Chapter 6 Functions

Opening Problem

Find the sum of integers from 1 to 10, from 20 to 37, and from 35 to 49, respectively.

Problem

```
sum = 0
for i in range(1, 11):
    sum += i
print("Sum from 1 to 10 is", sum)

sum = 0
for i in range(20, 38):
    sum += i
print("Sum from 20 to 37 is", sum)

sum = 0
for i in range(35, 50):
    sum += i
print("Sum from 35 to 49 is", sum)
```

Problem

```
sum = 0
for i in range(1, 11):
    sum += i
print("Sum from 1 to 10 is", sum)
```

```
sum = 0
for i in range(20, 38):
    sum += i
print("Sum from 20 to 37 is", sum)
```

```
sum = 0
for i in range(35, 50):
    sum += i
print("Sum from 35 to 49 is", sum)
```

Solution

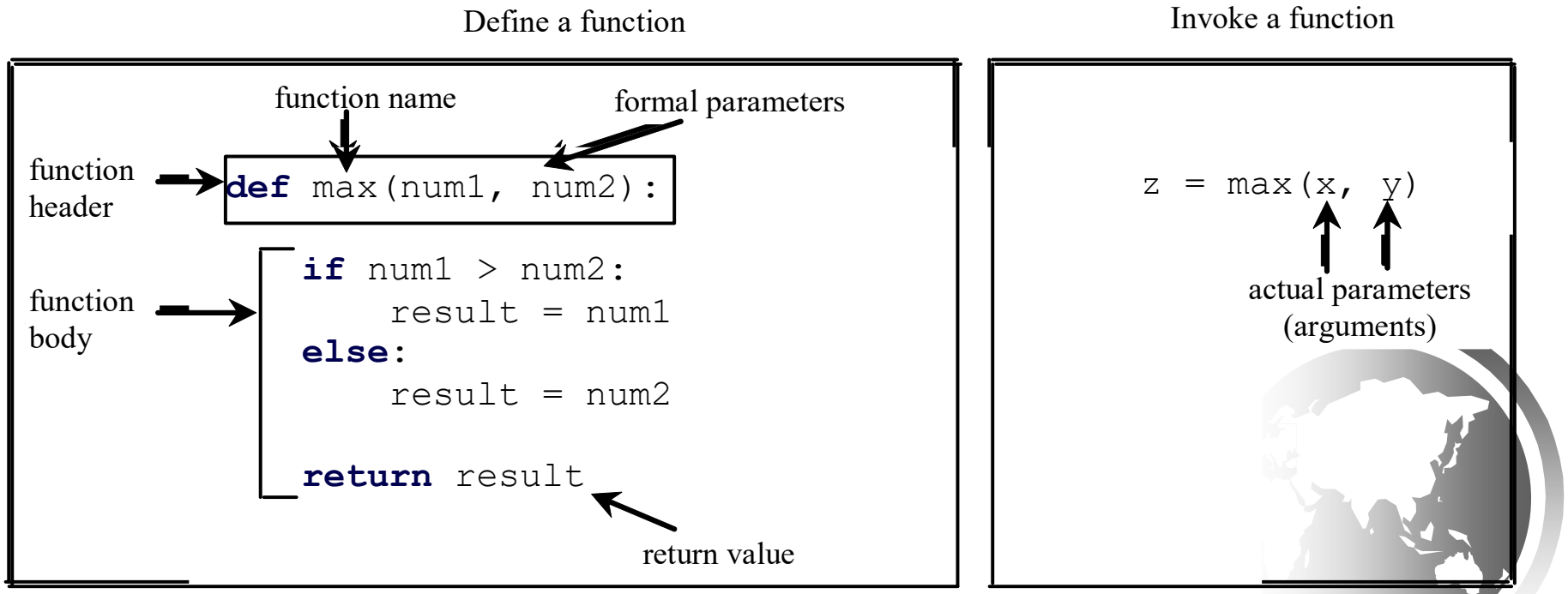
```
def sum(i1, i2):
    result = 0
    for i in range(i1, i2 + 1):
        result += i
    return result

def main():
    print("Sum from 1 to 10 is", sum(1, 10))
    print("Sum from 20 to 37 is", sum(20, 37))
    print("Sum from 35 to 49 is", sum(35, 49))

main() # Call the main function
```

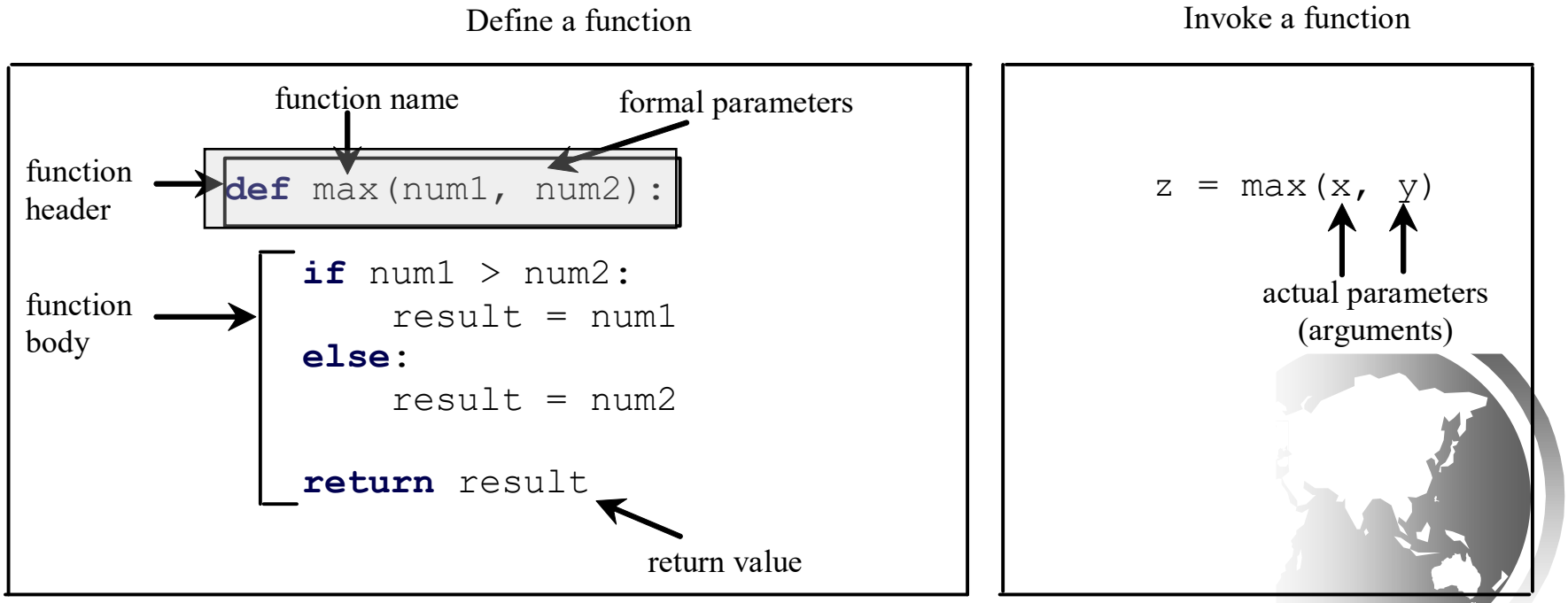
Defining Functions

A function is a collection of statements that are grouped together to perform an operation.



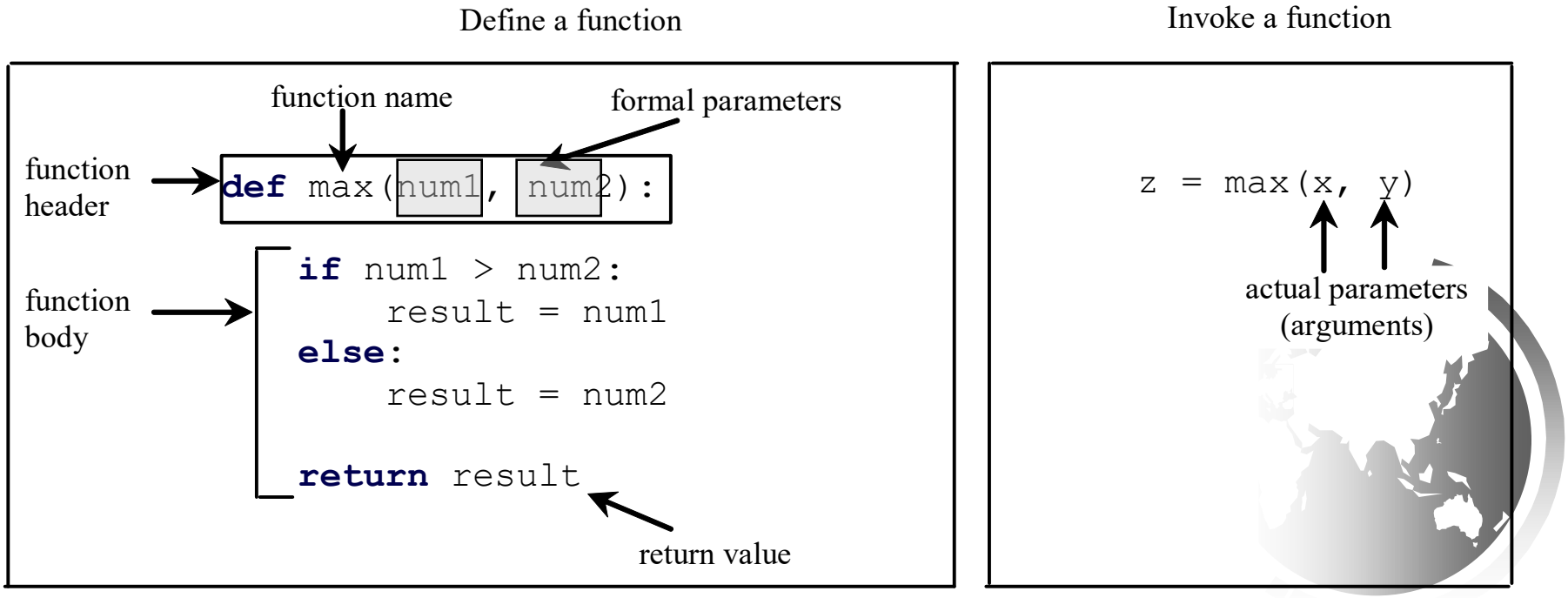
Function Header

A function contains a header and body. The header begins with the **def** keyword, followed by function's name and parameters, followed by a colon.



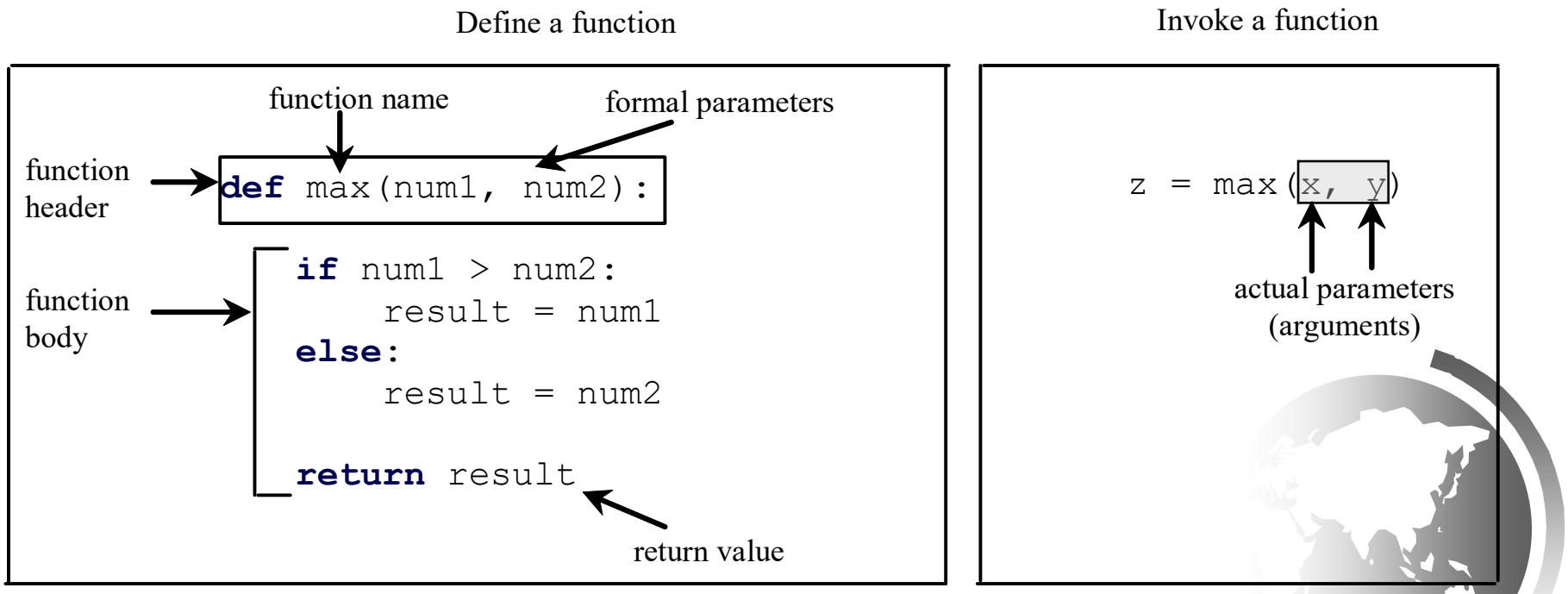
Formal Parameters

The variables defined in the function header are known as *formal parameters*.



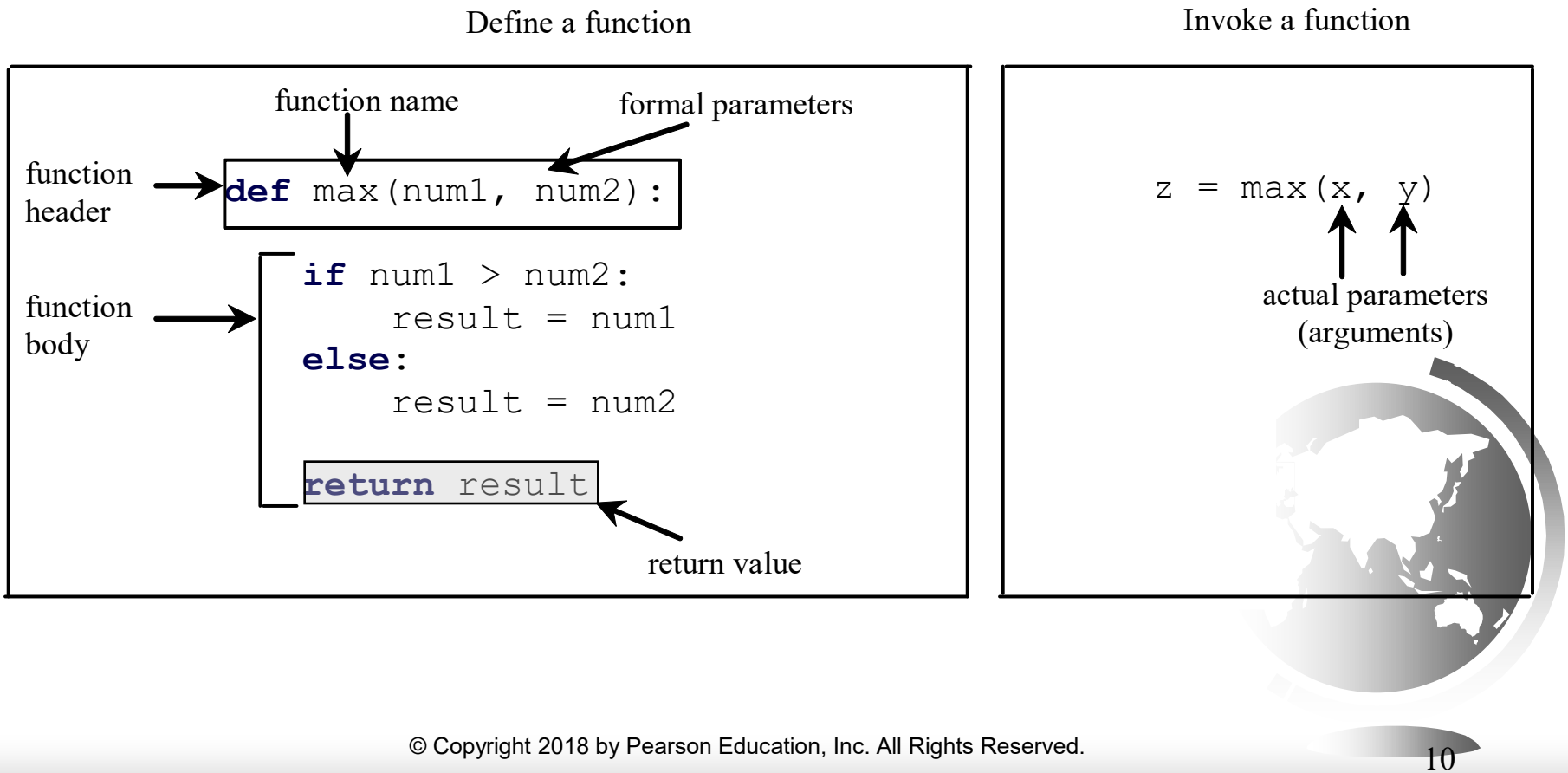
Actual Parameters

When a function is invoked, you pass a value to the parameter. This value is referred to as *actual parameter or argument*.



Return Value

A function may return a value using the return keyword.



Calling Functions

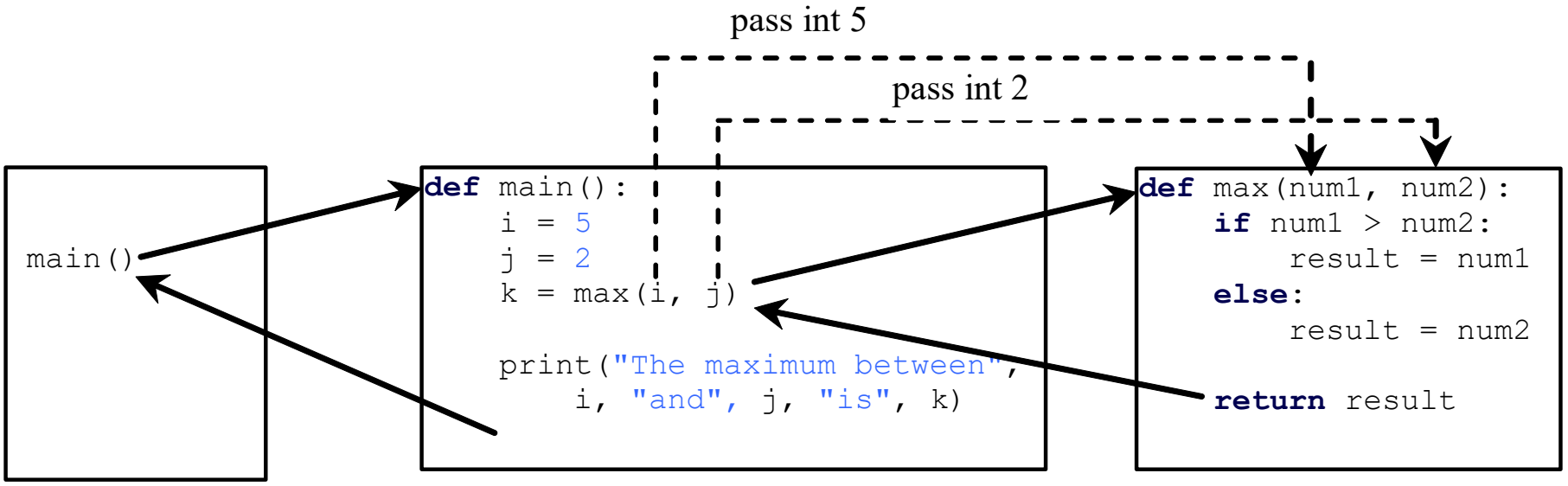
Testing the `max` function

This program demonstrates calling a function max to return the largest of the **`int`** values

TestMax

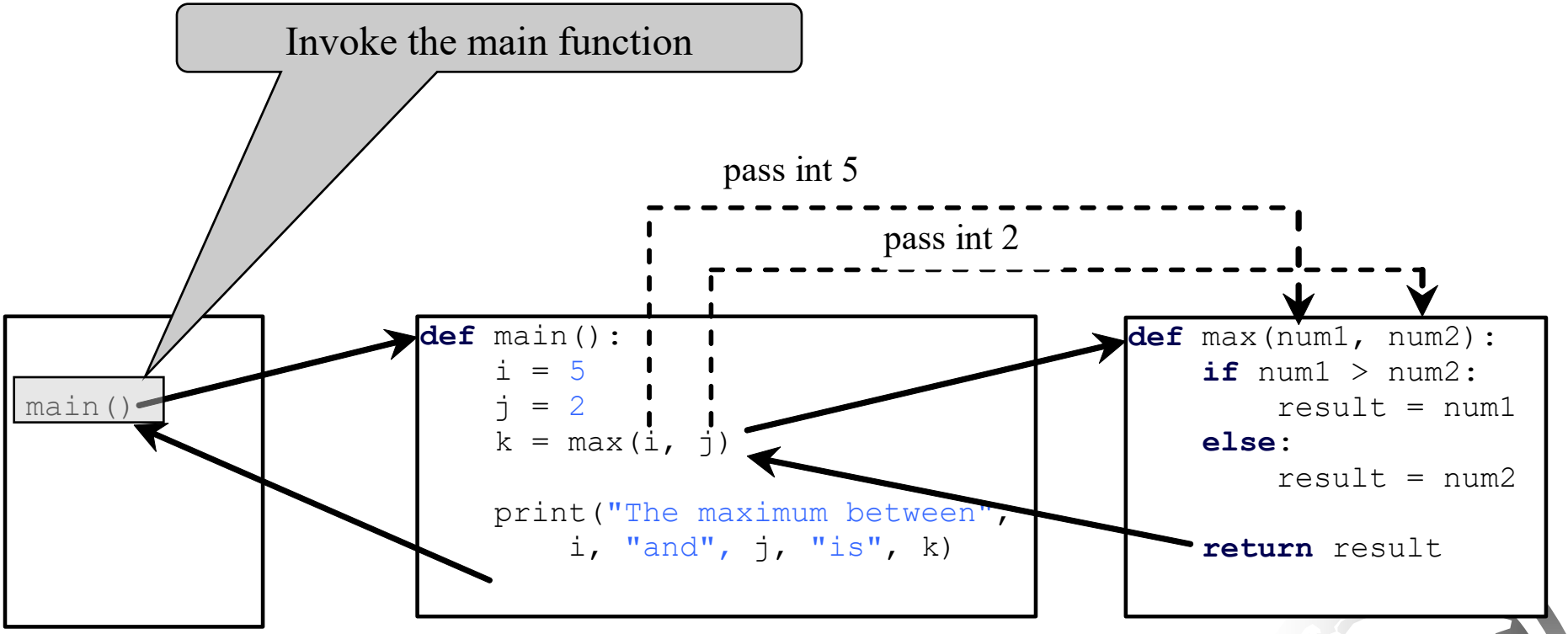
animation

Calling Functions, cont.



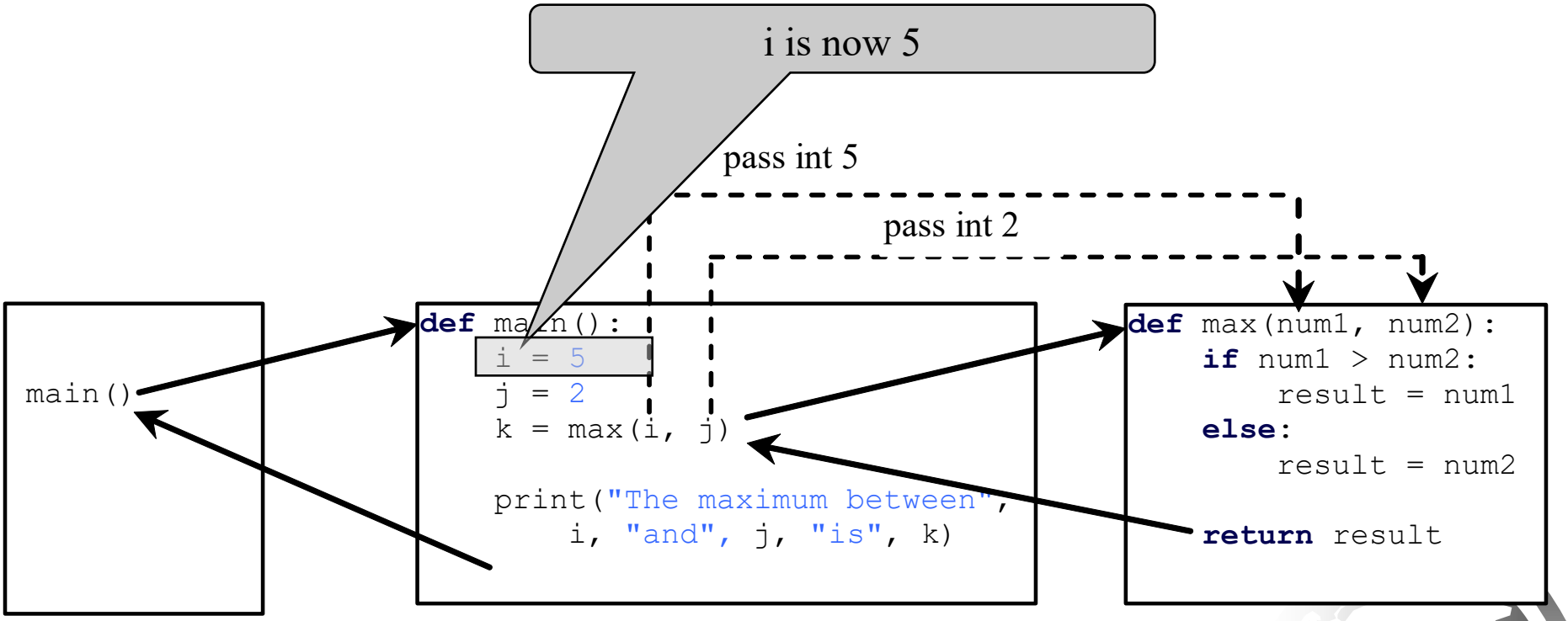
animation

Trace Function Invocation



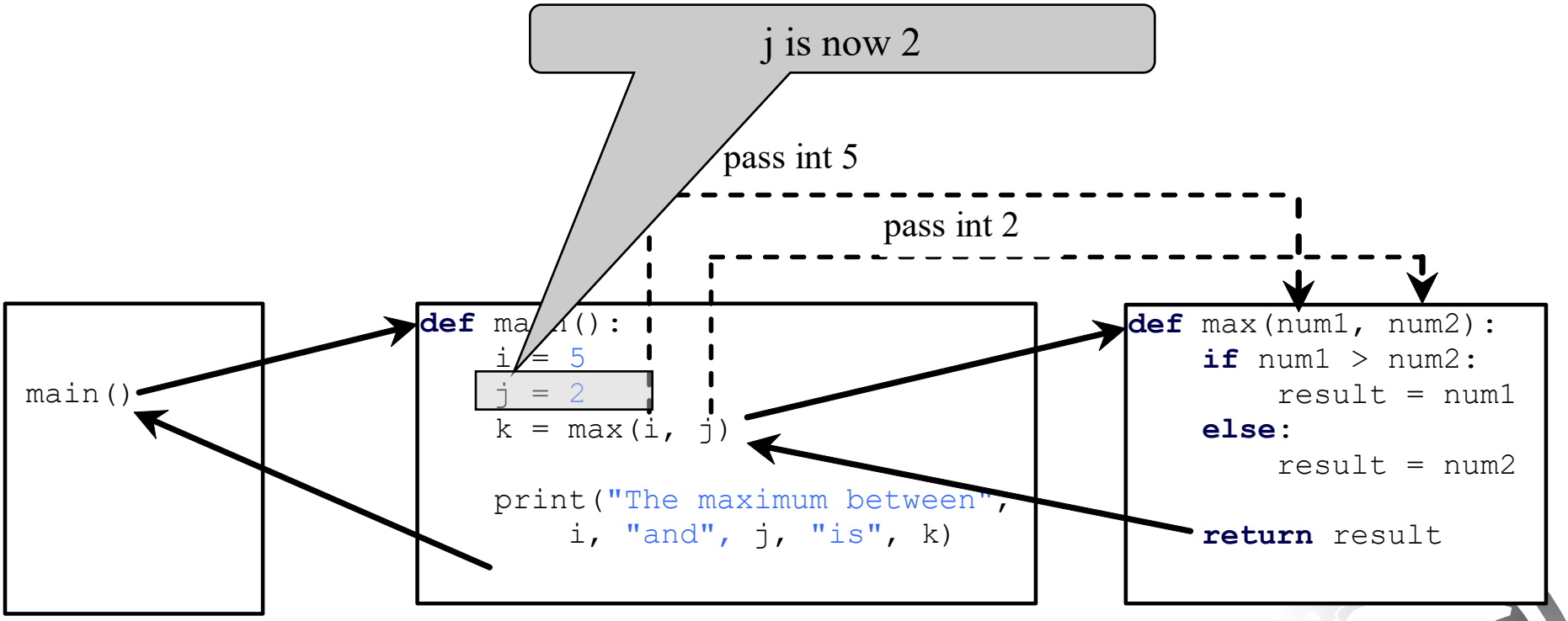
animation

Trace Function Invocation



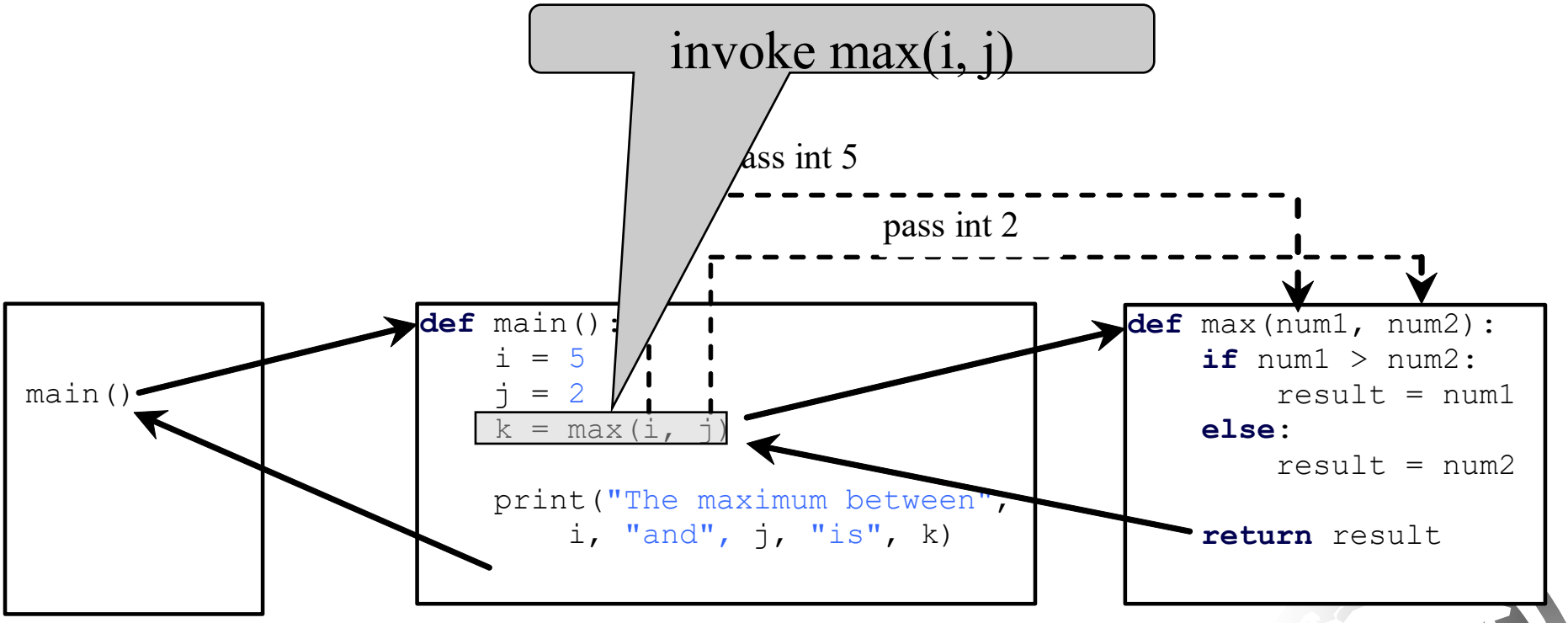
animation

Trace Function Invocation



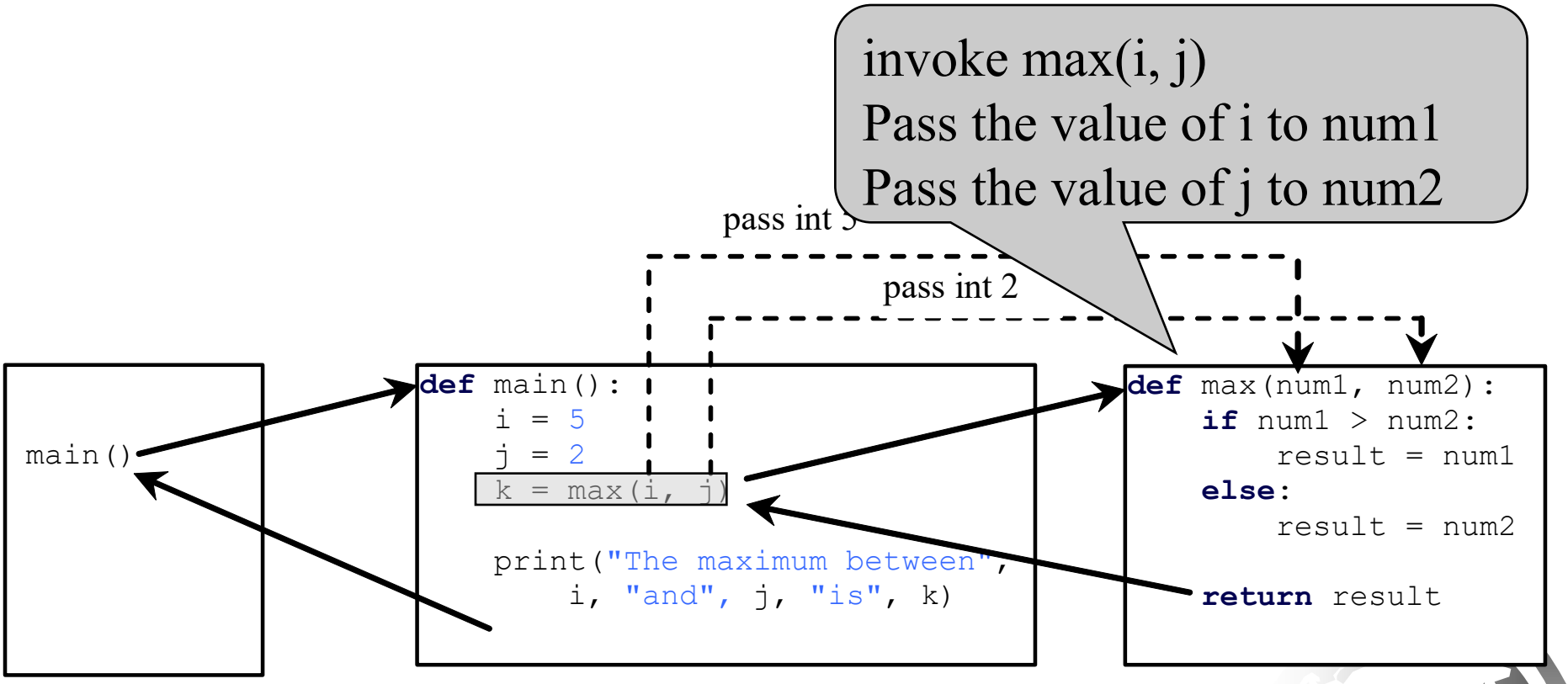
animation

Trace Function Invocation



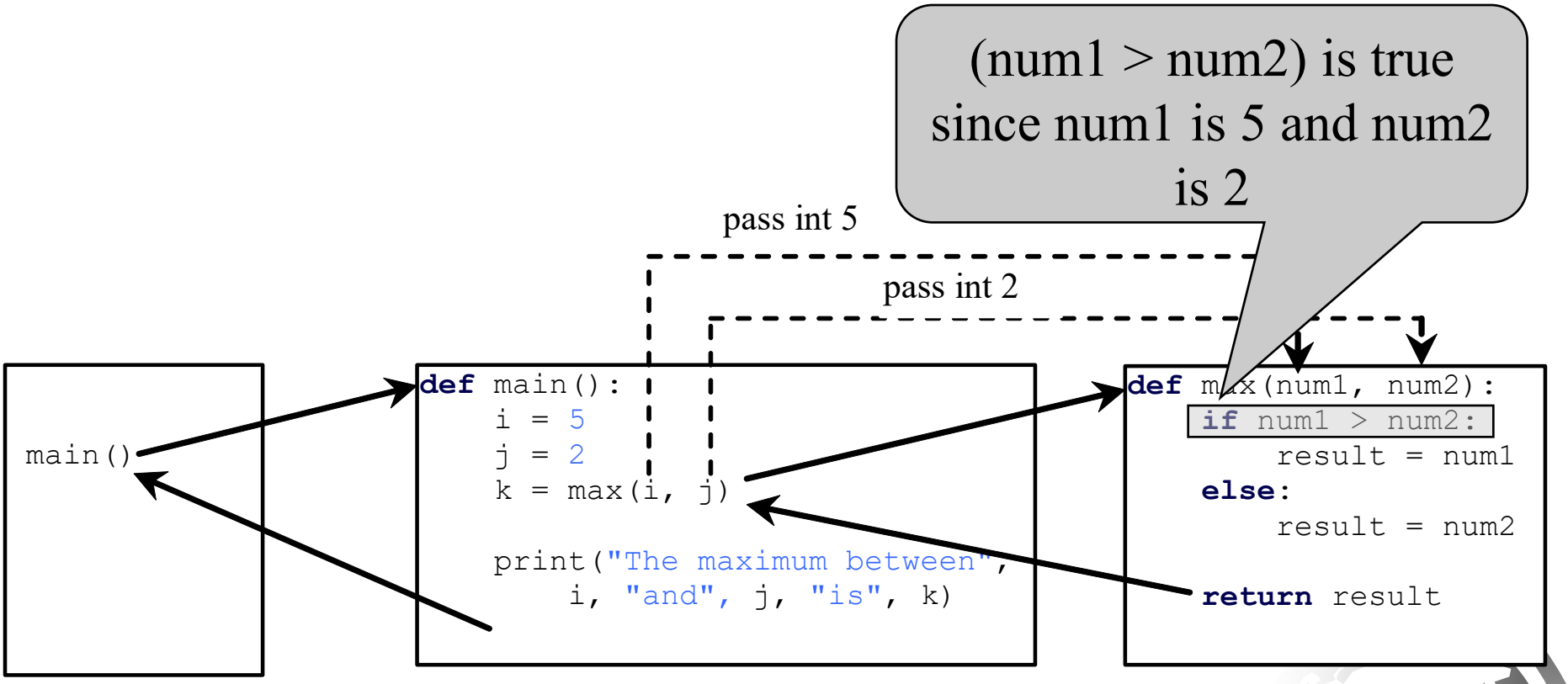
animation

Trace Function Invocation



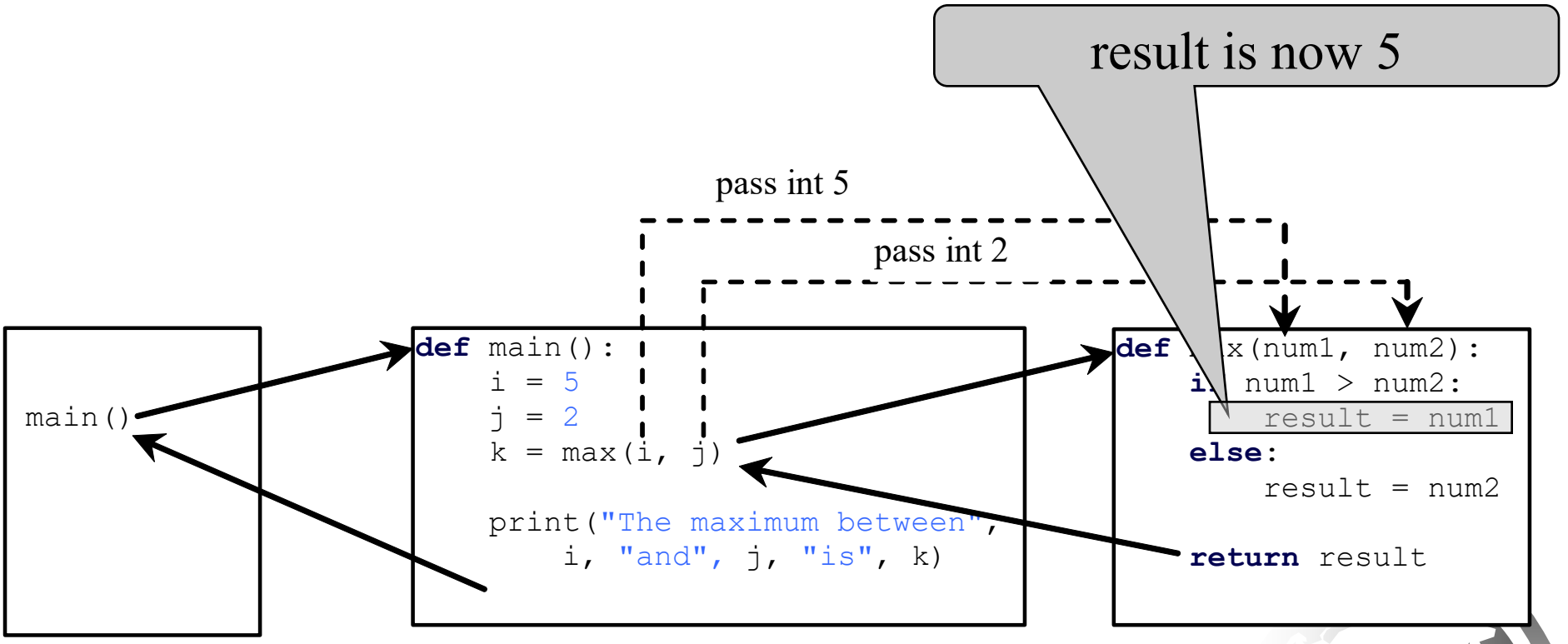
animation

Trace Function Invocation



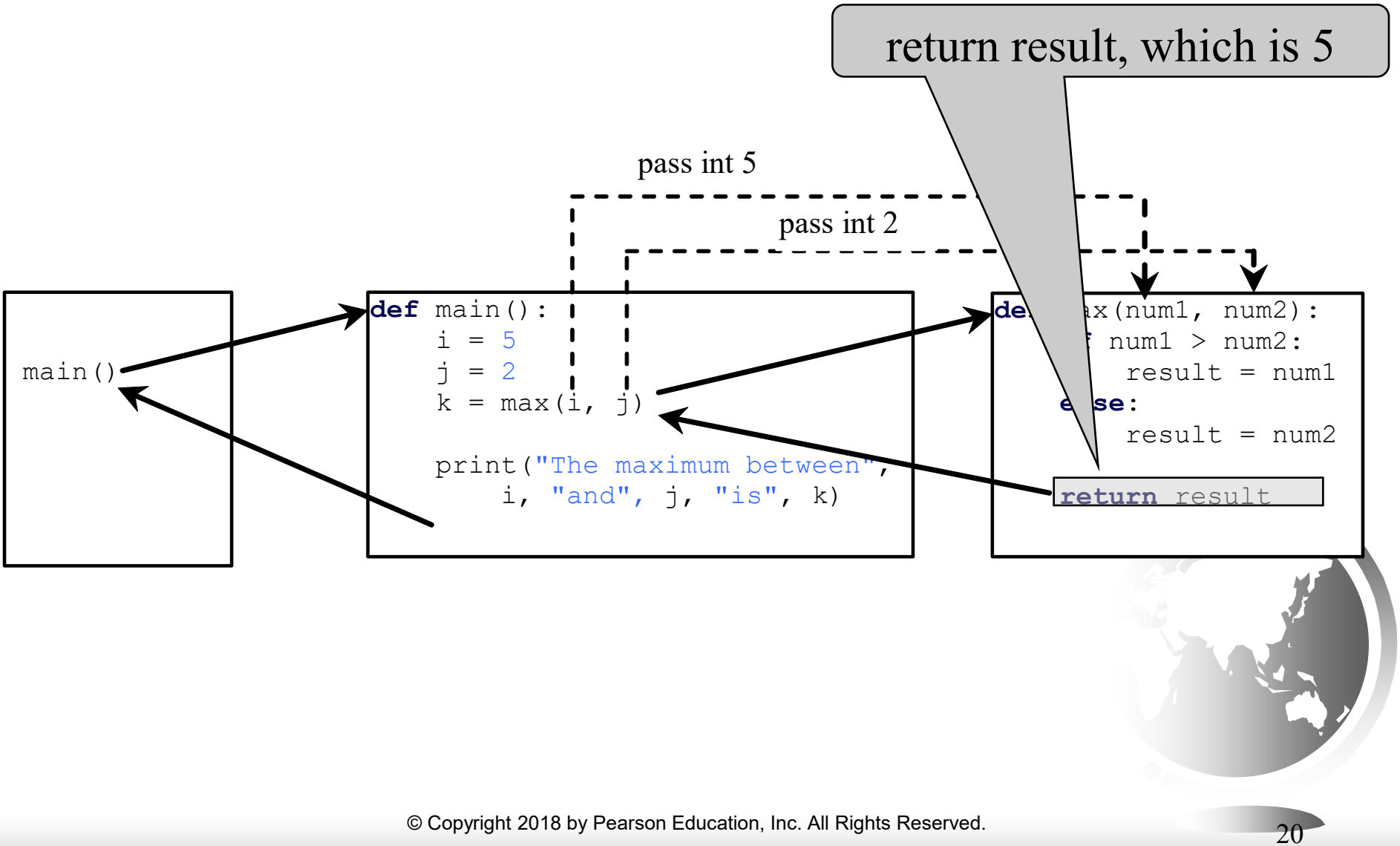
animation

Trace Function Invocation



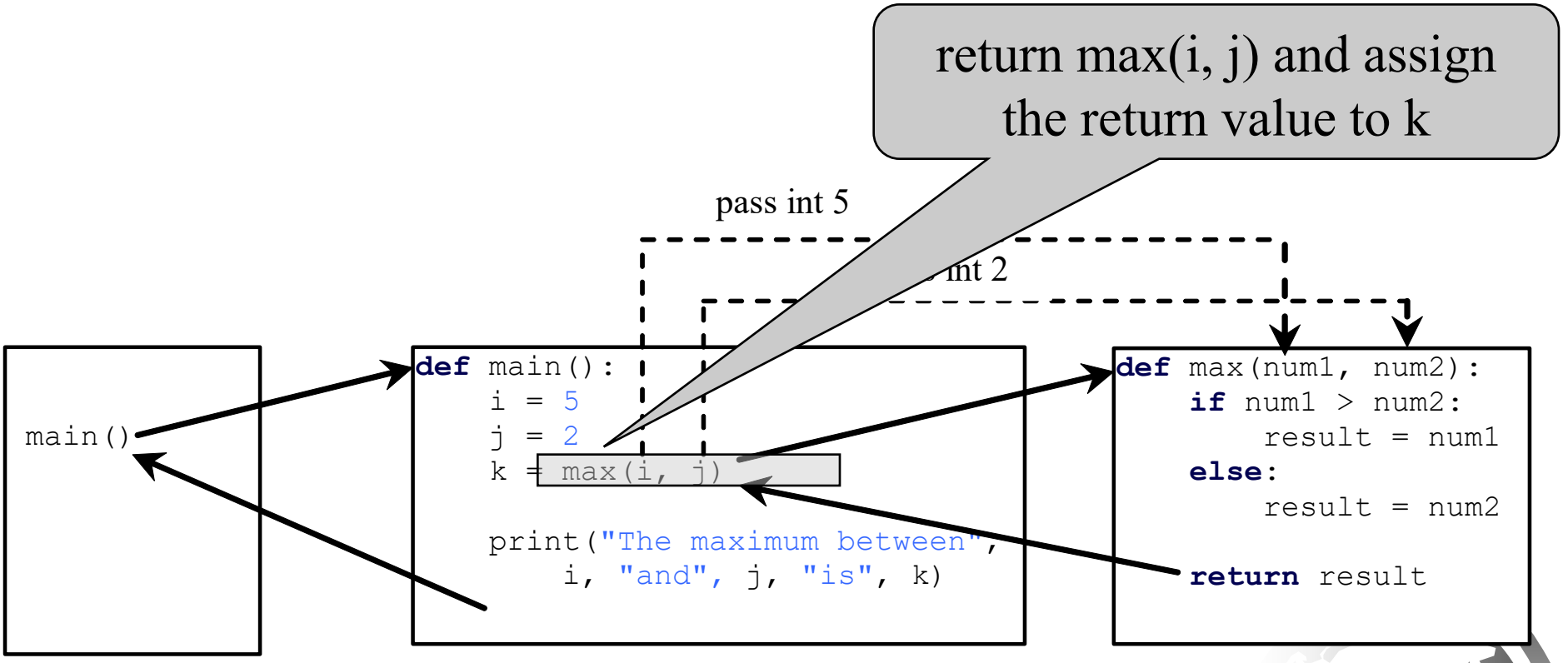
animation

Trace Function Invocation



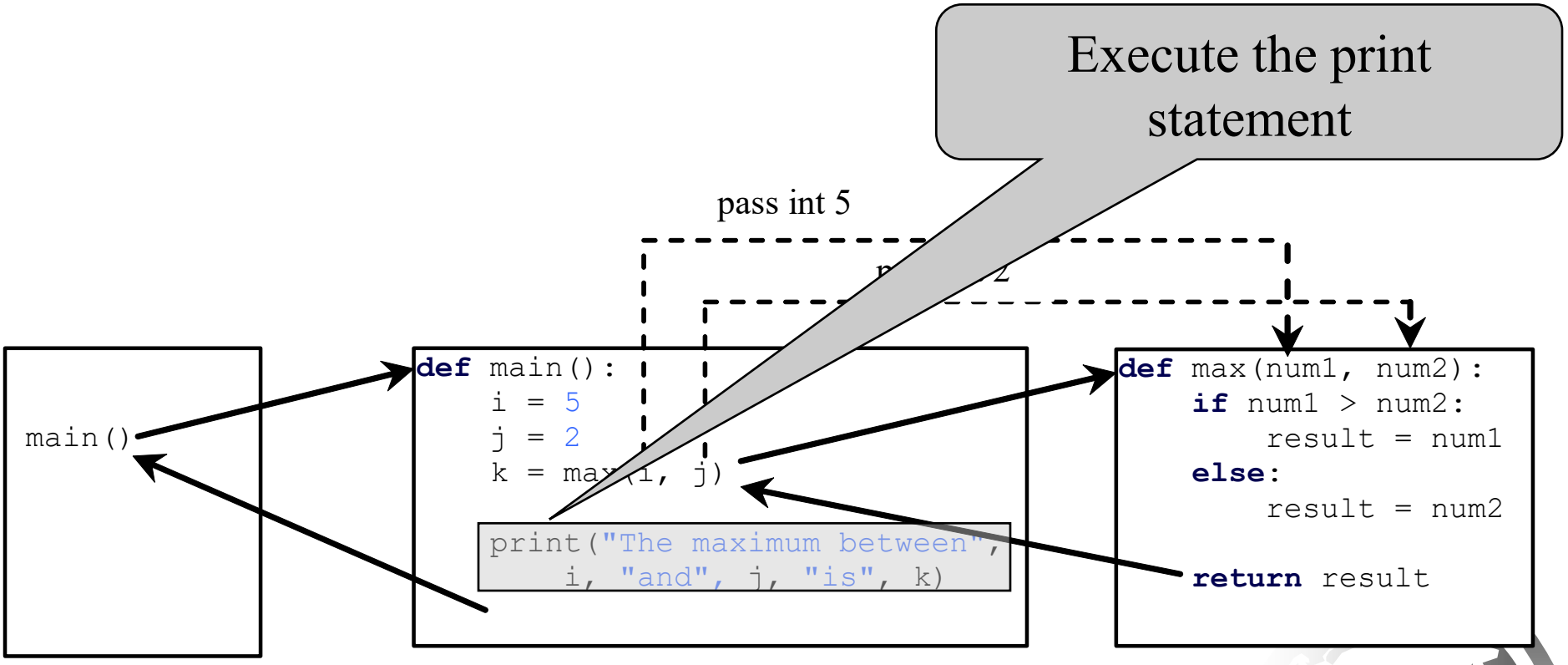
animation

Trace Function Invocation



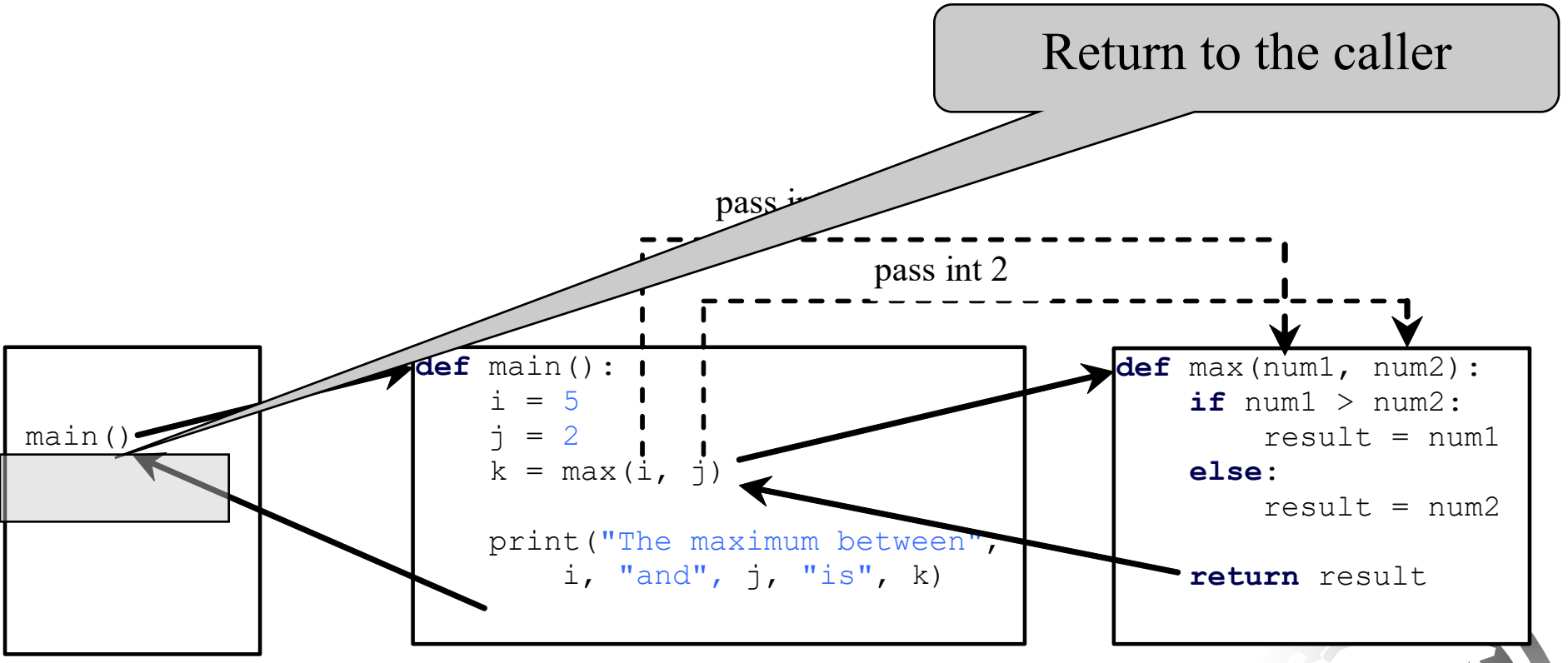
animation

Trace Function Invocation

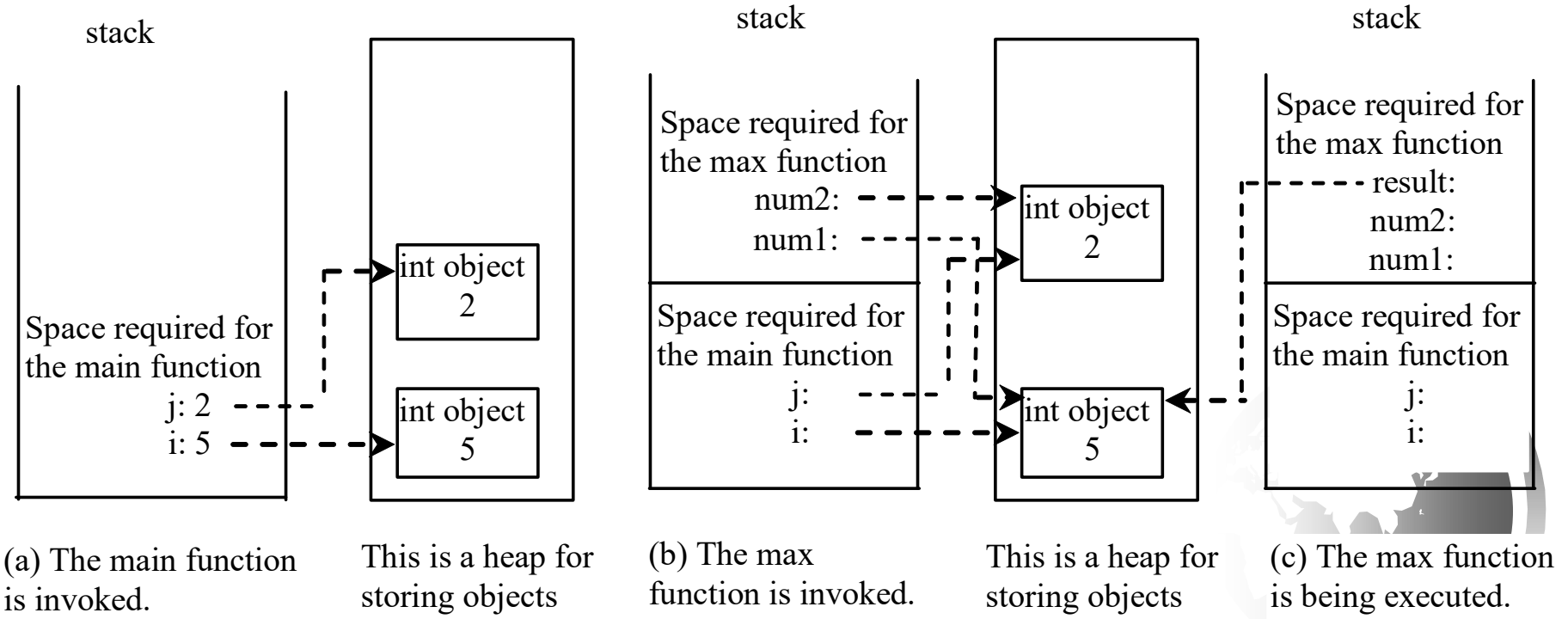


animation

Trace Function Invocation



Call Stacks



(a) The main function is invoked.

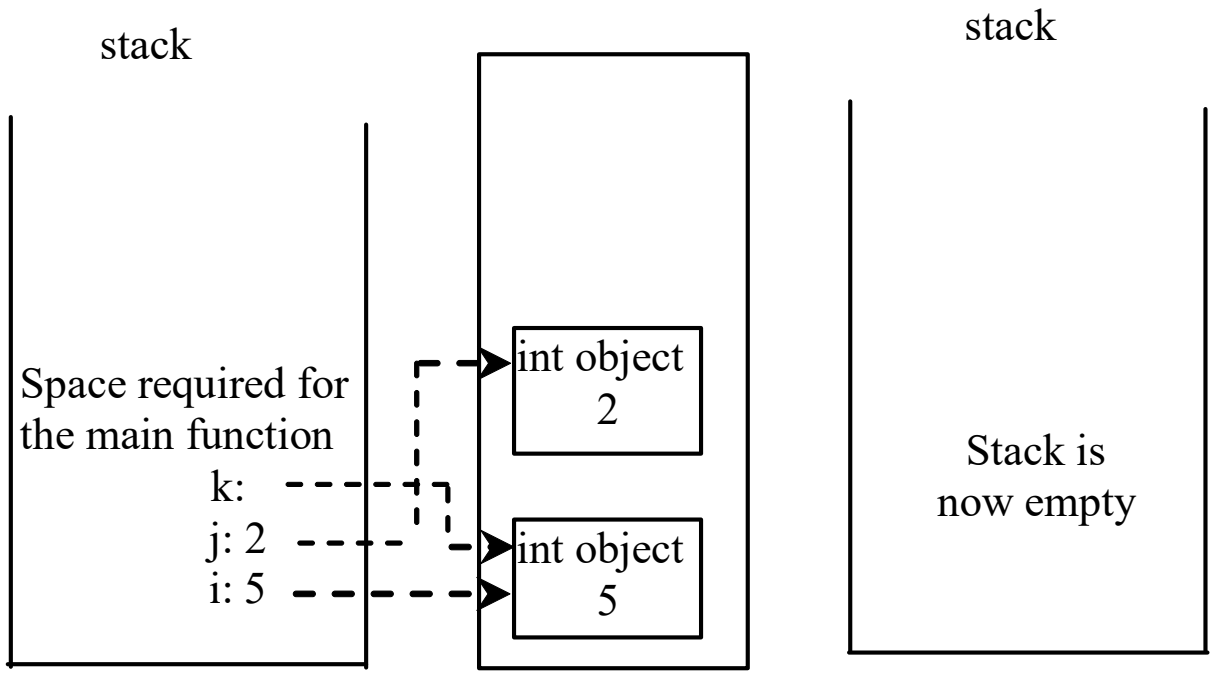
This is a heap for storing objects

(b) The max function is invoked.

This is a heap for storing objects

(c) The max function is being executed.

Call Stacks



(d) The max function is finished and the return value is sent to k.

This is a heap for storing objects

(e) The main function is finished.

Functions With/Without Return Values

This type of function does not return a value. The function performs some actions.

PrintGradeFunction

ReturnGradeFunction

The `None` value

A function that does not return a value is known as a *void* function in other programming languages such as Python, C++, and C#. In Python, such function returns a special value `None`.

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

Passing Arguments by Positions

```
def nPrintln(message, n):
    for i in range(0, n):
        print(message)
```

Suppose you invoke the function using

```
nPrintln("Welcome to Python", 5)
```

What is the output?

Suppose you invoke the function using

```
nPrintln("Computer Science", 15)
```

What is the output?

What is wrong

```
nPrintln(4, "Computer Science")
```

Keyword Arguments

```
def nPrintln(message, n):
    for i in range(0, n):
        print(message)
```

What is wrong

```
nPrintln(4, “Computer Science”)
```

Is this OK?

```
nPrintln(n = 4, message = “Computer Science”)
```

Pass by Value

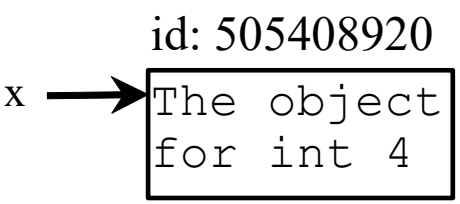
In Python, all data are objects. A variable for an object is actually a reference to the object. When you invoke a function with a parameter, the reference value of the argument is passed to the parameter. This is referred to as *pass-by-value*. For simplicity, we say that the value of an argument is passed to a parameter when invoking a function. Precisely, the value is actually a *reference value* to the object.

If the argument is a number or a string, the argument is not affected, regardless of the changes made to the parameter inside the function.

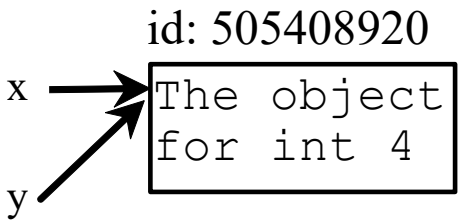
Increment

Pass by Value

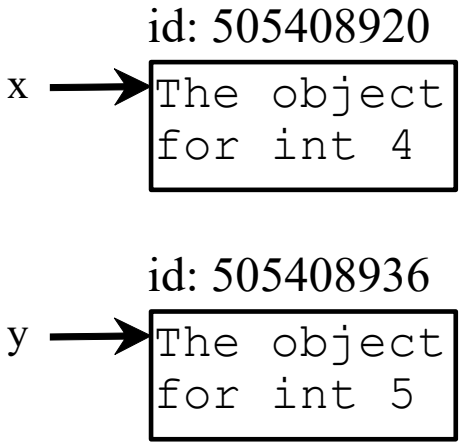
x = 4



y = x



y = y + 1



Merk at tall er immutable objects
Strenger er også immutable

Modularizing Code

Functions can be used to reduce redundant coding and enable code reuse. Functions can also be used to modularize code and improve the quality of the program.

GCDFunction

TestGCDFunction

PrimeNumberFunction

Problem: Converting Decimals to Hexadecimals

Write a function that converts a decimal integer to a hexadecimal.

Decimal2HexConversion

Scope of Variables

Scope: the part of the program where the variable can be referenced.

A variable created inside a function is referred to as a *local variable*. Local variables can only be accessed inside a function. The scope of a local variable starts from its creation and continues to the end of the function that contains the variable.

In Python, you can also use *global variables*. They are created outside all functions and are accessible to all functions in their scope.

Example 1

```
globalVar = 1
def f1():
    localVar = 2
    print(globalVar)
    print(localVar)
f1()
print(globalVar) # ok!
print(localVar)  # Out of scope. This gives an error
```

Example 2

```
x = 1 # not the same as x in f1
def f1():
    x = 2 # not the same as outer x
    print(x) # Displays 2
f1()
print(x) # Displays 1
```

Example 3

```
x = int(input("Enter a number: "))
if x > 0:
    y = 4
print(y) # This gives an error if y is not created
```

Example 4

```
sum = 0
for i in range(5):
    sum += i
print(i)# ok, unlike Java
```

Example 5

```
x = 1
def increase():
    global x
    x = x + 1
    print(x) # Displays 2
increase()
print(x) # Displays 2
```

Example 6

```
for foo in range(10):
    bar = 2
print(foo, bar)
```

The above will print (9,2).

Default and keyword arguments

Python allows you to define functions with default argument values. The default values are passed to the parameters when a function is invoked without the arguments.

In definition: non default cannot follow default.

DefaultArgumentDemo

Returning Multiple Values

Python allows a function to return multiple values. Listing 6.9 defines a function that takes two numbers and returns them in non-descending order.

MultipleReturnValueDemo

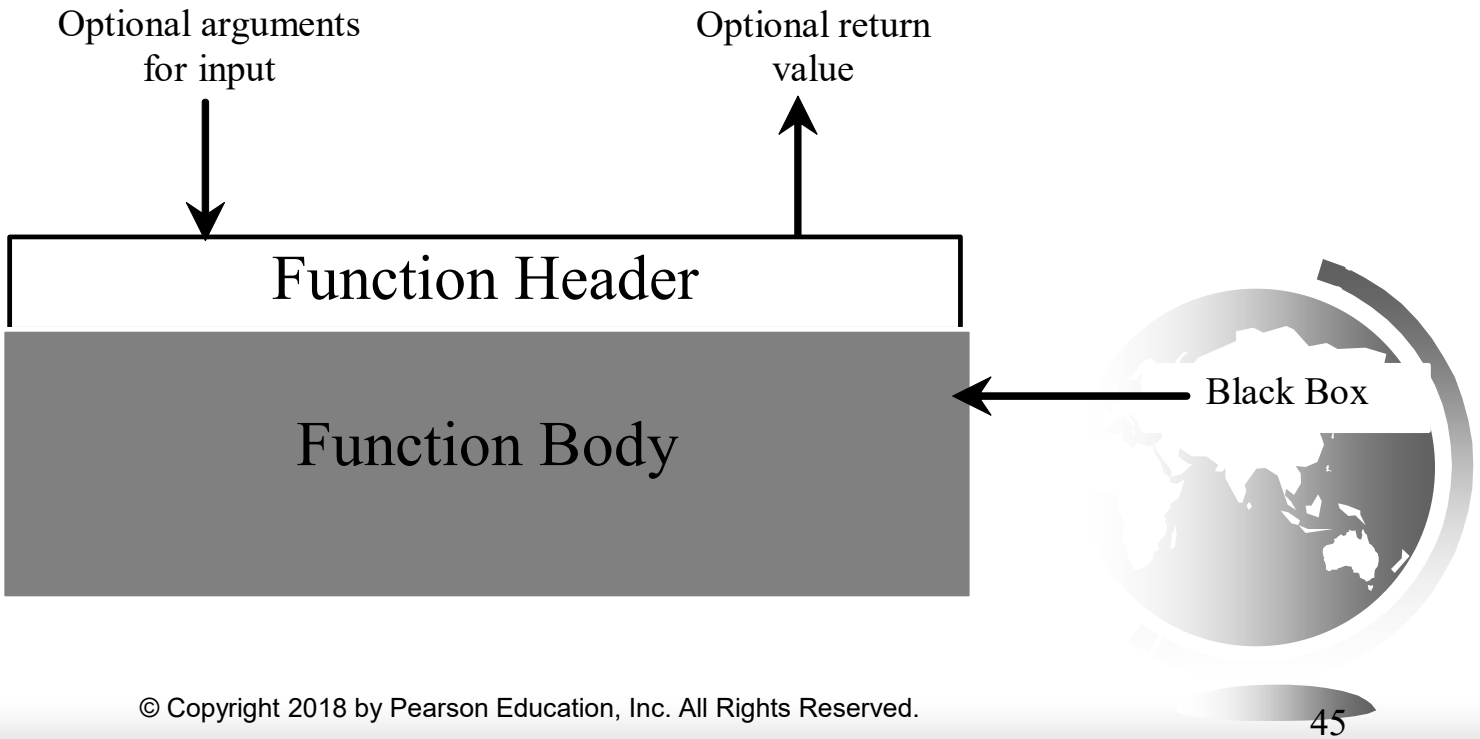
Generating Random Characters

RandomCharacter

TestRandomCharacter

Function Abstraction

You can think of the function body as a black box that contains the detailed implementation for the function.



Benefits of Functions

- Write a function once and reuse it anywhere.
- Information hiding. Hide the implementation from the user.
- Reduce complexity.

Stepwise Refinement

The concept of function abstraction can be applied to the process of developing programs. When writing a large program, you can use the "divide and conquer" strategy, also known as *stepwise refinement*, to decompose it into *subproblems*. The subproblems can be further decomposed into smaller, more manageable problems.

PrintCalender Case Study

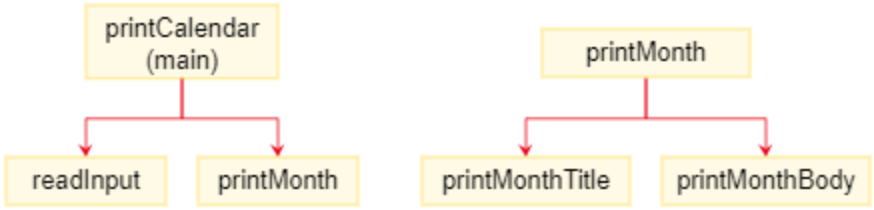
Let us use the PrintCalendar example to demonstrate the stepwise refinement approach.

Suppose you write a program that displays the calendar for a given month of the year. The program prompts the user to enter the year and the month, and then it displays the entire calendar for the month, as shown in the following sample run:

```
Enter full year (e.g., 2001): 2022
Enter month as number between 1 and 12: 9
          September   2022
------------------------------
 Sun Mon Tue Wed Thu Fri Sat
                   1   2   3
   4   5   6   7   8   9  10
  11  12  13  14  15  16  17
  18  19  20  21  22  23  24
  25  26  27  28  29  30
```

Grov skisse

1. Les input (år og måned som tall)
2. Skriv måned
    1. Skriv *måned header* (måned navn og navn på dagene: Sun, Mon, Tue etc)
        - Må ta et valg: begynne søndag eller mandag?
    2. Skriv selve måneden med uker og ukedager
        - Starte på hvilken ukedag?
        - Skrive ut ukenummer? (valg = NEI)
        - Hvor mange dager er det i denne måneden?



Hvordan starte på rett dag?

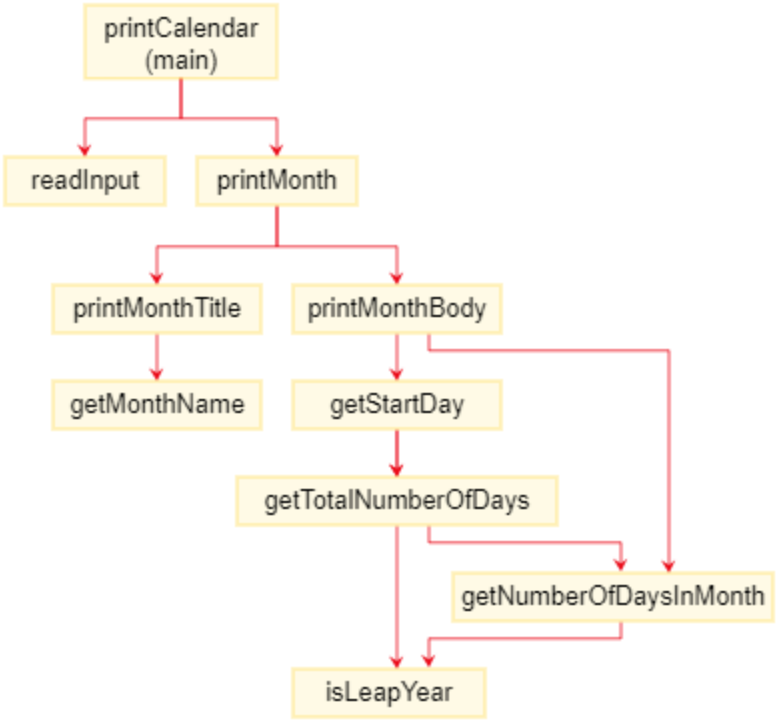
- Trenger å vite hvilken ukedag en gitt dato var på
  - Vet at den 1. januar 1800 var på en onsdag
    - Søn = 0, da er onsdag 3
  - Trenger å beregne antall dager..
    - For hvert år fra 1800 opp til aktuelt år -1
    - For hver måned fram til aktuell måned – 1
    - Legge til 3
    - Modulo 7 på dette tallet gir riktig startdato

The `isLeapYear(year)` function can be implemented using the following code (see **Section 3.11**):

```
return year % 400 == 0 or (year % 4 == 0 and year % 100 != 0)
```

Use the following facts to implement **`getTotalNumberOfDaysInMonth(year, month)`**:

- January, March, May, July, August, October, and December have 31 days.
- April, June, September, and November have 30 days.
- February has 28 days during a regular year and 29 days during a leap year. A regular year, therefore, has 365 days, and a leap year has 366 days.



Implementation: Top-Down

Top-down approach is to implement one function in the structure chart at a time from the top to the bottom. Stubs can be used for the functions waiting to be implemented. A stub is a simple but incomplete version of a function. The use of stubs enables you to test invoking the function from a caller. Implement the main function first and then use a stub for the printMonth function. For example, let printMonth display the year and the month in the stub. Thus, your program may begin like this:

A Skeleton for PrintCalendar

Implementation: Bottom-Up

Bottom-up approach is to implement one function in the structure chart at a time from the bottom to the top. For each function implemented, write a test program to test it. Both top-down and bottom-up functions are fine. Both approaches implement the functions incrementally and help to isolate programming errors and makes debugging easy. Sometimes, they can be used together.

Funksjoner er «first class objects»

First class objects er objekter som..

1. **Kan tilordnes til en variabel**: Du kan tilordne funksjoner eller objekter til variabler.
2. **Kan sendes som argument til en funksjon**: Du kan sende funksjoner eller objekter som argumenter til andre funksjoner.
3. **Kan returneres fra en funksjon**: En funksjon kan returnere andre funksjoner eller objekter.
4. **Kan lagres i datastrukturer**: Funksjoner eller objekter kan lagres i datastrukturer som lister, tupler og dictionaries