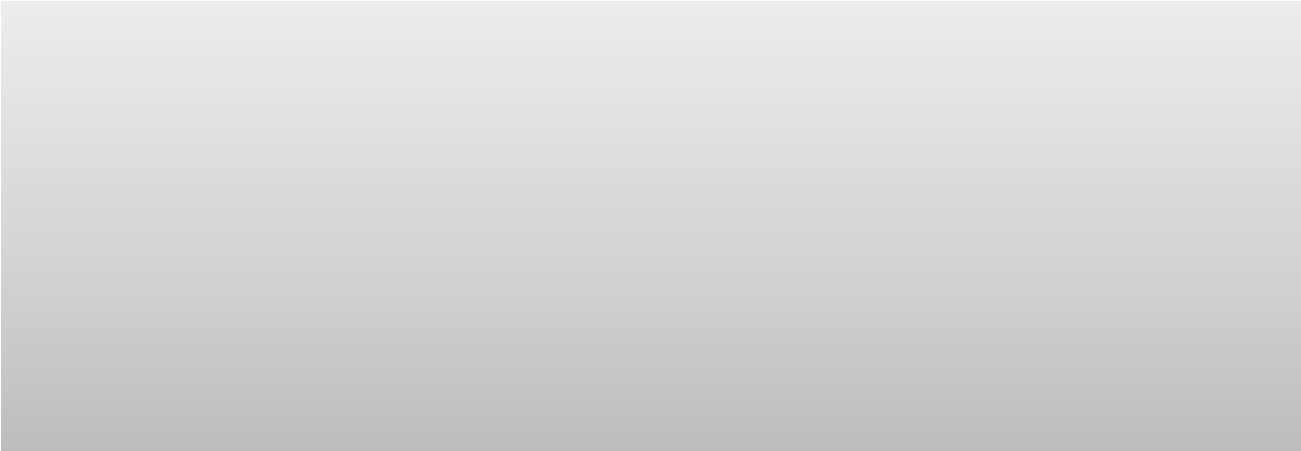
8

13

21

34

55



**# Importing specific functions**

>>>

from fibonacciimport gen\_fib

**# Importing all functions**

>>>

from fibonacciimport \*

## Module Search Path

* If you import a module, let's say "import abc", the interpreter searches for this module in the following locations and in the order given:
  + The directory of the top-level file, i.e. the file being executed
  + The directories of PYTHONPATH, if this global variable is set – Standard installation path Linux/Unix e.g. in /usr/lib/python3.1
* Location of any module can be seen as follows in 3.x:
  + In 2.x use math.\_\_file\_\_

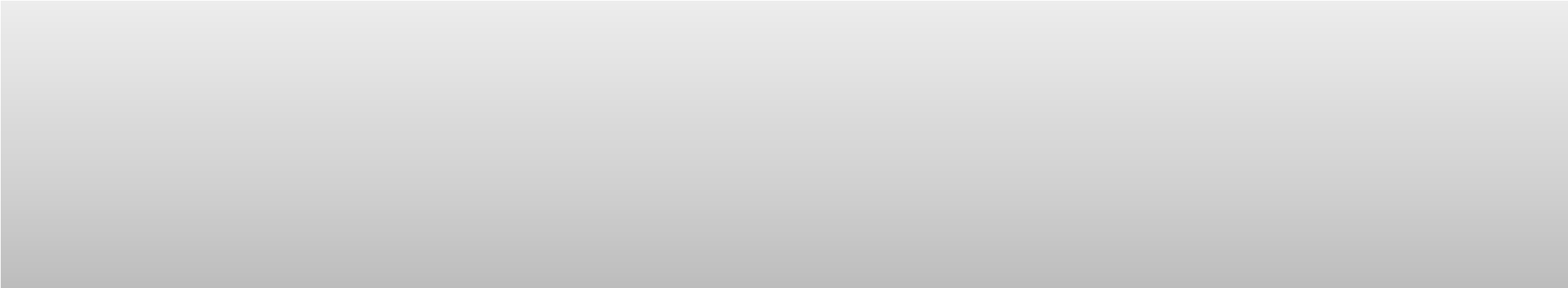
>>> os.path.dirname(os.path.realpath('\_\_file\_\_'))

'E:\\Python31‘

>>> os.path.dirname(os.path.realpath('math.\_\_file\_\_'))

'E:\\Python31'

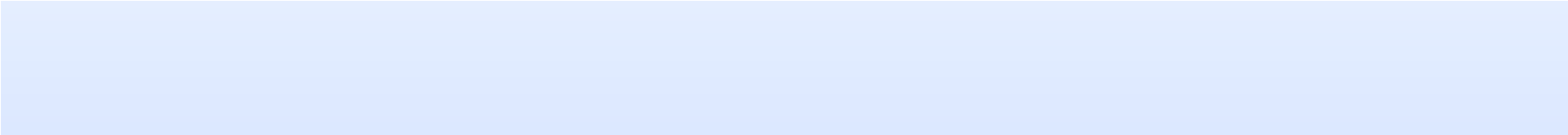
>>> os.path.dirname(os.path.realpath('random.\_\_file\_\_'))



'E:\\Python31'

## Unit Tests with **\_\_name\_\_**

• The technique of using **\_\_name\_\_** discussed previously can be effectively used to create some self-test code which you may not want to be executed when it is executed



# minmax.py



print('I am:', \_\_name\_\_)

def minmax(test, \*args):

res = args[0]

for argin args[1:]:

if test(arg, res):

res = arg

return res

def lessthan(x, y): return x < y

def grtrthan(x, y): return x > y

if \_\_name\_\_ == '\_\_main\_\_':

print(minmax(lessthan, 4, 2, 1, 5, 6, 3))

# Self-test code

print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))

## Packages

* Packages are a way of structuring Python’s module namespace by using “dotted module names”.
  + For example, the module name A.B designates a submodule named B in a package named A
* In simple terms, a package is a collection of modules arranged in a directory structure
* Every directory should contain an **\_\_init\_\_.py** file
  + The **\_\_init\_\_.py** files are required to make Python treat the directories as containing packages
  + **\_\_init\_\_.py** can just be an empty file, but it can also execute initialization code for the package or set the **\_\_all\_\_** variable

Refer: https://docs.python.org/2/tutorial/modules.html



## Package Structure



sound/ Top-level package

Initialize the sound package

\_\_init\_\_.py

formats/ Subpackagefor file format conversions

\_\_init\_\_.py

wavread.py

wavwrite.py

aiffread.py

aiffwrite.py

auread.py

auwrite.py



...

effects/ Subpackagefor sound effects

\_\_init\_\_.py

echo.py

surround.py

reverse.py

...

filters/ Subpackagefor filters

\_\_init\_\_.py

equalizer.py

vocoder.py

karaoke.py

...



**# Importing all functions**

>>>

import sound.effects.echo

## Challenge #1

* Convert a given number into its equivalent statement form
* Do your research on text to speech conversion online
* Adopt it to spell out the statement