Functions

CSE 1310 – Introduction to Computers and Programming
Vassilis Athitsos
University of Texas at Arlington

Why Do We Need Functions

- To write better code:
  - More correct, easier to read/write/change. Sdfsdf
- Functions help us organize code.

YOU CANNOT WRITE NON-TRIVIAL PROGRAMS IF YOU DO NOT USE FUNCTIONS

An Example: Numerical User Input

- In lots of programs, we use a line like:

```
number = input("please enter a number: ")
```

- If the user does not enter a number, the program crashes.
  - What is wrong with that?

An Example: Numerical User Input

- In lots of programs, we use a line like:

```
number = input("please enter a number: ")
```

- If the user does not enter a number, the program crashes.
  - What is wrong with that?
- Imagine registering for classes. You have to enter a course number. If you enter by accident 131a instead of 1310, do you want:
  - To get a useful error message, or
  - Your browser to crash.

An Example: Numerical User Input

- We need to devise a strategy for getting a number from the user, without any possibility of the program crashing in the process.

- Note: the following is NOT a solution, will also crash if the user does not enter a valid number:

```
>>> text = raw_input("please enter a number: ")
>>> number = int(text)
```

An Example: Numerical User Input

- We need to devise a strategy for getting a number from the user, without any possibility of the program crashing in the process.

- Note: the following is NOT a solution, will also crash if the user does not enter a valid number:

```
>>> text = raw_input("please enter a number: ")
>>> number = int(text)
```

ANY IDEAS FOR A STRATEGY?

Verifying Numerical User Input

- A strategy:
  - Call **raw_input** to get the user input.
    - This never crashes.
  - Verify (writing our own code) that the text entered by the user is a valid integer or float.
  - After verification, call **int** or **float** to do the conversion.

Verifying Numerical User Input

- A strategy:
  - Call **raw_input** to get the user input.
    - This never crashes.
  - Verify (writing our own code) that the text entered by the user is a valid integer or float.
  - After verification, call **int** or **float** to do the conversion (if the text has passed our code's check, the **int** and **float** functions will not crash).

How do we do this verification?

Verifying Numerical User Input

- In order for a string to represent an integer, what should be legal format for the string?

Verifying Numerical User Input

- In order for a string to represent an integer, what should be legal format for the string?
- Can have spaces at the beginning.
- First non-space character should be number or minus sign.
- Rest of non-space characters should be digits from 0 to 9.
- Can have spaces at the end.
- No spaces allowed except at beginning and end.

How Do We Use This Strategy?

- Implementing this strategy takes several tens of lines of code.
  - See verify_integer1.py, verify_integer2.py
- How do we use that to get an integer from the user without the program crashing?

Using Our Solution

- Previously, we saw this program for computing the sum of numbers from 1 to N (where the user specifies N):

```
# get N from the user
N_string = raw_input("please enter N: ")
N = int(N_string)

# compute the sum
total = 0
for i in range(0, N+1):
    total = total + i

# print the result
print "total =", total
```

Using Our Solution

- That program crashes if the user does not input a valid integer.
- Let's incorporate our solution, to make the program not crash.
  - Result: see summing_to_N_no_functions.py

```
# get N from the user
N_string = raw_input("please enter N: ")
N = int(N_string)

# compute the sum
total = 0
for i in range(0, N+1):
    total = total + i

# print the result
print "total =", total
```

Using Our Solution

- That program crashes if the user does not input a valid integer.
- Let's incorporate our solution, to make the program not crash.
  - Result: see summing_to_N_no_functions.py
- **What is the problem with this new code?**

Using Our Solution

- That program crashes if the user does not input a valid integer.
- Let's incorporate our solution, to make the program not crash.
  - Result: see summing_to_N_no_functions.py
- **What is the problem with this new code?**
- Plus:
  - Compared to previous version, it doesn't crash.
- Minus:
  - The code is hard to read.

Problem of Copying and Pasting Code

- In general, what is wrong with copying and pasting our integer-checking code to any function that needs it?
  - Code becomes ugly (hard to read) wherever we do that.
  - If we ever want to change our integer-checking code (let's say to allow scientific notation, like 10e3), we have to **MODIFY EVERY SINGLE FILE WHERE WE COPIED AND PASTED THIS CODE**.

    This is horrible, a recipe for disaster in large software projects.

Using Functions, Step 1 (Defining)

- Look at summing_with_functions_1.py
- The logical structure of the code is clearer.
- The programmer or reader of the code knows that the main part begins at a specific point, and the rest is just auxiliary code.
- Problem: we still have not fixed the need to copy-paste our function.

Using Functions, Step 2 (importing)

- Look at:
  - number_check.py
  - summing_to_N_with_functions_2.py
- number_check.py defines our function.
- summing_to_N_with_functions_2.py uses the function.
- We have achieved our goals:
  - The code does not crash.
  - The code is easy to read.
  - **The integer-checking function is easy to modify.**
    - **Hundreds of files can import number_check.py**
    - **If we want to make changes to support scientific notation, fix a bug, or whatever, we only need to change number_check.py.**

Using Functions, Step 2 (importing)

- Look at:
  - number_check.py
  - summing_to_N_with_functions_2.py
- In file summing_to_N_with_functions_2.py, we need to call the check_integer function, which is defined in number_check.py
- To do that, we need to tell Python that, in summing_to_N_with_functions_2.py we are using code defined in number_check.py.
- This is done using an **import** statement.

Importing, First Alternative

- To tell Python that summing_to_N_with_functions_2.py uses code defined in number_check.py, we can put the following line in the beginning of summing_to_N_with_functions_2.py:

  import number_check

- Then, whenever we need to call the check_integer function within summing_to_N_with_functions_2.py, the name of the file where the function is defined must precede the function name.

  number_check.check_integer(my_string)

summing_to_N_with_functions_2.py, v1

```
import number_check

# get N from the user, keep asking until an integer is entered
while True:
    my_string = raw_input("please enter N: ")
    if (number_check.check_integer(my_string) == False):
        print "string", my_string, "is not a valid integer"
    else:
        break

N = int(my_string)
# compute the sum
total = 0
for i in range(0, N+1):
    total = total + i
```

Importing, Second Alternative

- To tell Python that summing_to_N_with_functions_2.py uses code defined in number_check.py, we can put the following line in the beginning of summing_to_N_with_functions_2.py:

  ```
  from number_check import *
  ```

- The difference from the **"import number_check"** alternative is that now, when calling check_integer, the filename **does not need to precede** the function name.

  ```
  check_integer(my_string)
  ```

summing_to_N_with_functions_2.py, v2

```
from number_check import *

# get N from the user, keep asking until an integer is entered
while True:
    my_string = raw_input("please enter N: ")
    if (check_integer(my_string) == False):
        print "string", my_string, "is not a valid integer"
    else:
        break

N = int(my_string)
# compute the sum
total = 0
for i in range(0, N+1):
    total = total + i
```

Importing, Third Alternative

- To tell Python that summing_to_N_with_functions_2.py uses code defined in number_check.py, we can put the following line in the beginning of summing_to_N_with_functions_2.py:

  from number_check import check_integer

- With this alternative, if file number_check.py defines many functions, only the check_integer function is visible from summing_to_N_with_functions_2.py (using the second alternative, all functions would be visible). To call check_integer, we still use this line:

  check_integer(my_string)

summing_to_N_with_functions_2.py, v3

```
from number_check import check_integer

# get N from the user, keep asking until an integer is entered
while True:
    my_string = raw_input("please enter N: ")
    if (check_integer(my_string) == False):
        print "string", my_string, "is not a valid integer"
    else:
        break

N = int(my_string)
# compute the sum
total = 0
for i in range(0, N+1):
    total = total + i
```

What is a Function

- Consider a toy function:

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

- This function defines a new type of expression. From now on, if Python sees expression **square(5)**, it will evaluate this expression according to the definition of the function.
- In general, a function definition defines a **new expression** or **a new statement**.
  – It defines an expression if it returns a value.
  – It defines a statement if it **returns no value.**

A Function Defining a Statement

```
def print_greeting(name):
    print "hello,", name, ", how are you?"
```

```
>>> print_greeting("mary")
hello, mary, how are you?
```

- Function print_greeting does not compute and return a value, it just does something useful.

Function Arguments

- Functions have arguments.
- To call a function XYZ, you use this form:

  XYZ(argument_1, …, argument_N)

- How would you know how many arguments to use?

Function Arguments

- Functions have arguments.
- To call a function XYZ, you use this form:

  XYZ(argument_1, …, argument_N)

- How would you know how many arguments to use?
  - From the function definition, which (among other things) defines EXACTLY how many arguments to provide, and in what order.

How a Function is Evaluated

- Function definition:

```
def square(x):
        result = x*x
        return result
```

- Example function call:

```
n = input("enter a number:")
sq = square(n)
print "the square of", n, "is", sq
```

- Processing:

How a Function is Evaluated

- Function definition:

```
def square(x):
        result = x*x
        return result
```

- Example function call:

```
n = input("enter a number:")
sq = square(n)
print "the square of", n, "is", sq
```

- Processing this line:
  - Assume the user entered number 15.

```
Main Namespace:

n = 15
```

How a Function is Evaluated

- Function definition:

```
def square(x):
        result = x*x
        return result
```

- Example function call:

```
n = input("enter a number:")
sq = square(n)
print "the square of", n, "is", sq
```

- Processing this line:
  - Function call.

Main Namespace:

n = 15

How a Function is Evaluated

- Function definition:

```
def square(x):
        result = x*x
        return result
```

- Example function call:

```
n = input("enter a number:")
sq = square(n)
print "the square of", n, "is", sq
```

- Initializing function call:
  - Create new namespace.
  - Assign values to arguments.

Main Namespace:

n = 15

Square Namespace:

x = 15

How a Function is Evaluated

- Function definition:

```
def square(x):
        result = x*x
        return result
```

- Example function call:

```
n = input("enter a number:")
sq = square(n)
print "the square of", n, "is", sq
```

- Assigning values to args:
  - It is an **assignment** operation:
  - x = value of n **from caller namespace**

Square Namespace:

x = 15

Main Namespace:

n = 15

How a Function is Evaluated

- Function definition:

```
def square(x):
    result = x*x
    return result
```

- Example function call:

```
n = input("enter a number:")
sq = square(n)
print "the square of", n, "is", sq
```

- Processing this line:

Square Namespace:

x = 15
result = 225

Main Namespace:

n = 15

How a Function is Evaluated

- Function definition:

```
def square(x):
        result = x*x
        return result
```

- Example function call:

```
n = input("enter a number:")
sq = square(n)
print "the square of", n, "is", sq
```

- Processing this line:
  - Function returns a value.

Square Namespace:

x = 15
result = 225

Main Namespace:

n = 15

How a Function is Evaluated

- Function definition:

```
def square(x):
        result = x*x
        return result
```

- Example function call:

```
n = input("enter a number:")
sq = square(n)
print "the square of", n, "is", sq
```

- Processing this line:
  - Function returns a value.
  - Must **transfer that value to caller.**

Main Namespace:

n = 15
sq = 225

Square Namespace:

x = 15
result = 225

How a Function is Evaluated

- Function definition:

```
def square(x):
        result = x*x
        return result
```

- Example function call:

```
n = input("enter a number:")
sq = square(n)
print "the square of", n, "is", sq
```

- Processing this line:
  - Normal print statement, prints something.

```
Main Namespace:

n = 15
sq = 225
```

How a Function is Evaluated

- Function definition:

```
def square(x):
        result = x*x
        return result
```

- Example function call:

```
n = input("enter a number:")
sq = square(n)
print "the square of", n, "is", sq
```

- Processing this line:
  - Note: the namespace for square has disappeared.

Main Namespace:

n = 15
sq = 225

Understanding Function Evaluation

- Every function evaluation involves:
  - a calling line of code
  - the namespace of the calling line
  - arguments provided by the calling line of code
  - the function that we actually call
  - the namespace of the function call
  - the names of the arguments that the function uses
  - the body of the function
  - (optionally) a return value of the function

Calling a Function

- When we call a function:
  - A new namespace is created.
  - The first variables in the new namespace are the arguments of the function.
  - The arguments are assigned values obtained from the calling line, **using the namespace of the calling line.**
  - The next line of code that is executed is the first line of the body of the function.

Executing a Function Call

- When the body of the function starts executing, code execution follows the same rules we have been using, except that:
  - The only namespace visible is the namespace of the function call.
  - The namespace of the calling line (or any other namespaces) is invisible.

Completing a Function Call That Returns a Value

- When, while executing the body of a function, we find a **return** statement:
  - The expression after the **return** keyword is evaluated.
  - The value of that expression is **transferred to the calling line and used by the calling line.**
- From that point on, the code resumes execution from the calling line.
  - The visible namespace becomes again the namespace of the calling line.
  - **The namespace of the function call disappears.**
  - Any values computed by the function call, that were not returned, are lost forever.

The "Main" Code

- In order for a Python file to do something, it must include some code outside function definitions.
  - Until we did functions, the entire code was outside function definitions.
- This code that is outside definitions is called the **main code** of the program.
- Now that we have started using functions, the main code will be a relatively small part of the program.
- Why?
  - Some reasons we have seen (code easier to read/change).
  - Some reasons we will see (e.g., code easier to **write/design)**

Where Do We Store Functions?

- For toy programs (especially programs written in class), we will often put functions in the same file with the main code.
- In exams, obviously you do not need to specify different files.
- For your assignments, for each task where you write code, there will be two files:
  - task*xxx*_functions.py
  - task*xxx*_main.py

Computing the Divisors of a Number

- Function specifications:
  – Input: an integer *x.*
  – Output: a **list** of divisors of *x*.
  – Error handling: returns None if x is not a positive integer.

Computing the Divisors of a Number

- Function specifications:
  - Input: an integer *x.*
  - Output: a **list** of divisors of *x*.
  - Error handling: returns None if x is not a positive integer.

    None is a value of type NoneType.
    NoneType only has one legal value: None
    Good choice for returning a value signifying something went wrong.

Computing the Divisors of a Number

```
def list_divisors(n):
    if not(type(n) is int):
        return None
    if n < 1:
        return None

    result = []
    for i in range(1, n+1):
        remainder = n % i
        if (remainder == 0):
            result.append(i)

    return result
```

Using the list_divisors function

```
from divisors import *

# small main code to check function divisors
while True:
    number = input("enter a number, -1 to quit: ")
    if number == -1:
        break
    divisors = list_divisors(number)
    print "divisors: ", divisors
```

Converting a List of Characters to a String

- list_to_string(n):
  - Input: a list of characters list1.
  - Output: a string of those characters
  - Error handling: returns False if list1 is not a list, or if any element of the list is not a character.
  - A character is defined to be a string of length 1.

Converting a List of Characters to a String

```
def list_to_string(list1):
    if not(type(list1) is list):
        return None

    result = ""
    for i in list1:
        if not(type(i) is str):
            return None
        if not(len(i) == 1):
            return None

        result = result + i;

    return result
```

Using the list_to_string Function

```
from list_to_string import *

# small main code to check function list_to_string
list1 = ['h', 'e', 'l', 'l', 'o']
print list_to_string(list1)

list2 = ['h', 'ell', 'o']
print list_to_string(list2)
```

What Will This Print?

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2


var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2
```

Main Namespace:

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Current line

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2
```

Main Namespace:

var1 = "hello"

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Current line adds var1

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2
```

Main Namespace:

var1 = "hello"

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Current line

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2
```

Main Namespace:

var1 = "hello"
var2 = "goodbye"

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Current line adds var2

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2
```

Main Namespace:

var1 = "hello"
var2 = "goodbye"

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Current line

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2
```

Main Namespace:

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
```

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Current line adds var3

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2
```

Main Namespace:

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
```

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Current line

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2
```

Main Namespace:

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
```

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Current line adds var4

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2
```

```
Main Namespace:

var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
```

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Current line is function call

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2
```

Main Namespace:

var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Current line is function call
Must be processed together
with header of foo.

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2


var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Main Namespace:

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
```

foo Namespace:

Current line is function call
Must be processed together
with header of foo.
Step 1: create new name space

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2



var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

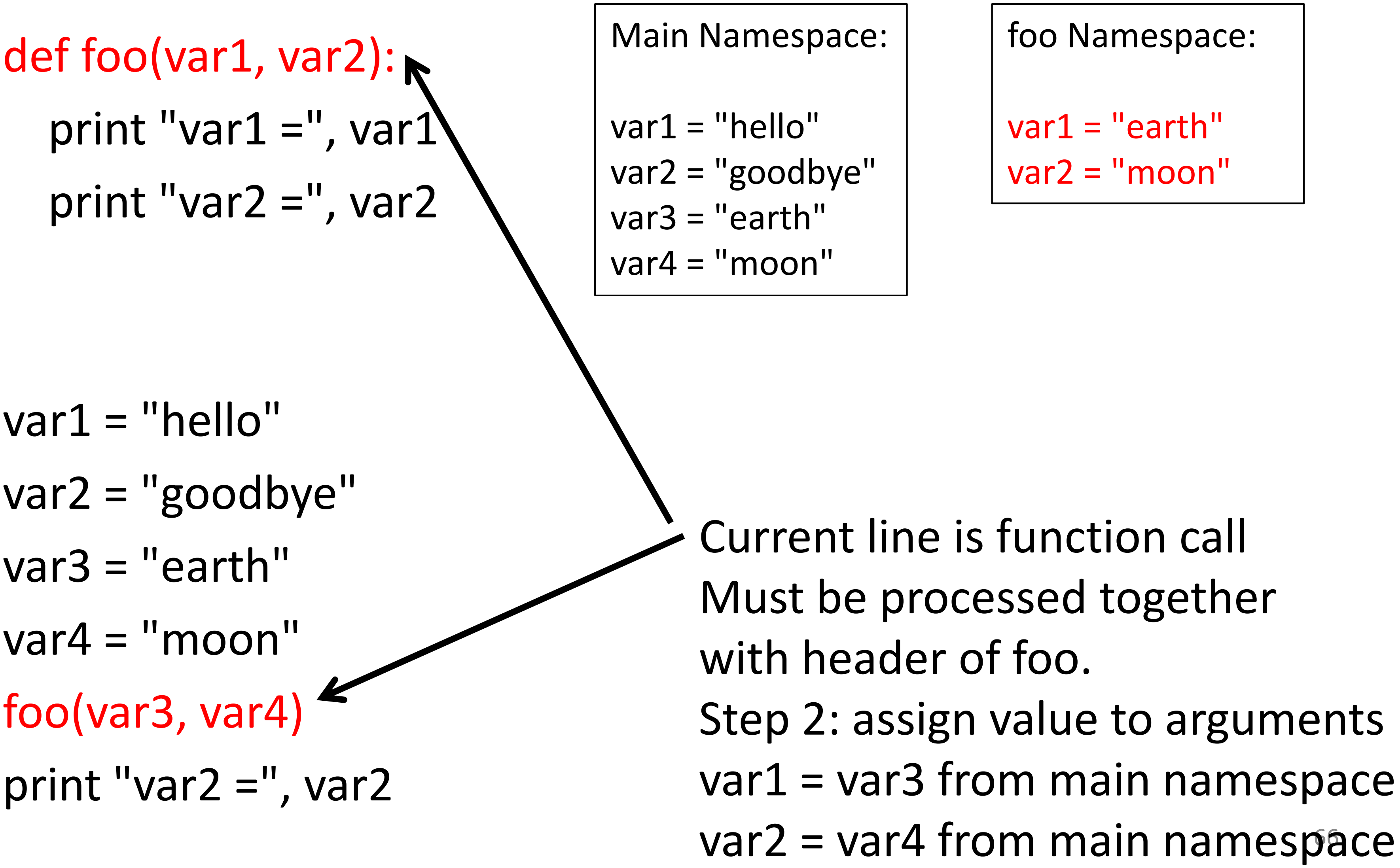
Main Namespace:

var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"

foo Namespace:

Current line is function call
Must be processed together
with header of foo.
Step 2: assign value to arguments
var1 = ???
var2 = ???

Step-by-step Execution



```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2


var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Main Namespace:

var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"

foo Namespace:

var1 = "earth"
var2 = "moon"

Current line is function call
Must be processed together
with header of foo.
Step 2: assign value to arguments
var1 = var3 from main namespace
var2 = var4 from main namespace

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2


var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Main Namespace:

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
```

foo Namespace:

```
var1 = "earth"
var2 = "moon"
```

Current line

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2


var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

```
Main Namespace:

var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
```

```
foo Namespace:

var1 = "earth"
var2 = "moon"
```

Current line

How does Python know which var1 to use?

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2


var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

```
Main Namespace:

var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
```

```
foo Namespace:

var1 = "earth"
var2 = "moon"
```

Current line
How does Python know which var1 to use?

PYTHON ALWAYS USES THE NAMESPACE OF THE CURRENT FUNCTION CALL

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2


var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Current line

Main Namespace:

var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"

foo Namespace:

var1 = "earth"
var2 = "moon"

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2



var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Main Namespace:

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
```

foo Namespace:

```
var1 = "earth"
var2 = "moon"
```

Next line?

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2
```

Main Namespace:

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
```

foo Namespace:

```
var1 = "earth"
var2 = "moon"
```

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Done with the function call.

Calling line does nothing after the function call (does not receive any return value).

Thus, we proceed to the next line in the main code.

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2
```

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Main Namespace:

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
```

foo Namespace:

```
var1 = "earth"
var2 = "moon"
```

How should we update our namespaces?

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2
```

Main Namespace:

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
```

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

How should we update our namespaces?

The foo namespace disappears!

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2
```

Main Namespace:

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
```

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Current line

How does Python know which var2 to use?

Step-by-step Execution

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2
```

Main Namespace:

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
```

```
var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Current line

How does Python know which var2 to use?

PYTHON ALWAYS USES THE NAMESPACE OF THE CURRENT FUNCTION CALL

Summary of Program Output

```
def foo(var1, var2):
    print "var1 =", var1
    print "var2 =", var2


var1 = "hello"
var2 = "goodbye"
var3 = "earth"
var4 = "moon"
foo(var3, var4)
print "var2 =", var2
```

Output:

```
var1 = "earth"
var2 = "moon"
var2 = "goodbye"
```

Multiple Outputs: **find_substrings**

- See find_substrings.py
- This function:
  - finds all positions in string1 where string2 occurs.
  - returns the total number of such positions, and the list of the positions.
- Thus, find_substrings needs to return two values:
  - a number (count of occurences)
  - a list (list of positions of occurrences).
- Solution: return a container (list or tuple) of the two values.

Multiple Outputs: **find_substrings**

- When we call find_substrings, we have two options for handling the return values.
- Option 1: store the list of return values into a list.

```
>>> result = find_substrings("asfdjaskdlfjsdlkfjds", "as")
>>> result
[2, [0, 5]]
```

Multiple Outputs: **find_substrings**

- When we call find_substrings, we have two options for handling the return values.
- Option 2: simultaneously assign two variables. **This is a new Python trick for you.**

```
>>> [count, positions] = find_substrings("asfdjaskdlfjsdlkfjds", "as")
>>> count
2
>>> positions
[0, 5]
```