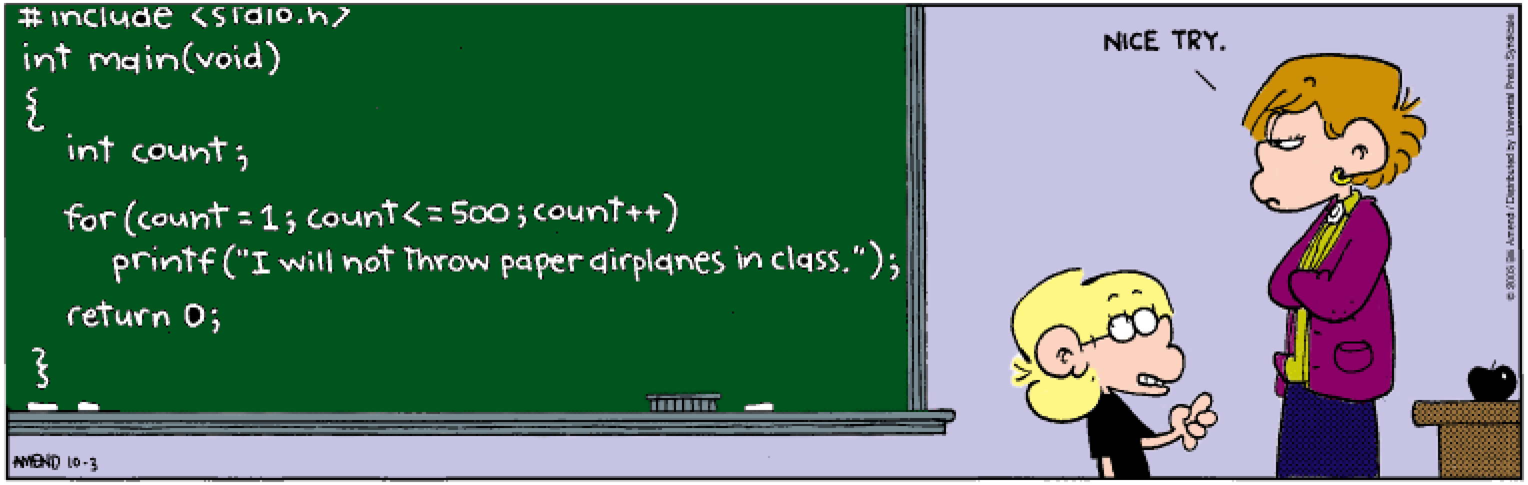
PYTHON BOOT CAMP

Module 6: Functions

CS Jokes


#include <stdio.h>
int main(void)
{
int count;
for (count = 1; count<=500; count++)
printf("I will not throw paper airplanes in class.");
return 0;
}
NICE TRY.
AMEND 10-3

Program 1: Sum Numbers

- Write a program that will sum three sets of numbers and then display the sum of each:
  - sum of integers from 1 to 10
  - sum of integers from 20 to 37
  - sum of integers from 35 to 49
- Remember:
  - Step 1: Problem-solving Phase
  - Step 2: Implementation Phase

Program 1: Sum Numbers

- Step 1: Problem-solving Phase
  - Algorithm:
    - This program is really easy
    - For each set of numbers:
      - make a variable sum
      - make a for loop and sum from the first number to the second number
      - print the final sum
    - So this is very easy to do
    - Expected Output:

```
>>> %Run sumnumbers_threetimes.py
 Sum from 1 to 10 is 55.
 Sum from 20 to 37 is 513.
 Sum from 35 to 49 is 630.
```

    - Go ahead and code this up…

Program 1: Sum Numbers

- Step 2: Implementation Phase

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

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

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

  - This works just fine…but what's the problem?
  - We are repeating the same code three times!

Program 1: Sum Numbers

- Observation
  - Each sum is doing something very similar
  - In fact, each sum is essentially doing the same thing
  - The only difference is the range of numbers
    - the starting and ending numbers of the sum
  - So **why** do we *__repeat__* our code three times?
  - Wouldn't it be nice if we could write "common" code and then **reuse** it when needed?
    - That would be PERFECT!
  - This is the idea of **functions**!

Program 1: Sum Numbers

- Step 2: Implementation

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

print("Sum from 1 to 10 is ", compute_sum(1, 10), ".", sep = "")
print("Sum from 20 to 37 is ", compute_sum(20, 37), ".", sep = "")
print("Sum from 35 to 49 is ", compute_sum(35, 49), ".", sep = "")
```

  - Here, we write a function to calculate the sum
    - And then, inside main, we call/invoke the function three times
  - You don't need to understand this perfectly right now
  - We will spend the next week or so understanding it!

Introduction

- What is a function?
  - A function is a collection of statements grouped together to perform an operation.
  - Guess what?
    - You've already used something kinda similar!
    - `random.randint(a, b)` or `eval(something here)`
      - These are predefined methods.
      - Methods are similar to functions in the way the work
        - Specifically, methods are connected to objects
        - …and functions are independent
        - but the idea is the same
    - In this chapter, we'll learn how to define our own functions and return the results from them
    - We'll also apply function abstraction to solve complex problems!

Chapter Objectives

- To define functions (§6.2).
- To invoke value-returning functions (§6.3).
- To invoke functions that does not return a value (§6.4).
- To pass arguments by values (§6.5).
- To pass arguments by values (§6.6).
- To develop reusable code that is modular, easy to read, easy to debug, and easy to maintain (§6.7).
- To create modules for reusing functions (§§6.7-6.8).
- To determine the scope of variables (§6.9).
- To define functions with default arguments (§6.10).
- To return multiple values from a function (§6.11).
- To apply the concept of function abstraction in software development (§6.12).
- To design and implement functions using stepwise refinement (§6.13).

Defining Functions

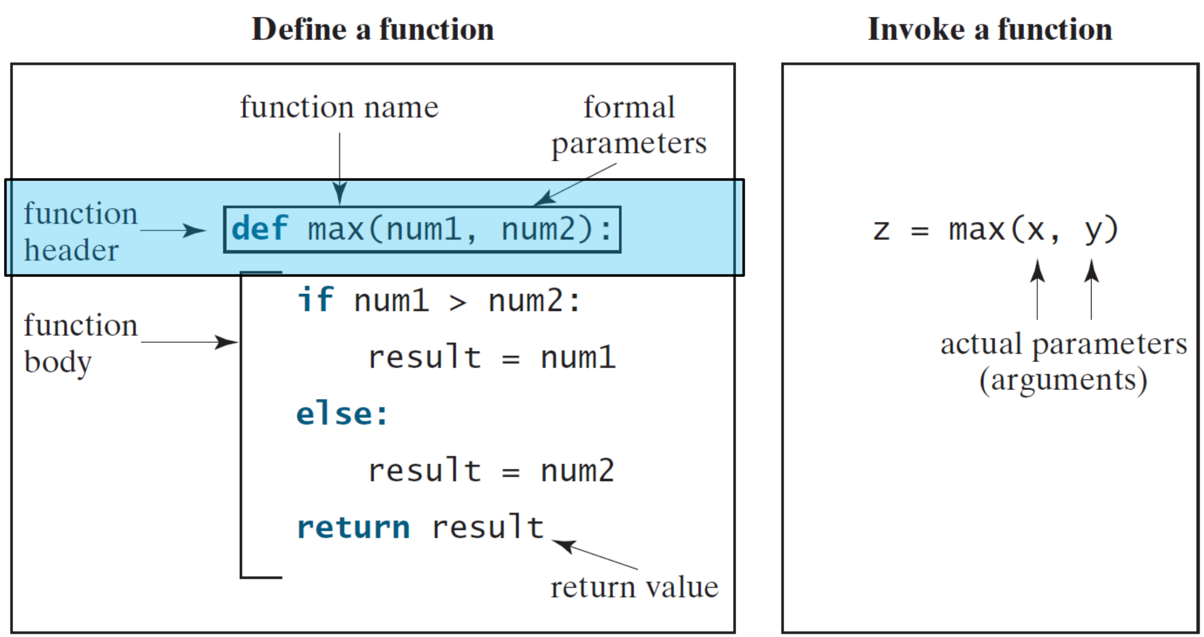
- What is a function?
  - A function is a collection of statements grouped together to perform an operation.
  - A function definition consists of:
    - The function's name
    - The parameters of the function
    - The body of the function
  - Syntax:

```
def functionName(list of parameters)
    # Function body
```

  - To understand the anatomy of a function, we start with a simple example: find the maximum of two numbers…

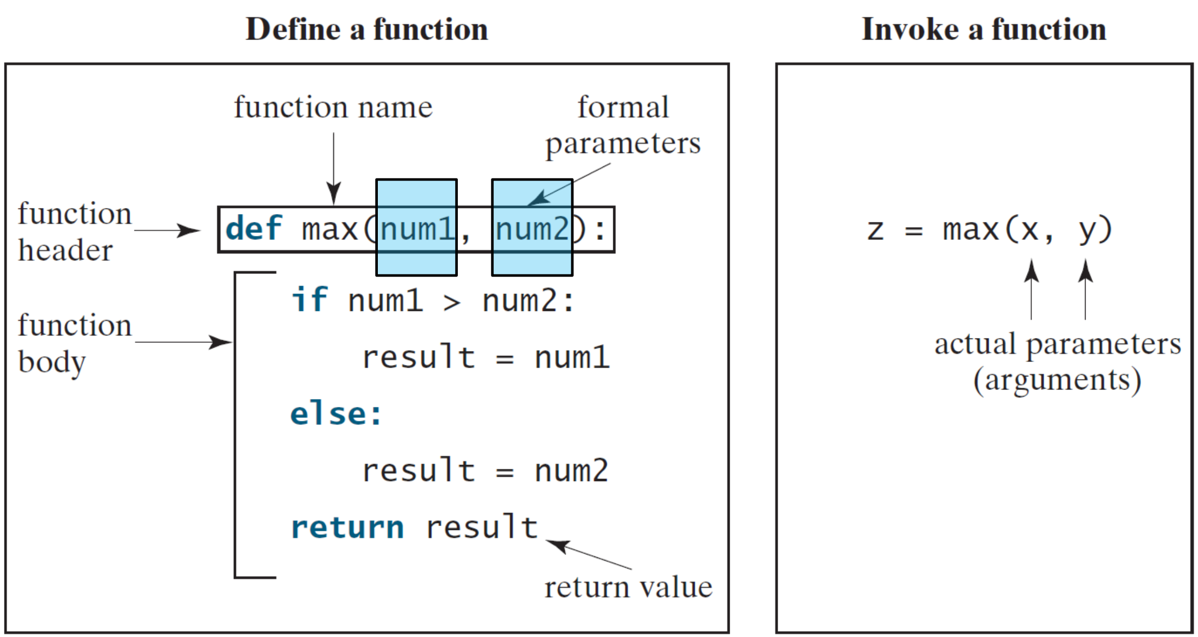
Defining Functions

- Anatomy of Sample Function:
  - **Function Header**:
    - Begins with the `def` keyword, followed by the function's name and parameters, followed by a colon.



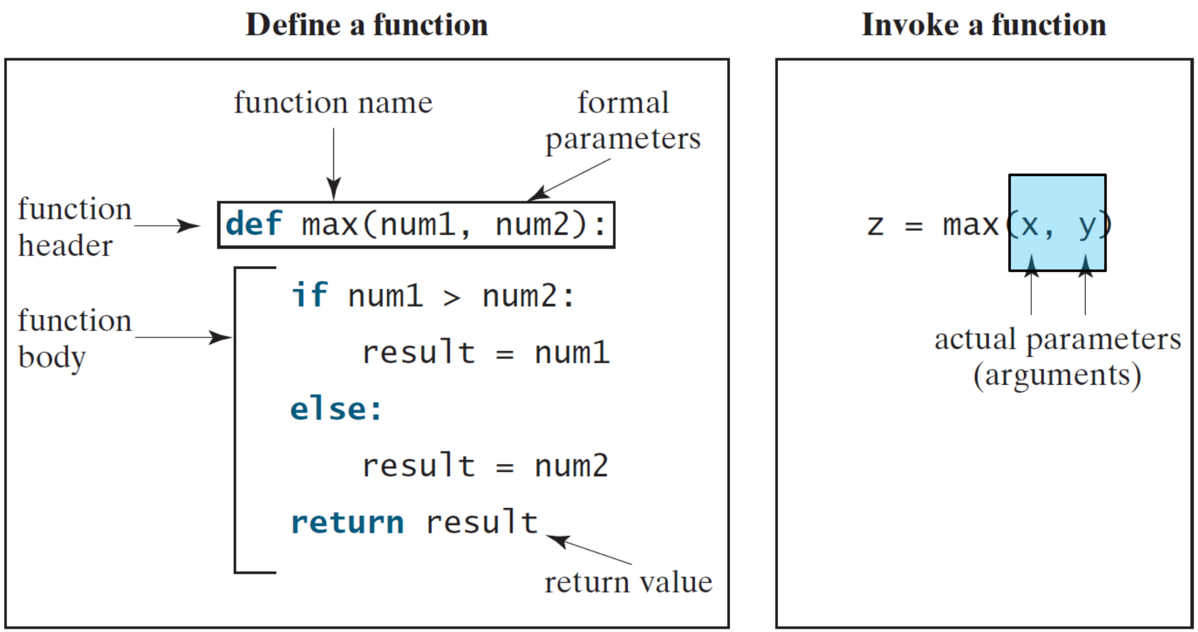
Defining Functions

- Anatomy of Sample Function:
  - **Formal Parameters**:
    - Variables shown or defined in the function header are called formal parameters (think of these as placeholders).



Defining Functions

- Anatomy of Sample Function:
  - **Actual Parameters**:
    - When you call/invoke a function, you send a value to the formal parameter placeholders.

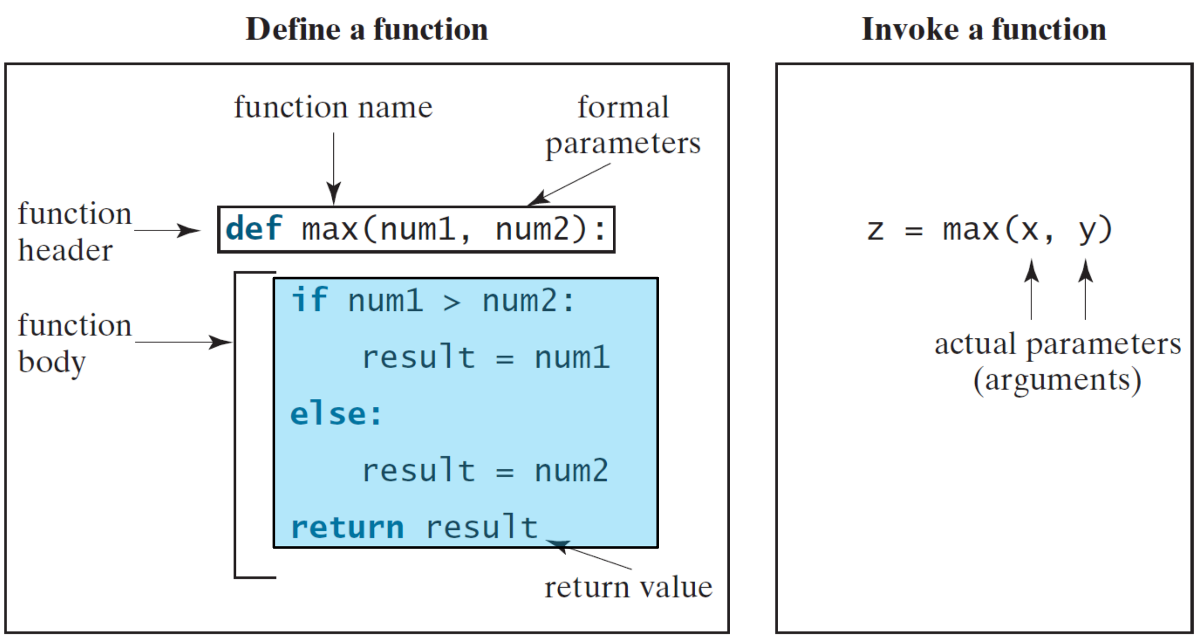


Defining Functions

- Anatomy of Sample Function:
  - **Actual Parameters**:
    - When you call/invoke a function, you send a value to the formal parameter placeholders.
    - These actual (real) values are called actual parameters
    - Note:
      - You can use the word "parameters" or the word "arguments"
      - BOTH are well-known
    - The parameter list (or the argument list) refers to the function's type, order, and number of parameters
    - Parameters are optional
    - This means that some functions may have no parameters

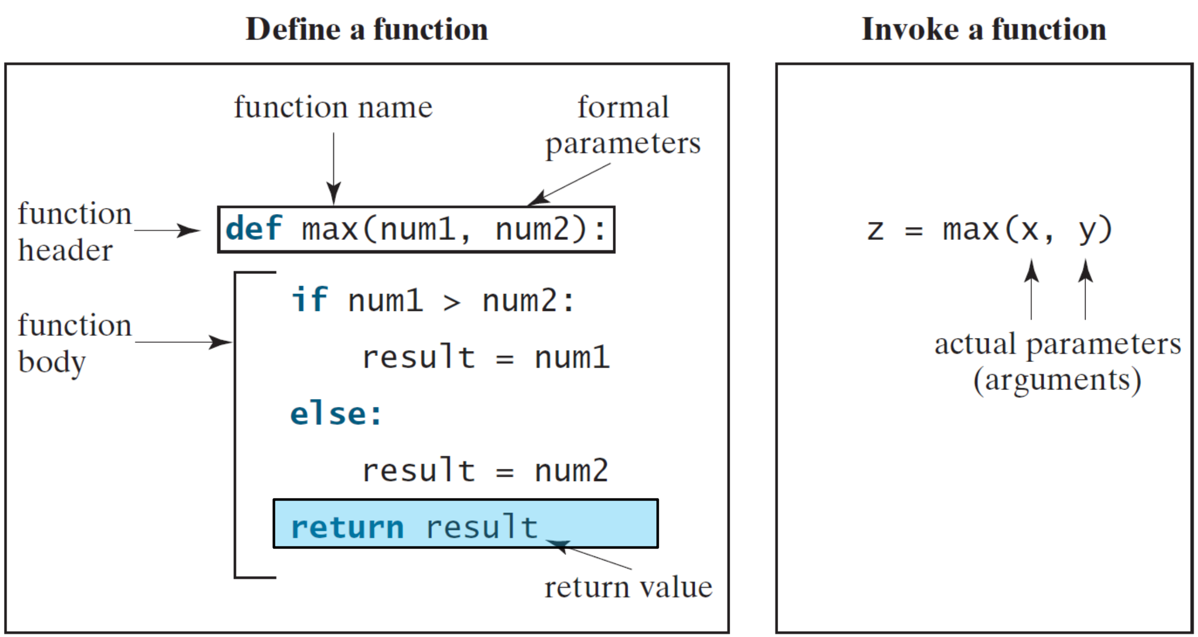
Defining Functions

- Anatomy of Sample Function:
  - **Function Body**:
    - This is the collection of statements that implement the function.



Defining Functions

- Anatomy of Sample Function:
  - **Return Value**
    - Not all functions are used to calculate and the return a value.
    - But a function can return a value using the `return` keyword.



Calling a Function

- Remember:
  - A function is a collection of statements grouped together to perform an action
  - So inside the function, you define the actions
    - You “code up” everything that you want the function to “do”
  - Question:
    - How do we "start" the function? How do we run it?
  - Answer:
    - We call or invoke the function.

Calling a Function

- Two ways to call a function, depending on whether the function returns a value or not
  1. If the function returns a value, the "call" is usually treated as a value:
     - Example:

       ```
       larger_number = max(3, 4)
       ```

       - Here, we "call" the function, `max(3, 4)`
       - The maximum number, which is 4, will get returned
       - We save that value (4) into the variable `larger_number`
     - Example:

       ```
       print(max(3, 4))
       ```

       - Here, we directly print the result, which is 4

Calling a Function

- Two ways to call a function, depending on whether the function returns a value or not
  2. If the function does not return a value, the "call" to the function is a basic statement
     - Example:

```python
def print_this():
    print("Hi. We are printing from inside a function.")

# Call the print_this function
print_this()
```

- So there are no actual parameters of the `print_this()` function
- And it does not return a value…it simple prints inside the function

Calling a Function

- Program Control
  - When you run a program, the control of the program is in the regular area of your program
    - We'll refer to this as "**`main`**"
    - This is called program control
  - When you call a function from **`main`**, program control is transferred to the function you called
  - **`main`** is basically waiting for the function to finish
    - Once the function finishes, program control returns to **`main`**
    - <u>A called function returns control</u> to the caller
      - <u>when</u> its return statement is executed, or
      - <u>when</u> the last line of the function is reached

Program 2: Test Max

- Write a program that will call another function, **`max`**, to determine the maximum of two numbers. Function **`max`** should return the maximum value.
- Remember:
  - Step 1: Problem-solving Phase
  - Step 2: Implementation Phase

Program 2: Test Max

- **Step 1**: Problem-solving Phase
  - Algorithm:
    - In our "main" working area, we just make two integers and give values for each
      - Of course, we could ask the user for two numbers
      - Or we could generate two random numbers
        - These are easy things and are not the purpose of this example
    - Next, we call the **`max`** function
    - This means we need to write a **`max`** function!
      - **`max`** function should be easy
      - Just check which number is larger
      - Save the larger number into a variable
      - Finally, return that variable (the larger number)

Program 2: Test Max

- **Step 2**: Implementation Phase
  - A possible solution:

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

    return max_number

a = int(input("Enter an integer: "))
b = int(input("Enter an integer: "))
max_num = max(a, b)
print("\nThe maximum of", a, "and", b, "is", max_num)
```

The Main Function

- Main:
  - We referred to the working area (the non-function area) of your program as "main"
  - Why?
  - Because many (or most) languages actually define main
    - This is the standard entry point into your program
  - By default, Python doesn't need this
    - You just start coding on line 1
  - But because using main is so common, most Python programmers define a "main" function and then invoke this function to start their program

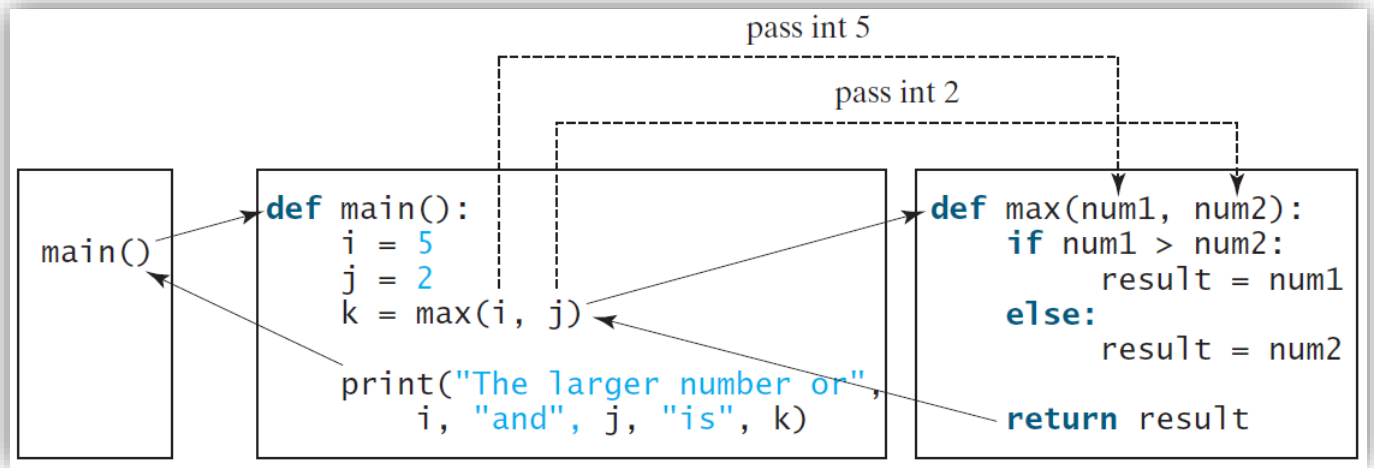
Program 2: Test Max

- **Step 2**: Implementation Phase
  - Another possible solution:

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

    return max_number

def main():
    a = int(input("Enter an integer: "))
    b = int(input("Enter an integer: "))
    max_num = max(a, b)
    print("\nThe maximum of", a, "and", b, "is", max_num)

main()
```

Program 2: Test Max

- Tracing Program Control
  - Do yourself a HUGE favor:
    - Run this program through Thonny's debugger
    - You can see precisely how the functions are called
    - And what values are sent between the various functions
    - Here's a graphic, although it doesn't come close to Thonny



Functions without Return Values

- The previous example (`max` function) was a value-returning function
  - meaning, it returned a value (the `max`) to the caller
- Some functions do not return anything at all
- This type of function is called a `void` function in programming terminology
- The following program defines a function named `print_grade` and invokes (calls) it to print the grade based on a given score

Program 3: Print Grade

- Write a program that will call another function, **`print_grade`**, to determine and print the letter grade based on a given score. Your function should not return anything.
- Remember:
  - Step 1: Problem-solving Phase
  - Step 2: Implementation Phase

Program 3: Print Grade

- **Step 1**: Problem-solving Phase
  - Write a function that does the following:
    - It takes in one parameter, a score
    - It then prints the letter grade based off of that score
  - Also, make a function called main:
    - Ask the user to enter a score
    - Print out “The grade is ”
      - but you won’t print the numeric score at that point
      - The goal is to have the function print the letter grade
      - So remember to <u>not</u> print a newline
        - Cause we want the letter grade on the same line
    - Next you simply call the function that you made above
  - Give this a shot…

```
Enter a score: 84
The grade is B
```

Program 3: Print Grade

- **Step 2**: Implementation Phase

```python
# Print grade for the score
def printGrade(score):
    if score >= 90.0:
        print('A')
    elif score >= 80.0:
        print('B')
    elif score >= 70.0:
        print('C')
    elif score >= 60.0:
        print('D')
    else:
        print('F')

def main():
    score = int(input("Enter a score: "))
    print("The grade is ", end = "")
    printGrade(score)

main() # Call the main function
```

Program 4: Return Grade

- Write a program that will call another function, **`get_letter_grade`**, to determine and then <u>return</u> the letter grade based on a given score.

```
Enter a score: 78.5 ↵Enter
The grade is C
```

- Remember:
  - Step 1: Problem-solving Phase
  - Step 2: Implementation Phase

Program 4: Return Grade

- **Step 1**: Problem-solving Phase
  - Firstly, DO make a new code for this problem
    - copy your last code
    - Make a new program
    - Paste the code into the new program
    - Edit it accordingly
  - This program is identical to the last problem
  - Only thing is you should not print inside the function
  - Instead, you should return a value
  - And then, in main, you should invoke your function correctly…

Program 4: Return Grade

- **Step 2**: Implementation Phase

```python
# Return letter grade based on the score
def get_letter_grade(score):
    if score >= 90.0:
        return "A"
    elif score >= 80.0:
        return "B"
    elif score >= 70.0:
        return "C"
    elif score >= 60.0:
        return "D"
    else:
        return "F"

def main():
    score = int(input("Enter a score: "))
    print("The grade is ", get_letter_grade(score))

main() # Call the main function
```

- Start here

None Functions

- What is a None Function?
  - Technically, every Python program returns a value
    - Even if you do not explicitly return something
    - Meaning, whether or not you use the return statement, something is returned
  - By default, Python returns a special value, **None**
  - Thus, functions that do not explicitly return a value are referred to as None functions in Python
  - Note:
    - A return statement is not needed by a None function
    - But you can include one by typing either:
    - `return` or `return None`

Function Call Stacks

- What happens when a function is called:
  - The system creates an *activation record*
    - This *activation record* stores the parameters and variables, specific to the function
  - The *activation record* is then stored in an area of memory known as the **call stack**
    - Often referred to just as “the stack” (like a stack of books)
  - Each time a function is called, a new activation record is made and placed on **the stack** of called functions
    - Note: the caller’s activation record is kept intact
      - and it’s still on the stack
      - It’s just that the activation record for the new/called function is placed on top of it on the stack

Function Call Stacks

- The Call Stack
  - What happens when a function finishes execution?
  - Answer:
    - Program control returns to the caller
      - The function that called the one that is now finishing
    - and the activation record is removed from the stack
  - The Call Stack stores information in LIFO order
    - Stands for **L**ast **I**n **F**irst **O**ut
    - So the last activation pushed into the stack will be the first activation record removed from the stack
    - Then program control returns to the previous function on the stack

Positional and Keyword Arguments/Parameters

- Power of functions comes with parameters
  - We can pass values (arguments/parameters) to our functions
  - In Python, there are two kinds of arguments:
    - Positional Arguments
    - Keyword Arguments
  - Positional Arguments:
    - This simply means that the arguments sent to the function MUST be in the exact same order as their respective placeholders (formal parameters) in the function header

Positional and Keyword Arguments/Parameters

- Power of functions comes with parameters
  - Positional Arguments:
    - Consider the following function that prints a line ***n*** times:

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

    - We could call this function with `nPrintln("Hello", 3)`
    - The result:
      - The word "Hello" gets passed to the variable **`message`**
      - The integer 3 gets passed to the variable **`n`**
      - The word "Hello" would be printed 3 times
    - We could not call this function with `nPrintln(3, "Hello")`
    - Why?
      - Because the order of the sent arguments wouldn't match the placeholders

Positional and Keyword Arguments/Parameters

- Power of functions comes with parameters
  - Positional Arguments:
    - Important to remember:
      When using Positional Arguments, the arguments absolutely must match the formal parameters with respect to their order, their number, and their compatible type
  - Keyword Arguments
    - With Python, we can also use Keyword Arguments
    - You can pass each argument in the form name = value
    - Example:
      - `nPrintln(n = 3, message = "Hello")`
    - Because the arguments use Keywords/names, you can pass them in any order

Program 5: Roll Dice Game

- Write a program to simulate two users rolling a pair of dice. You should then print the result of each player's dice roll, along with who won (or if a tie).

```
Player 1 rolled a 7 and Player 2 rolled a 9
Player 2, you win!
```

  - You should use two functions:
    - `main()`
    - `roll_pair_dice()`
- Remember:
  - Step 1: Problem-solving Phase
  - Step 2: Implementation Phase

Program 5: Roll Dice Game

```
Player 1 rolled a 7 and Player 2 rolled a 9
Player 2, you win!
```

- Step 1: Problem-solving Phase
  - How do you code up the `roll_pair_dice()` function?
    - What's the first thing we realize we need?
      - Random!
    - We need to randomly choose a value of the six-sided dice
      - So a random number between 1 and 6
      - and we need to do this two times…once for each dice
    - The result is then returned to the main() function
  - What goes into `main()`?
    - You need to keep the score of both players
    - You need to call the roll_pair_dice() function for each player
    - You need to print the result

Program 5: Roll Dice Game

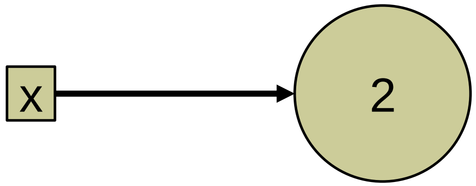
```
Player 1 rolled a 7 and Player 2 rolled a 9
Player 2, you win!
```

- Step 2: Implementation Phase

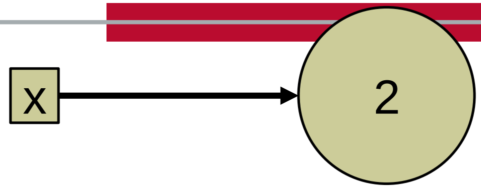
```
import random

def main():

    # Roll both pairs of dice.
    score1 = rollPairDice()
    score2 = rollPairDice()

    print("Player 1 rolled a", score1,"and Player 2 rolled a", score2)

    # Print out the winner.
    if score1 > score2:
        print("Player 1, you win!")
    elif score2 > score1:
        print("Player 2, you win!")
    else:
        print("It's a tie!")

def rollPairDice():
    return random.randint(1,6) + random.randint(1,6)

main()
```

Passing Arguments by Reference Values

- Remember: in Python, all data are actually objects
  - a variable for an object is actually a reference variable that points to (refers to) the actual object
    - Even something as simple as “x = 2”
    - An object is created.
    - Then the value, 2, is stored in that object
    - Then, the reference of that object is saved inside the variable x



Passing Arguments by Reference Values



- So here's a question for you:
  - When we call a function and pass to it arguments, what actually gets sent to the function?
  - Does the reference (address) of the object get sent?
  - Or does the actual value, saved in the object, get sent?
- Answer:
  - Python uses what is known as "call by object"
  - In short, a reference to the actual object is sent to the function
  - So the value inside the variable, x, is sent to the function
    - And that value is simple a reference to the object storing 2

Passing Arguments by Reference Values

- Some Python objects are immutable!
  - Objects containing numbers or strings are immutable
  - This is a fancy word for saying they cannot be changed!
  - More generally, the contents of immutable objects cannot be changed
  - Try typing the following code and then debugging it in Thonny while viewing both variables and the Heap

```
x = 2
x = 3
y = x
```

- Start here Wednesday

Passing Arguments by Reference Values

- Consider the following program:

```
def main():
    x = 1
    print("Before the call, x is", x)
    increment(x)
    print("After the call, x is", x)

def increment(n):
    print("\tInside function, before increment, n is", n)
    n += 1
    print("\tInside function, after increment, n is", n)

main() # Call the main function
```

  - What is the output?

Passing Arguments by Reference Values

- Consider the following program:
  - Output:

```
Before the call, x is 1
       Inside function, before increment, n is 1
       Inside function, after increment, n is 2
After the call, x is 1
```

  - So we see that the value saved in the object referenced by variable x did not change.
  - Why?
    - The reference stored in `x` was passed and saved inside `n`
    - Then the value was incremented by 1
    - But numbers are immutable! So a new object was made, and a reference for that object was saved in the variable `n`

Passing Arguments by Reference Values

- Consider the following snippet of code:

```
x = 2
x = 3
y = x
y += 1
```

  - How many objects do you think Python creates?
  - Answer:
    - If you said 3, you were close…but wrong
    - There’s definitely an object for the 2, the 3, and even the 4
    - But python even creates an object for the 1 that is added to 3
    - So 4 total objects

Passing Arguments by Reference Values

- Consider the following snippet of code:

```
x = 4
y = x
print(id(x))
print(id(y))
y = y + 1
print(id(y))
```

  - Here's a graphic explaining what happens:

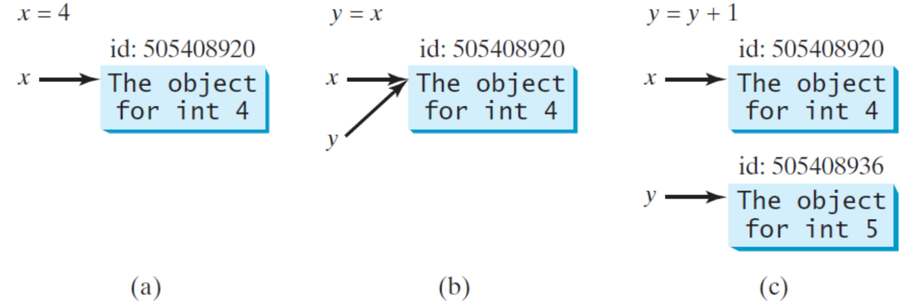


FIGURE 6.4 (a) 4 is assigned to x; (b) x is assigned to y; (c) y + 1 is assigned to y.

Modularizing Code

- What is the main purpose of functions?
  - Code reuse!
    - We can write code once and then reuse it over and over
- A secondary purpose of functions:
  - Modularize our code
  - With longer programs, code can be hard to read
    - Perhaps no organization…just one long block of code
  - Better to break it into chunks (functions)
    - This is the idea of modularizing one's code
    - Also, what's cool is that these chunks can be offloaded into other files and then imported into the current program…

Modularizing Code

- Consider the GCD program we wrote previously…
  - We can write the function to compute the gcd
  - And we can then save that function <u>in its **own** file</u>
    - called `gcd_function.py`

```
def gcd(num1, num2):
    # find the smaller of num1 and num2
    smaller_num = min(num1, num2)

    # Loop from 1 up to (and including) the smaller_num
    # Test if each value of i is a factor of num1 and num2
    for i in range(1, smaller_num + 1):
        # IF i is a factor of num1 and num2
        if num1 % i == 0 and num2 % i == 0:
            # save i as our new "best" answer
            answer = i

    return answer
```

Modularizing Code

- Consider the GCD program we wrote previously…
  - Now we make another program
    - Called test_gcd_function.py
    - Here, we import the function from the other program

test_gcd_function.py

```
from gcd_function import gcd

n1 = int(input("Enter an integer: "))
n2 = int(input("Enter an integer: "))

print("The GCD of {} and {} is {}.".format(n1, n2, gcd(n1, n2)))
```

- Notice the syntax:

  `from gcd_function import gcd`

  - `from` instructs the interpreter where to find the function
  - `import` tells the interpreter exactly which function to import

Modularizing Code

- Reasons why modularization is helpful:

Modularizing Code

- Reasons why modularization is helpful:
  - It isolates the problem for computing the gcd from the rest of the code in the program.
    - Thus, the logic becomes clear and the program is easier to read
  - Any errors for computing the gcd are confined to the gcd function…this narrows the scope of debugging
  - The gcd function now can be reused by other programs
- Encapsulation
  - This is another popular programming word
  - We've just <u>encapsulated</u> (captured and then enclosed) the gcd code in its own function and then program

Scope of Variables

- Chapter 2 introduced the idea of scope
- What is scope?
  - Short answer:
  - The scope of a variable is the area of the program where the variable is understood
    - where the variable can be referenced and used
- We now look at scope within the context of functions
  - Variables created inside functions are called local variables

Scope of Variables

- Scope within the context of functions
  - Variables created inside functions are called local variables
  - Local variables can only be accessed within that function
  - The scope of a local variable starts from its creation and continues to the end of the function that contains that variable
- Python also has global variables
  - These variables are created outside all functions
  - And they are accessible anywhere

Scope of Variables

- Examples of local and global variables

Example 1

```
1 globalVar = 1
2 def f1():
3     localVar = 2
4     print(globalVar)
5     print(localVar)
6
7 f1()
8 print(globalVar)
9 print(localVar) # Out of scope, so this gives an error
```

  - Global variable on line 1 is accessed inside and outside the function with no problem
  - Local variable created on line 3 cannot be accessed outside the function

Scope of Variables

- Examples of local and global variables

Example 2

```
1  x = 1
2  def f1():
3      x = 2
4      print(x) # Displays 2
5
6  f1()
7  print(x) # Displays 1
```

  - Notice the x is declared twice
    - Once as a global variable and once as a local variable
    - Thus, from line 3 and onward, inside the function, the global variable is no longer accessible
    - Outside the function (line 7), the global variable is accessible

Scope of Variables

- Examples of local and global variables

Example 3

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

- Notice the y is declared conditionally
  - y is only declared if the condition (x > 0) is true
  - Thus, if x is greater than zero, line 5 prints just fine
  - But if x is nonpositive, line 5 will produce an error
    - because, in fact, y was never defined

Scope of Variables

- Examples of local and global variables

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

increase()
print(x) # Displays 1
```

  - The local variable x is different than the global variable x
  - The result:
    - The increment inside the function does not change the global x
  - But what if we have a global variable and would like to modify it inside the function, can we do that?

Scope of Variables

- Examples of local and global variables

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

increase()
print(x) # Displays 2
```

  - Here, we did not declare a new x inside the function
  - Instead, we typed “global x”
  - This effectively binds (glues) the usage of x inside the function to the global variable x

Scope of Variables

Check Yourself:

- What if the output of the following code?

```
def function(x):
    print(x)
    x = 4.5
    y = 3.4
    print(y)

x = 2
y = 4
function(x)
print(x)
print(y)
```

```
Output:
2
3.4
2
4
```

Scope of Variables

Check Yourself:

- What if the output of the following code?

```
def f(x, y = 1, z = 2):
    return x + y + z

print(f(1, 1, 1))
print(f(y = 1, x = 2, z = 3))
print(f(1, z = 3))
```

```
Output:
3
6
5
```

Scope of Variables

Check Yourself:

- What is wrong with the following code?

```
1  def function():
2      x = 4.5
3      y = 3.4
4      print(x)
5      print(y)
6
7  function()
8  print(x)
9  print(y)
```

Answer:
x and y are not defined outside the scope of the function

Thus, lines 8 and 9 will produce errors.

Returning Multiple Values

- Python allows you to return multiple values
  - This is cool
  - And something most languages do not allow

LISTING 6.10 MultipleReturnValueDemo.py

```
def sort(number1, number2):
    if number1 < number2:
        return number1, number2
    else:
        return number2, number1

n1, n2 = sort(3, 2)
print("n1 is", n1)
print("n2 is", n2)
```

Program 6: Generate Random Characters

- Write a program that will generate 175 random lowercase letters and print them 25 per line.

```
gmjsohezfkgtazqgmswfclrao
pnrunulnwmaztlfjedmpchcif
lalqdgivxkxpbzulrmqmbhikr
lbnrjlsopfxahssqhwuuljvbe
xbhdotzhpehbqmuwsfktwsoli
cbuwkzgxpmtzihgatdslvbwbz
bfesoklwbhnooygiigzdxuqni
```

- Remember:
  - Step 1: Problem-solving Phase
  - Step 2: Implementation Phase

Program 6: Generate Random Characters

- **<u>Step 1</u>**: Problem-solving Phase
  - How do we print a random character?
    - For sure, we need to import random
    - But what else?
  - We learned in Chapter 3 that every ASCII character has a unique code between 0 and 127
  - So generating a random character really amounts to generating a random integer between 0 and 127!
  - Then we just use the `chr` function to obtain the integer value from the randomly generated int
    - `chr(randint(0, 127))`

Program 6: Generate Random Characters

- **<u>Step 1</u>**: Problem-solving Phase
  - What about random lowercase letters?
  - One solution is to remember the ASCII values of a and z:
    - Lowercase ‘a’ is 97
    - Lowercase ‘z’ is 122
    - So now you just create a random int value between those values
    - `chr(randint(97, 122))`
  - But no one wants to remember that!
  - Thankfully, we can use Python’s built-in `ord` function
    - We saw this in Chapter 3 as well
    - The `ord` function returns the ASCII value of a character
    - `print(ord(‘a’))      # 97 is printed`

Program 6: Generate Random Characters

- **<u>Step 1</u>**: Problem-solving Phase
  - What about random lowercase letters?
  - So what we need is a random integer between:
    - `ord('a') and ord('z')`
  - Thus:
    - `randint(ord('a'), ord('z'))`
  - And now we get the character value of the
    - `chr(randint(ord('a'), ord('z')))`
  - And finally, a random character between any two characters, ch1 and ch2 (ch1 must be less than ch2) can be made as follows:
    - `chr(randint(ord(ch1), ord(ch2)))`

Program 6: Generate Random Characters

- **<u>Step 2</u>**: Implementation Phase
  - This gives as another chance to practice modularization
  - Let’s remove the functionality of generating random characters from the program that actually prints them
  - So we first make a program containing only functions
  - Then we make our program to print the characters
    - In this program, we import the functions

Program 6: Generate Random Characters

- **<u>Step 2</u>**: Implementation Phase

```
from random import randint # import randint

# Generate a random character between ch1 and ch2
def get_random_character(ch1, ch2):
    return chr(randint(ord(ch1), ord(ch2)))

# Generate a random lowercase letter
def get_random_lowercase_letter:
    return get_random_character('a', 'z')

# Generate a random uppercase letter
def get_random_uppercase_letter():
    return get_random_character('A', 'Z')

# Generate a random digit character
def get_random_digit_character:
    return get_random_character('0', '9')

# Generate a random character
def get_random_ASCII_character:
    return chr(randint(0, 127))
```

Discuss in groups what is going on here.

Notice that we first make a generic function, which is then called by the other functions.

The first function generates a random character between "ch1" and "ch2" (inclusive).

Next, for example, the second function shown calls the first function by sending to it the characters 'a' and 'z'.

Program 6: Generate Random Characters

- **<u>Step 2</u>**: Implementation Phase

```
import random_characters

NUMBER_OF_CHARS = 175 # Number of characters to generate
CHARS_PER_LINE = 25 # Number of characters to display per line

# Print random characters between 'a' and 'z', 25 chars per line
for i in range(1, NUMBER_OF_CHARS + 1):
    print(random_characters.get_random_lowercase_letter() , end = " ")
    if i % CHARS_PER_LINE == 0:
        print() # Jump to the new line
```

Function Abstraction

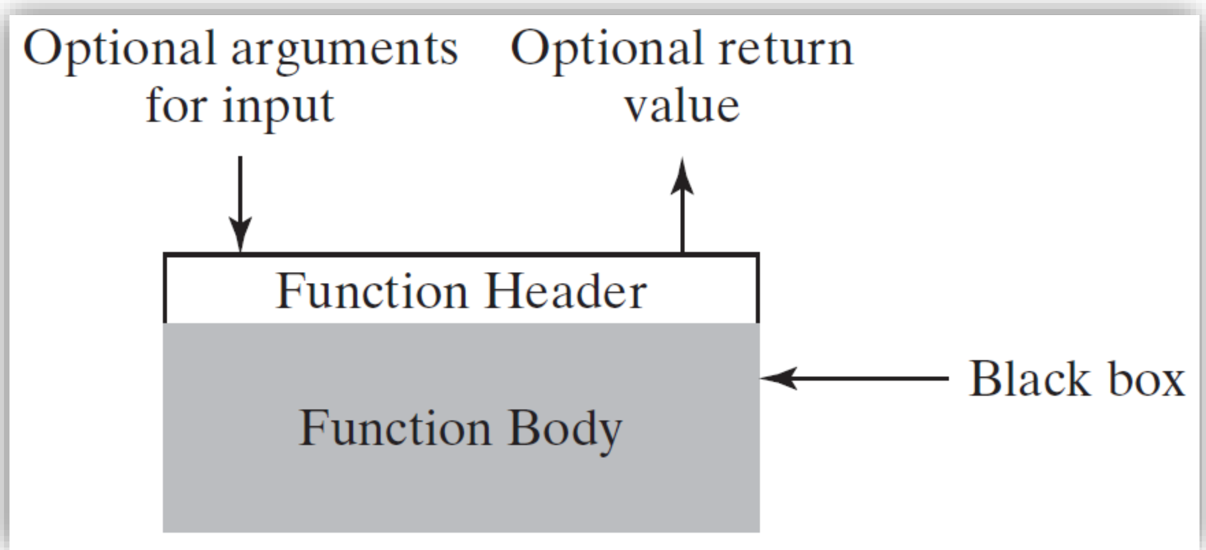
- Main idea for developing software!
  - To develop quality software, programmers must fully understand and be comfortable with the idea of function abstraction
- What is function abstraction?
  - We separate the implementation of a function from the actual use of the function
  - The client/customer can use a function without knowing how to actually code it
  - The details of the function are hidden from the client

Function Abstraction

- Information Hiding (Encapsulation)
  - Again, the details of the implementation are encapsulated inside the function
  - And they are hidden from the client
  - This is called Information Hiding or encapsulation
  - The client has access to the function header
    - They can call the function with certain parameters
    - And they hope to get a return value from the function
  - But what is inside the function is hidden from them
  - In fact, they don't care…they just want it to work!

Function Abstraction

- Information Hiding (Encapsulation)
  - So think of the function as a “BLACK BOX” that contains the implementation…but it is hidden



Stepwise Refinement

- Function Abstraction helps makes programs easier
  - because the implementation of a specific idea is removed from the main body of the program
  - So the program is easier to read and understand
- This idea is part of Stepwise Refinement
- What is stepwise refinement?
  - The idea of solving a larger problem/program in smaller steps
  - Certainly, solving something small is easier than solving something larger

Program 7: Print Calendar

- Write a program that prompts the user to enter the calendar year and month and then displays the exact calendar for that input.

- Remember:
  - Step 1: Problem-solving Phase
  - Step 2: Implementation Phase

Program 7: Print Calendar

- **Step 1**: Problem-solving Phase
  - Expected output:

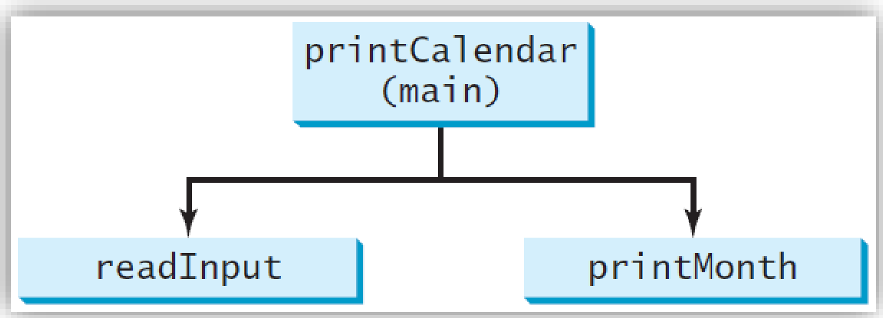
```
Enter full year (e.g., 2012): 2012 ↵Enter

Enter month as number between 1 and 12: 3 ↵Enter

         March 2012
-----------------------------
 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
```

Program 7: Print Calendar

- **Step 1**: Problem-solving Phase
  - Requirements:
    - First requirement: do NOT START CODING!!!
    - New programmers want to start code right away
    - And they also care about the DETAILS of the program
    - Yes, details are important…but not at the beginning
  - The main requirement is to truly understand what the programming is asking of you
  - So for this problem, let us use function abstraction to isolate the details from the actual program design

Program 7: Print Calendar

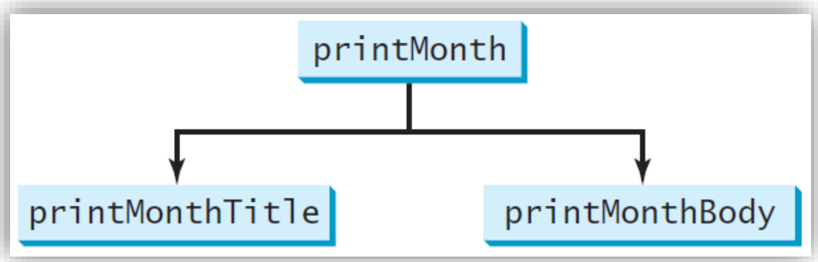
- **Step 1**: Problem-solving Phase
  - Problem Components:
    - We can start by breaking the program into two main components:
      - Get input from user
      - Print the calendar



    - Clearly, getting input from the user is easy and can be left for later discussion
    - The main work is in printing the calendar

Program 7: Print Calendar

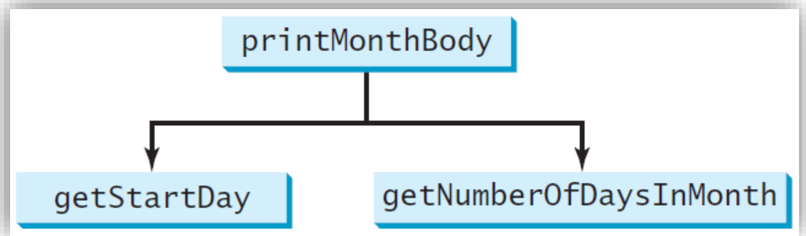
- **Step 1**: Problem-solving Phase
  - Problem Components:
    - And printing the calendar can also be broken down into two components:
      - Print the month title
      - Print the month body



    - Printing the month title is easy
      - It consists of three lines, month and year, a long dashed line, and then the names of the week
      - The only "calculation" here is determining the name of the month

Program 7: Print Calendar

- **Step 1**: Problem-solving Phase
  - Problem Components:
    - Printing the month body will take some thought
    - There are two main things we must compute
      - Starting day of month
      - # of days in month



    - So how can you get the starting day of the month?
      - This problem on its own can be complicated and requires its own thought and strategy

Program 7: Print Calendar

- **Step 1**: Problem-solving Phase
  - So how can you get the starting day of the month?
    - Assume we know that the start day for January 1, 1800 was a Wednesday

      ```
      START_DAY_FOR_JAN_1_1800 = 3
      ```
    - You could compute the total number of days between January 1, 1800 and the first date of the calendar month
    - The start day of the calendar month is:

      (totalNumberOfDays + START_DAY_FOR_JAN_1_1800) % 7
    - Summary: the problem of getting the starting day can be further broken down into the problem of getting the total number of days since January 1, 1800

Program 7: Print Calendar

- **Step 1**: Problem-solving Phase
  - Okay. So how can we get the total number of days?
    - Simple, each year is 365 days.
    - And then for the last year, you must count the number of days before that specific month
    - This means you need to save the number of days in each month
      - And you can write a separate function for this
    - But wait! There is something else to consider!
    - LEAP YEAR!
    - So you must also test for a leap year

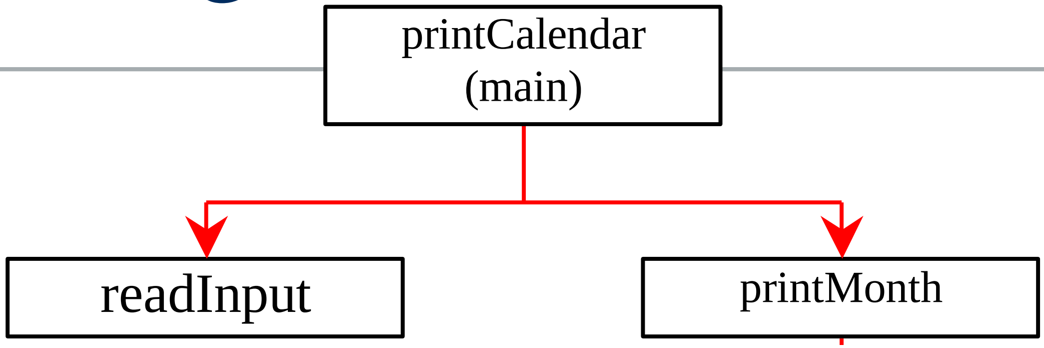
Program 7: Print Calendar

- **Step 1**: Problem-solving Phase
  - So you can see that many components are needed to solve this problem
  - You cannot just start coding immediately
  - Instead, you must identify, step-by-step, or component-by-component, what is needed for your program
  - What we just did was called the "Top-Down Approach"
  - The design diagram is shown on the next pages

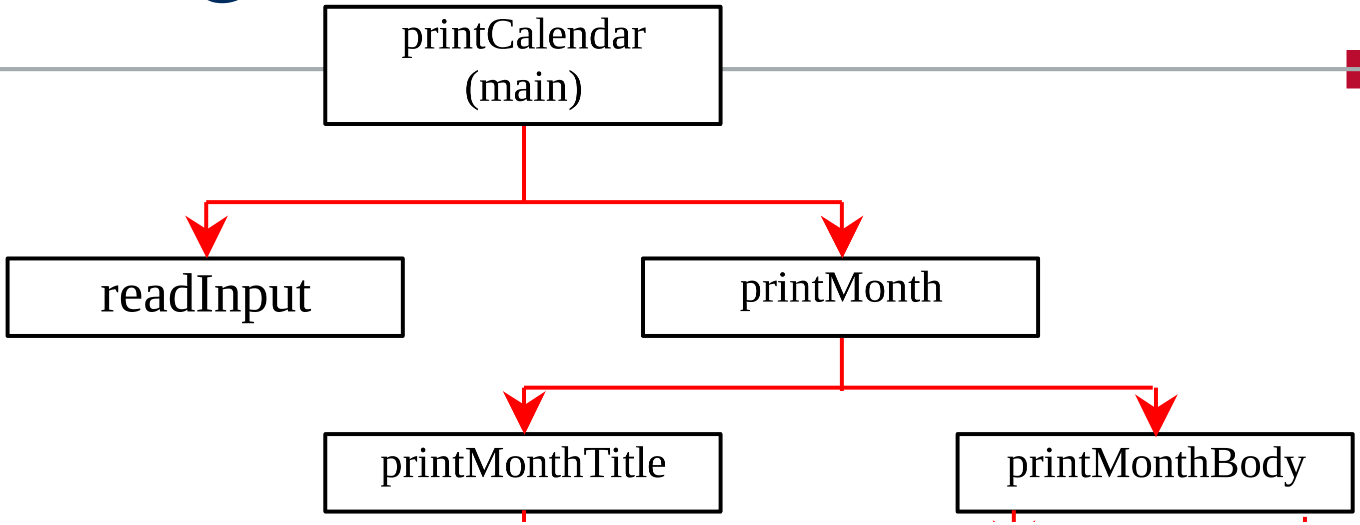
Program 7: Print Calendar

printCalendar
(main)

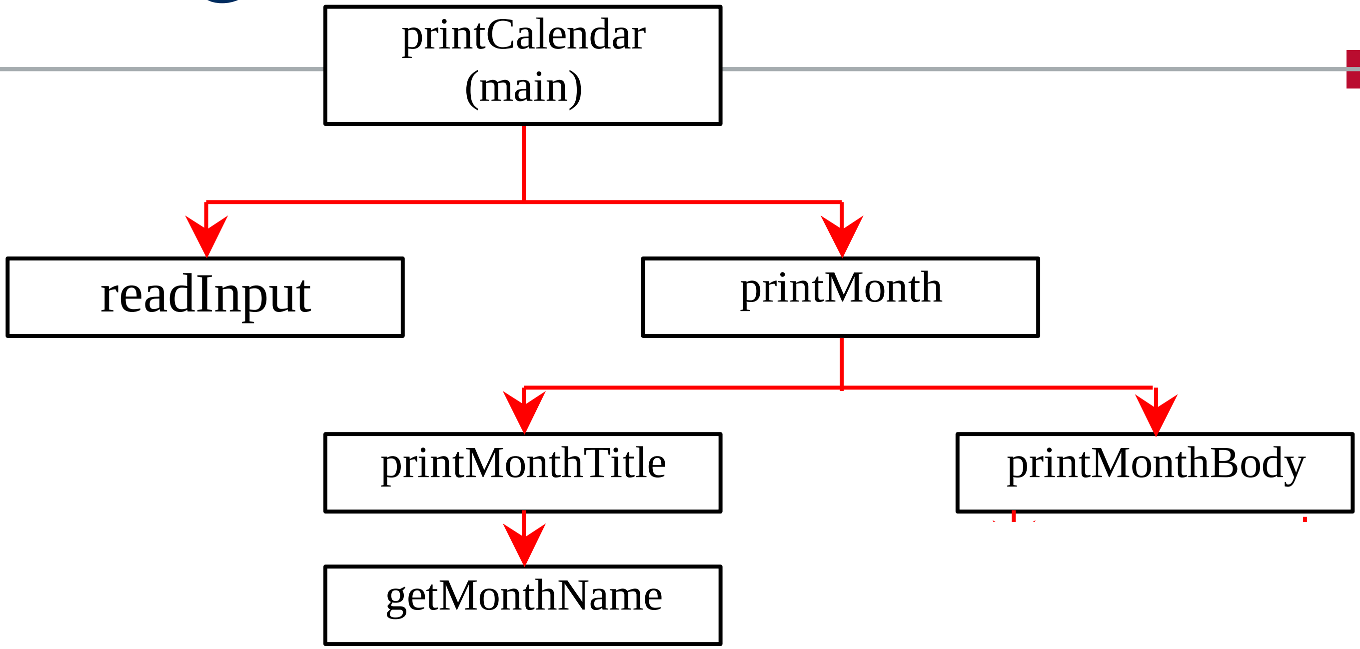
Program 7: Print Calendar



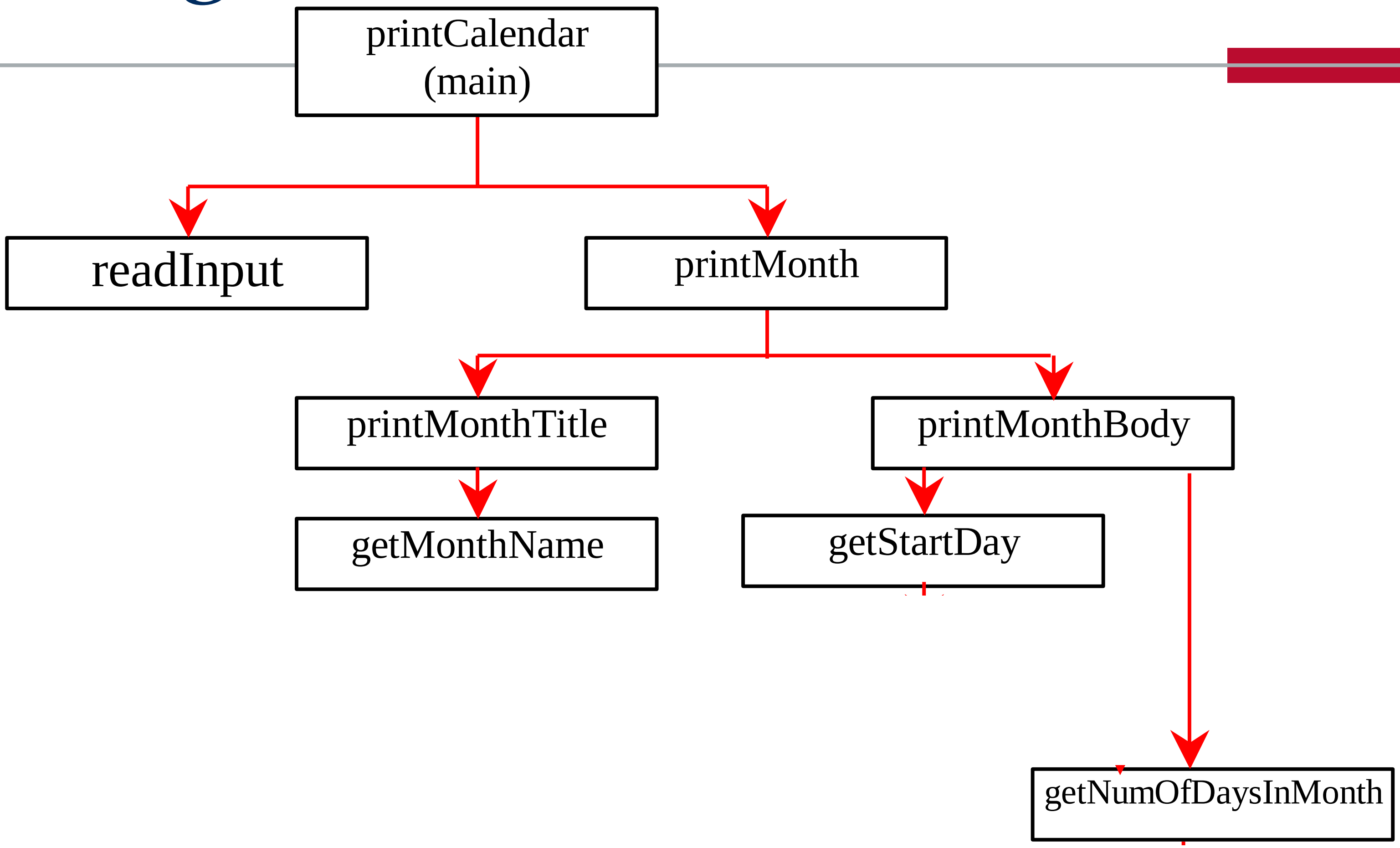
Program 7: Print Calendar



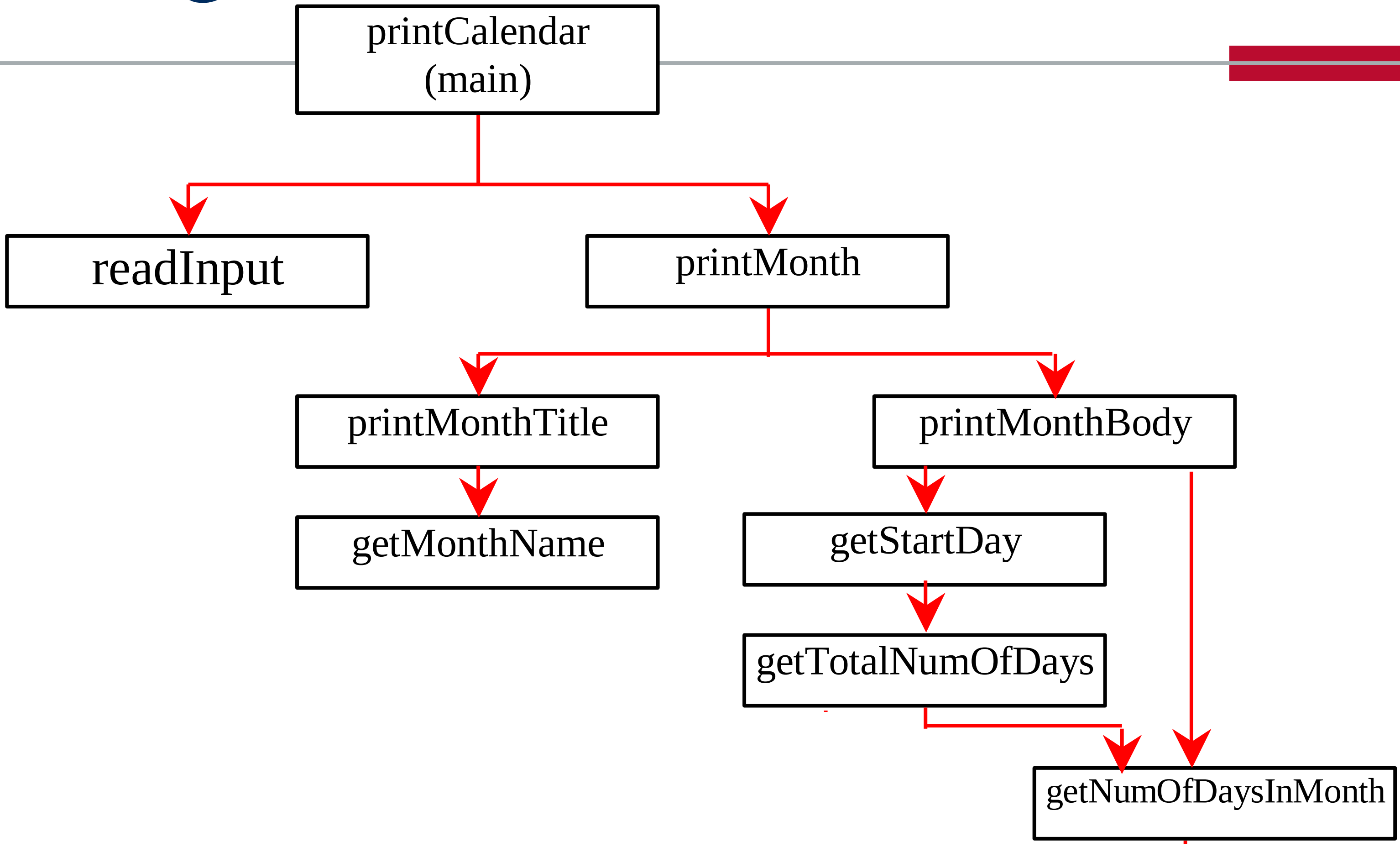
Program 7: Print Calendar



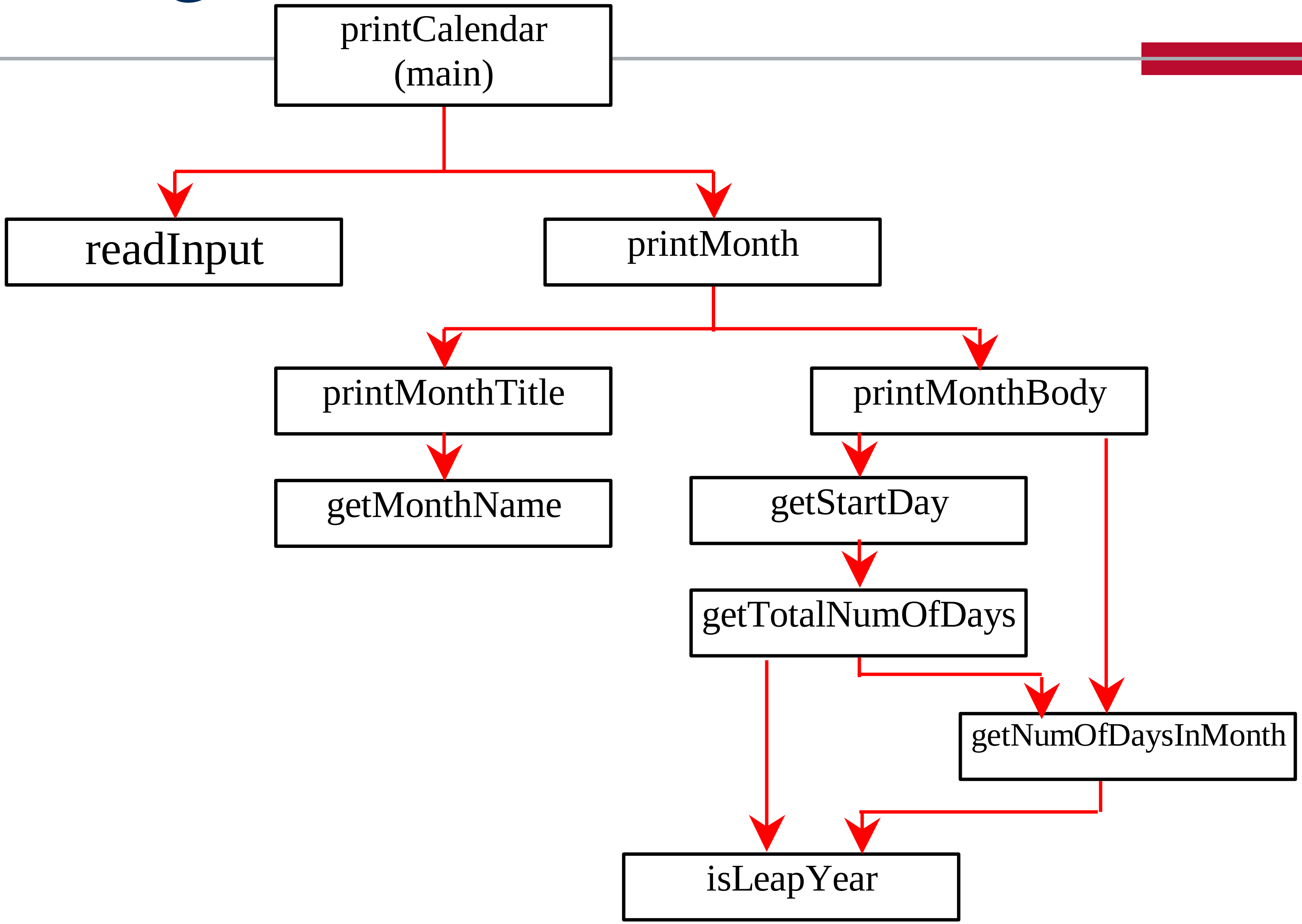
Program 7: Print Calendar



Program 7: Print Calendar



Program 7: Print Calendar



Program 7: Print Calendar

- **Step 1**: Problem-solving Phase
  - 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 `print_month` function and so on
    - This basically sets up a "skeleton version" of our code as shown on the **`print_calendar_stubs.py`** program on portal

Program 7: Print Calendar

- **Step 2**: Implementation Phase
  - Clearly this program is way too long to fit here on the slides
  - The stub/skeleton program is available on Portal
  - Also, the final working version of the program is available for you on Portal

Benefits of Stepwise Refinement

- Some programs can be very long
  - This last program was only 100 lines, but it had several logically independent components
- Stepwise refinement breaks the larger problem down into smaller, more manageable subproblems
- Each subproblem can be implemented using a function
- This approach makes the program easier to:
  - write, reuse, debug, test, modify, and maintain

Benefits of Stepwise Refinement

- Reusing functions:
  - Stepwise refinement encourages code reuse
  - The `is_leap_year` function is defined once
  - However, it is used twice:
    - Inside the `get_total_number_of_days`
    - Inside the `get_number_of_day_in_month`

Benefits of Stepwise Refinement

- Easier developing, debugging, and testing
  - Each subproblem is developed in a function
  - This means each subproblem can be developed, debugged, and tested independently of other components of the problem
  - This isolates errors
  - Whenever you develop large programs, use this stepwise refinement approach
  - It may seem to take longer at first
  - But it saves time and makes debugging much easier!

PYTHON BOOT CAMP

Module 6:
Functions