CITY
UNIVERSITY OF LONDON
EST 1894

Introduction to programming with Python

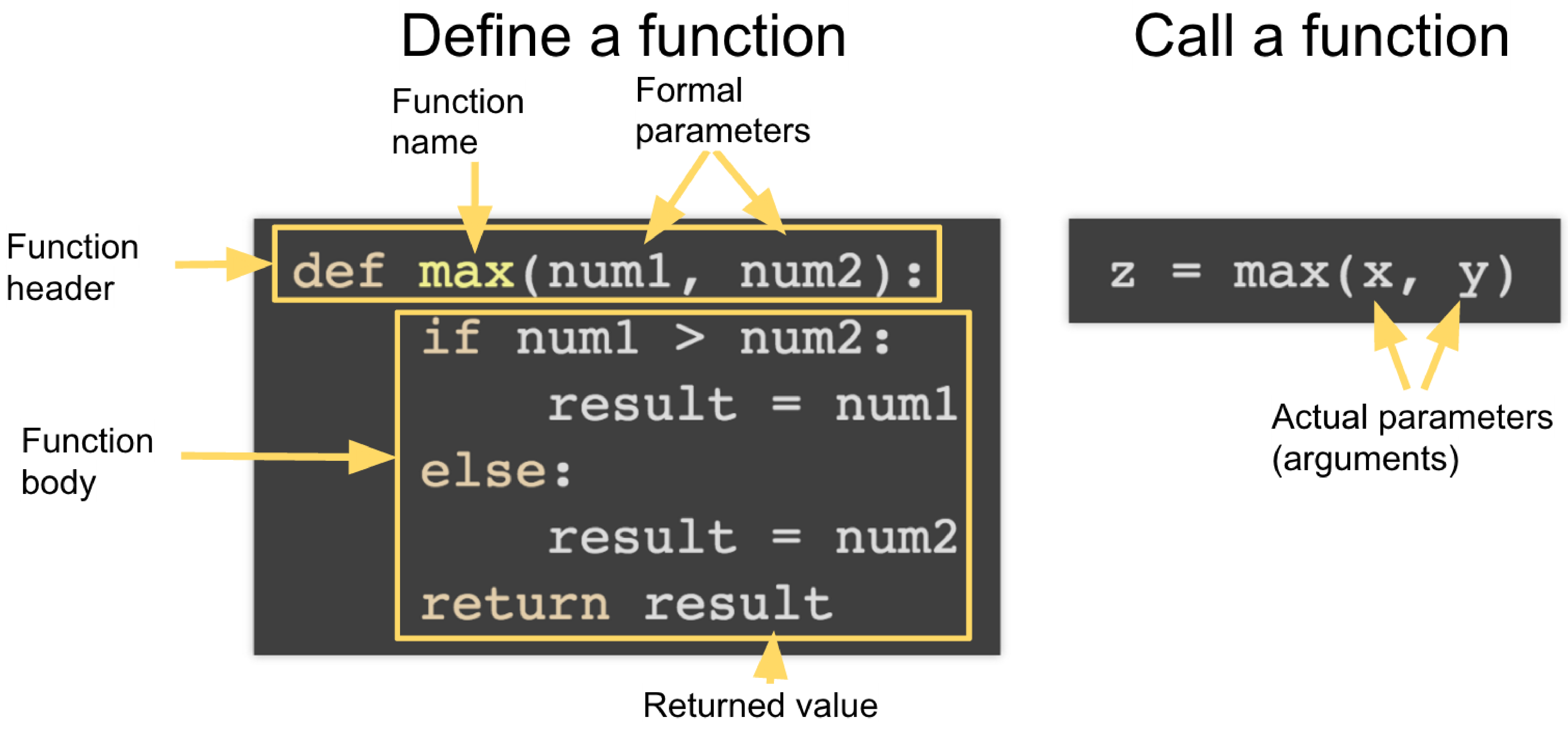
Session 5

Objectives

- To revise function definitions
- To invoke value-returning functions
- To invoke functions that does not return a value
- To pass arguments by value
- To develop reusable code that is modular, easy to read, easy to debug, and easy to maintain
- To create modules for reusing functions
- To determine the scope of variables
- To define functions with default arguments
- To return multiple values from a function
- To apply the concept of function abstraction in software development
- To design and implement functions using stepwise refinement

Defining and Calling Functions

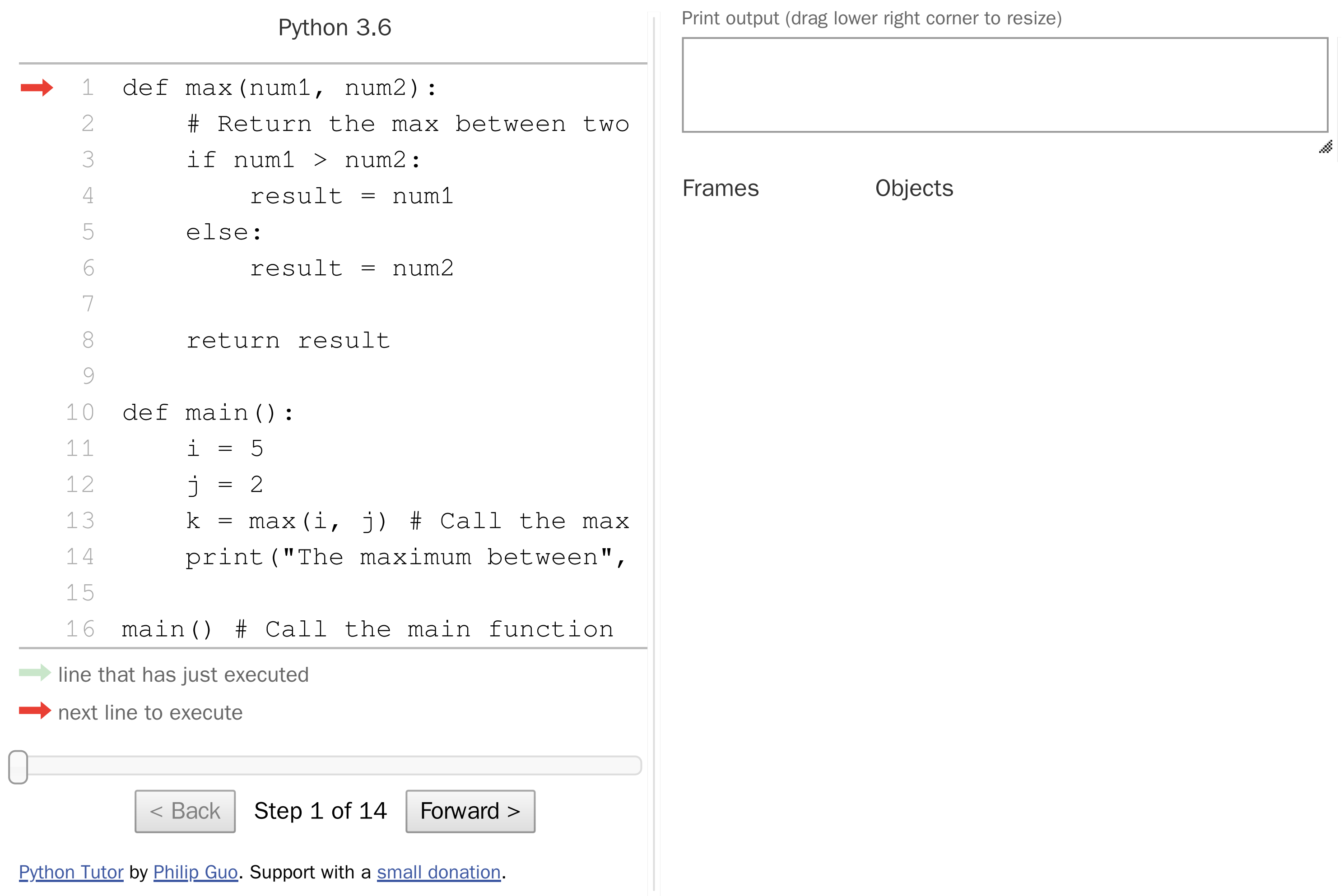
A function is a collection of statements that are grouped together to perform an operation.



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

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

How a function gets called



Python 3.6

```python
def max(num1, num2):
    # Return the max between two
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max
    print("The maximum between",

main() # Call the main function
```

line that has just executed
next line to execute
< Back
Step 1 of 14
Forward >
Print output (drag lower right corner to resize)
Frames
Objects

Python Tutor by Philip Guo. Support with a small donation.

Functions With/Without Return Values

- A function with a **return** keyword explicitly written, returns a **value**. For example the function max() in the previous program.
- A function **does something** but does not return a value. For example the function main() in the previous program.

Example of a function that does something without returning a value

```python
def printGrade(score):
    # Print grade for the 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 = eval(input("Enter a score: "))
    print("The grade is ", end = "")
    printGrade(score)

main() # Call the main function
```

Example of a function that returns a value

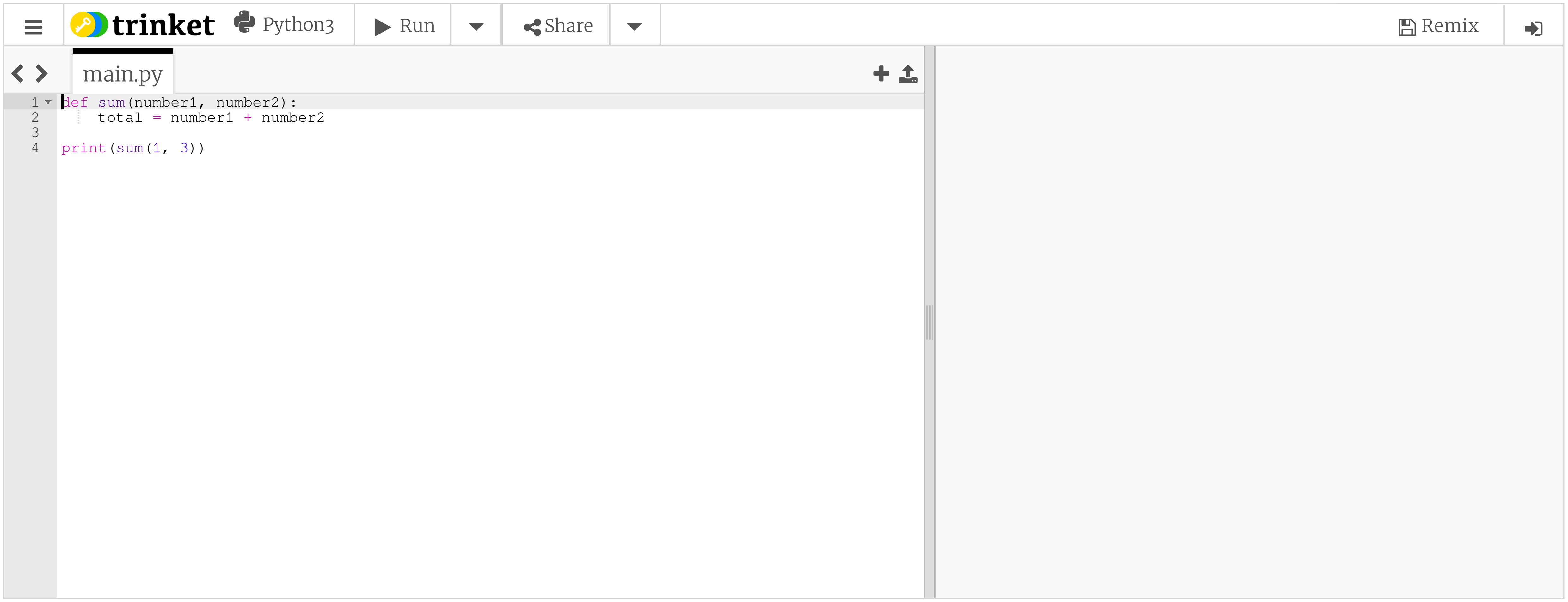
```
def getGrade(score):
    # Return the grade for the score
    if score >= 90.0:
        return 'A'
    elif score >= 80.0:
        return 'B'
    elif score >= 70.0:
        return 'C'
    elif score >= 60.0:
        return 'D'
    else:
        return 'F'

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

main() # Call the main function
```

The None Value

A function that does not return a value is known as a void function. In Python, such function returns a special value, called None.



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

print(sum(1, 3))
```

Passing Arguments by Position

Suppose you have the following function:

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

What is the ouput of **nPrintln("Welcome to Python", 5)**?

What is the ouput of **nPrintln(15, "Computer Science")**?

What is wrong? How to fix?

Keyword Arguments

With the same function:

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

What is the ouput of **nPrintln(message="Welcome to Python", n=5)**

What is the ouput of **nPrintln(n = 4, message = "Computer Science")**

What is wrong? How to fix?

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.

Example

Modularising Code

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

Examples

Download the following files and put them in the current Pycharm project (right click and save as):

- GCDFunction.py
- TestGCDFunction.py

Exercise: Use the isPrime Function

The program PrimeNumberFunction.py (right click and save as) provides the isPrime(number) function for testing whether a number is prime.

Use this function to find the number of prime numbers less than 10,000.

- Reuse the function in the same file
- Import the function in an other file

...ng the function in the same file

Solution

```
def main():
    count = 0
    N = 10000
    for number in range(2, N):
        if isPrime(number):
            count += 1
    print("The number of prime number <", 10000, "is", c

# Check whether number is prime
def isPrime(number):
    for divisor in range(2, number // 2 + 1):
        if number % divisor == 0: # If true, number is n
            return False # number is not a prime
    return True # number is prime

main()
```

....port the function from an other file

Solution

```
from PrimeNumberFunction import isPrime

def main():
    count = 0
    N = 10000
    for number in range(2, N):
        if isPrime(number):
            count += 1

    print("The number of prime number <", 10000, "is", c

main()
```

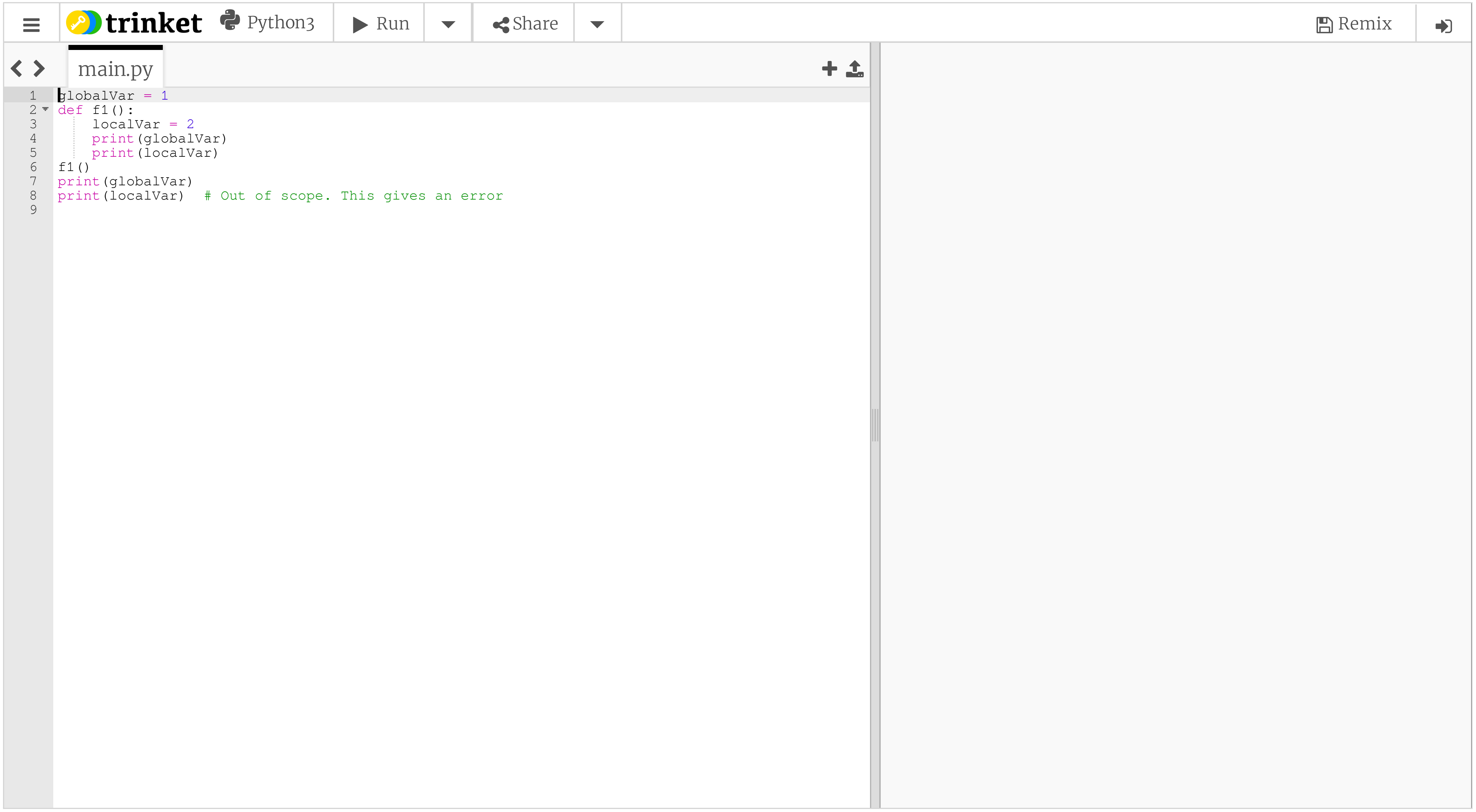
Scope of Variables

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.

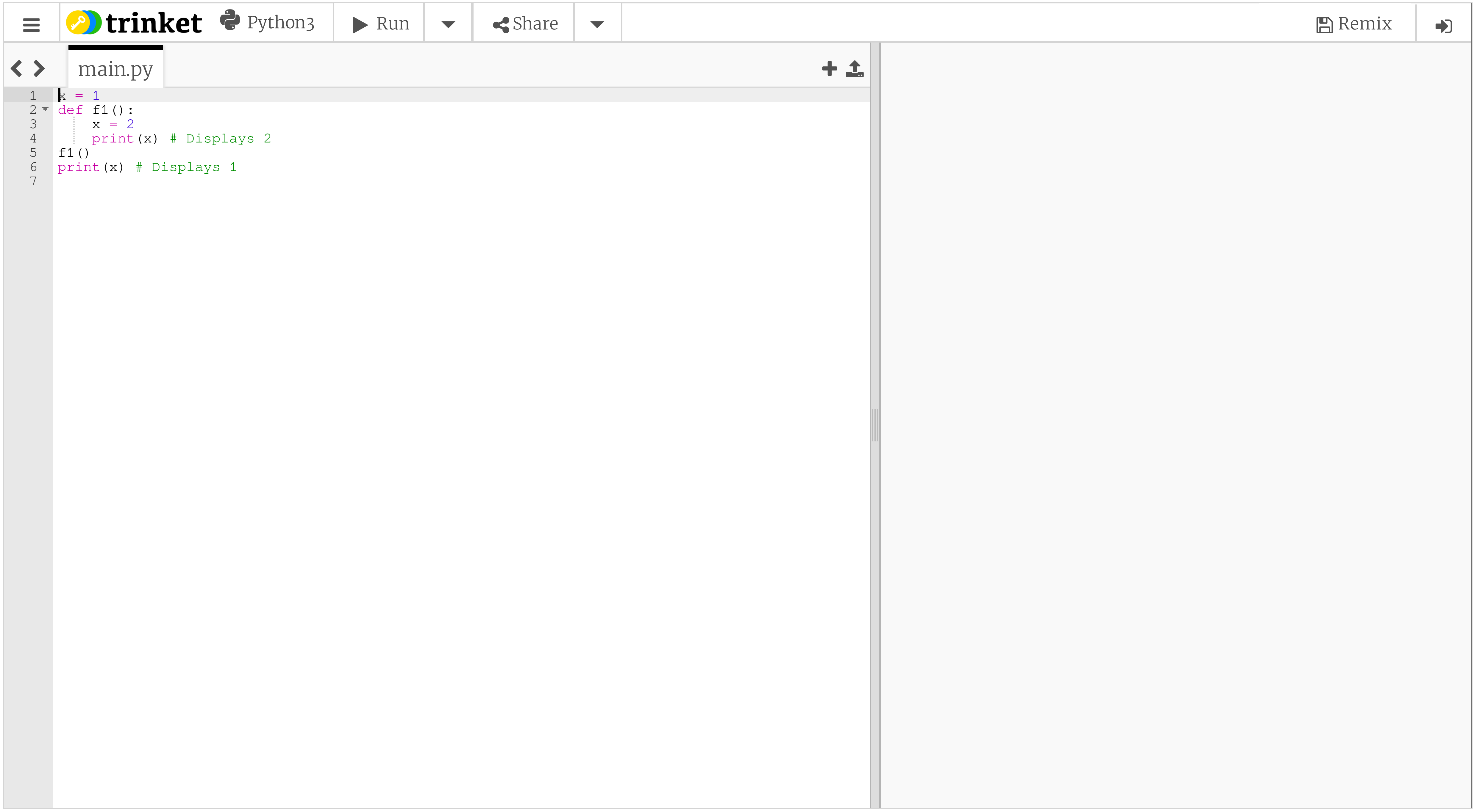
In Python, you can also use **global variables**. They are created outside all functions and are accessible to all functions in their scope.

Example 1



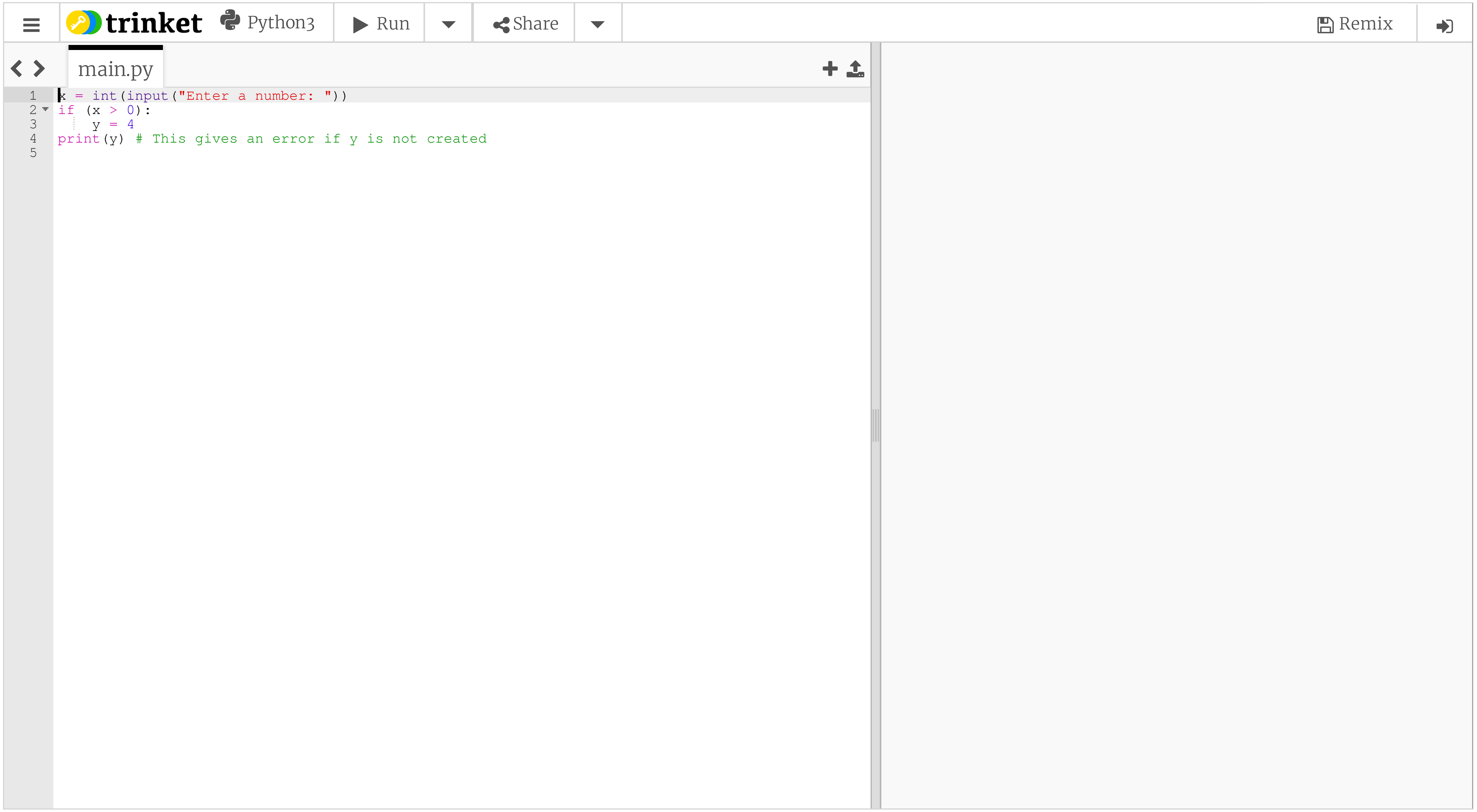
```python
globalVar = 1
def f1():
    localVar = 2
    print(globalVar)
    print(localVar)
f1()
print(globalVar)
print(localVar)   # Out of scope. This gives an error
```

Example 2



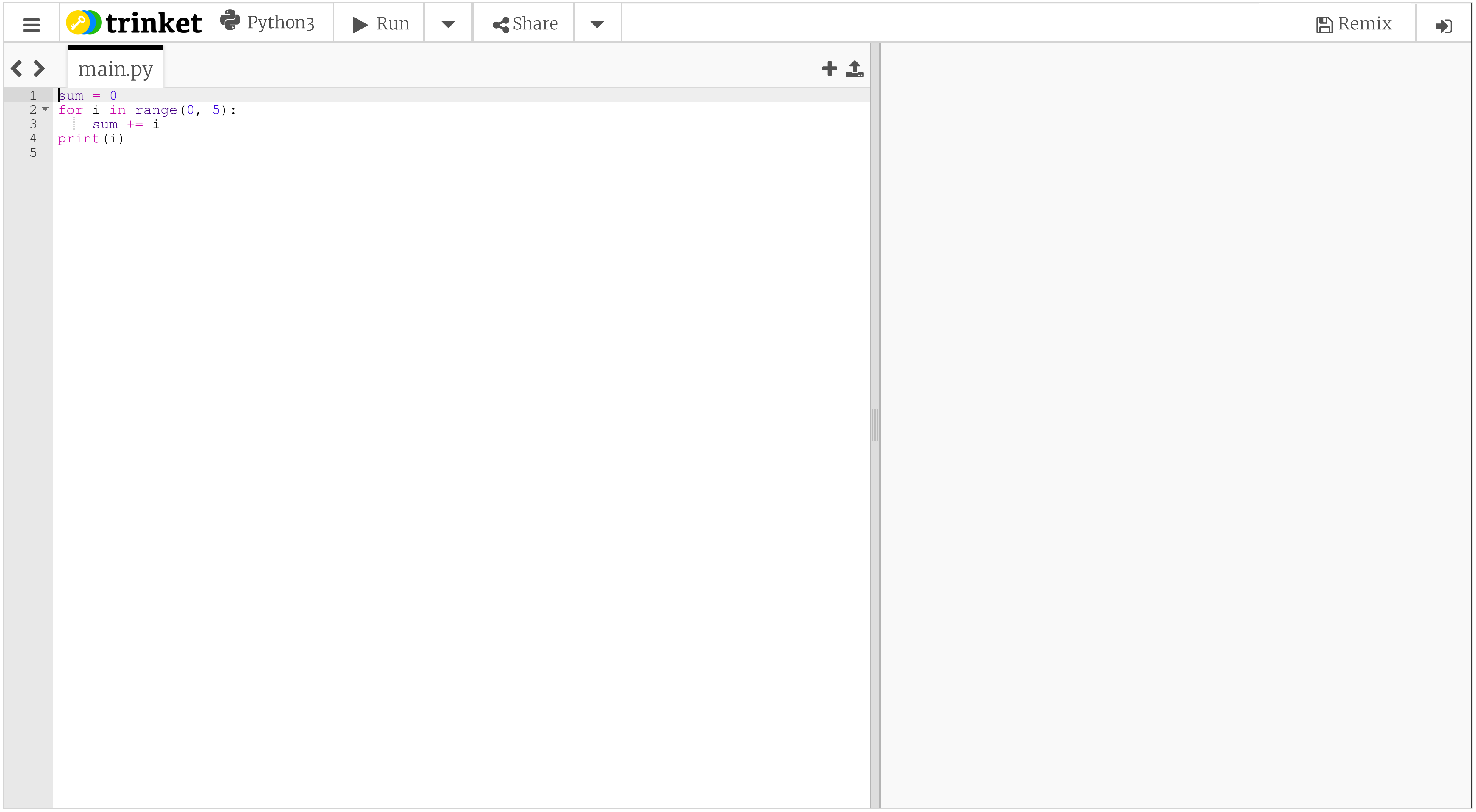
```python
x = 1
def f1():
    x = 2
    print(x) # Displays 2
f1()
print(x) # Displays 1
```

Example 3



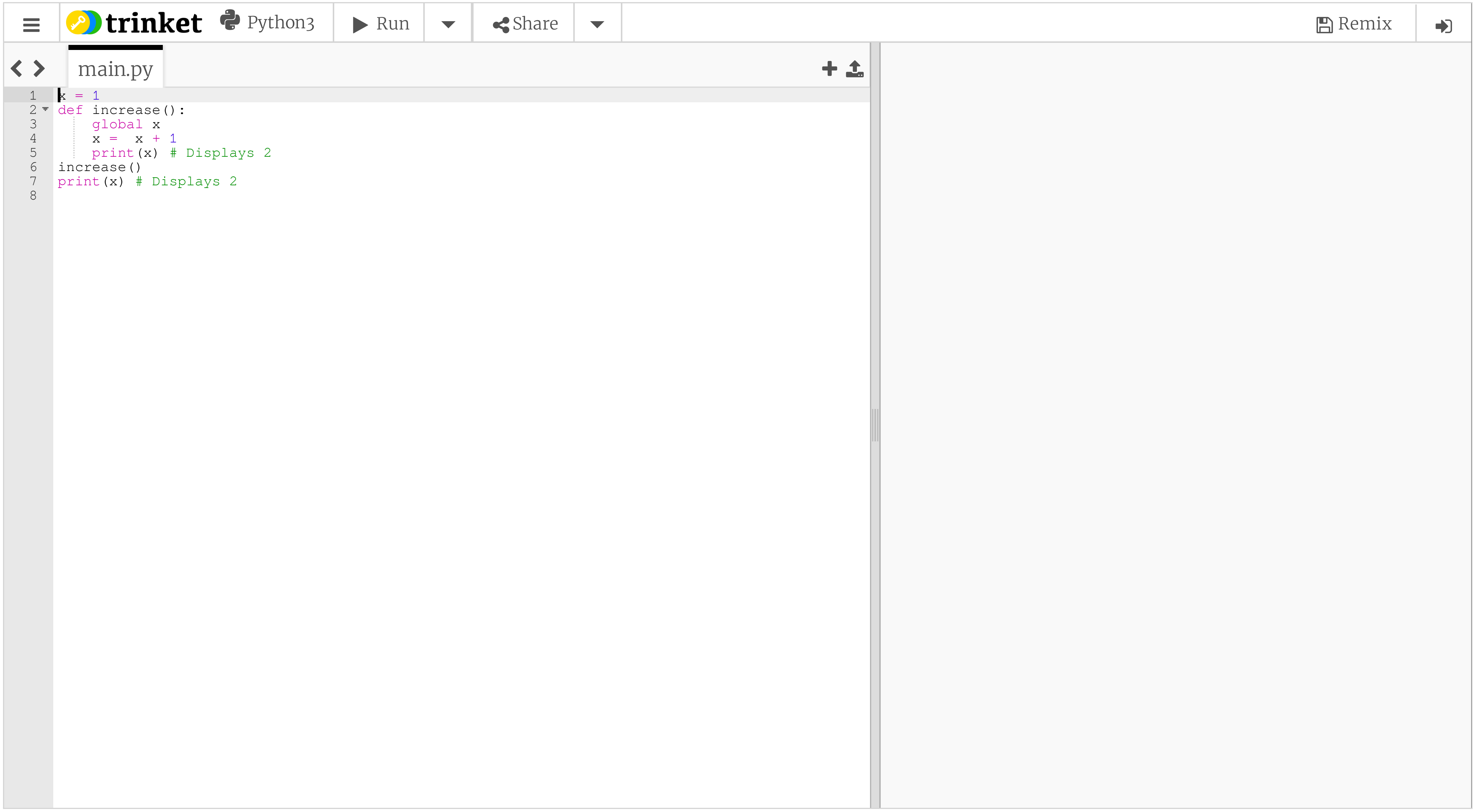
```python
x = int(input("Enter a number: "))
if (x > 0):
    y = 4
print(y) # This gives an error if y is not created
```

Example 4



```python
sum = 0
for i in range(0, 5):
    sum += i
print(i)
```

Example 5

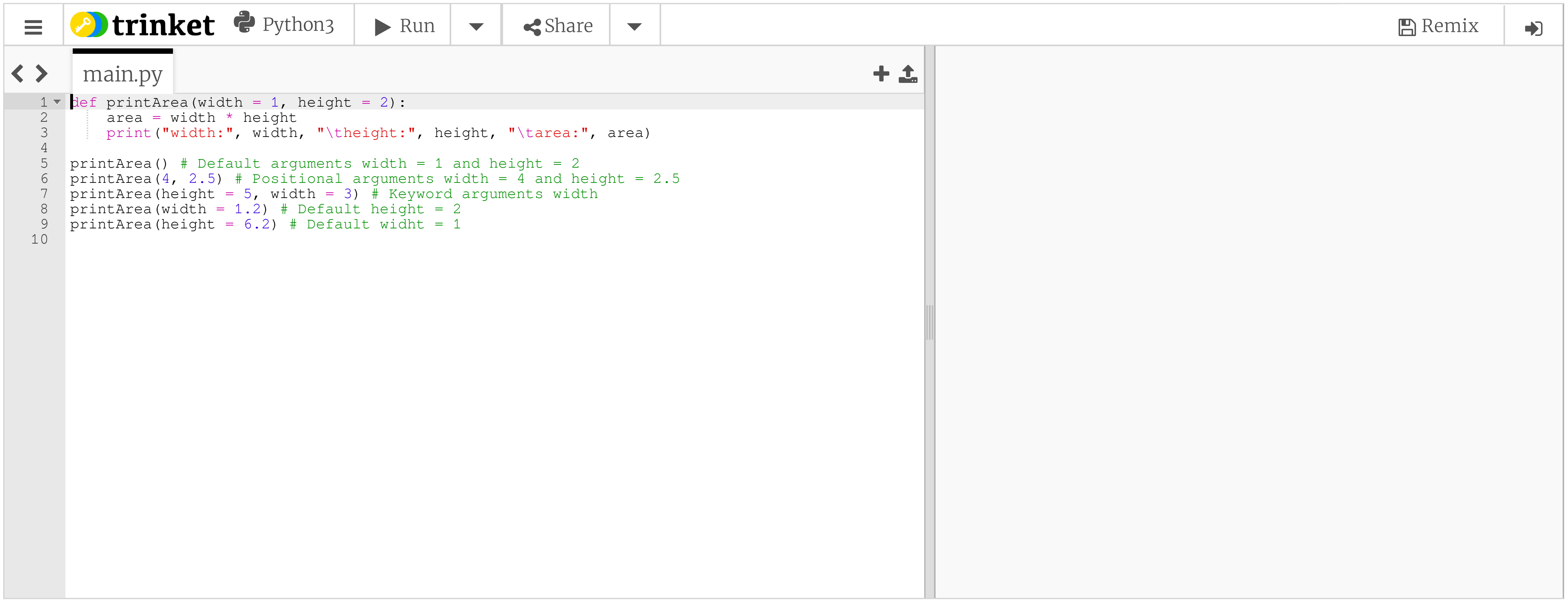


```python
x = 1
def increase():
    global x
    x =  x + 1
    print(x) # Displays 2
increase()
print(x) # Displays 2

```

Default 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.

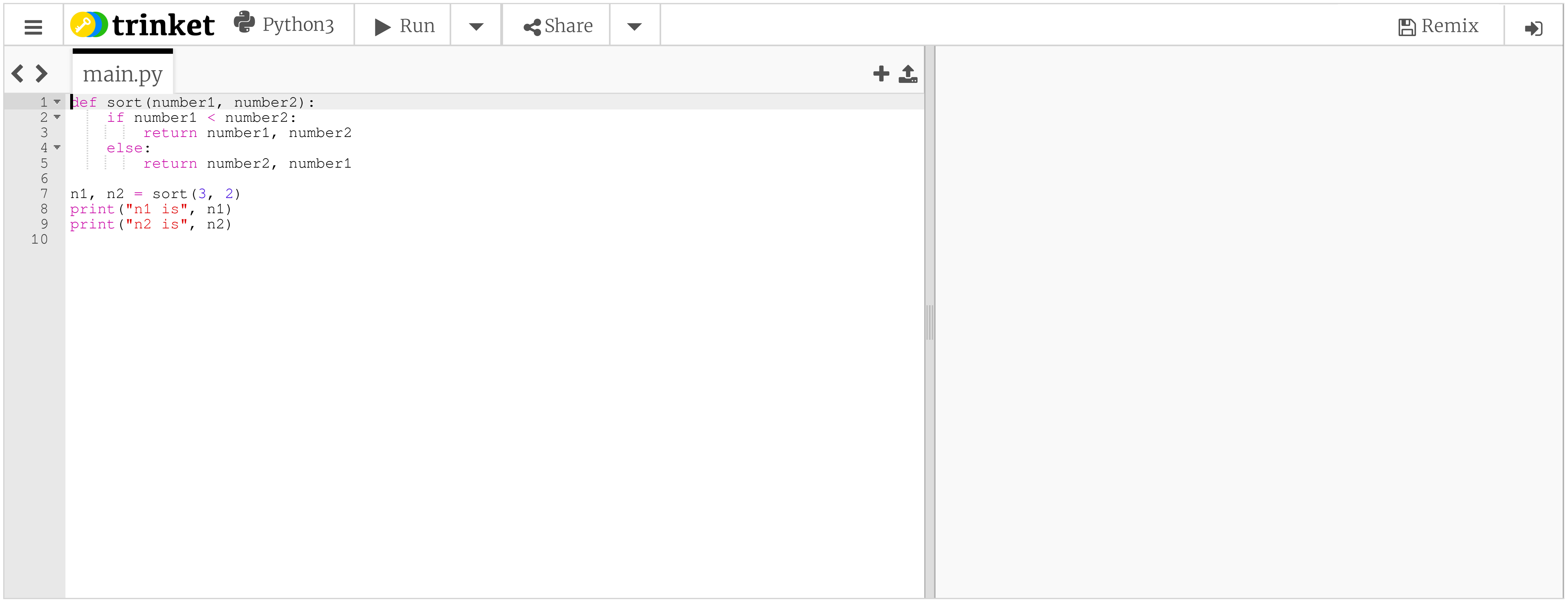


```python
def printArea(width = 1, height = 2):
    area = width * height
    print("width:", width, "\theight:", height, "\tarea:", area)

printArea() # Default arguments width = 1 and height = 2
printArea(4, 2.5) # Positional arguments width = 4 and height = 2.5
printArea(height = 5, width = 3) # Keyword arguments width
printArea(width = 1.2) # Default height = 2
printArea(height = 6.2) # Default widht = 1
```

Returning Multiple Values

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

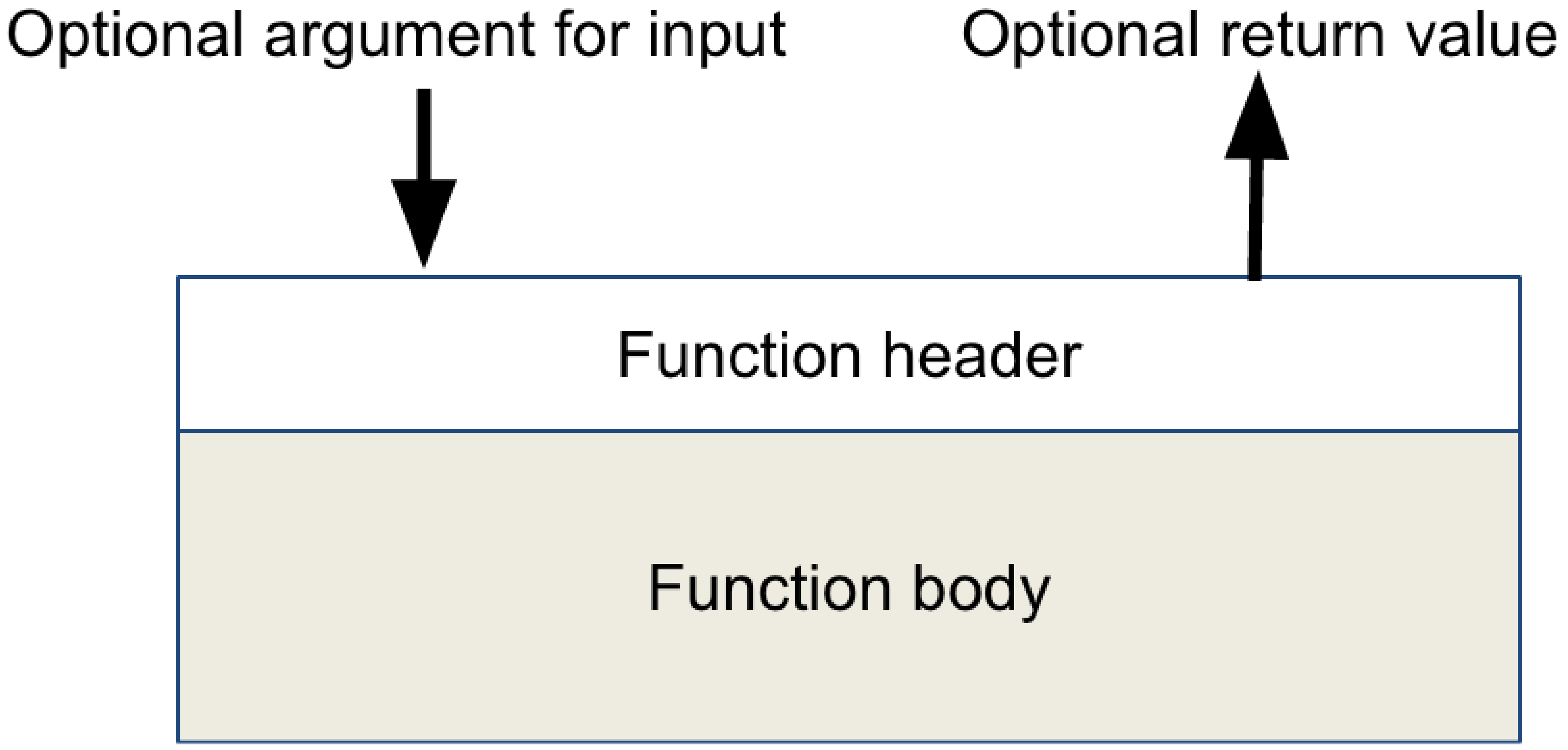


```python
def sort(number1, number2):
    if number1 < number2:
        return number1, number2
    else:
        return number2, number1

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

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.

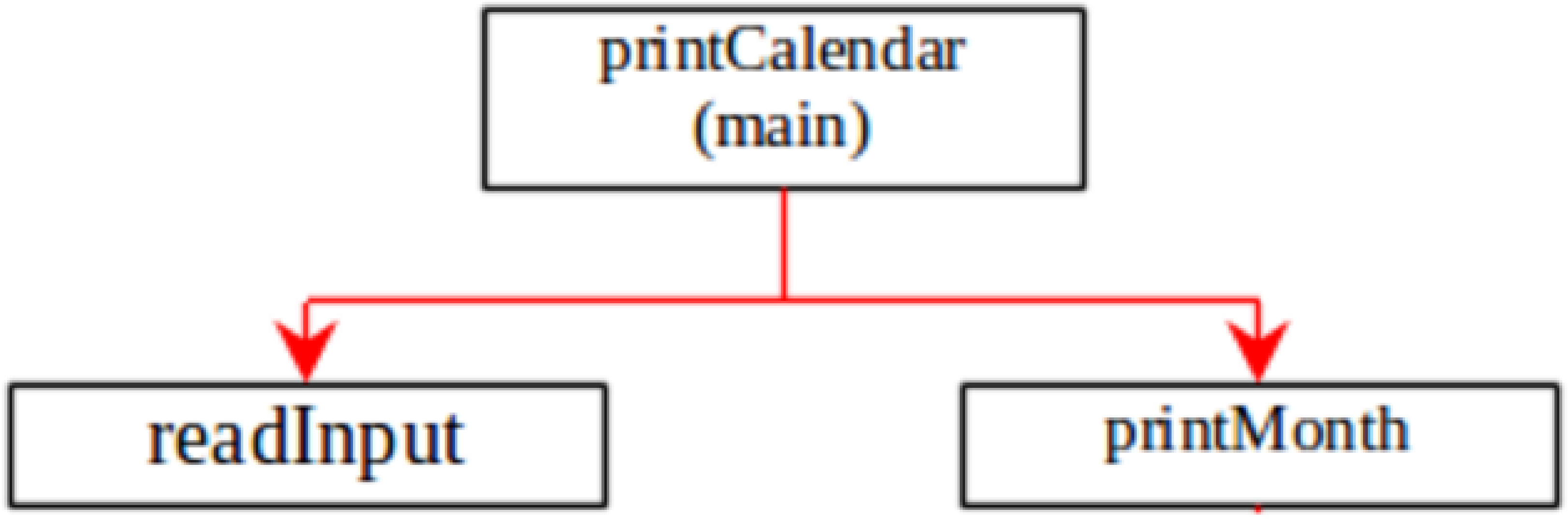
PrintCalendar Case Study

Let us use the PrintCalendar example to demonstrate the stepwise (or divide-and-conquer) 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:

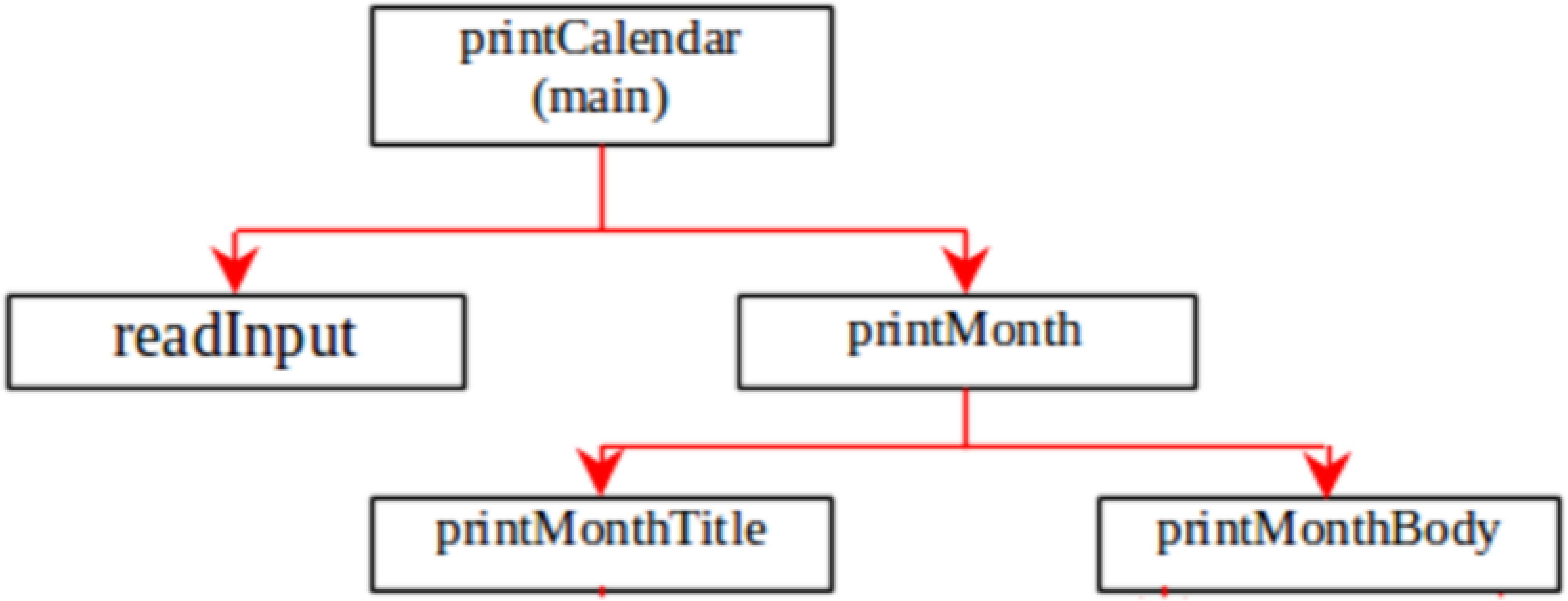
```
Enter full year (e.g., 2001): 2019
Enter month as number between 1 and 12: 8

        August 2019
___________________________
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  31
```

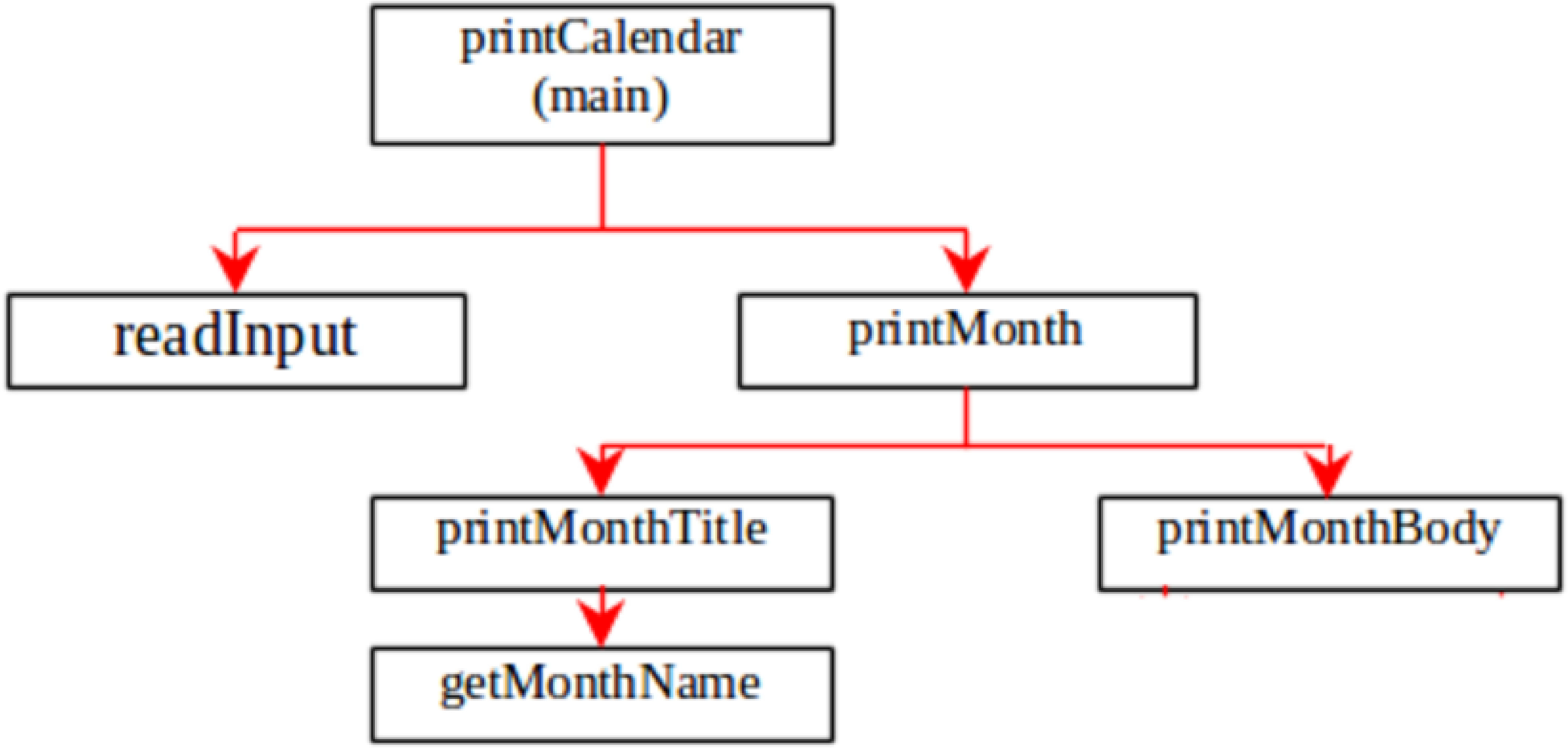
Design Diagram 1



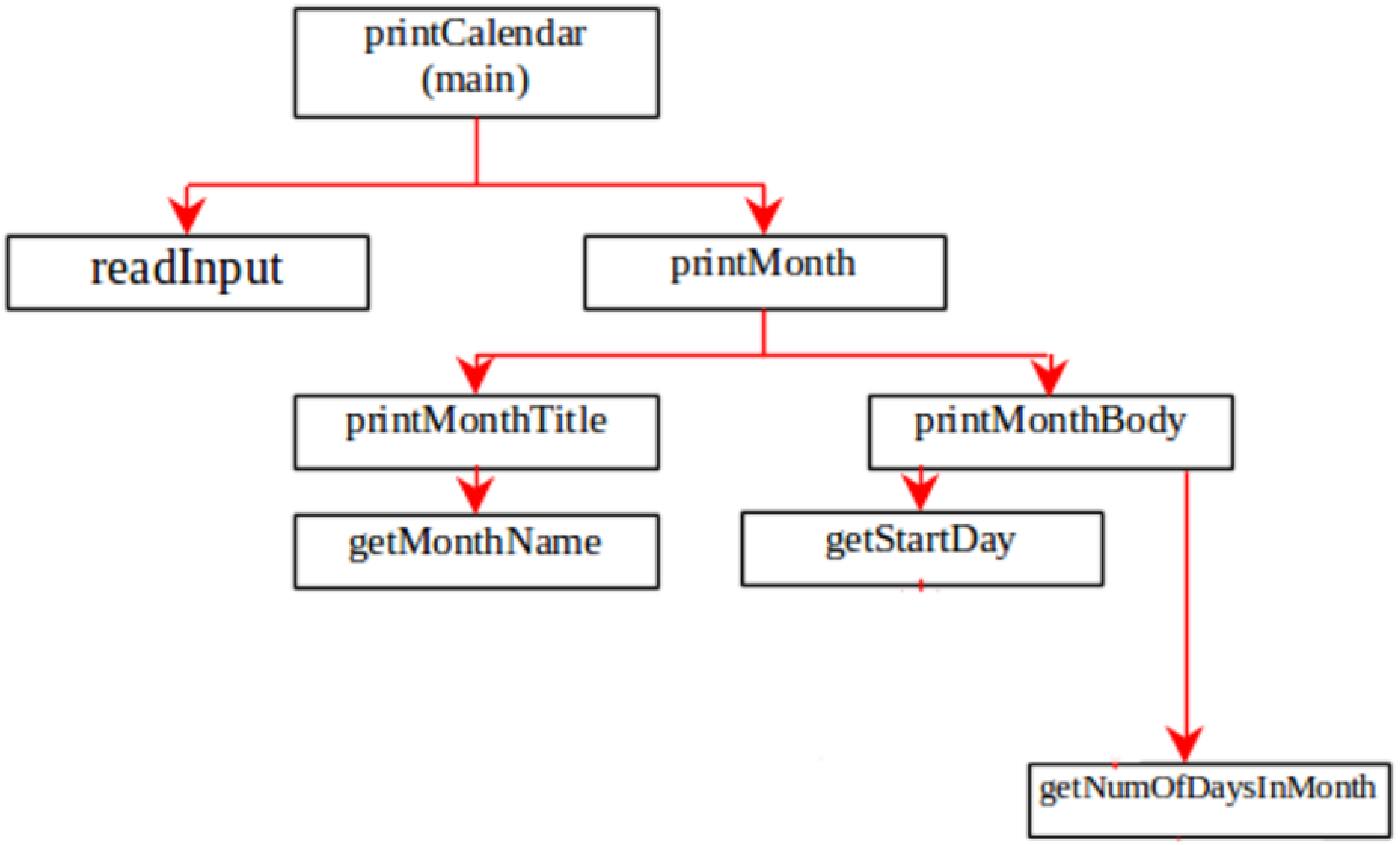
Design Diagram 2



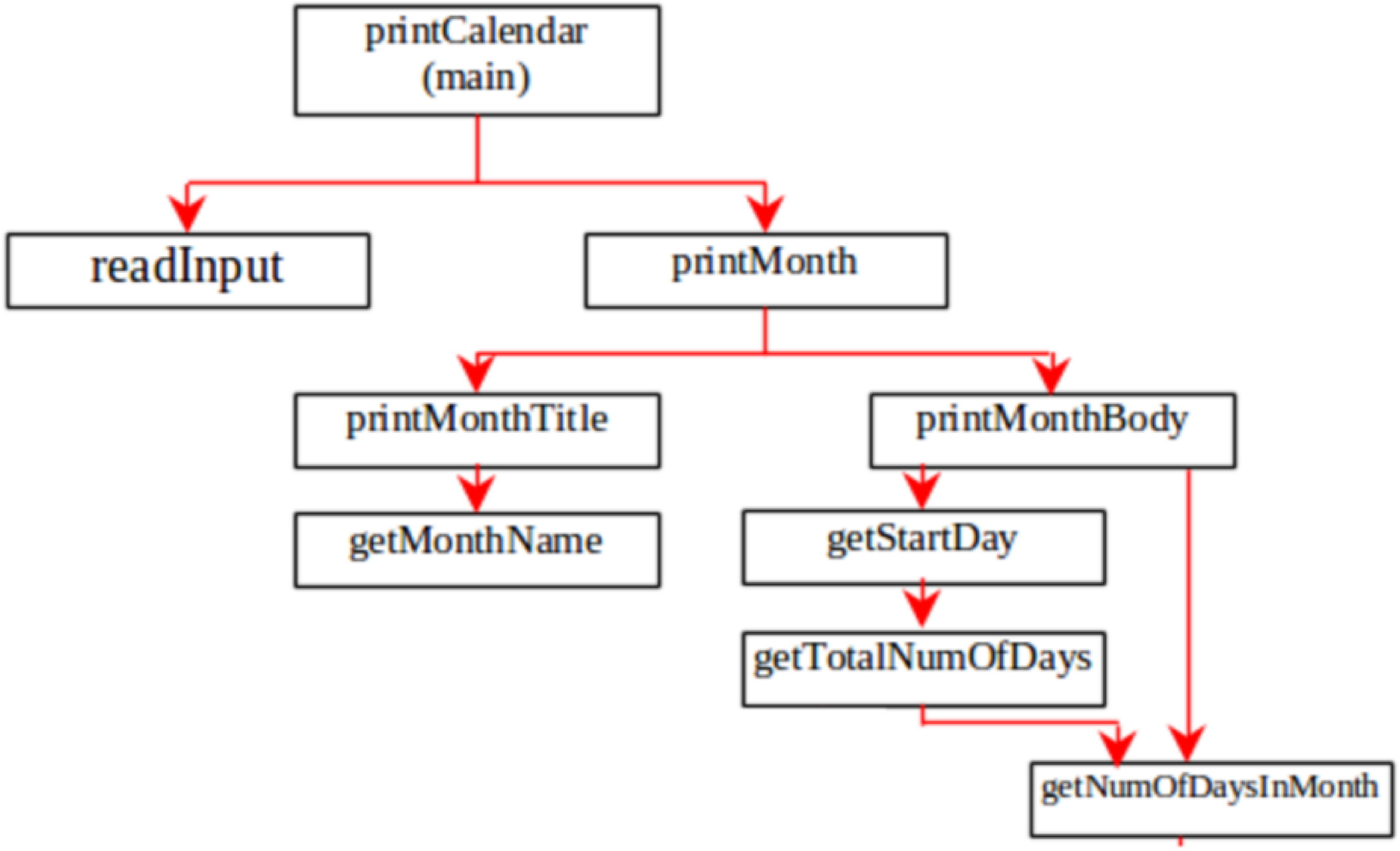
Design Diagram 3



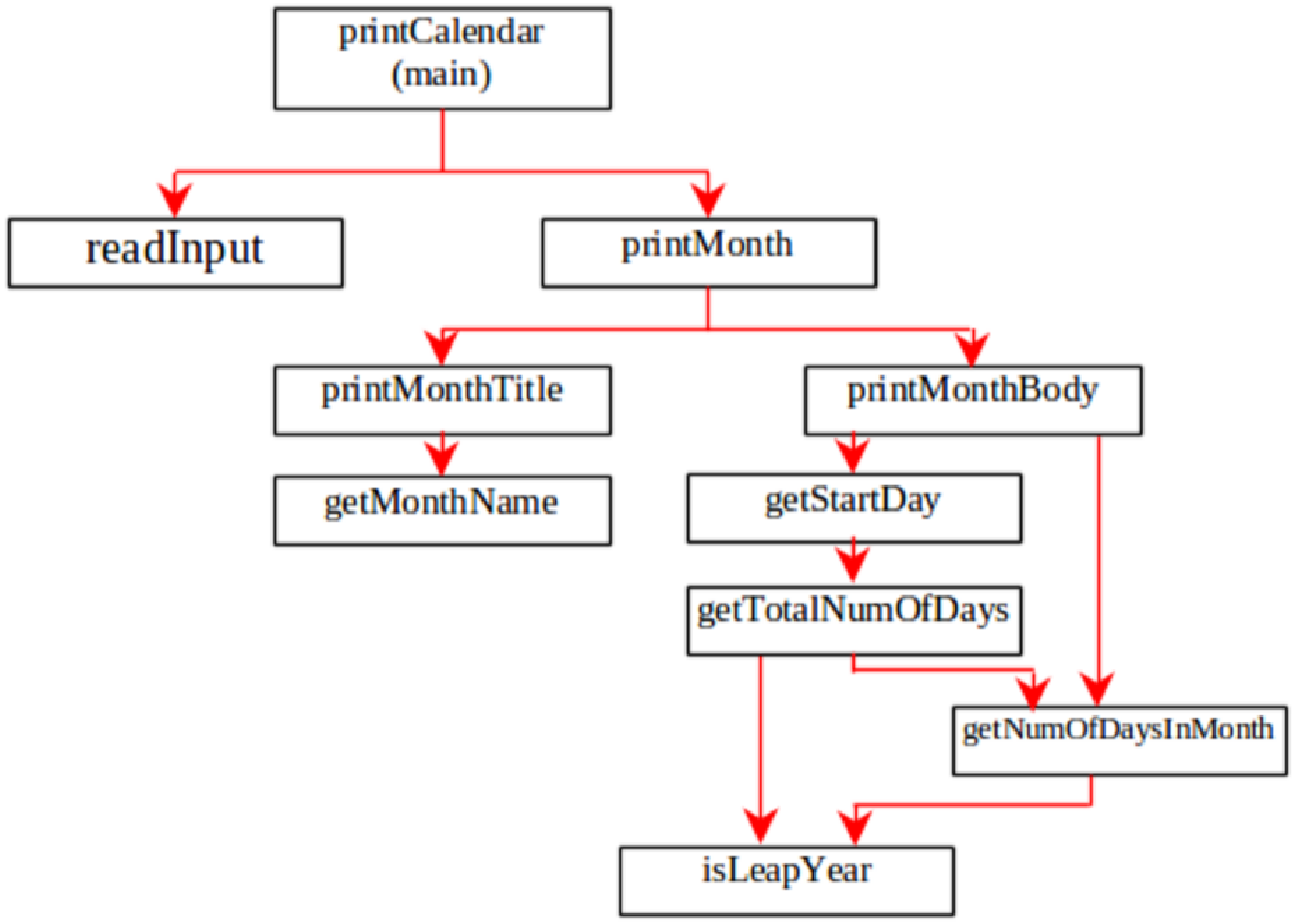
Design Diagram 4



Design Diagram 5



Design Diagram 6



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:

PrintCalendarSkeleton.py

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.

Implementation: Bottom-Up cont. (1)

The isLeapYear(year) function can be implemented using the following code:

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

Implementation: Bottom-Up cont. (2)

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.
- To implement getTotalNumberOfDays(year, month), you need to compute the total number of days (totalNumberOfDays) between January 1, 1800, and the first day of the calendar month. You could find the total number of days between the year 1800 and the calendar year and then figure out the total number of days prior to the calendar month in the calendar year. The sum of these two totals is totalNumberOfDays.
- To print the calendar's body, first pad some space before the start day and then print the lines for every week.

Solution

You can download a solution here: PrintCalendar.py

Benefit of stepwise refinement

This approach makes the program easier to write, reuse, debug, test, modify, and maintain.

- Simpler Program
- Reusing function
- Easier Developing, Debugging and Testing
- Better facilitating Teamwork

Exercise: credit card number validation (1)

Credit card numbers follow certain patterns: They must have between 13 and 16 digits, and the number must start with:

- 4 for Visa cards
- 5 for MasterCard credit cards
- 37 for American Express cards
- 6 for Discover cards

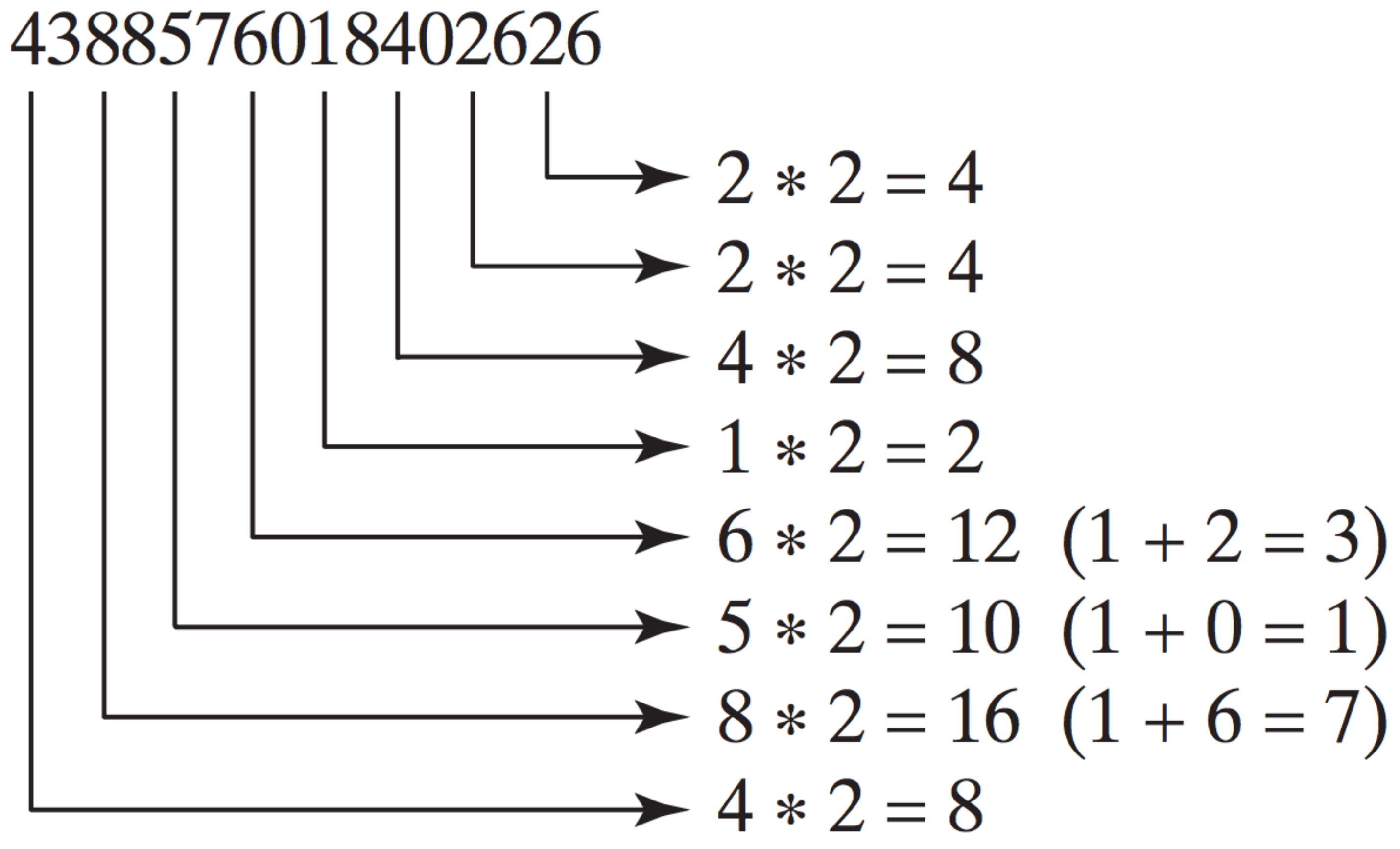
Exercise: credit card number validation (2)

In 1954, Hans Luhn of IBM proposed an algorithm for validating credit card numbers. The algorithm is useful to **determine whether a card number is entered correctly**.

Credit card numbers are generated following this validity check, commonly known as the Luhn check or the Mod 10 check, which can be described as follows (for illustration, consider the card number 4388576018402626):

Exercise: credit card number validation (3)

1. Double every second digit from right to left. If doubling of a digit results in a twodigit number, add up the two digits to get a singledigit number



Exercise: credit card number validation(4)

2. Now add all singledigit numbers from Step 1.
   4 + 4 + 8 + 2 + 3 + 1 + 7 + 8 = 37
3. Add all digits in the odd places from right to left in the card number.
   6 + 6 + 0 + 8 + 0 + 7 + 8 + 3 = 38
4. Sum the results from Steps 2 and 3.
   37 + 38 = 75
5. If the result from Step 4 is divisible by 10, the card number is valid; otherwise, it is invalid. For example, the number 4388576018402626 is invalid, but the number 4388576018410707 is valid.

Exercise: credit card number validation (5)

Write a program that prompts the user to enter a credit card number as an integer. Display whether the number is valid or invalid. Design your program to use the following functions:

```
#Return true if the card number is valid
def isValid(number):
#Get the result from Step2
def sumOfDoubleEvenPlace(number):
#Return this number if it is a single digit, otherwise,return
#the sum of the two digits
def getDigit(number):
#Return sum of odd place digits in number
def sumOfOddPlace(number):
#Return true if the digit d is a prefix for number
def prefixMatched(number,d):
#Return the number of digits in d
def getSize(d):
#Return the first k number of digits from number.If the
#number of digits in number is less than k, return number.
def getPrefix(number,k):
```

Solution

You can download a solution here:
credit_card_number_validation.py