Information, Computation, Communication
Learning Python

Functions – Part II

Agenda

- Functions in brief
- Local and global variables
  - Example 1 and 2 and 3
  - Example with lists
  - Variable scope, summary
- Recursive functions
  - Definition
  - Program flow
  - Advantages and disadvantages
  - Example
- Importing functions from files

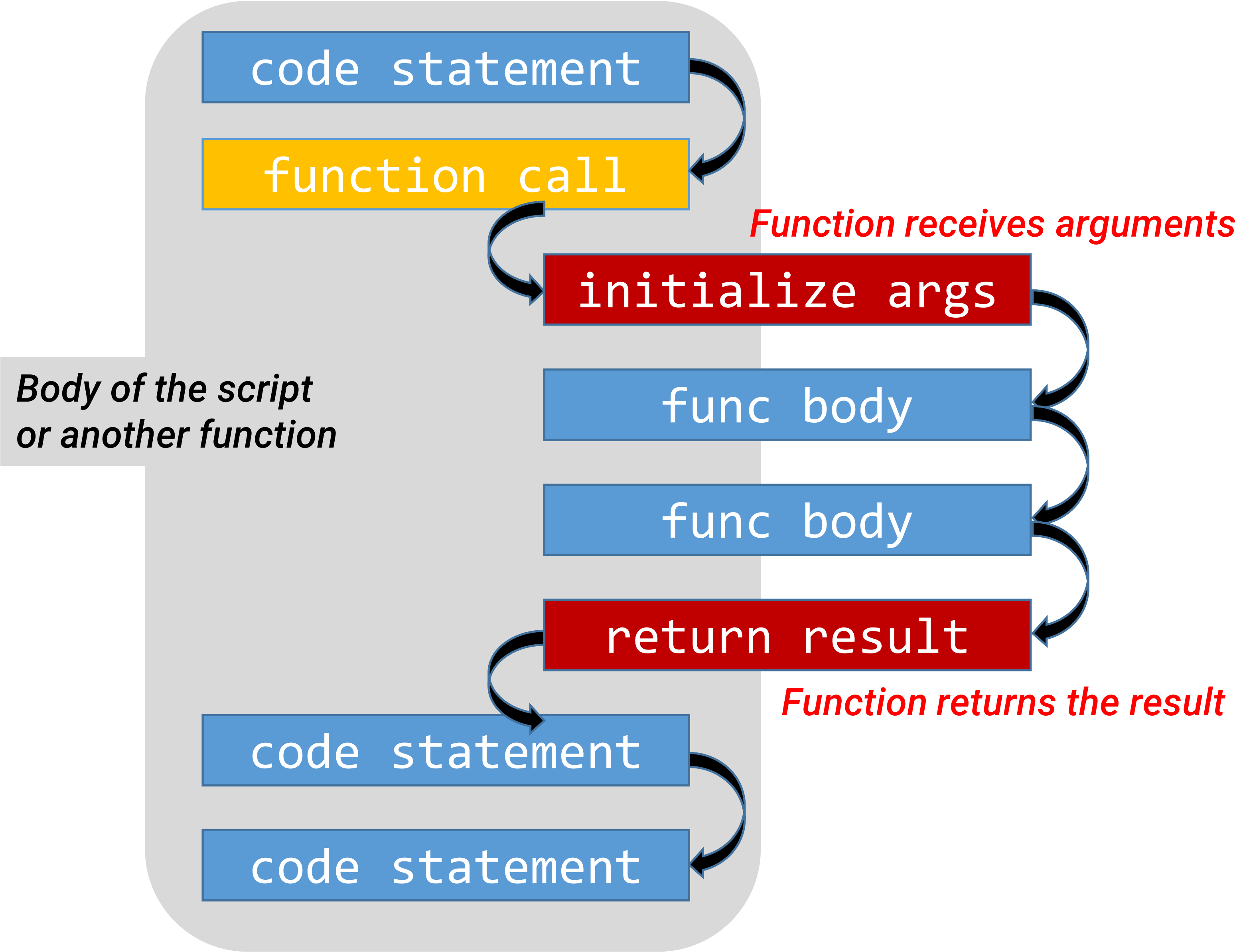
© kras99 / Adobe Stock

Functions – In Brief

- Group lines of code in a way that allows reuse without repetition
    - High code reuse
    - High code readability
    - Low code redundancy
    - Low number of error sources
    - High code maintainability

```
def name(arg1, arg2, …, argN):
    code
    return result
```

Program Flow When Calling a Function



Local and Global Variables

© kras99 / Adobe Stock

Local Variables

- A variable is **local** if it is created in the function body
- Function **arguments** are considered **local** variables
- Local variables are **created** when the function is **called** and **destroyed** when the function **terminates**

- Function can read or modify a variable created elsewhere (outside of it) only if the following is satisfied:
  - The variable name is preceded by the keyword **global** or
  - The variable type is **list, set, or dictionary**

Example 1: Local vs. Global Variables

EXAMPLES

What does this code output?

```
a = 1
b = 100
c = -55

def f_sum(a):
    b = 99
    return a + b

print(a, b, c)
print(f_sum(c), a, b, c)
```

EXAMPLES

Example 1: Local vs. Global Variables

What does this code output?

```
a = 1
b = 100
c = -55

def f_sum(a):    # f_local_a = -55
    b = 99       # f_local_b = 99
    return a + b # f_local_a + f_local_b = 44

print(a, b, c)   # 1 100 -55
print(f_sum(c), a, b, c) # 44 1 100 -55
```

Example 2: Local vs. Global Variables

EXAMPLES

What does this code output?

```
a = 1
b = 100

def f_sub(a):
    global b
    a += 1
    return a - b

print(a, b)
print(f_sub(a), a, b)
```

EXAMPLES

Example 2: Local vs. Global Variables

What does this code output?

```
a = 1
b = 100

def f_sub(a):     # f_local_a = 1
    global b      # b = 100
    a += 1        # f_local_a = 2
    return a – b  # f_local_a – b = -98

print(a, b)       # 1 100
print(f_sub(a), a, b)  # -98 1 100
```

Example 3: Local vs. Global Variables

What does this code output?

```
a = 1
b = 100

def f_sub(a):
    global b
    b += 1
    return a - b

print(a, b)
print(f_sub(a), a, b)
```

Example 3: Local vs. Global Variables

What does this code output?

```
a = 1
b = 100

def f_sub(a):          # f_local_a = 1
    global b           # b = 100
    b += 1             # b = 101
    return a – b       # a - b = 1 – 101 = -100

print(a, b)            # 1 100
print(f_sub(a), a, b) # -100 1 101
```

What if Arguments are Lists/Sets/Dictionaries?

Mutable Types

Then, functions can modify lists/sets/dictionaries created "outside"

```
def f_extender(my_list, factor):
    my_list *= factor

numbers = ['N', 0, 'v']

f_extender(numbers, 2)
print(numbers)

f_extender(numbers, 2)
print(numbers)
```

What if Arguments are Lists/Sets/Dictionaries?

Mutable Types

EXAMPLES

Then, functions can modify lists/sets/dictionaries created "outside"

```
def f_extender(my_list, factor):
    my_list *= factor

numbers = ['N', 0, 'v']

f_extender(numbers, 2) # ['N', 0, 'v', 'N', 0, 'v']
print(numbers)         # numbers was changed by f_extender()

f_extender(numbers, 2) # ['N',0,'v','N',0,'v','N',0,'v','N',0,'v']
print(numbers)         # numbers was changed by f_extender()
```

Variable Scope, Summary

Function **argument** types

- **Booleans, integers, strings, floating-point numbers**
  - Equivalent to local variables
  - Local variables do not exist before the function is called
  - Local variables are destroyed after the function returns
  - Only the code inside the function can use them
- **Lists, dictionaries, sets**
  - Function can modify the external variable passed as the argument
  - All changes made by the function are persistent
    (i.e., visible after the function returns)

Global variables

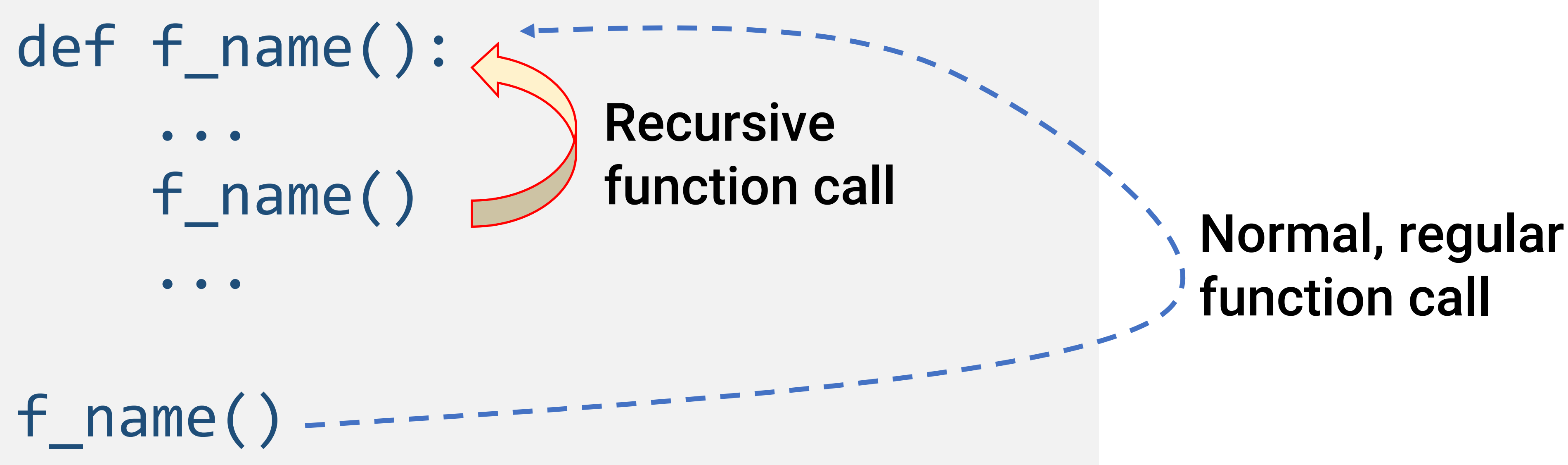
- Created outside of functions but can be read/modified by the code inside the function if preceded by **keyword global**

Recursive Functions

© kras99 / Adobe Stock

Recursive Functions

- Recursive functions are the functions that call themselves



```
def f_name():
    ...
    f_name()
    ...

f_name()
```

Recursive function call

Normal, regular function call

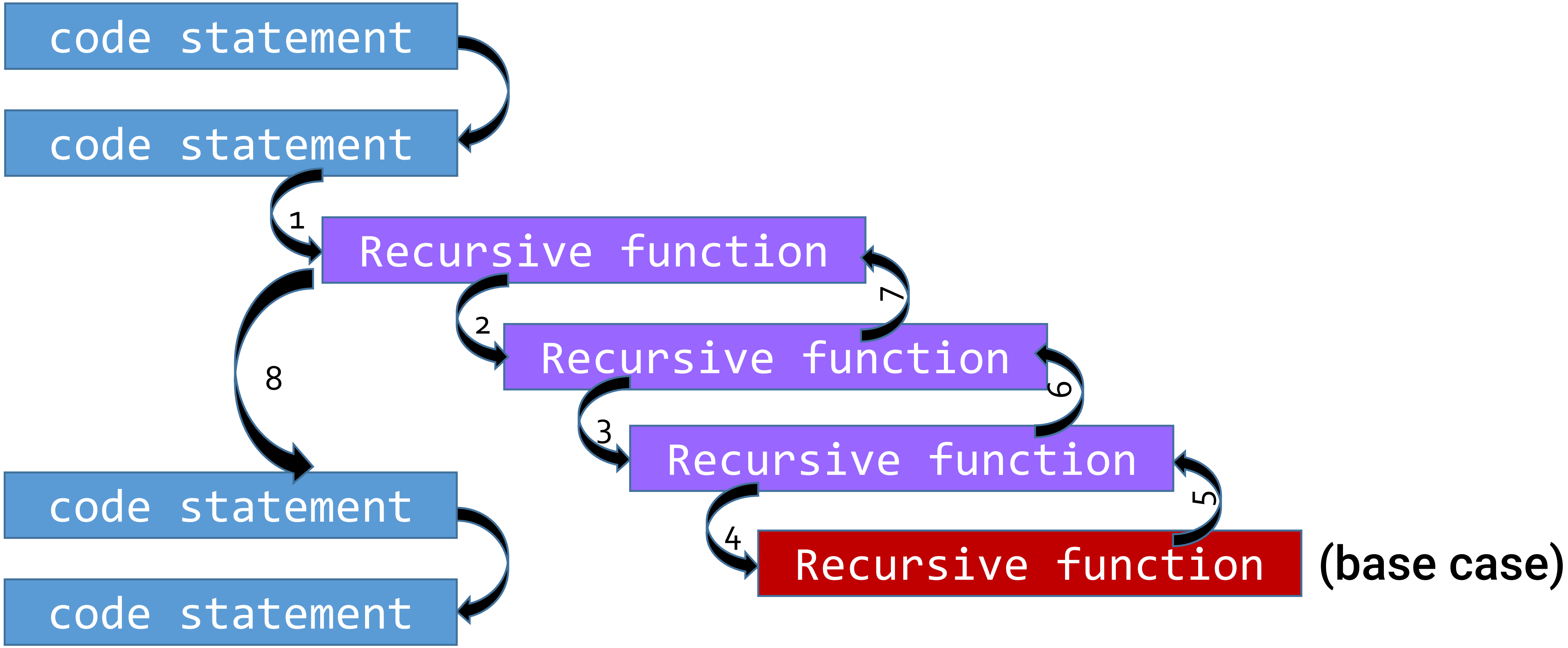
Recursive Functions

- Recursive functions are the functions that call themselves
- Every recursive function must have a **base condition** that stops recursion, or else the function calls itself indefinitely
- In Python, by default, max recursive calls is limited to 1000; past that number, **RecursionError** occurs

Pros and Cons of Recursive Functions

- Advantages
  - Code is clean and elegant
  - Complex task is broken into simpler subproblems
- Disadvantages
  - Sometimes, the logic behind recursion is hard to follow
  - Recursive calls are expensive: take up memory and run time
  - Hard to debug

Recursive Functions: Program Flow



Example: Recursion

f(a, b) is recursive function to calculate the sum of elements in list **a,** from index **b** down to index 3. What does this code output?

```
def f(a, b):
    # Base case
    if b < 3:  # if 'b' < 3, stop the recursion and return 0
        return 0
    # Recursive step: add the element at index 'b' to
    # the result of calling 'f' with 'b-1'
    res = a[b] + f(a, b - 1)
    return res

# Example list of integers
s = [2, 7, 1, 4, 6, 9, 4, 3, 0, 0]
# Call the function 'f' starting from index 8 and print the result
print(f(s, 8))
```

Recursion: Solution

s = [2, 7, 1, 4, 6, 9, 4, 3, 0, 0]

1) f(s, 8) = s[8] + f(s, 7)

2) f(s, 7) = s[7] + f(s, 6)

3) f(s, 6) = s[6] + f(s, 5)

4) f(s, 5) = s[5] + f(s, 4)

5) f(s, 4) = s[4] + f(s, 3)

6) f(s, 3) = s[3] + f(s, 2)

7) f(s, 2) = 0

```
def f(a, b):
    if b < 3:
        return 0
    res = a[b] + f(a, b - 1)
    return res

s = [2, 7, 1, 4, 6, 9, 4, 3, 0, 0]
print(f(s, 8))
```

Once we have reached the end of the recursion, we can start computing the intermediate values

EXAMPLES

Recursion: Solution

s = [2, 7, 1, 4, 6, 9, 4, 3, 0, 0]

1) f(s, 8) = s[8] + f(s, 7)

2) f(s, 7) = s[7] + f(s, 6)

3) f(s, 6) = s[6] + f(s, 5)

4) f(s, 5) = s[5] + f(s, 4)

5) f(s, 4) = s[4] + f(s, 3)

6) f(s, 3) = s[3] + f(s, 2)

7) f(s, 2) = 0

1) f(s, 8) = s[8] + f(s, 7) **= 0 + 26 = 26**

2) f(s, 7) = s[7] + f(s, 6) **= 3 + 23 = 26**

3) f(s, 6) = s[6] + f(s, 5) **= 4 + 19 = 23**

4) f(s, 5) = s[5] + f(s, 4) **= 9 + 10 = 19**

5) f(s, 4) = s[4] + f(s, 3) **= 6 + 4 = 10**

6) f(s, 3) = s[3] + f(s, 2) **= 4 + 0 = 4**

7) f(s, 2) **= 0**

Once we have reached the end of the recursion, we can start computing the intermediate values, one by one

Importing Functions

© kras99 / Adobe Stock

Importing Functions from Files

- Importing **one** function from a file

```
from file_name import function_name
```

- Importing **several** functions from a file

```
from file_name import function1, function2
```

- Importing **all** functions from a file

```
from file_name import *
```

Important: The file should reside in the same directory as your script

EXAMPLES

Importing Functions from Files

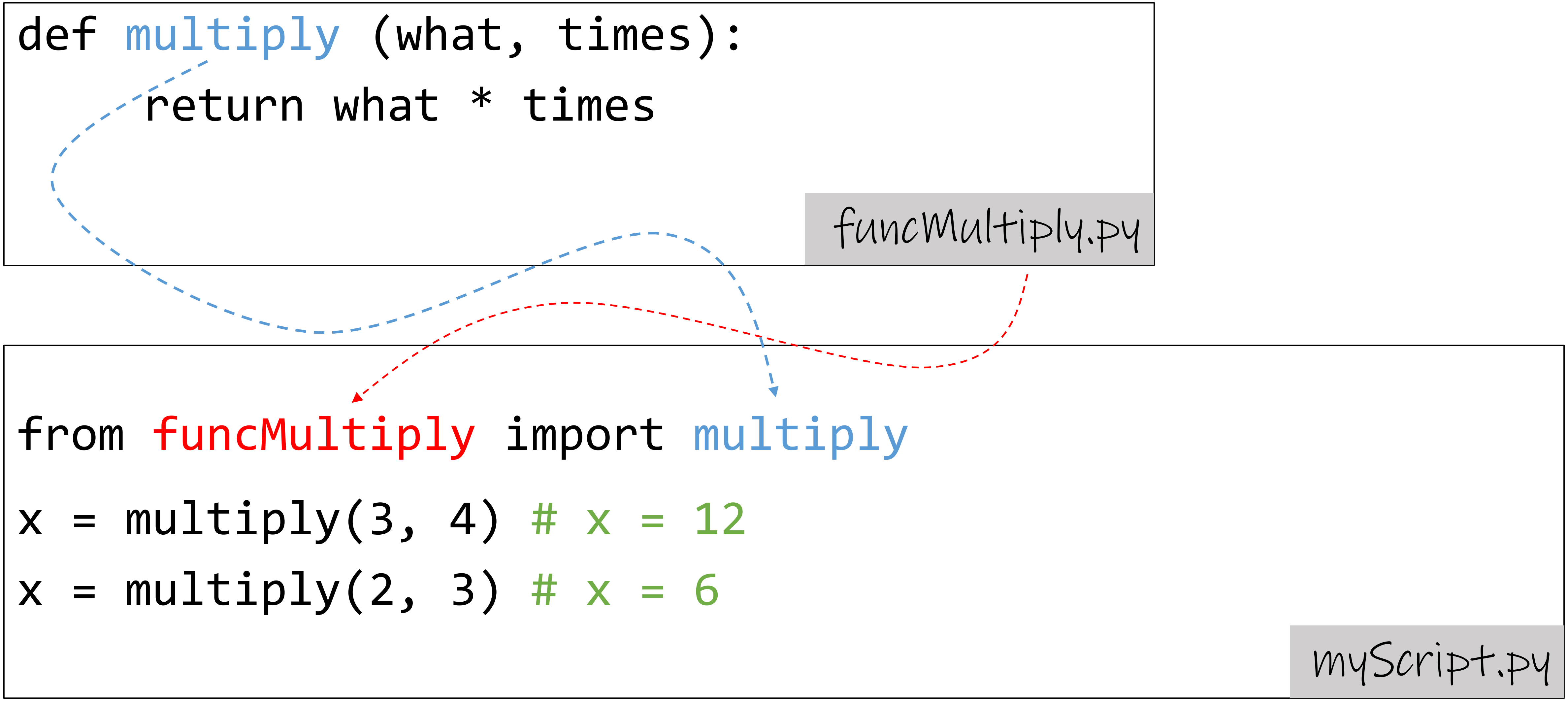
```
def multiply (what, times):
     return what * times
```

funcMultiply.py

```
# How to link this function call to the correct function definition?
x = multiply(3, 4) # x = 12
x = multiply(2, 3) # x = 6
```

myScript.py

Importing Functions from Files



```
def multiply (what, times):
    return what * times
```

funcMultiply.py

```
from funcMultiply import multiply
x = multiply(3, 4) # x = 12
x = multiply(2, 3) # x = 6
```

myScript.py

Next topic:
Tuples and Sets