Information, Computation, Communication

Learning Python

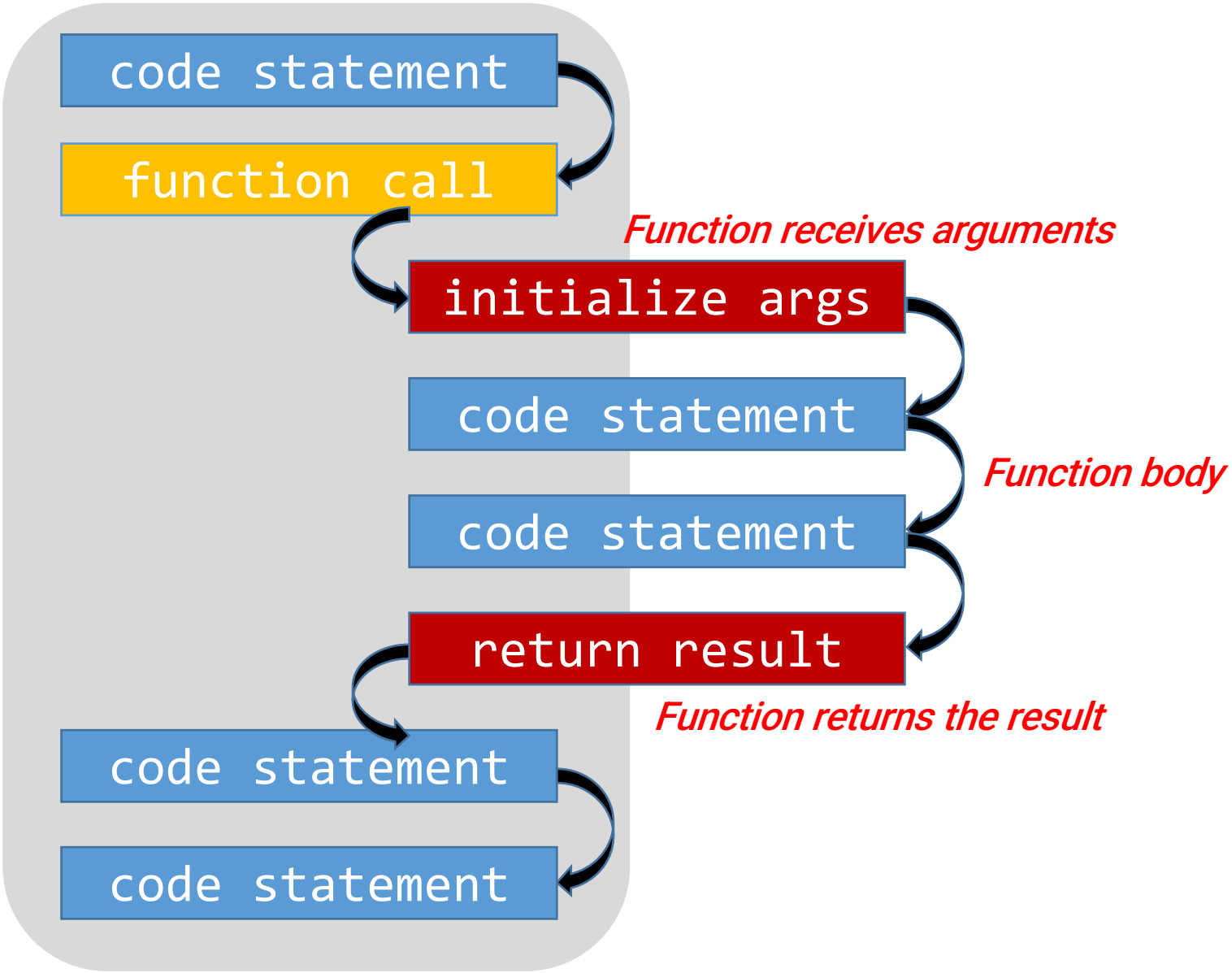
Functions – Part II

Quick Reminder: Functions

- 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



Agenda

- Variable scope
  - Local variables
  - Global variables
  - Function arguments of mutable types

Recursive functions

  - Definition
  - Program flow
  - Advantages and disadvantages
  - Example

- Importing functions from files



© kras99 / Adobe Stock

Variable Scope

Local Variables



© kras99 / Adobe Stock

Local Variables

- A variable is **local to the function**
  - if it is created **inside its** body or
  - if it is one of its **arguments**

- Local variables are
  - **created** when the function is **called** and
  - **destroyed** when the function **terminates**

Example 1: Local Variables

What does this code output?

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

a = 1    # an integer variable
b = 100  # an integer variable
c = -55  # an integer variable

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

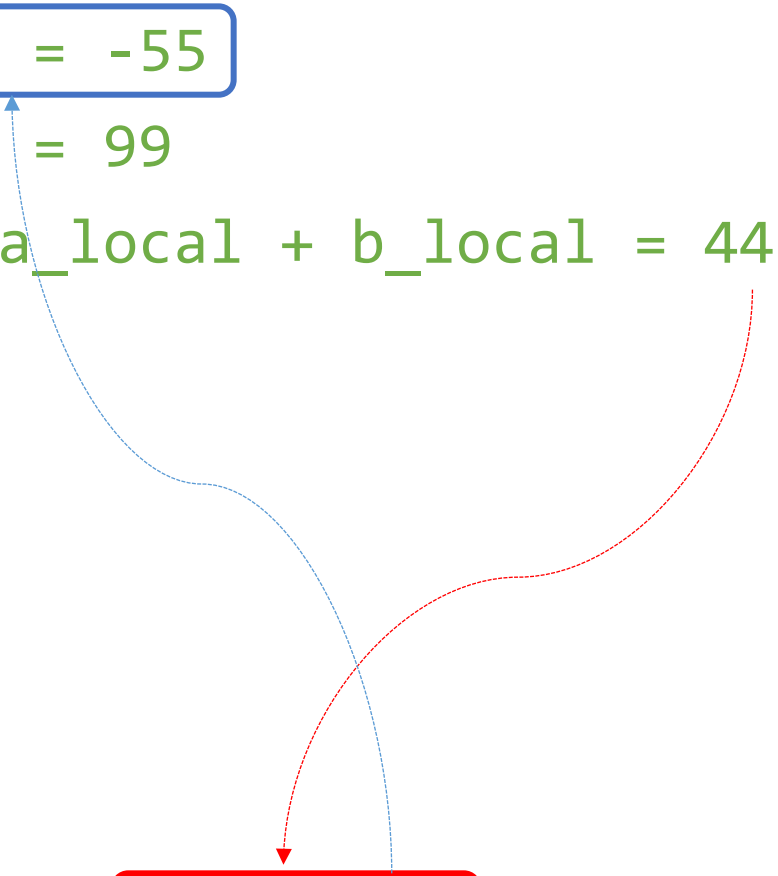
Example 1: Local Variables

What does this code output?

```
def f_sum(a):                  # a_local = -55
    b = 99                     # b_local = 99
    return a + b               # return a_local + b_local = 44


a = 1
b = 100
c = -55

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



Variable Scope

Global Variables



© kras99 / Adobe Stock

Global Variables

- A variable is **external** with respect to the function if it is created **outside** it and outside any other function body

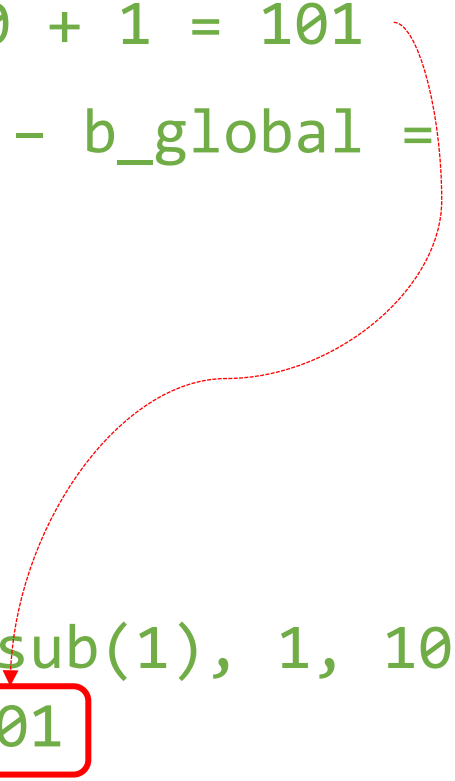
- Normally, a function cannot modify an external variable
- For a function to gain access to an external variable (and, for example, modify it), the function body must include the keyword **global** alongside the variable name

Example 2: Global Variables

What does this code output?

```
def f_sub(a):       # a_local = 1
    global b        # b is external variable, value 100
    b += 1          # b_global  = 100 + 1 = 101
    return a - b    # return a_local - b_global = 1 - 101 = -100


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



Example 3: Global Variables

What does this code output?

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

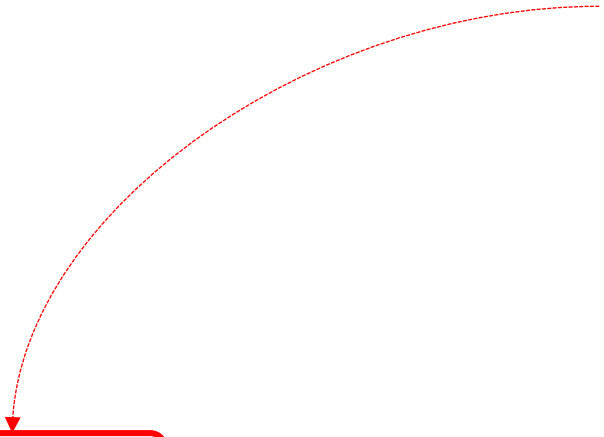

a = 1
b = 100
print(a, b)  # 1 100
print(f_sub(a), a, b)  # print(f_sub(1), 1, 100)
```

Example 3: Global Variables

What does this code output?

```
def f_sub(a):       # a_local = 1
    global b        # b is external variable, value 100
    a += 1          # a_local = 1 + 1 = 2
    return a - b    # return a_local - b = 2 - 100 = -98


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



EXAMPLES

Variable Scope

Function Arguments of Mutable Types (i.e., lists, sets, dictionaries)



© kras99 / Adobe Stock

What if Arguments are Lists/Sets/Dictionaries?

Mutable Types

If arguments are of mutable type, functions **can** modify the external variables they correspond to

```
def f_extender(my_list, factor):
    my_list *= factor       # external list gets modified
```

What if Arguments are Lists/Sets/Dictionaries?

Mutable Types

If arguments are of mutable type, functions can modify the external variables they correspond to

```
def f_extender(my_list, factor):
    my_list *= factor       # external list gets modified

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

f_extender(numbers, 2)
# numbers = ['N', 0, 'v', 'N', 0, 'v']

f_extender(numbers, 2)
# numbers = ['N',0,'v','N',0,'v','N',0,'v','N',0,'v']
```

Variable Scope

Summary

If function **arguments** have the following types

- **Immutable: Boolean, integer, string, floating-point numbers**
  - All changes made inside the function remain local to the function (i.e., last until the function returns/terminates and do not affect any external variable)
- **Mutable: Lists, dictionaries, sets**
  - All changes made inside the function are persistent (i.e., last even after the function returns/terminates and affect external variables)

Global variables

- Functions can use (read, modify) a variable created outside this and any other function, provided the keyword **global** precedes it

Local variables

- Variables created inside the function; they can only be used by the function and last (we also say **live**) until the function returns/terminates

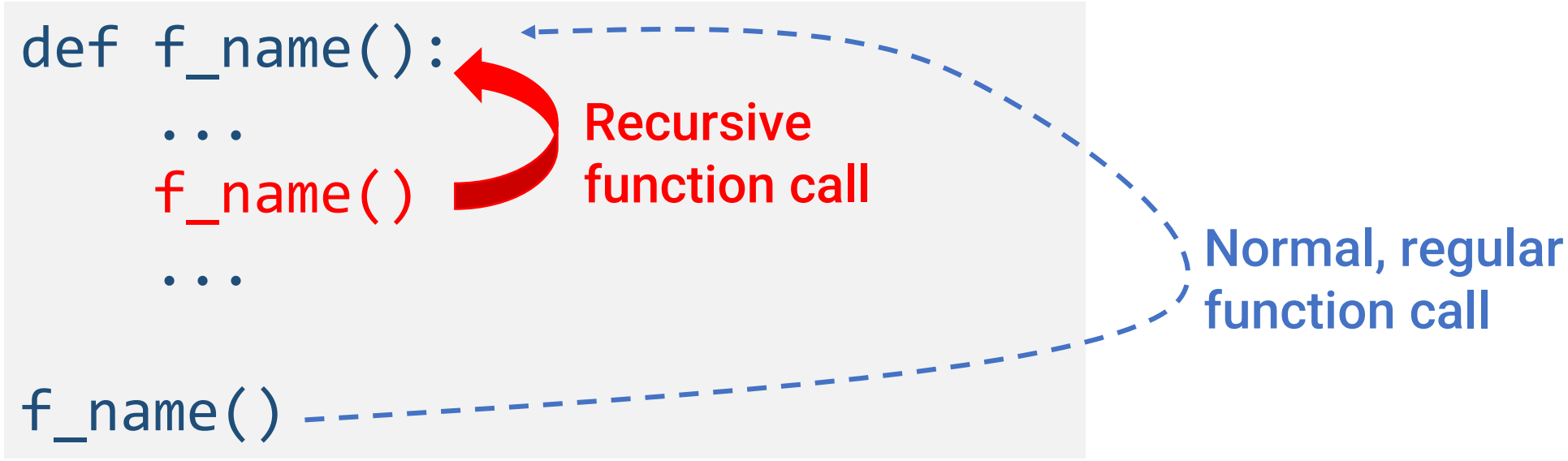
Recursive Functions



© kras99 / Adobe Stock

Recursive Functions

- Recursive functions are the functions that call themselves



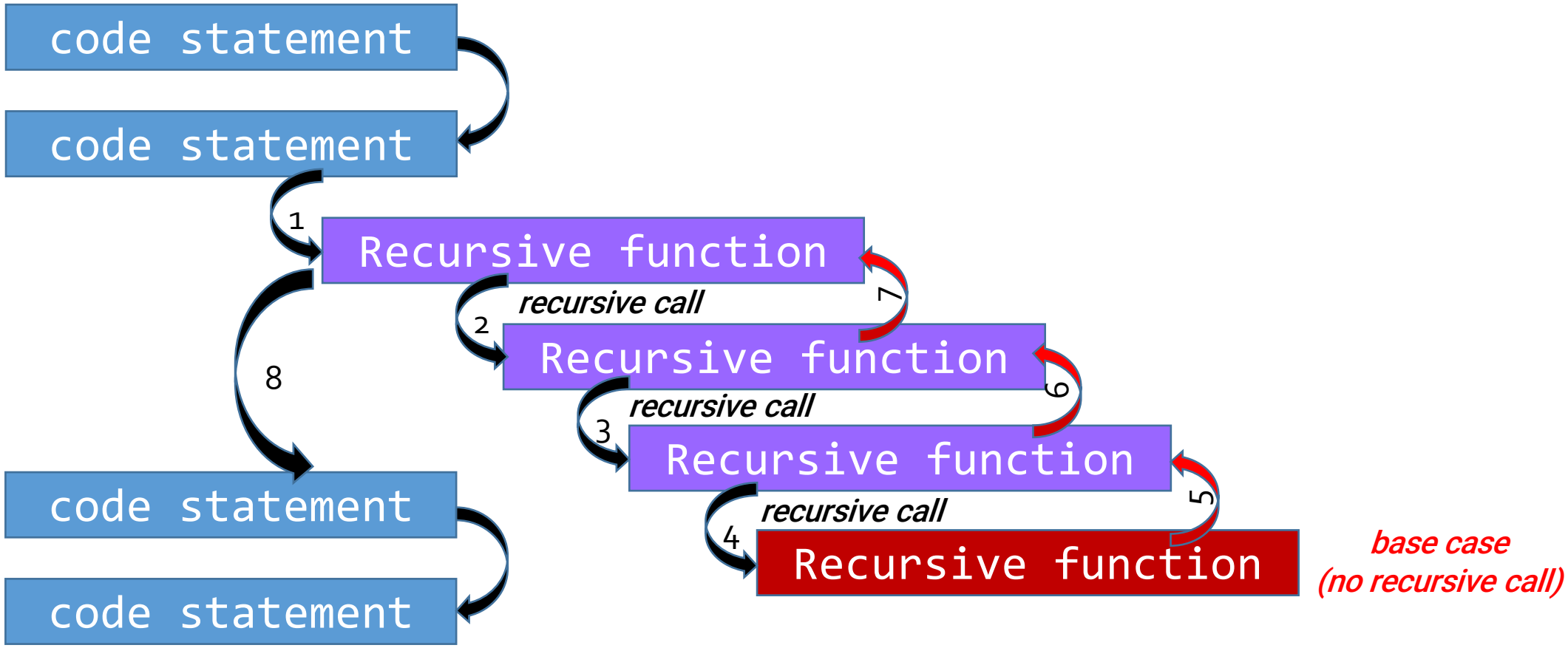
Recursive Functions

- 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
  - *Analogous to preventing infinite loops*
- 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 repetitive subproblems
- Disadvantages
  - Sometimes, the logic behind recursion is hard to follow
  - Recursive calls are expensive: they take up memory and run time
  - Hard to debug

Recursive Functions: Program Flow



Example: Recursion



f(a, b) is a 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        # 2nd return: 0 is a neutral value for the sum
    # Recursive step: add the list element at index b to
    # the sum of all elements from index b-1 down to …
    res = a[b] + f(a, b - 1)
    return res          # 1st return

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

All function calls (six recursive, one base)



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 *base case*

```
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

Recursion: Solution

Computing the result



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 *base case*

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

EXAMPLES

Importing Functions

Python allows us to write functions in separate files and then use/import them in our scripts



© kras99 / Adobe Stock

Importing Functions from Files

- Importing **one** function from a file

```
from file_name import func_name
```

- Importing **several** functions from a file

```
from file_name import func_name_1, func_name_2
```

- Importing **all** functions from a file

```
from file_name import *
```

Important: The file containing function definitions should reside in the same directory as your script that is using those functions

Importing Functions from Files

EXAMPLES

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

funcMultiply.py

```
# How to link this function call and
# 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