Information, Computation, Communication

Learning Python

Functions – Part I

Agenda

- Why functions?
  - Example: Intersection of two lists
- Function definition
  - def, arguments, body, return
- Calling functions
  - Example: multiply and add
- Function arguments, continued
- Type-dependent behavior
  - Example: nested function call
- Example: bubble sort



© kras99 / Adobe Stock

Why Functions?



© kras99 / Adobe Stock

Why Functions?

- Group lines of code in a way that allows reuse without repetition
  - **Maximize code reuse**
  - **Improve code readability**
- Avoid copy-paste (code repetition)
  - **Minimize code redundancy**
  - Minimize the number of **error sources**
- Test code once, and then reuse it as frequently as needed to perform computations on **various** data
  - **Good code maintainability**
- In some programming languages, functions are referred to as subroutines or procedures

Example: Intersection Of Two Lists

- Write a code that finds all elements shared between input lists **seq1** and **seq2** and creates a new list containing only those elements, sorted in increasing order

```
result = []  # Initialize an empty list to store the intersection.
for x in seq1:  # Iterate over each item in the first sequence.
    if x in seq2:  # Check if the item is also in the second sequence.
        if not x in result:
            result.append(x)  # If so, add it to the result list.
result.sort()  # Sort the resulting list to ensure items are in order.
```

Example: Intersection Of Two Lists

- Write a **function** that finds all elements shared between input lists **seq1** and **seq2** and returns a new list containing only those elements, sorted in increasing order

```
def seq_intersect(seq1, seq2):
    result = []  # Initialize an empty list to store the intersection.
    for x in seq1:  # Iterate over each item in the first sequence.
        if x in seq2:  # Check if the item is also in the second sequence.
            if not x in result:
                result.append(x)  # If so, add it to the result list.
    result.sort()  # Sort the resulting list to ensure items are in order.
    return result  # Return the sorted intersection list.
```

Function Definition

Before it can be used, a function must be defined



© kras99 / Adobe Stock

Function Definition

General format and syntax:

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

Function Definition: `def` Statement

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

- **def** keyword creates a function (object) and assigns to it a function **name**
  - Choose a name that best describes what the function does
  - PEP 8 Style Guide recommends:
    - Function names should be lowercase, with words separated by underscores as necessary to improve readability
- Function definition is often part of your script (.py file)

Function Definition: Parameters (Arguments)

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

- The function name is followed by **parentheses**, grouping zero or more function **parameters**
- Parentheses are followed by a **colon**
- Parameters are variables (with **names** and **values**) acting as **inputs** to the function
  - Function code uses those variables to perform computation
- Parameters **connect** the function with the program that uses it

Function Definition: Function Body

- …is simply the code inside the function

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

Function Definition: return Statement

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

- **return** keyword permits the function to answer with a value it obtained for the given set of arguments
- We say that the **function returns a value**
- Return is **not mandatory**
  - Function anyway terminates when it reaches the last line of its body
  - If no return is specified, the function returns **None**
    - The None keyword is used to define a null value or no value at all

Calling Functions

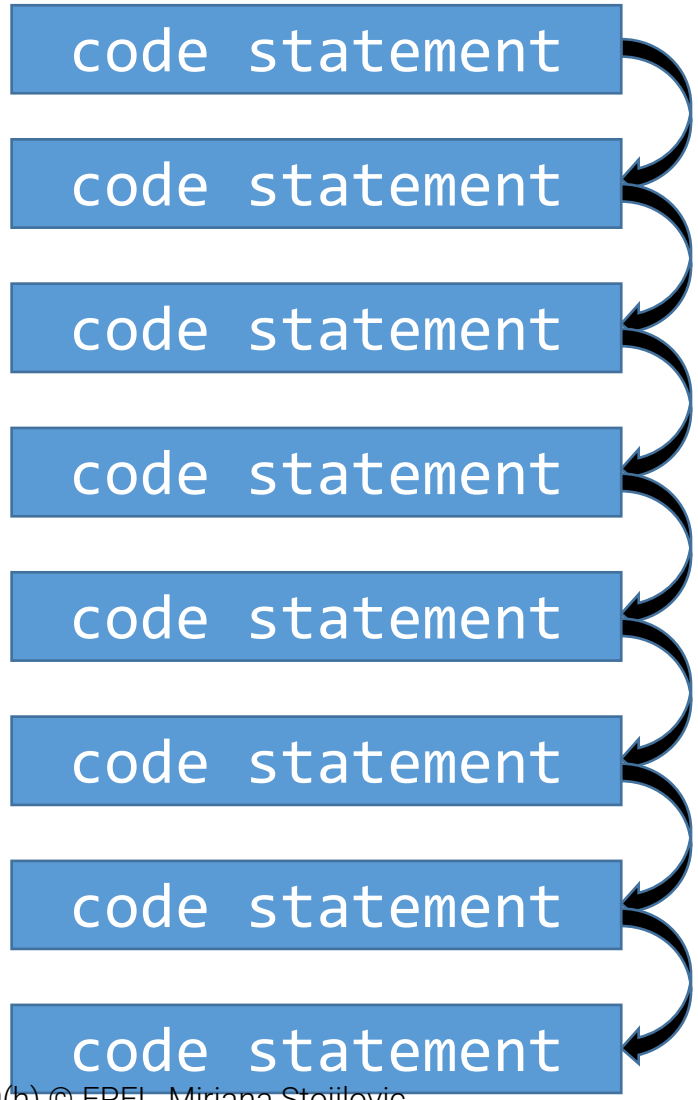
To make use of a function, we call it



© kras99 / Adobe Stock

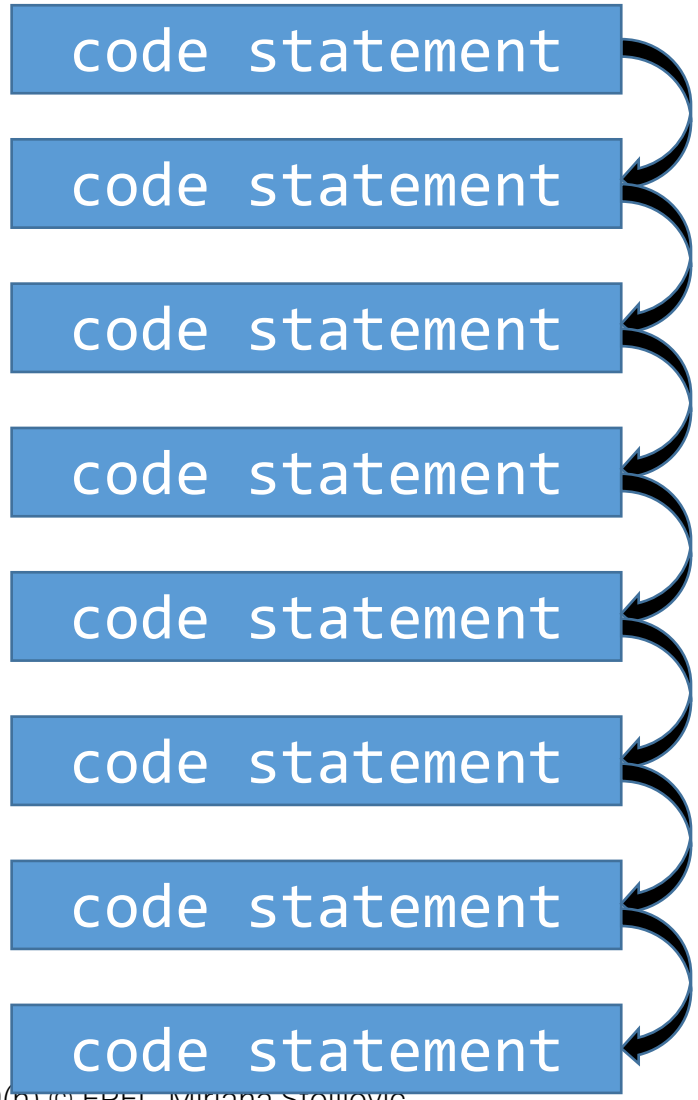
Program Flow

Without functions

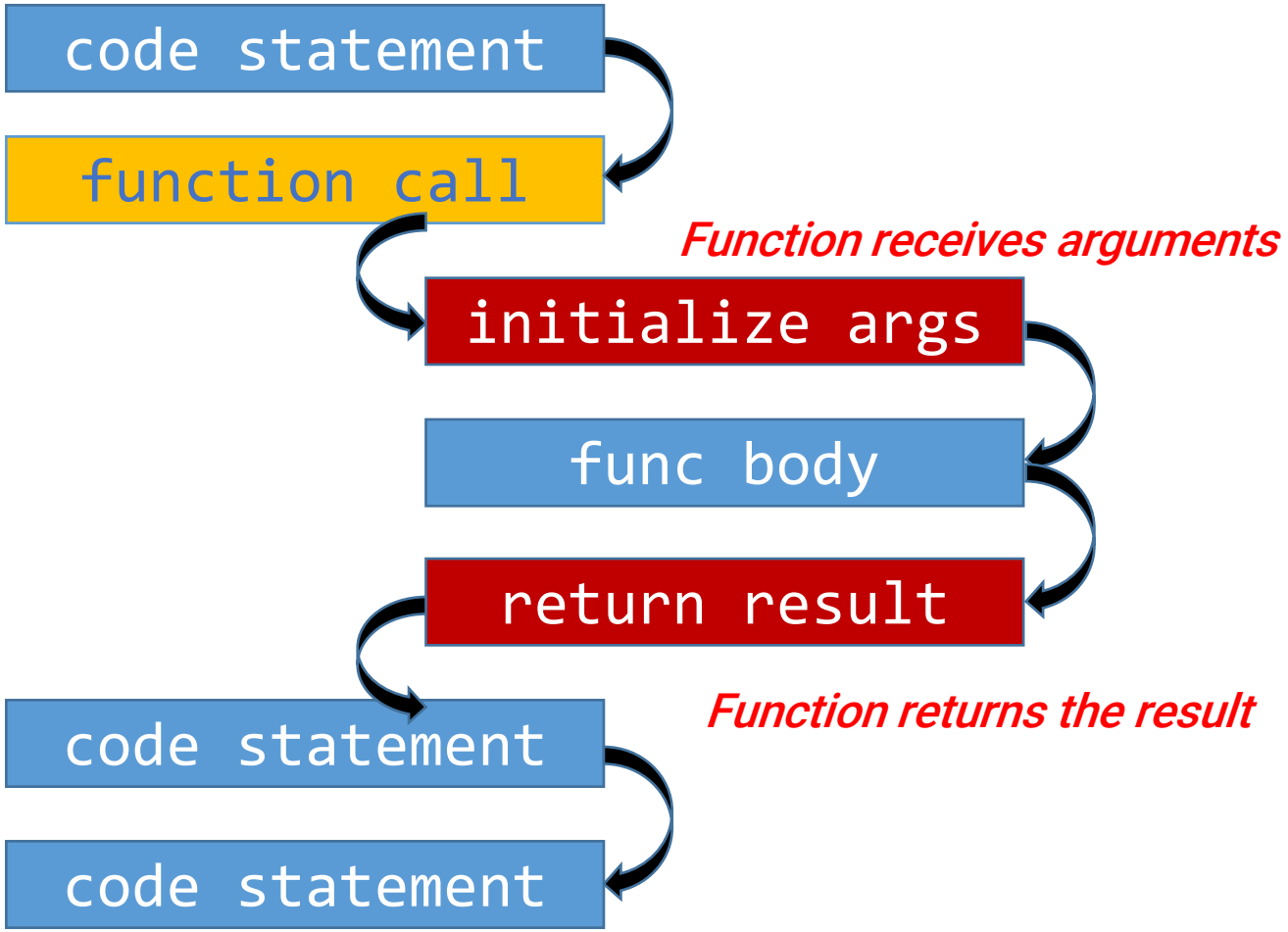


Program Flow

Without functions



With functions



Example: `multiply_and_add_scalar` function

- Consider the example function below, which multiplies two variables and sums the result with the third variable

```
# Start by function definition
def multiply_and_add_scalar(m, n, p):
    return m * n + p
```

- The function receives three arguments: **m**, **n**, and **p**. Therefore when called, this function expects three values. *Then*, it performs the computation on them and, *finally*, returns the result.

A Complete Script, Including Function Calls

```
# Start by function definition
def multiply_and_add_scalar(m, n, p):
    return m * n + p

# Having defined it, we can now use the function (call it)
# m = 3, n = 4, p = 2
# Let us save the function result in variable t
t = multiply_and_add_scalar(3, 4, 2) # t = 14

# m = 5, n = 2, p = t (= 14)
t = multiply_and_add_scalar(5, 2, t) # t = 24
```

Function Arguments

Continued…



© kras99 / Adobe Stock

Function Arguments, Continued…

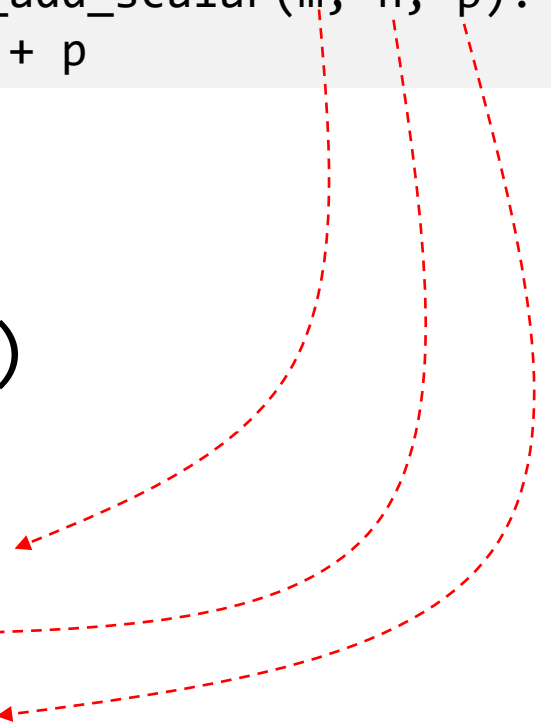
```
def multiply_and_add_scalar(m, n, p):
    return m * n + p
```

- In the previous example

```
t = multiply_and_add_scalar(3, 4, 2)
```

arguments were assigned by **their position**

- 1st position, value 3, was assigned to the 1st argument: **m**
- 2nd position, value 4, was assigned to 2nd argument: **n**
- 3rd position, value 2, was assigned to the 3rd argument: **p**



Function Arguments, Continued…

```
def multiply_and_add_scalar(m, n, p):
    return m * n + p
```

- In Python, arguments can also be assigned **explicitly**

```
multiply_and_add_scalar(m=3, n=4, p=2)
```

- When explicitly assigned, the order becomes irrelevant
- Examples of calls equivalent to the one above
  - `multiply_and_add_scalar(p=2, n=4, m=3)`, or
  - `multiply_and_add_scalar(n=4, p=2, m=3)`, etc.

Function Arguments, Continued…

- Python allows setting **default** values to parameters
- Default values are used if no other value is specified

```
def name(arg1=default_value1, arg2=default_value2, …, argN=default_valueN):
    code # function body
    return result
```

- **Note:** Parameters with default values must be written last in the parameter list; otherwise, the position-based assignment of variables/values to arguments would be incorrect

Example: multiply_and_add_scalar function

```
# Start by function definition
def multiply_and_add(m, n=0.5, p=2):
    return m * n + p

if (some_condition == True):
      t = multiply_and_add_scalar(3)
      # m = 3, n = 0.5 (default), p = 2 (default)
      # Result: t = 3.5
else:
      t = multiply_and_add_scalar(5, 2)
      # m = 5, n = 2, p = 2 (default)
      # Result: t = 12
t = multiply_and_add_scalar()
# Result: the function call throws an error, as m has no value
```

Polymorphism

Type-dependent behavior



© kras99 / Adobe Stock

Type-Dependent Behavior

- In Python, the meaning of every expression depends upon the types of its objects
- This type-dependent behavior is called **polymorphism**
- Why does this matter?
  - The output of your function **may change** depending on the type of values (arguments) it was given

Nested Function Call




```
def multiply_and_add(m, n, p):
    return m * n + p

# What is the value of t?
t = multiply_and_add("I", 1, multiply_and_add("C", 2, "-P"))
print(t)



# Answer: ICC-P
# first, multiply_and_add("C", 2, "-P") returns "CC-P"
# then, multiply_and_add("I", 1, "CC-P") returns "ICC-P"
```

Example: Bubble Sort



Example: Bubble Sort

Write a function **bubble_sort_descending( )** which takes a list of numbers as an argument and returns a list sorted in descending order

[Wiki] Bubble sort is a simple sorting algorithm that repeatedly steps through the input list, element by element, comparing the current element with the next and swapping the values if needed. The process repeats until the list is sorted.

Bubble Sort Descending: Step by Step

- **Initial Step**
  - Start with an unsorted list of elements
  - The algorithm examines each pair of adjacent elements
- **Comparison and Swap**
  - For each adjacent pair, if the first element is smaller than the second, swap them;
    If not, leave them as is
  - This way, the smallest unsorted element "bubbles" up to the end of the list with each pass
- **Repeat Passes**
  - After one pass, the smallest element will be in its correct position at the end
  - Repeat the process for the remaining **unsorted** part of the list, which keeps shrinking as more elements are placed in their final positions
- **Completion**
  - Continue repeating the passes until no swaps are left to do, at which point the list is sorted

Example: Bubble Sort

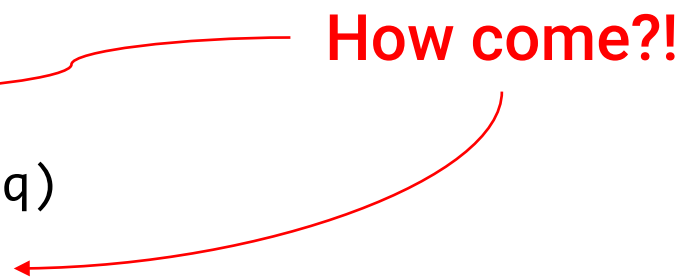
```
# Function
def bubble_sort_descending(arr):
    n = len(arr)
    # Each pass places the smallest remaining element at the end
    # of the unsorted portion.
    for i in range(n-1):
        # compare adjacent elements in the unsorted portion
        for j in range(n - i - 1):
            # swap if the left element is smaller than the right
            if arr[j] < arr[j + 1]:
                temp = arr[j]
                arr[j] = arr[j+1]
                arr[j+1] = temp
    return arr # return the sorted array
```

Example Usage of Bubble Sort Function

```
seq = [5, 3, 8, 4, 2]
print(seq)
# [5, 3, 8, 4, 2]

sorted_seq = bubble_sort_descending(seq)
print(sorted_seq)
# [8, 5, 4, 3, 2]

# Then add this line. What is printed?
seq[0] = -1
print("Original array seq = ", seq)
# Original array seq = [-1, 5, 4, 3, 2]
print("Sorted array sorted_seq = ", sorted_seq)
# Sorted array sorted_seq = [-1, 5, 4, 3, 2]
```



Example: Bubble Sort Returning NEW Array

```
# Now that you know array parameters can be modified inside functions,
# update your function and code to return a NEW array
def bubble_sort_descending_returns_new(unsorted_arr):
    n = len(unsorted_arr)
    arr = unsorted_arr.copy()
    for i in range(n-1):
        for j in range(n - i - 1):
            if arr[j] < arr[j + 1]:
                temp = arr[j]
                arr[j] = arr[j+1]
                arr[j+1] = temp
    return arr
```

Example Usage

```
seq = [5, 3, 8, 4, 2]
print(seq)
# [5, 3, 8, 4, 2]
sorted_seq = bubble_sort_descending_returns_new(seq)
print(seq)
# [5, 3, 8, 4, 2]
print(sorted_seq)
# [8, 5, 4, 3, 2]
# Then add this line. What gets printed?
seq[0] = -1
print("NEW: Original array seq = ", seq)
# NEW: Original array seq = [-1, 3, 8, 4, 2]
print("NEW: Sorted array sorted_seq = ", sorted_seq)
# NEW: Sorted array sorted_seq = [8, 5, 4, 3, 2]
```

As expected for two independent lists

Next topic:
Recursive Functions