STSCI 4060

Lecture File 2

Xiaolong Yang
(xy44@cornell.edu)

Python Functions

A Python function is a named sequence of statements that performs a desired operation. It is common to say that a function “takes” an argument(s) and “returns” a result. However, in Python a function may or may not take an argument(s) and may or may not return a result.

Some functions are built in with the Python interpreter, some are defined in special modules, and some will have to be defined by the users. They are respectively called:

- Built-in functions
- Importable functions, e.g., math functions
- User-defined functions (UDFs)

Python Built-in Functions

These function are built into the Python interpreter and are always available.

abs()
all()
any()
basestring()
bin()
bool()
bytearray()
callable()
chr()
classmethod()
cmp()
compile()
complex()
delattr()
dict()
dir()
divmod()
enumerate()
eval()
execfile()
file()
filter()
float()
format()
frozenset()
getattr()
globals()
hasattr()
hash()
help()
hex()
id()
input()
int()
isinstance()
issubclass()
iter()
len()
list()
locals()
long()
map()
max()
memoryview()
min()
next()
object()
oct()
open()
ord()
pow()
print()
property()
range()
raw_input()
reduce()
reload()
repr()
reversed()
round()
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()
tuple()
type()
unichr()
unicode()
vars()
xrange()
zip()
__import__()
apply()
buffer()
coerce()
intern()

Built-in Function Examples: bool()

bool([x]): Convert a value to a Boolean, using the standard truth testing procedure. If x is zero, false, missing or omitted, it returns False; otherwise it returns True.

```
>>> bool(0)
False
>>> bool()
False
>>> bool(False)
False
>>> bool(10 )
True
>>> bool('a')
True
>>> bool(1<2)
True
>>> bool(1>2)
False
```

```
>>> bool(True)
True
>>> bool('Cornell University')
True
```

How about the following?

```
>>> bool(false)

>>> bool(true)

>>> bool('false')

>>> bool('true')

>>> bool('')
```

Built-in Function Examples: cmp()

cmp(x, y): Compare the two objects *x* and *y* and return an integer according to the outcome. The return value is negative (-1) if x < y, zero if x == y and positive (+1) if x > y.

```
>>> cmp(1,2)
-1
>>> cmp(2,1)
1
>>> cmp(10,10)
0
>>> cmp('Cornell',
'University')
-1
>>> cmp(True, True)
0
```

How about the following?

```
>>> cmp(True, False)

>>> cmp(False, True)

>>> cmp(False, False)

>>> cmp(true, true)
```

Built-in Function Examples: dict()

dict([iterable,…]): Returns a new dictionary.

```
>>> d=dict()
>>> type(d)
<type 'dict'>
>>> d
{}
>>> d2=dict((('car', 100), ('bus', 20), ('truck', 30), ('bike', 500)))
>>> d2
{'car': 100, 'bike': 500, 'truck': 30, 'bus': 20}
```

Built-in Function Examples: float()

float([x]): Convert a string or a number to floating point number. If the argument is a string, it must contain a possibly signed decimal or floating point number, possibly embedded in whitespaces.

```
>>> float(1)
1.0
>>> float('120')
120.0
>>> float("-32")
-32.0
>>> float("   -32  ")
-32.0
>>> float("123.45  ")
123.45
```

Built-in Function Examples: int()

int(x): Convert a number or string *x* to an integer, or return 0 if no arguments are given. If *x* is a number, it can be a plain integer, a long integer, or a floating point number.

```
>>> int(0)
0
>>> int()
0
>>> int(12.3)
12
>>> int('1234')
1234
```

```
>>> a=123456L
>>> type(a)
<type 'long'>
>>> type(a)
<type 'long'>
>>> b=int(a)
>>> type(b)
<type 'int'>
```

Built-in Function Examples: eval()

eval(expression[, globals[,locals]]): The arguments are a string and optional globals and locals. The return value is the result of the evaluated expression.

```
>>> eval('1234')
1234
>>> eval('2+3')
5
>>> x=10
>>> y=20
>>> eval('x+y+100')
130
```

Built-in Function Examples: raw_input()

raw_input([prompt]): If the *prompt* argument is present, it is written to standard output without a trailing new line. The function then reads a line from the keyboard input, converts it to a string, and returns the string. This function is to input a string from keyboard.

```
>>> raw_input('input a string ')
input a string MPS in Applied Statistics
'MPS in Applied Statistics'
>>> raw_input('input a string ')
input a string 123456
'123456'
>>> raw_input('input a string ')
input a string kjfkl9098
'kjfkl9098'
```

Built-in Function Examples: input()

input([prompt]): Writes the prompt if provided, then reads a line of input from the keyboard. It then applies the Python eval(*string*) function to evaluate the input. Typically, we expect to decode *a string of digits* as an integer or floating point number. Actually, it will evaluate any legal Python expression! This function is to input a number from keyboard.

```
>>> input('input a number ')
input a number 123.45
123.45
>>> input('input a number ')
input a number 100
100
>>> input('input a math operation ')
input a math operation 12+13
25
```

Built-in Function Examples: open()

open(name[, mode[, buffering]]): Opens a file, returning a file object. When opening a file, it’s preferable to use open() instead of invoking the file constructor directly. *name:* file name to be opened; *mode:* 'r' (reading), 'w' (writing, which truncates the file if it already exists), or 'a' (appending). If *mode* is omitted, it defaults to 'r'. *buffering:* 0, unbuffered; 1, line buffered; any other positive value, a buffer of that size; a negative value or omitted, use the system default.

```
>>> colors = ['red\n', 'yellow\n', 'blue\n', 'green\n']
>>> f = open('K:\STSCI4060\colors1.txt', 'w')
>>> f.writelines(colors)  #write to the file
>>> f.close()   #the file shows the contents after this.
>>> f = open('K:\STSCI4060\colors1.txt')
>>> f.readlines() #read the lines at once
['red\n', 'yellow\n', 'blue\n', 'green\n']
>>> f.close() #it must be closed again
>>> f = open('K:\STSCI4060\colors1.txt')
>>> f.readline() #read one line at a time; 1st line
'red\n'
>>> f.readline() #read the 2nd line
'yellow\n'
>>> f.readline() #read the 3rd line
```

```
'blue\n'
>>> f.readline() # read 4th line
'green\n‘
>>> f.readline() #end of file
''
>>> f.close()
>>> f = open('K:\STSCI4060\colors1.txt')
>>> for line in f:
            print line
red
yellow
blue
green
>>> f.close()
```

Built-in Function Examples: range()

range([start=0,] stop[, step=1]): This is a versatile function that creates lists containing arithmetic progressions. It is most often used in for loops. The arguments must be plain integers. If the *start* argument is omitted, it defaults to 0. If the *step* argument is omitted, it defaults to 1; but it cannot be zero. You must provide the stop value. The full form returns a list of plain integers.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(0, 40, 5)
[0, 5, 10, 15, 20, 25, 30, 35]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
>>> range(1, 0)
[]
```

Built-in Function Examples: round()

round(number[, ndigits]): Return the floating point value *number* rounded to *ndigits* digits after the decimal point. If *ndigits* is omitted, it defaults to zero.

```
>>> round(10.2345,2)
10.23
>>> round(10.2345,3)
10.235
>>>round(3.14159265*5**2, 4)
78.5398
>>>round(3.14159265*5**2)
79.0
```

Built-in Function Examples: zip()

zip([iterable, ...]): Returns a list of tuples, where the ith tuple contains the ith element from each of the argument sequences or iterables.

```
>>> s=['dog', 'cat', 'bird', 'snake']
>>> n=[10, 15, 8, 50]
>>> zipped = zip(s,n)
>>> zipped
[('dog', 10), ('cat', 15), ('bird', 8), ('snake', 50)]
```

Can you do these?

```
zip(s, n, [1,2,3,4,5])
zip(s)
```

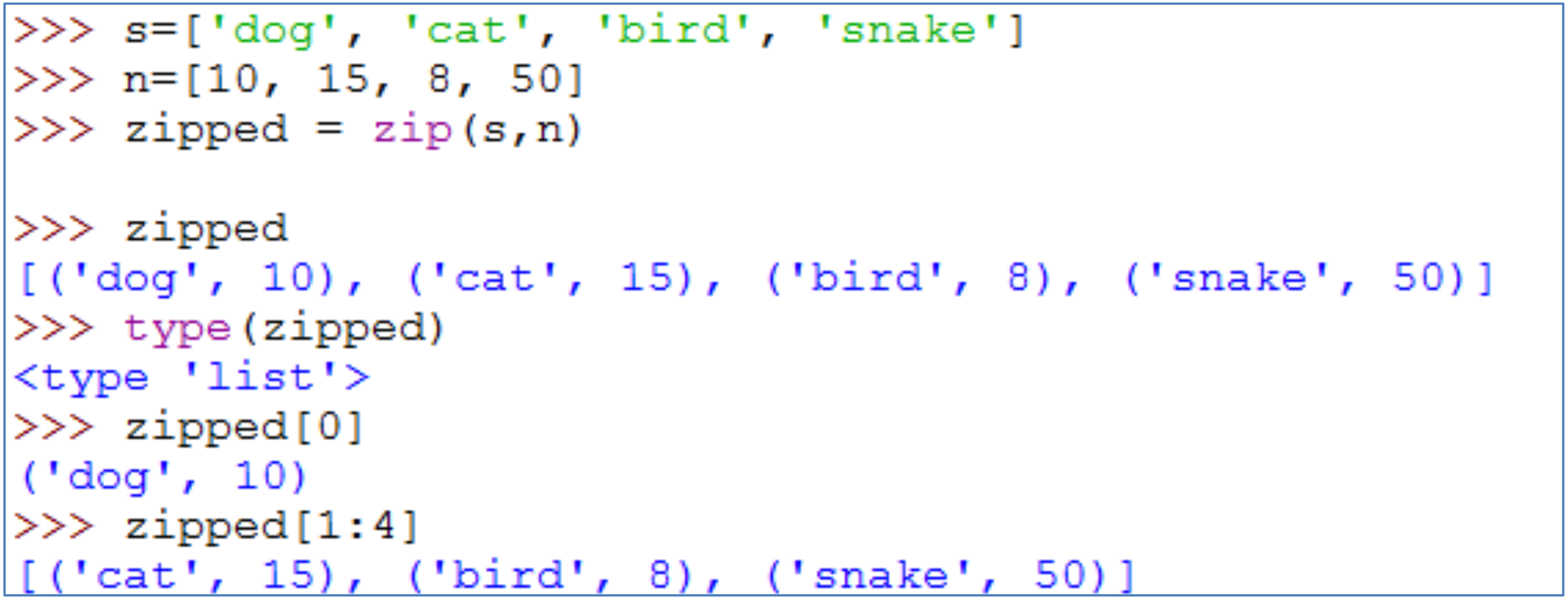
You can unzip a list of tuples with the *** operator**

```
>>> unzipped = zip(*zipped)
>>> unzipped
[('dog', 'cat', 'bird', 'snake'), (10, 15, 8, 50)]
```

Use zip() and dict() Together to Produce a Dictionary from Two Sequences

```
>>> k=['car', 'bus', 'truck', 'bike']
>>> v=[100, 20, 30, 500]
>>> myDict =dict(zip(k,v))
>>> myDict
{'car': 100, 'bike': 500, 'truck': 30, 'bus': 20}
```

Slice the Zipped Result



```
>>> s=['dog', 'cat', 'bird', 'snake']
>>> n=[10, 15, 8, 50]
>>> zipped = zip(s,n)

>>> zipped
[('dog', 10), ('cat', 15), ('bird', 8), ('snake', 50)]
>>> type(zipped)
<type 'list'>
>>> zipped[0]
('dog', 10)
>>> zipped[1:4]
[('cat', 15), ('bird', 8), ('snake', 50)]
```

More on Creating Python Dictionaries

```
>>> a = {'one': 1, 'two': 2, 'three': 3}
>>> b = dict(one=1, two=2, three=3)
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
```

Importable Python Functions: Math Functions

The Python **math** module contains many useful non-built-in math functions. They are defined by C standard. To use, you need to import the math module. Here are some examples. Refer to the Python docs for details.

- math.factorial(*x*): Returns *x* factorial.

```
>>> math. factoiral(5)
120
```

- math.fsum(iterable): Returns an accurate sum of values in the iterable

```
>>> math.fsum([.1,.1,.1,.1,.1,.1,.1,.1])
0.8
>>> sum([.1,.1,.1,.1,.1,.1,.1,.1])
0.7999999999999999
```

- math.log(x[, base]):

```
>>> math.log(2) #natural logarithm of 2
0.6931471805599453
>>> math.log(2, 10) #base-10 log. of 2
0.30102999566398114
```

- math.log10(x): Returns base-10 log. of x

```
>>> math.log10(2)  # often more
accurate than math.log(2, 10)
0.3010299956639812
```

- math.degrees(*x*): Convert radians x to degree

```
>>> math. degrees(math.pi)
180.0
```

- math.radians(*x*): Convert degrees x to radians

```
>>> math. radians(180.0)
3.141592653589793
```

- math.gamma(*x*): Returns the Gamma function at x

```
>>> math.gamma(5)
24.0
```

Importable Python Functions: Random Number Functions

The Python **random** module contains many useful non-built-in functions to generate random numbers. To use, you need to import the random module. Here are some examples. Refer to the Python docs for details.

- **random.randrange(*start*, *stop*[, *step*])**: Return a randomly selected element from range(start, stop, step). Note stop is excluded from the range.

```
>>> random.randrange(20,80)
47
>>> random.randrange(20,800,5)
245
```

- **random.randint(*a*, *b*)**: Return a random integer *N* such that a <= N <= b.

```
>>> random.randint(1,10)
2
>>> random.randint(1,10)
10
```

- **random.choice(*seq*)**: Return a random element from the non-empty sequence *seq*.

```
>>> a=[2,4,6,78,3,5,67,112,45]
>>> b=['he', 'she', 'we', 'they', 'you']
>>> random.choice(a)
67
>>> random.choice(b)
'she'
```

- **random.sample(*population*, *k*)**: Return a *k* length list of unique elements chosen from the population sequence. Used for random sampling without replacement.

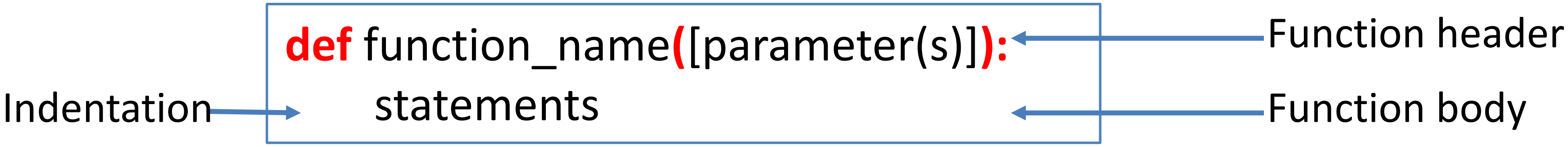
```
>>> random.sample(a,3)
[2, 3, 67]
>>> random.sample(b,2)
['she', 'you']
```

- **random.random()**: Return the next random floating point number in the range [0.0, 1.0).

```
>>> random.random()
0.940211236763933
>>> random.random()
0.578941897582904‌9
```

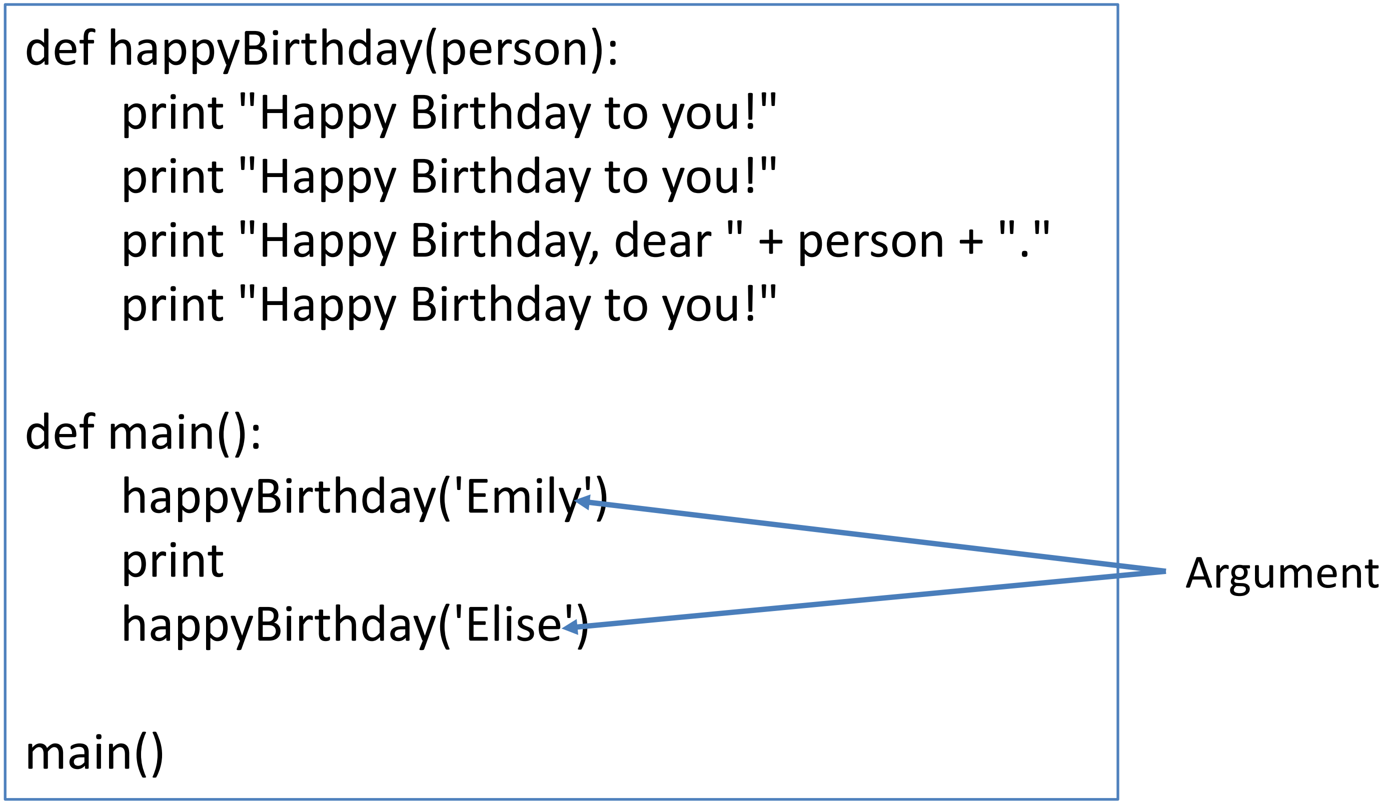
Python User-Defined Functions (UDFs)

A user can define a function to achieve his/her own computation needs. The general syntax is



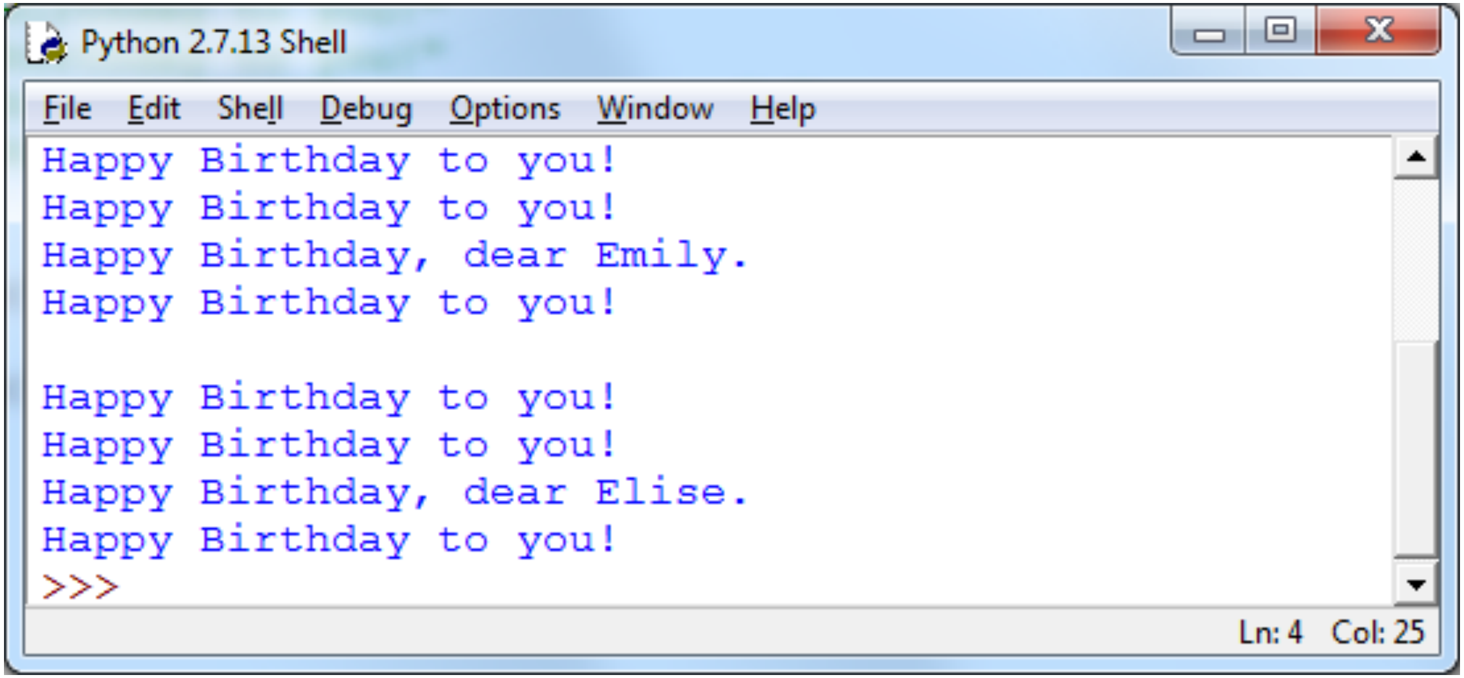
```
def function_name([parameter(s)]):
        statements
```

Example: the Happy Birthday Song



```
def happyBirthday(person):
        print "Happy Birthday to you!"
        print "Happy Birthday to you!"
        print "Happy Birthday, dear " + person + "."
        print "Happy Birthday to you!"

def main():
        happyBirthday('Emily')
        print
        happyBirthday('Elise')

main()
```

The Output of the happyBirthday() Function



```
Happy Birthday to you!
Happy Birthday to you!
Happy Birthday, dear Emily.
Happy Birthday to you!

Happy Birthday to you!
Happy Birthday to you!
Happy Birthday, dear Elise.
Happy Birthday to you!
>>>
```

Python User-Defined Functions (UDFs) (cont’d)

A few functions that do some simple math:

```
# Addition
>>> def add(x,y):
        return(float(x)+float(y))
>>> add(1.1, 3.3)
4.4

# Substraction
>>> def sub(x,y):
        return(float(x)-float(y))
>>> sub(4, 2)
2.0

# Multiplication
>>>
def mult(x, y):
        return(float(x) * float(y))
>>> mult(12,12)
144.0
```

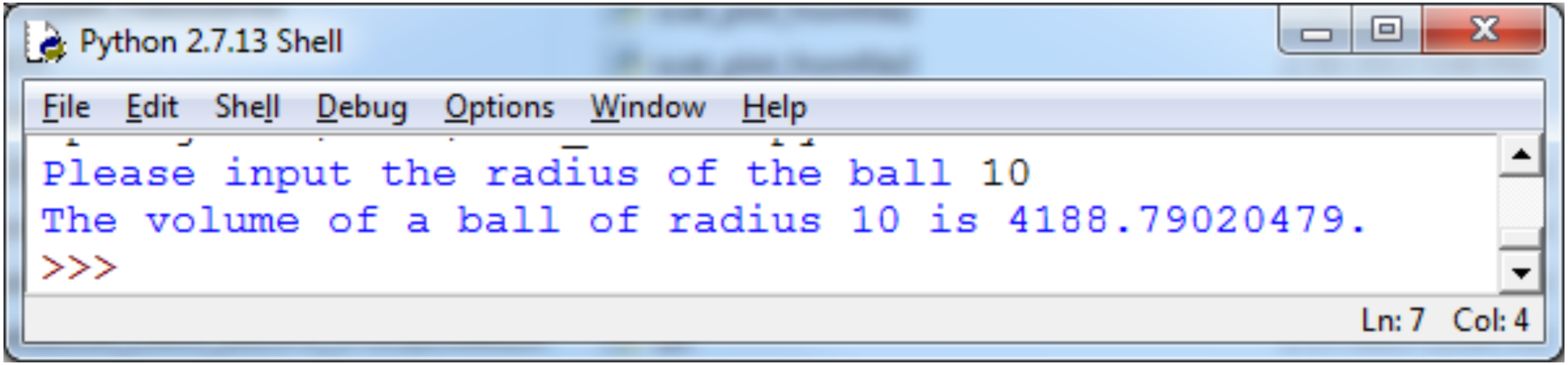
```
# Factorial
def myFact(n):
        if(n==0):
                return 1
        else:
                return n*myFact(n-1)
>>> myFact(1)
1
>>> myFact(2)
2
>>> myFact(3)
6
>>> myFact(5)
120
```

Python User-Defined Functions (UDFs) (cont'd)

Another example: a function that calculates the volume of a sphere

```
#The following function returns the volume of a ball of known radius.
def ballVolume(radius):
    import math
    volume = (4*math.pi*radius**3)/3
    return volume

def main():
    r = input('Please input the radius of the ball ')
    v= ballVolume(r)
    print 'The volume of a ball of radius %s is %s.' % (r, v)

main()
```

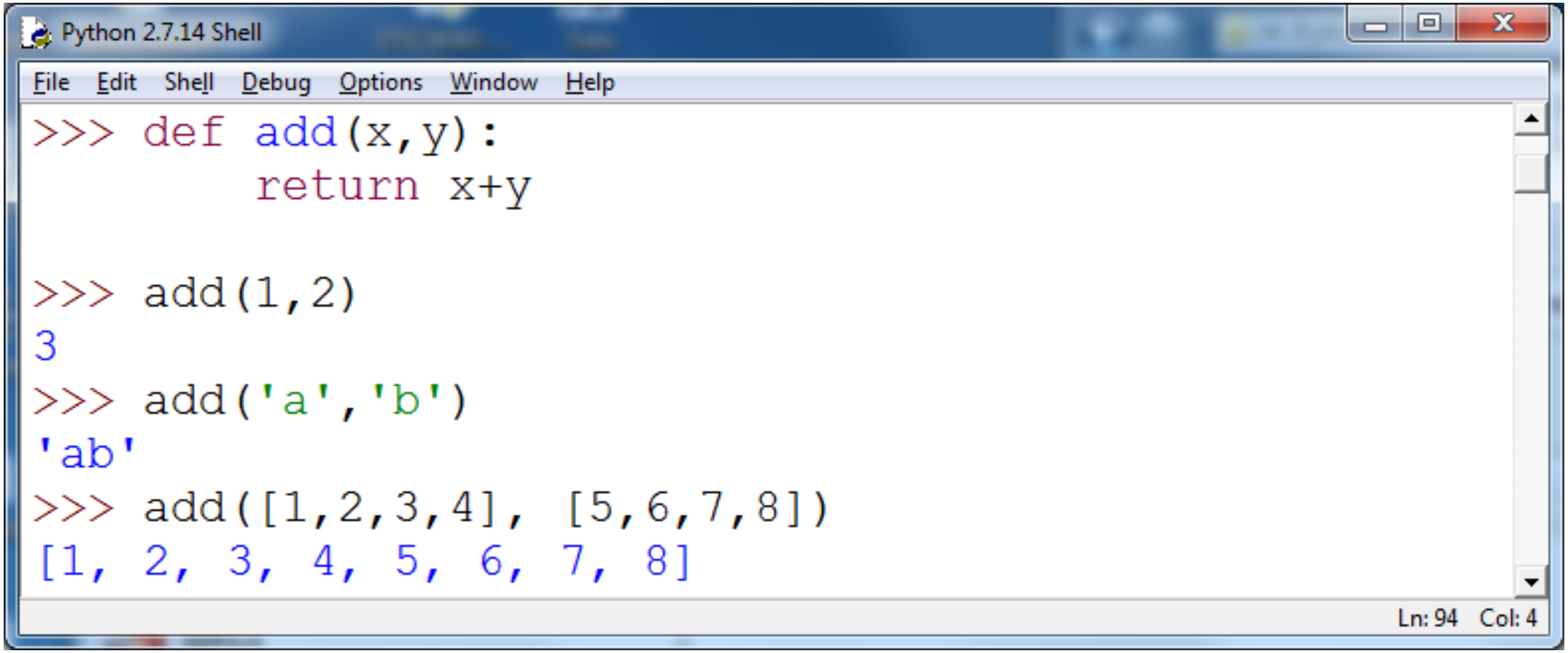
The Output of the main() Functions Calling the ballVolume() function



```
Please input the radius of the ball 10
The volume of a ball of radius 10 is 4188.79020479.
>>>
```

Python Function Polymorphism

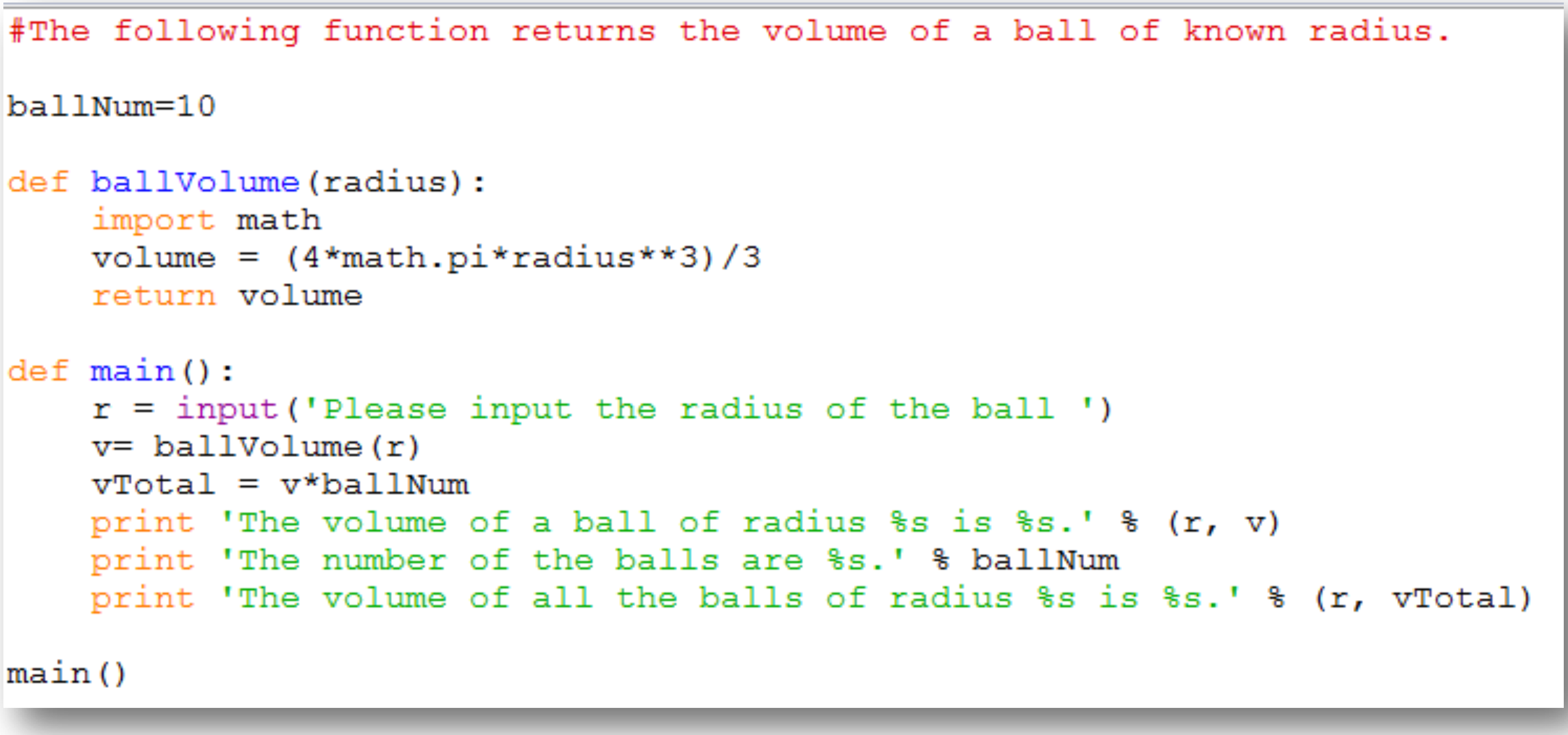
Python polymorphism is the ability to leverage the same name for different underlying forms such as data types, which permits functions to use entities of different types at different times.



```
>>> def add(x,y):
	return x+y

>>> add(1,2)
3
>>> add('a','b')
'ab'
>>> add([1,2,3,4], [5,6,7,8])
[1, 2, 3, 4, 5, 6, 7, 8]
```

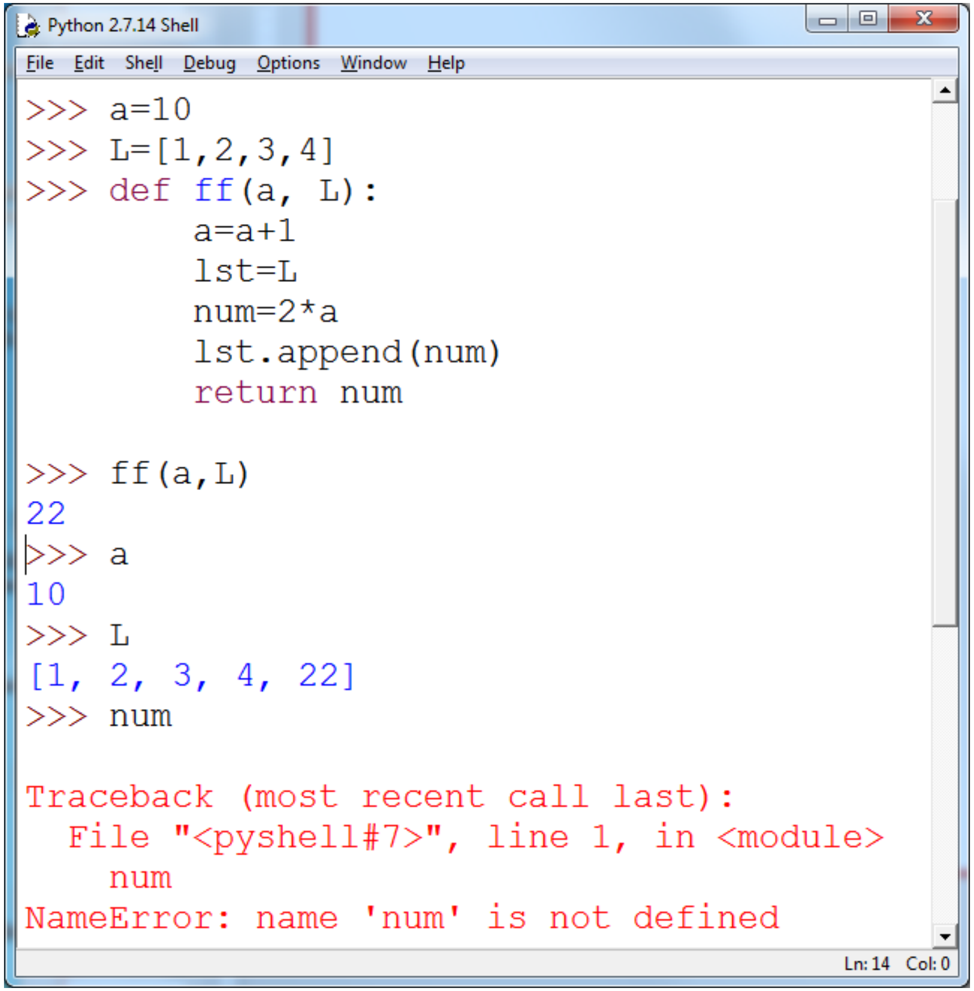
Python Function Variable Scopes

- Global variables: accessible inside and outside of functions.
- Local variables: only accessible inside the function.

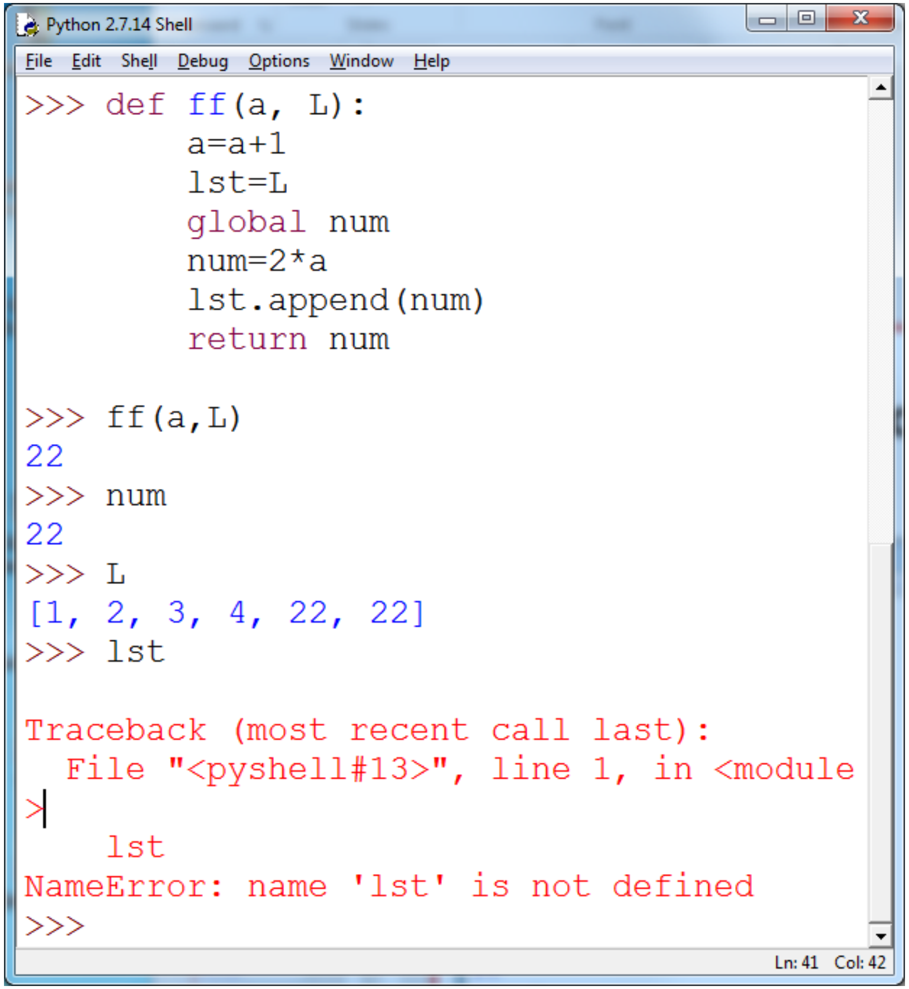


```
#The following function returns the volume of a ball of known radius.

ballNum=10

def ballVolume(radius):
    import math
    volume = (4*math.pi*radius**3)/3
    return volume

def main():
    r = input('Please input the radius of the ball ')
    v= ballVolume(r)
    vTotal = v*ballNum
    print 'The volume of a ball of radius %s is %s.' % (r, v)
    print 'The number of the balls are %s.' % ballNum
    print 'The volume of all the balls of radius %s is %s.' % (r, vTotal)

main()
```

Python Function Variable Scope Examples



```
>>> a=10
>>> L=[1,2,3,4]
>>> def ff(a, L):
        a=a+1
        lst=L
        num=2*a
        lst.append(num)
        return num

>>> ff(a,L)
22
>>> a
10
>>> L
[1, 2, 3, 4, 22]
>>> num

Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    num
NameError: name 'num' is not defined
```

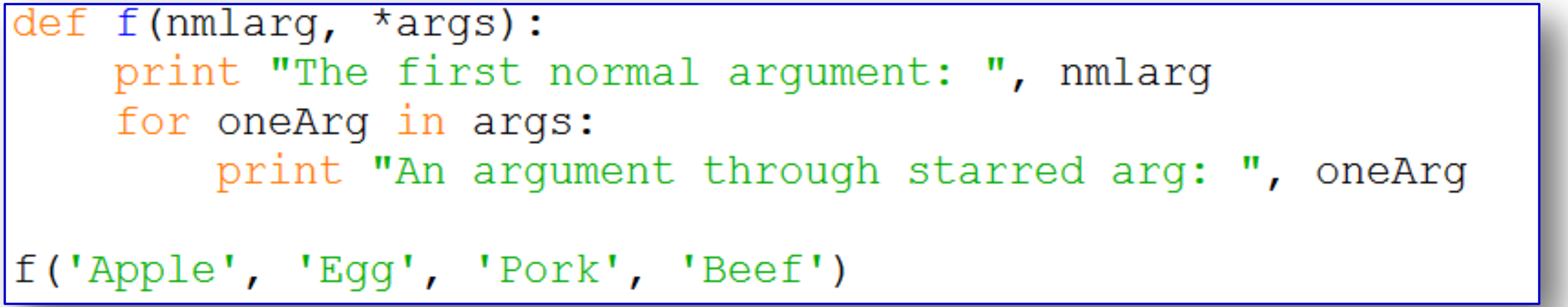


```
>>> def ff(a, L):
        a=a+1
        lst=L
        global num
        num=2*a
        lst.append(num)
        return num

>>> ff(a,L)
22
>>> num
22
>>> L
[1, 2, 3, 4, 22, 22]
>>> lst

Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module
>
    lst
NameError: name 'lst' is not defined
>>>
```

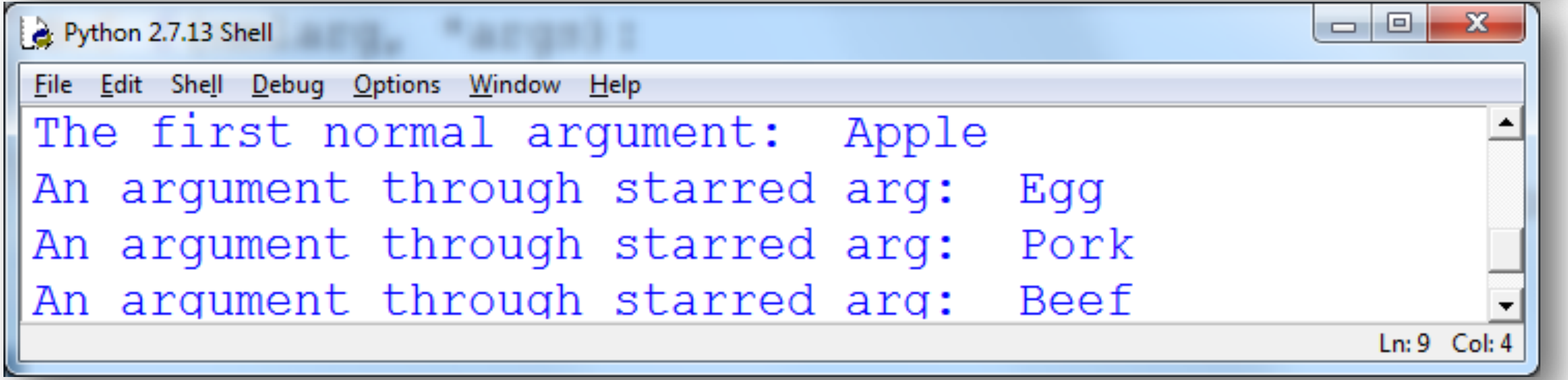
Python Starred-Argument Syntax for Arbitrary Arguments

Python functions can be designed to take any arbitrary number of arguments using *args or **kwargs.

- Usage of *args: for passing a variable length of argument list.



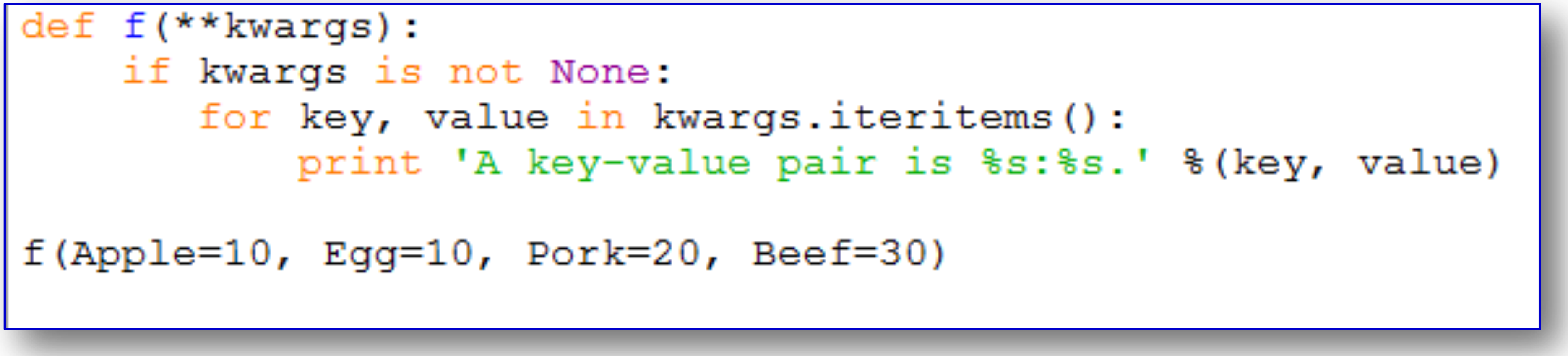
```
def f(nmlarg, *args):
    print "The first normal argument: ", nmlarg
    for oneArg in args:
        print "An argument through starred arg: ", oneArg

f('Apple', 'Egg', 'Pork', 'Beef')
```



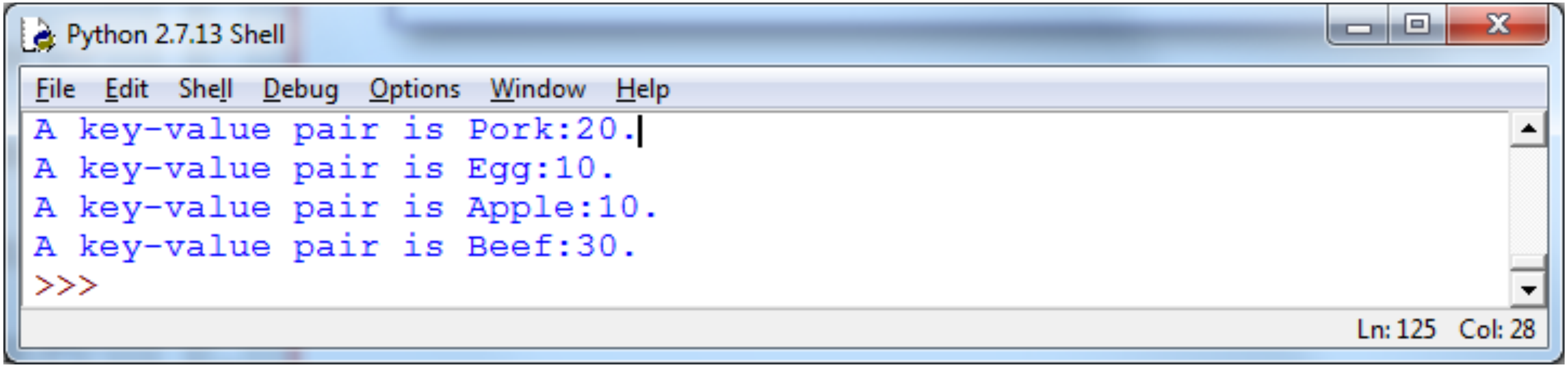
```
The first normal argument:  Apple
An argument through starred arg:  Egg
An argument through starred arg:  Pork
An argument through starred arg:  Beef
```

Python Starred-Argument Syntax for Arbitrary Arguments

- Usage of **kwargs: for passing a variable length of keyworded arguments.



```
def f(**kwargs):
    if kwargs is not None:
        for key, value in kwargs.iteritems():
            print 'A key-value pair is %s:%s.' %(key, value)

f(Apple=10, Egg=10, Pork=20, Beef=30)
```



```
A key-value pair is Pork:20.
A key-value pair is Egg:10.
A key-value pair is Apple:10.
A key-value pair is Beef:30.
>>>
```

Python Files

A file is a Python object that gives us access to a file on the storage (e.g., disk) system. A file object can be created for reading ("r" mode), for writing ("w“ mode), or for appending ("a" mode) to a file. Opening a file for writing erases an existing file with that path/name. Opening a file for append does not. The file() function can be used to create a file object.

```
outFile = file('K:\STSCI4060\STSCI 4060_spring_2017\pfile.txt', 'w')
outFile.write('Line 1 written')
outFile.close()
```

```
appendFile = file('K:\STSCI4060\STSCI 4060_spring_2017\pfile.txt', 'a')
appendFile.write('\nLine 2 appended\n')
appendFile.write('Line 3 appended\n')
appendFile.close()
```

```
infile = file('K:\STSCI4060\STSCI 4060_spring_2017\pfile.txt', 'r')
contents = infile.read()
print contents
```

Note: ‘r’ is optional if mode is reading

Python **Class** and Objects

Some terms:

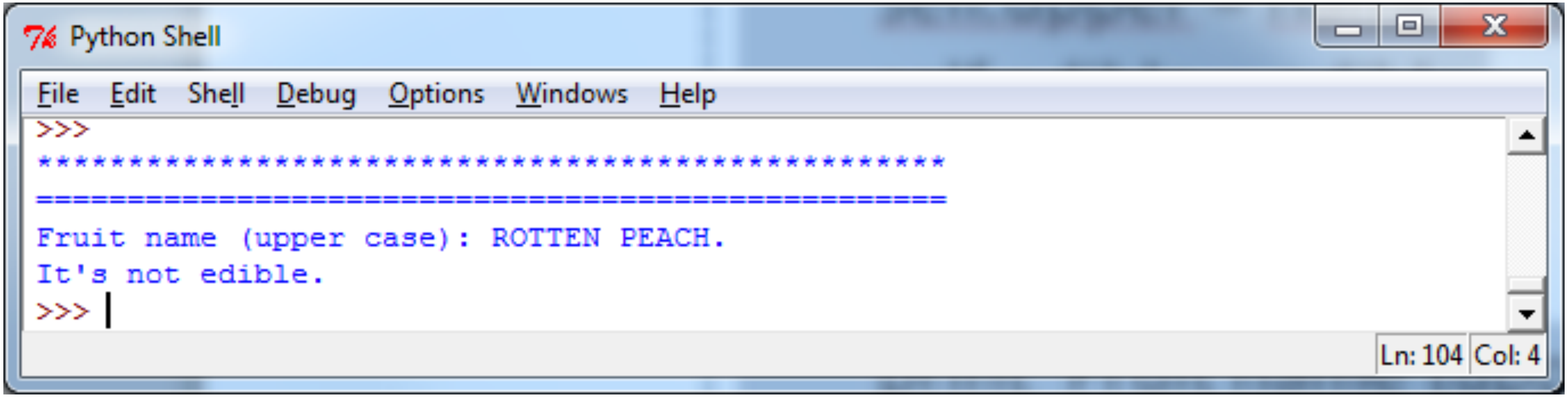
- A Python class is often a user-defined data type, e.g., **MyClass**.
- An instance of the class is called an object, e.g., **f**. However, in Python any value is an object.
- Creating a new instance of a class is called **instantiation**.
- To instantiate an object you need to call a function with the same name as the class you defined, which is called a **constructor**, for example, f = **MyClass()**, where f is an object or an instance of MyClass, and MyClass() is a constructor.

Python Class and Objects

```
class Fruit:
    def __init__(self, name, edible): # class constructor
        self.upper = name.upper()       # instance variable
        self.edible = edible            # instance variable
    def show(self):
        print '=' * 50
        print 'Fruit name (upper case): %s.' % self.upper
        if self.edible:
            print "It's edible."
        else:
            print "It's not edible."

def test():
    print '*' * 50
    obj = Fruit('Rotten Peach', 0)
    obj.show()

test()
```

Python Class and Objects

The result:



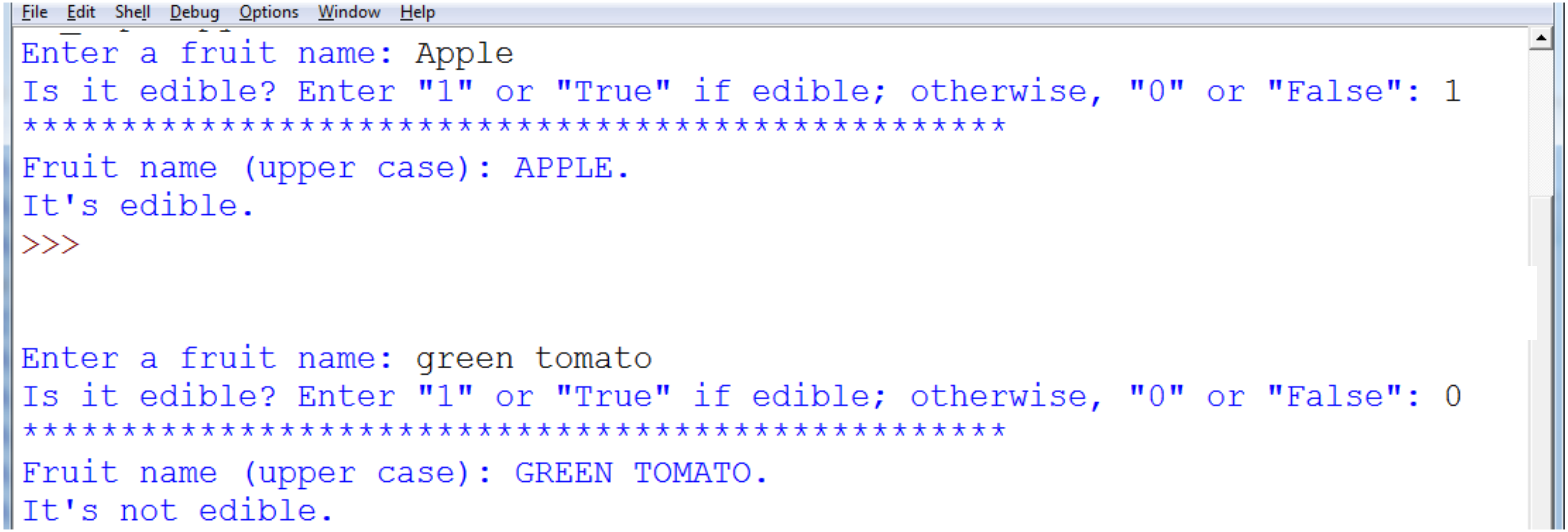
```
>>>
**************************************************
==================================================
Fruit name (upper case): ROTTEN PEACH.
It's not edible.
>>>
```

Python Class and Objects

```
class Fruit:
    def __init__(self, name, edible):
        self.upper = name.upper()
        self.edible = edible
    def show(self):
        print '=' * 50
        print 'Fruit name (upper case): %s.' % self.upper
        if self.edible:
            print "It's edible."
        else:
            print "It's not edible."

a=raw_input('Enter a fruit name: ')
b=input('Is it edible? Enter "1" or "True" if edible; otherwise, "0" or "False": ')
obj = Fruit(a, b)
obj.show()
```

Python Class and Objects

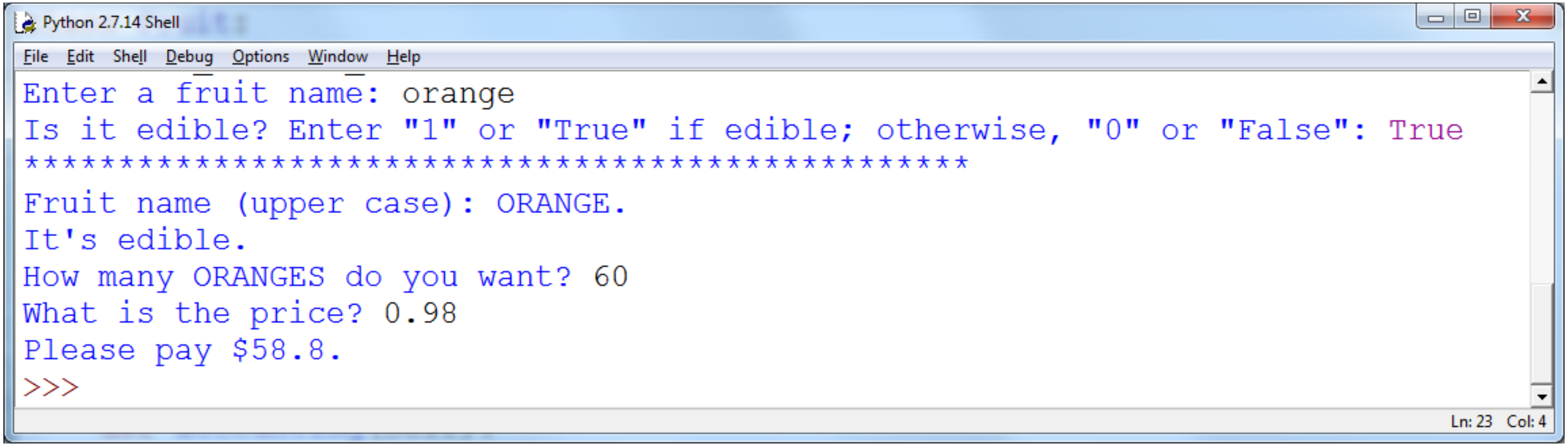
The result:



```
Enter a fruit name: Apple
Is it edible? Enter "1" or "True" if edible; otherwise, "0" or "False": 1
**************************************************
Fruit name (upper case): APPLE.
It's edible.
>>>


Enter a fruit name: green tomato
Is it edible? Enter "1" or "True" if edible; otherwise, "0" or "False": 0
**************************************************
Fruit name (upper case): GREEN TOMATO.
It's not edible.
```

Python Class and Objects

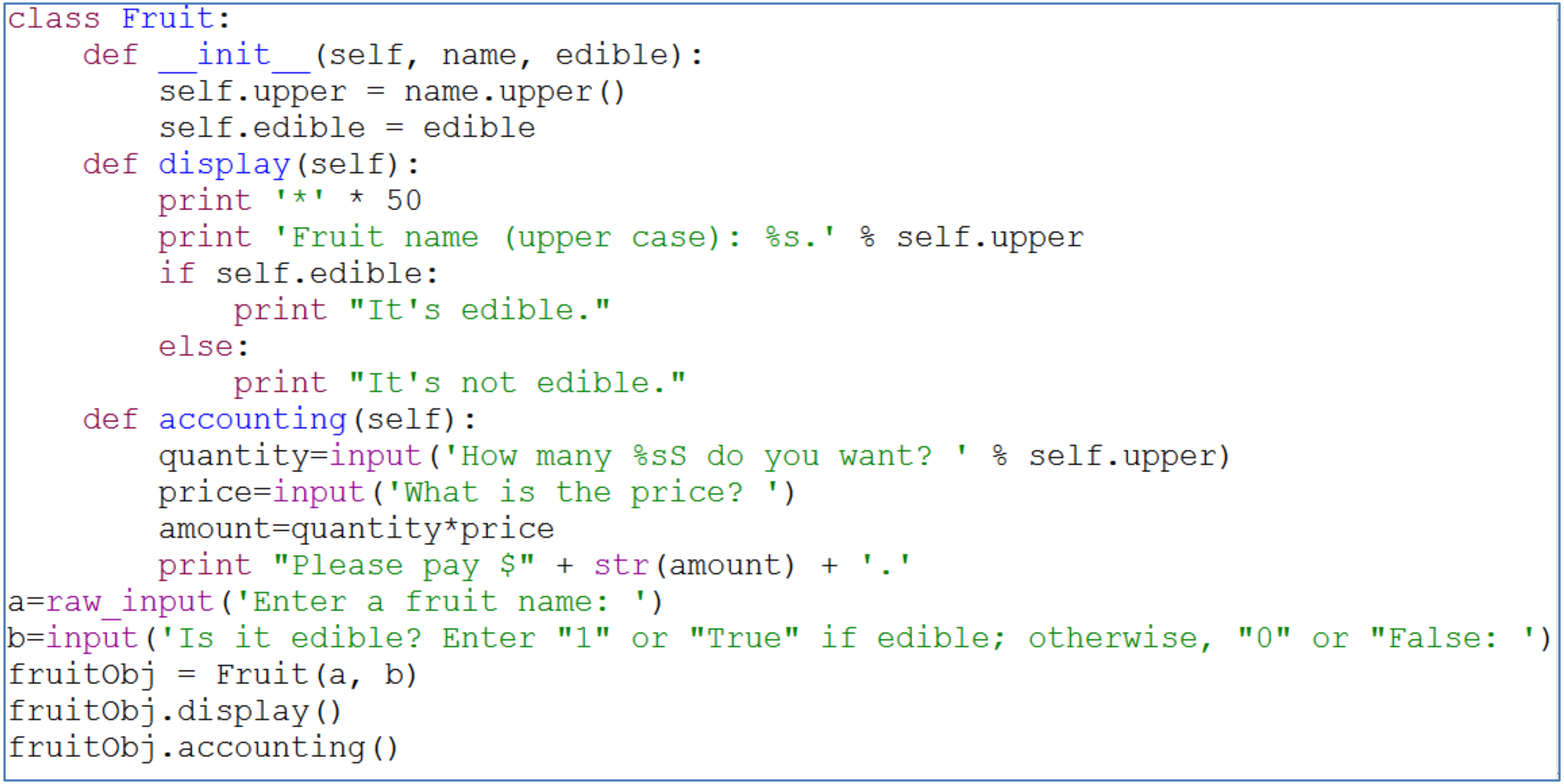
Add a method to the Fruit class so that you can produce the following output.



```
Enter a fruit name: orange
Is it edible? Enter "1" or "True" if edible; otherwise, "0" or "False": True
**************************************************
Fruit name (upper case): ORANGE.
It's edible.
How many ORANGES do you want? 60
What is the price? 0.98
Please pay $58.8.
>>>
```

Python Class and Objects

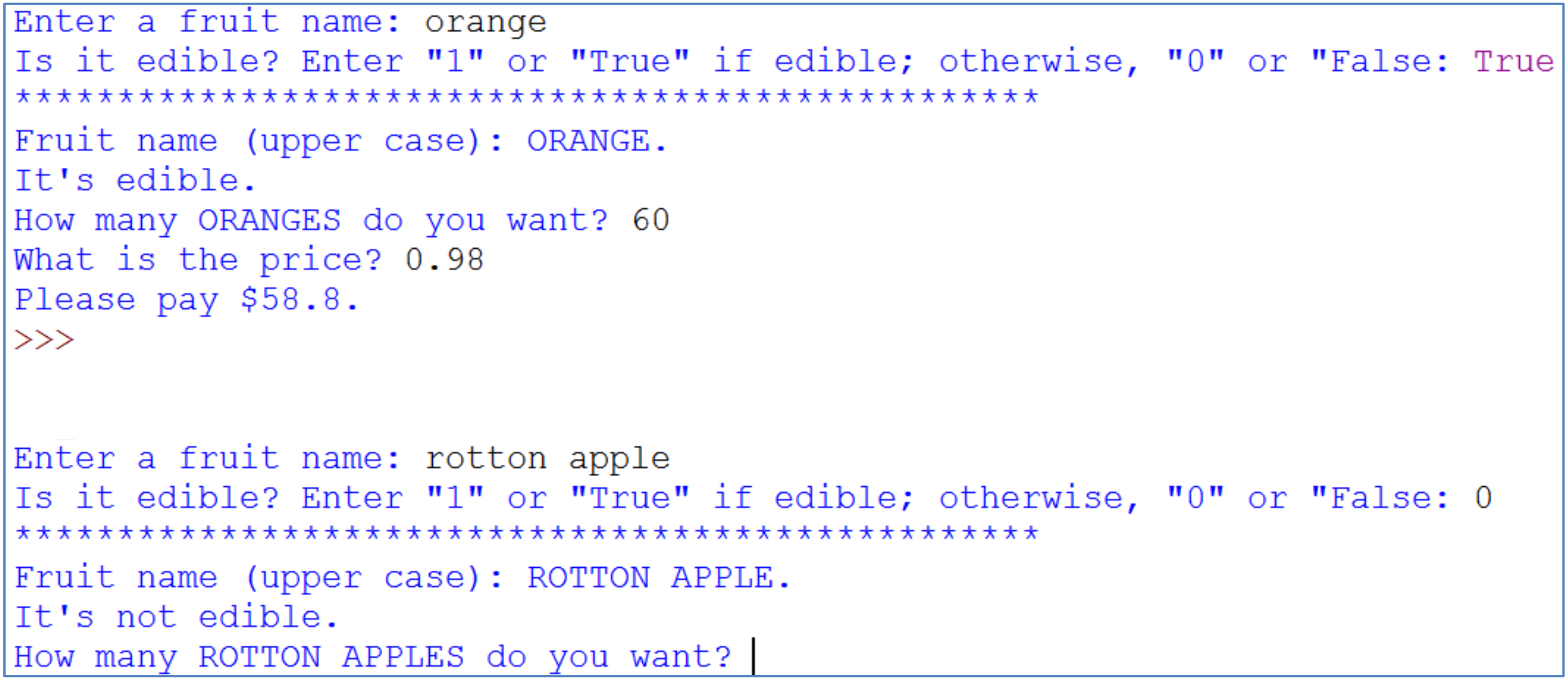
The code



```
class Fruit:
    def __init__(self, name, edible):
        self.upper = name.upper()
        self.edible = edible
    def display(self):
        print '*' * 50
        print 'Fruit name (upper case): %s.' % self.upper
        if self.edible:
            print "It's edible."
        else:
            print "It's not edible."
    def accounting(self):
        quantity=input('How many %sS do you want? ' % self.upper)
        price=input('What is the price? ')
        amount=quantity*price
        print "Please pay $" + str(amount) + '.'
a=raw_input('Enter a fruit name: ')
b=input('Is it edible? Enter "1" or "True" if edible; otherwise, "0" or "False: ')
fruitObj = Fruit(a, b)
fruitObj.display()
fruitObj.accounting()
```

Python Class and Objects

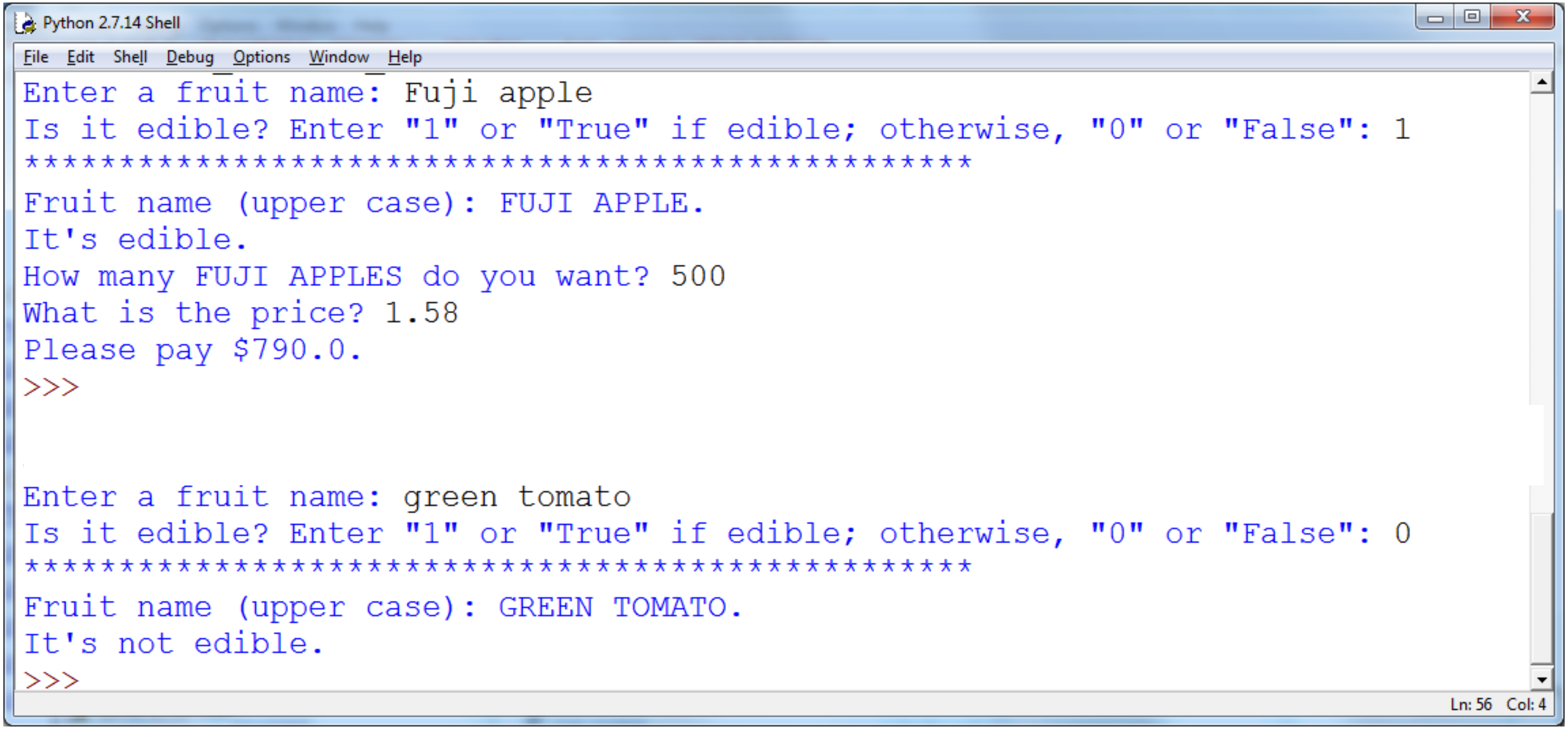
Some output showing that the code does not respond well to all the situations.



```
Enter a fruit name: orange
Is it edible? Enter "1" or "True" if edible; otherwise, "0" or "False: True
**************************************************
Fruit name (upper case): ORANGE.
It's edible.
How many ORANGES do you want? 60
What is the price? 0.98
Please pay $58.8.
>>>


Enter a fruit name: rotton apple
Is it edible? Enter "1" or "True" if edible; otherwise, "0" or "False: 0
**************************************************
Fruit name (upper case): ROTTON APPLE.
It's not edible.
How many ROTTON APPLES do you want? 
```

Python Class and Objects

Further modify your Python code so that you only do accounting for the fruit that is edible.



```
Enter a fruit name: Fuji apple
Is it edible? Enter "1" or "True" if edible; otherwise, "0" or "False": 1
**************************************************
Fruit name (upper case): FUJI APPLE.
It's edible.
How many FUJI APPLES do you want? 500
What is the price? 1.58
Please pay $790.0.
>>>


Enter a fruit name: green tomato
Is it edible? Enter "1" or "True" if edible; otherwise, "0" or "False": 0
**************************************************
Fruit name (upper case): GREEN TOMATO.
It's not edible.
>>>
```