FUNCTIONS

Prepared By:

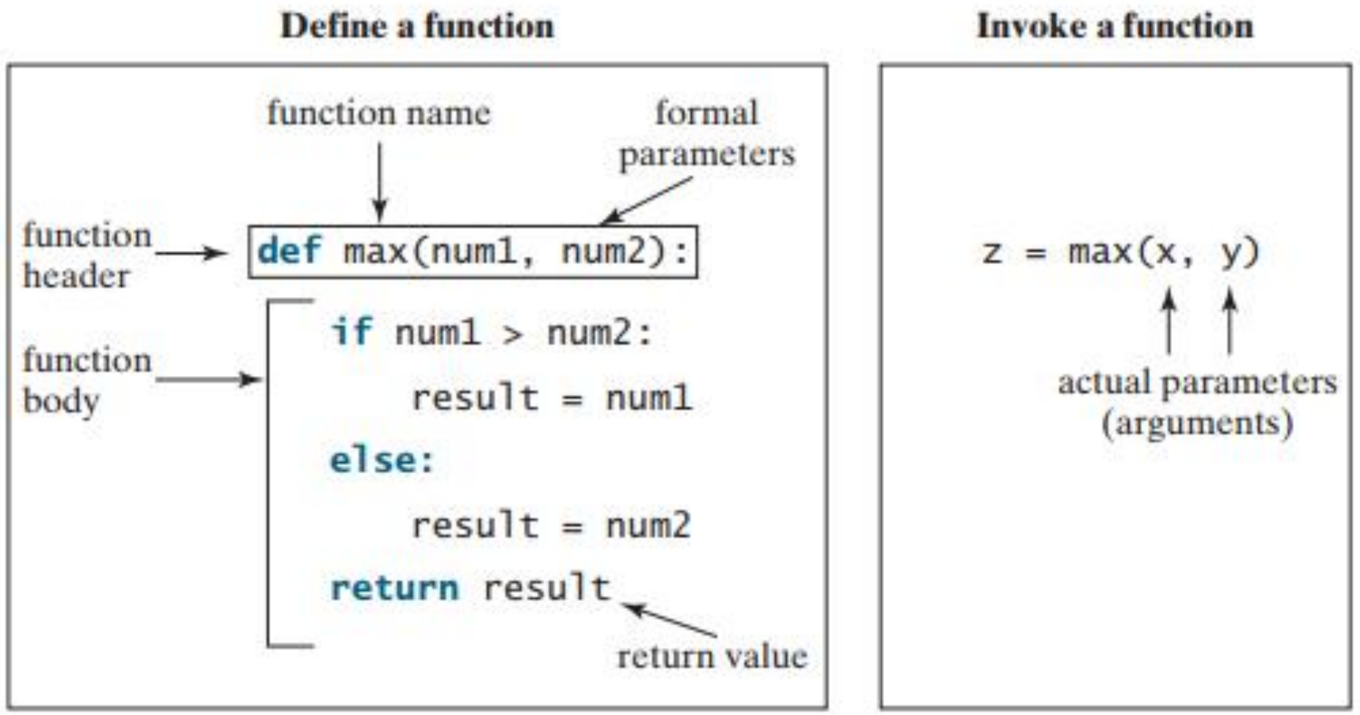
Prof. Prachi Pancholi

Ganpat University

Definition and syntax of function

- *A function definition consists of the function’s name, parameters, and body.*
  The syntax for defining a function is as follows:
  ```
  def functionName(list of parameters):
          # Function body
  ```

Let's look at a function created to find which of two numbers is bigger. This function, named `max`, has two parameters, `num1` and `num2`, the larger of which is returned by the function. Figure 6.1 illustrates the components of this function.



```
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2
    return result
```

```
z = max(x, y)
```

FIGURE 6.1 You can define a function and invoke it with arguments.

- When a function is invoked, you pass a value to the parameter. This value is referred to as an *actual parameter* or *argument*. Parameters are optional;that is, a function may not have any parameters. For example, the **random.random()** function has no parameters.
- Some functions return a value, while other functions perform desired operations without returning a value. If a function returns a value, it is called a ***value-returning function*****.**

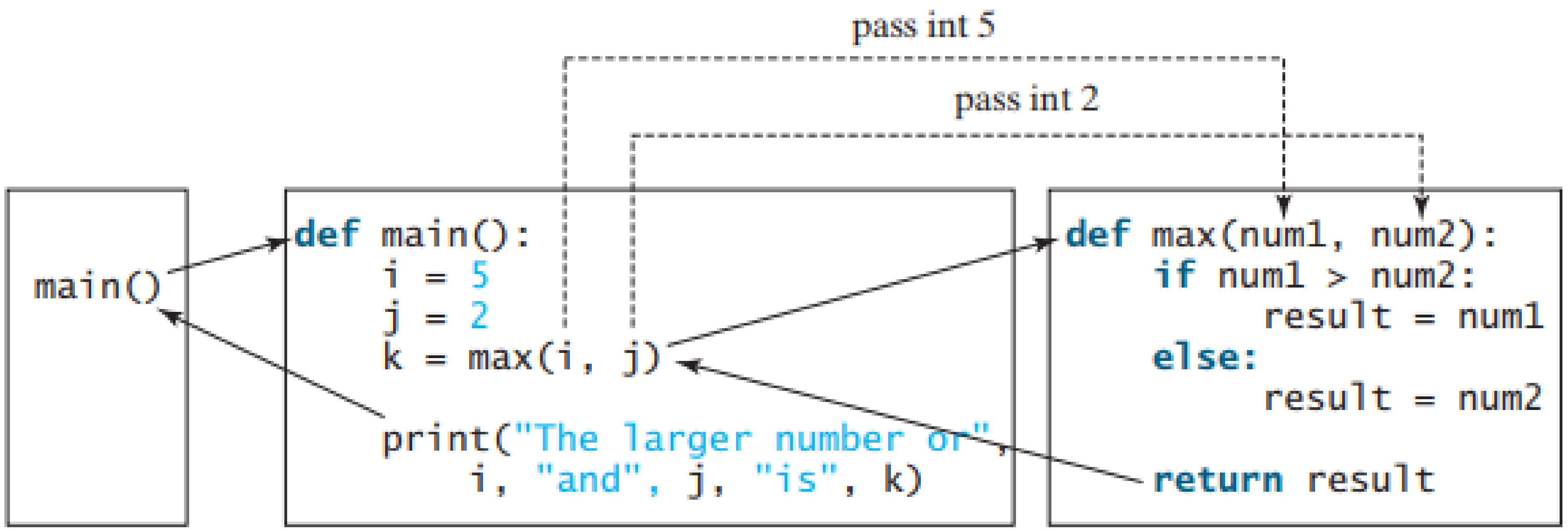
Calling a Function

- *Calling a function executes the code in the function*
- If the function returns a value, a call to that function is usually treated as a value. For example,
  larger = max(**3**, **4**)

calls **max(3, 4)** and assigns the result of the function to the variable **larger**.

- Another example of a call that is treated as a value is
  print(max(**3**, **4**))
  which prints the *return value* of the function call **max(3, 4)**.

Execution of any function



```
main()
```

```
def main():
    i = 5
    j = 2
    k = max(i, j)

    print("The larger number of",
        i, "and", j, "is", k)
```

```
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result
```

FIGURE 6.2 When a function is invoked, the control is transferred to the function. When the function is finished, the control is returned to where the function was called.

Call Stacks

- Each time a function is invoked, the system creates an *activation record* that stores its arguments and variables for the function and places the activation record in an area of memory known as a ***call stack***.
- A call stack is also known as an **execution stack, runtime stack, or machine stack,** and is often shortened to just "the stack."
- A call stack stores the activation records in a last-in, first-out fashion.
- Python creates and stores objects in a separate memory space called ***heap***.

Understanding Call Satcks

- When the **main** function is invoked, an activation record is created to store variables **i** and **j**, as shown in Figure 6.3a. Remember that all data in Python are objects. Variables **i** and **j** actually contain reference values to **int** objects **5** and **2**, as shown in Figure 6.3a. Invoking **max(i, j)** passes the values **i** and **j** to parameters **num1** and **num2** in the **max** function.
- Now **num1** and **num2** reference **int** objects **5** and **2**, as shown in Figure 6.3b. The **max** function finds the maximum number and assigns it to **result**, so **result** now references **int** object **5**, as shown in Figure 6.3c. The result is returned to the **main** function

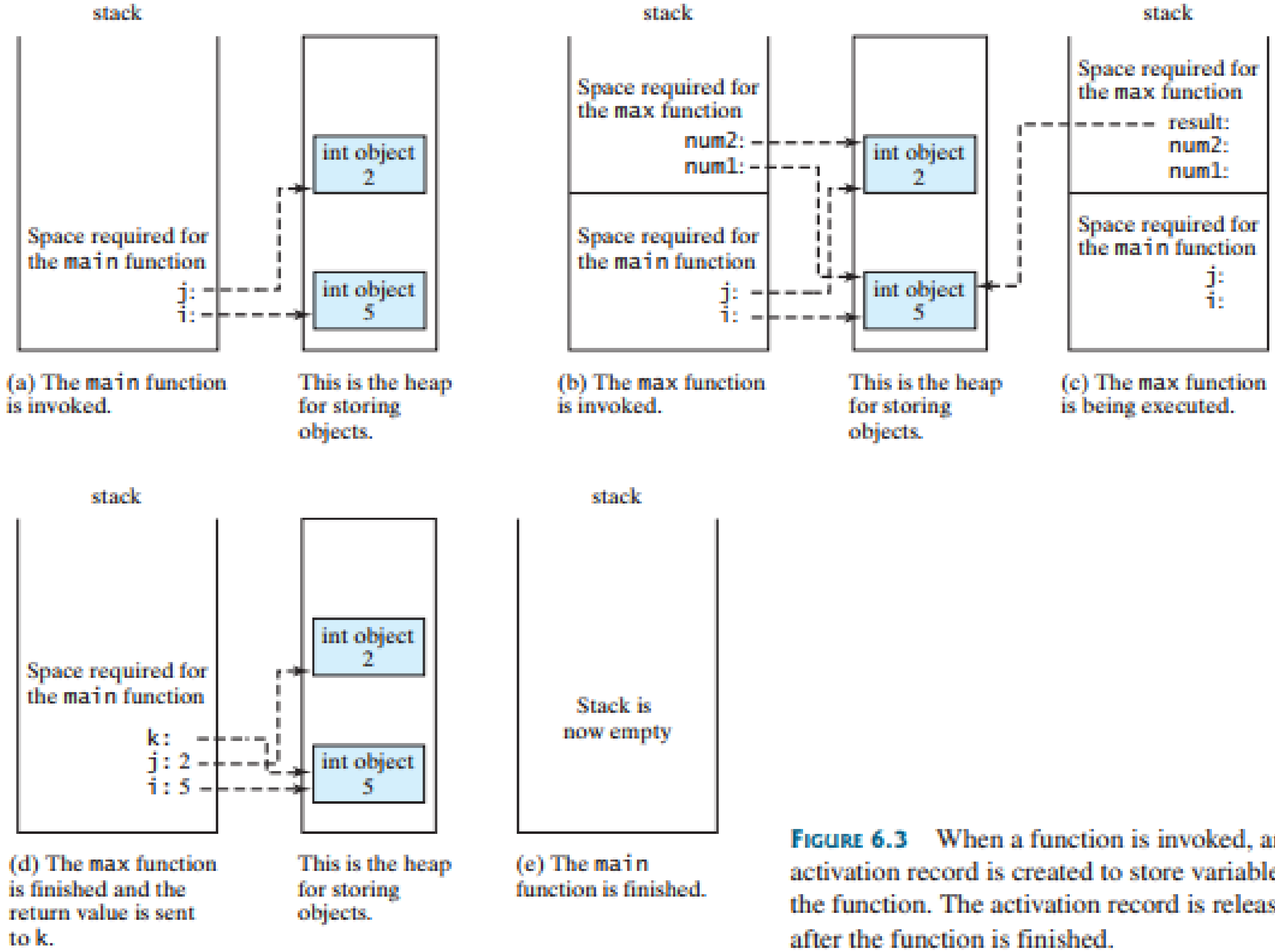

stack
Space required for the main function
j:
i:
int object 2
int object 5
(a) The main function is invoked.
This is the heap for storing objects.
stack
Space required for the max function
num2:
num1:
Space required for the main function
j:
i:
int object 2
int object 5
(b) The max function is invoked.
This is the heap for storing objects.
stack
Space required for the max function
result:
num2:
num1:
Space required for the main function
j:
i:
(c) The max function is being executed.
stack
Space required for the main function
k:
j: 2
i: 5
int object 2
int object 5
(d) The max function is finished and the return value is sent to k.
This is the heap for storing objects.
stack
Stack is now empty
(e) The main function is finished.

FIGURE 6.3 When a function is invoked, an activation record is created to store variables in the function. The activation record is released after the function is finished.

and assigned to variable k. Now k references int object 5, as shown in Figure 6.3d. After the main function is finished, the stack is empty, as shown in Figure 6.3e. The objects in the heap are automatically destroyed by the Python interpreter when they are no longer needed.

Why Use Functions?

- **Maximizing code reuse and minimizing redundancy**

Functions allow us to group and generalize code to be used arbitrarily many times later.

they allow us to reduce code redundancy in our programs, and thereby reduce maintenance effort.

- **Procedural decomposition:**

It’s easier to implement the smaller tasks in isolation than it is to implement the entire process at once.

NONE Function

- Technically, every function in Python returns a value whether you use **return** or not. If a function does not return a value, by default, it returns a special value **None**. For this reason, a function that does not return a value is also called a **None** *function*.
- The **None** value can be assigned to a variable to indicate that the variable does not reference any
  object.
- For example, if you run the following program:

```
def sum(number1, number2):
        total = number1 + number2
print(sum(1, 2))
```

you will see the output is **None**, because the **sum** function does not have a return statement. By default, it returns **None**.

return in None function

- A **return** statement is not needed for a **None** function, but it can be used for terminating the function and returning control to the function's caller. The syntax is
  simply
  return
  or
  return None

- For example, the following code has a
  return statement to terminate the function when the score is invalid.

```
# Print grade for the score
def printGrade(score):
    if score < 0 or score > 100:
        print("Invalid score")
    return
```

Same as return None

Scopes

- When you use a name in a program, Python creates, changes, or looks up the name in what is known as a *namespace*—a place where names live.
- *scope* refers to a namespace: that is, the location of a name’s assignment in your source code determines the scope of the name’s visibility to your code

variables may be assigned in three different places, corresponding to three different scopes:

- If a variable is assigned inside a def, it is *local* to that function.
- If a variable is assigned in an enclosing def, it is *nonlocal* to nested functions.
- If a variable is assigned outside all defs, it is *global* to the entire file.

Functions define a local scope and modules define a global scope with the following properties:

- The enclosing module is a global scope.
- The global scope spans a single file only.
- Assigned names are local unless declared global or nonlocal.
- All other names are enclosing function locals, globals, or built-ins.
- Each call to a function creates a new local scope.

Name Resolution: The LEGB Rule

- Python's name-resolution scheme is sometimes called the *LEGB rule*, after the scope names:
  • When you use an unqualified name inside a function, Python searches up to four scopes—the local (*L*) scope, then the local scopes of any enclosing (*E*) defs and lambdas, then the global (*G*) scope, and then the built-in (*B*) scope—and stops at the first place the name is found. If the name is not found during this search, Python reports an error.

Argument Matching Syntax

- *Positionals: matched from left to right*
- *Keywords: matched by argument name*
  (with the name=value syntax.)
- *Defaults:* Functions *specify values for optional arguments that aren't passed*
- *Varargs collecting: collect arbitrarily many positional or keyword arguments* Functions can use special arguments preceded with one or two * characters to collect an arbitrary number of possibly extra arguments.