Revised: 7/27/2020

Functions and Modules

Michael C. Hackett

Assistant Professor, Computer Science

Community College of Philadelphia

Lecture Topics

- Functions
- Local and Global Variables
- Parameters and Arguments
- Returning Data from Functions
- Recursive Functions
- Creating Modules
- Importing Select Functions from a Module
- The Random Module

Colors/Fonts

- Global Variable Names – Brown
- Local Variable Names – Lt Blue
- Literals – Blue
- Keywords – Orange
- Operators/Punctuation – Black
- Functions – Purple
- Parameters – Gold
- Comments – Gray
- Modules – Pink

Source Code – **Consolas**
Output – Courier New

Functions

- A ***function*** is a group of statements that are executed when called.
- Functions begin with the keyword **def** (short for define.)
- A parameter list is optional (but the parentheses are not.)
- Function names follow the same rules as variables.

```
def name(parameter list):
    #code that will be
    #executed
```

Indent one tab.

Creating a Main Function

- A common approach when developing a Python application is to place your program's primary statements and logic in a "main" function.
  - This ensures your program's primary code is in one place.

- Having a main function is not required, but it does help keep your code better organized.

Creating a Main Function

- Aside from naming the function “main”, there is nothing special that needs to be done.

```
def main():
    print("Hello World")

main()
```

```
Hello World
```

Functions

```
def main() :
  hello()       2
  goodbye()     4

def goodbye() :
  print("Goodbye World")  5

def hello() :
  print("Hello World")   3

main()  1
```

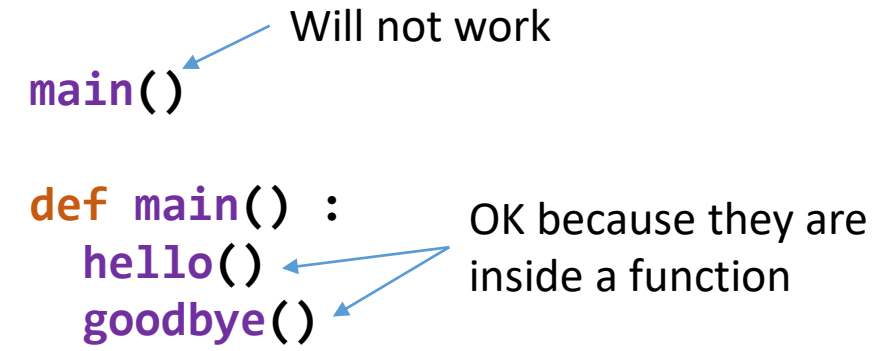
```
Hello World
Goodbye World
```

Functions

- A function must be defined before it can be called by statements outside of any function.

```
def main() :
  hello()
  goodbye()

def goodbye() :
  print("Goodbye World")

def hello() :
  print("Hello World")

main()
```

Correct Code




```
main()

def main() :
  hello()
  goodbye()

def goodbye() :
  print("Goodbye World")

def hello() :
  print("Hello World")
```

Incorrect Code

Global Variables

- A ***global variable*** is a variable that exists outside of a function.
  - We have been working only with global variables until this point.

- Global variables are normally declared at the beginning of the source code file.
  - Although, functions can access a global variable regardless of where it is declared.

Global Variables

```
global_variable = 3

def main() :
  print("global_variable main:", global_variable)
  test()

def test() :
  print("global_variable test:", global_variable)

main()
```

```
global_variable main: 3
global_variable test: 3
```

Local Variables

- A ***local variable*** is a variable exists only inside of a function.

- Local variables are inaccessible to code outside of the function.

Local Variables

```
global_variable = 3

def main() :
    local_variable = 8.6
    print("global_variable:", global_variable)
    print("local_variable:", local_variable)

main()
```

```
global_variable: 3
local_variable: 8.6
```

Global Variables Declared Locally

- A local variable can be made global using the **global** keyword.
  - Its declaration and assignment must be on separate lines.

```
global my_variable
my_variable = 75.4
```

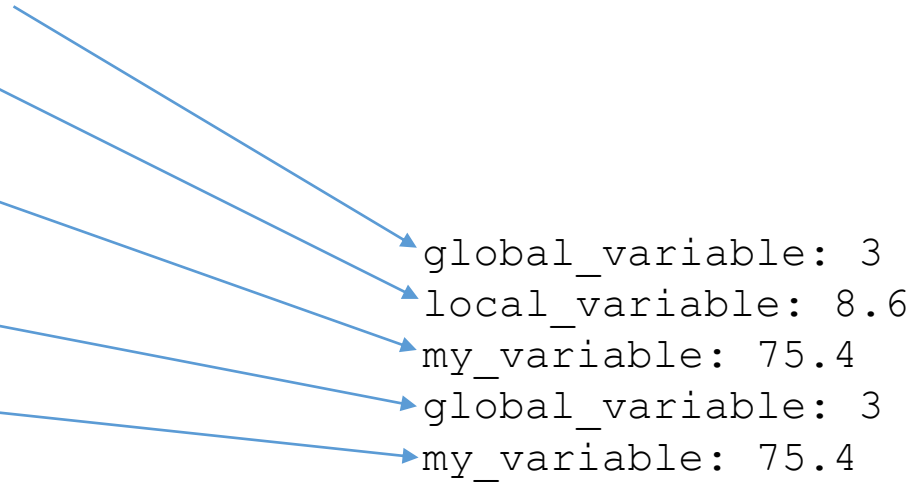
Global Variables Declared Locally

```
global_variable = 3

def main() :
  local_variable = 8.6
  global my_variable
  my_variable = 75.4
  print("global_variable:", global_variable)
  print("local_variable:", local_variable)
  print("my_variable:", my_variable)

print("global_variable:", global_variable)
print("my_variable:", my_variable)
main()
```

Will not work because its containing function has not been called yet.

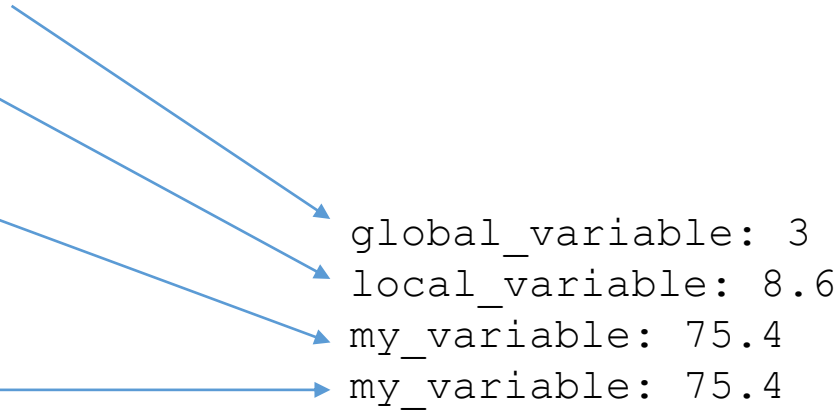
Global Variables Declared Locally

```
global_variable = 3

def main() :
  local_variable = 8.6
  global my_variable
  my_variable = 75.4
  print("global_variable:", global_variable)
  print("local_variable:", local_variable)
  print("my_variable:", my_variable)

main()
print("global_variable:", global_variable)
print("my_variable:", my_variable)
```



Global Variables Declared Locally

```
global_variable = 3

def main() :
  local_variable = 8.6
  global my_variable
  my_variable = 75.4
  print("global_variable:", global_variable)
  print("local_variable:", local_variable)
  print("my_variable:", my_variable)
  test()

def test() :
  print("my_variable:", my_variable)

main()
```



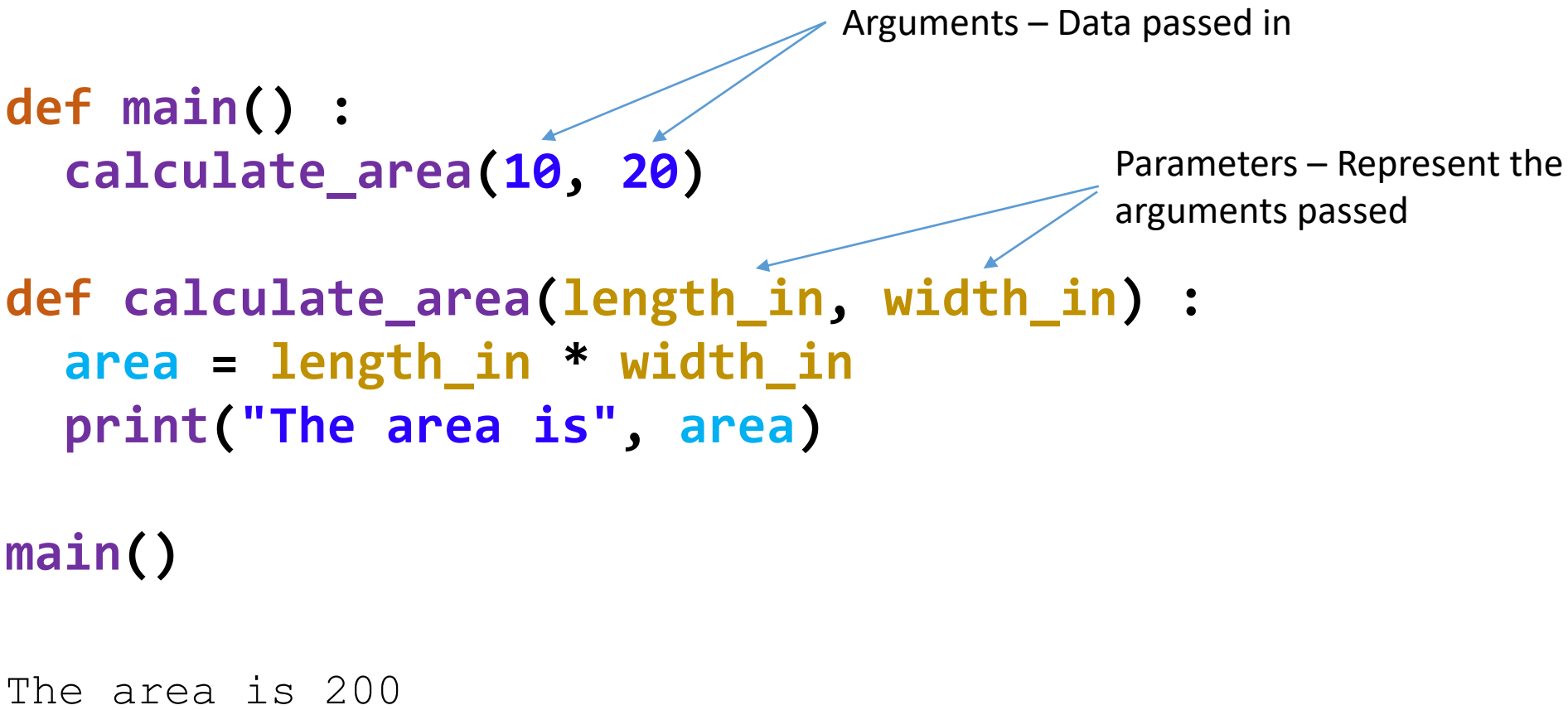
Parameters and Arguments

- Functions have the ability to accept arguments.
  - ***Arguments*** are data passed to a function

```
round(original_number)
format(temp, "f")
```

- The variables used in the source code of a function to represent the arguments are called ***parameters***.

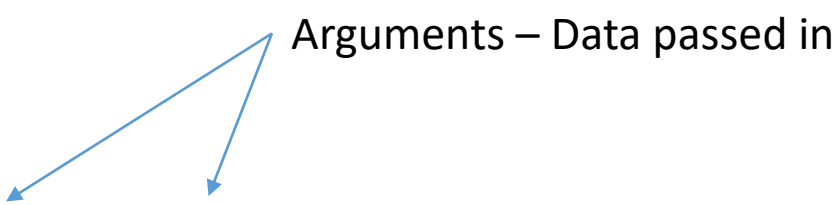
Parameters and Arguments




```
The area is 200
```

Parameters and Arguments

```
def main() :
  length = 10
  width = 20
  calculate_area(length, width)

def calculate_area(length_in, width_in) :
  area = length_in * width_in
  print("The area is", area)

main()
```




```
The area is 200
```

Parameters and Arguments

- The number of arguments must match the number of parameters.

```
def main() :
  calculate_area(10)

def calculate_area(length_in, width_in) :
  area = length_in * width_in
  print("The area is", area)

main()
```

Error

Returning Data From Functions

- A return statement allows a function to give data back when called.

```
def main() :
    area = calculate_area(10, 20)
    print("The area is", str(area))

def calculate_area(length_in, width_in) :
    area = length_in * width_in
    return area

main()
```

```
The area is 200
```

Note: The main function's local variable area and the calculate_area function's local variable area are two **different** variables.

They can have the same name because they are local to their respective function.

Returning Data From Functions

- Functions can have more than one return statement.

```
def main() :
  eligible = can_vote(22)
  print("Eligible to vote:", ("Yes" if eligible else "No"))

def can_vote(age_in) :
  if age_in >= 18 :
    return True
  else :
    return False

main()
```

Conditional Operation
(Shorthand if/else)

```
Eligible to vote: Yes
```

Void Functions

- A ***void function*** is one that does not return any data when called.

- Functions that have no return statement are implicitly void.

Void Functions

- There are two other ways to have a void function.
  - A statement that consists only of the return keyword

```
return
```

  - A return statement that returns None.

```
return None
```

    - The None keyword is used to indicate no reference to any data in memory.

Void Functions

- All three void functions below would behave identically.

```
def calculate_area(length_in, width_in) :
  area = length_in * width_in
  print("The area is", area)
```

```
def calculate_area(length_in, width_in) :
  area = length_in * width_in
  print("The area is", area)
  return
```

```
def calculate_area(length_in, width_in) :
  area = length_in * width_in
  print("The area is ", area)
  return None
```

Recursive Functions

- A ***recursive*** function is a function that calls itself.
  - Without any logic controlling the number of times it calls itself, it will call itself forever.

```
def main() :
  hello()

def hello() :
  print("Hello World")
  hello()

main()
```

```
Hello World
Hello World
Hello World
Hello World
Hello World
…
```

Recursive Functions

- Any task that can be completed using a repetitive algorithm can be solved using a recursive algorithm (and vice versa.)

```
def hello() :
  while True :
    print("Hello World")

hello()
```

```
def hello() :
  print("Hello World")
  hello()

hello()
```

Recursive Functions

- A function that prints a count down using a repetitive algorithm.

```
def main() :
  countdown(3)

def countdown(number_in) :
  for number in range(number_in, 0, -1) :
    print(number)

main()
```

```
3
2
1
```

Recursive Functions

- A similar function that again uses a repetitive algorithm.

```
def main() :
  countdown(3)

def countdown(number_in) :
  number = number_in
  while number > 0 :
    print(number)
    number -= 1

main()
```

```
3
2
1
```

Recursive Functions

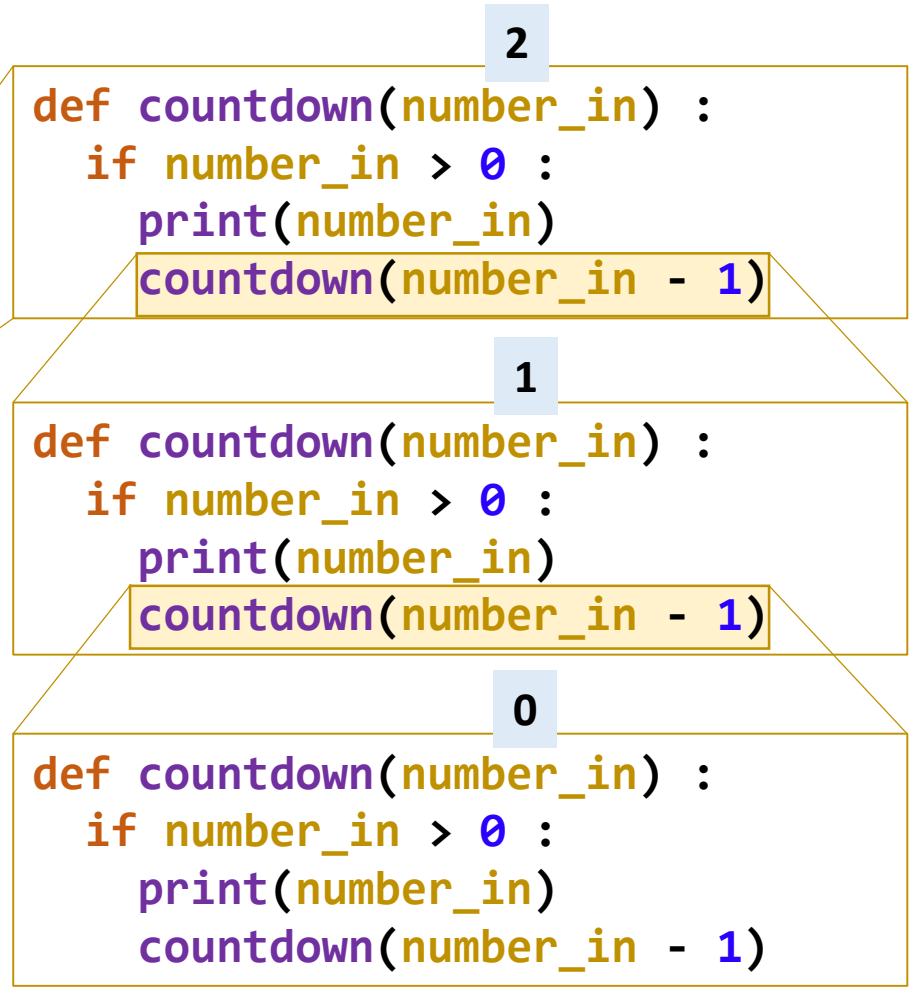
- A function that uses a recursive algorithm to print the count down.

```
def main() :
  countdown(3)

def countdown(number_in) :
  if number_in > 0 :
    print(number_in)
    countdown(number_in - 1)

main()
```

```
3
2
1
```

Recursive Functions

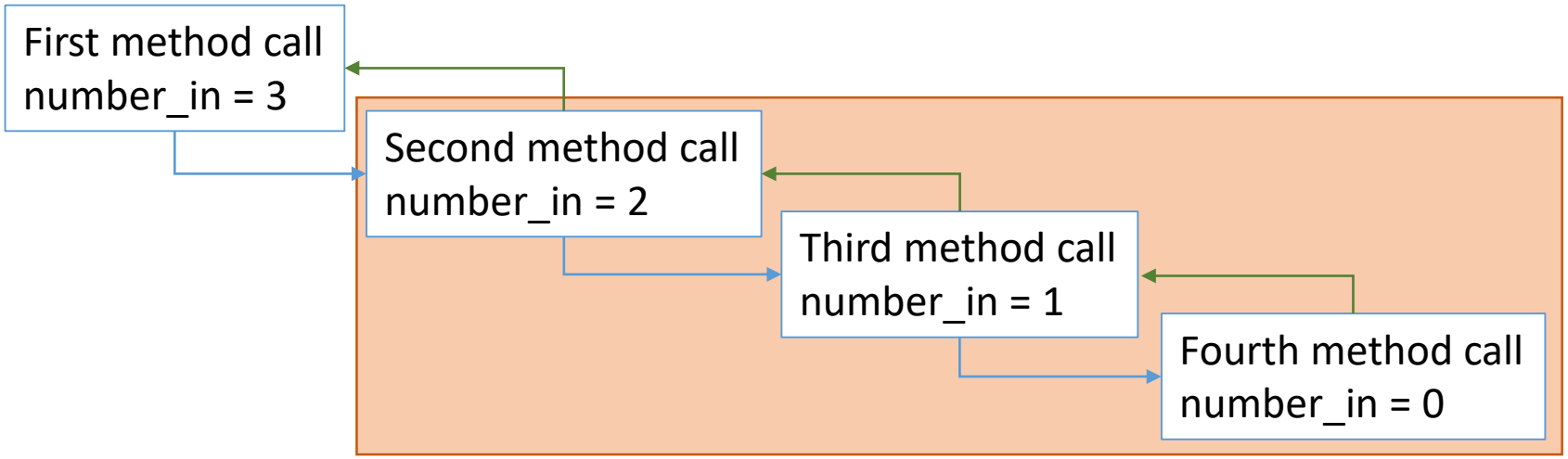
```
def main() :
    countdown(3)

                        3
def countdown(number_in) :
  if number_in > 0 :
    print(number_in)
    countdown(number_in - 1)

main()
```




```
3
2
1
```

Depth

- The number of times a function calls itself is the ***depth*** of recursion.



- In the previous example, the depth of recursion is 3 since the countdown function calls itself a total of three times.

Using Recursion to Solve a Factorial

- In mathematics, the notation *n!* represents the factorial of some number, *n*.
- The factorial of a non-negative number is defined by the following rules:
  - If n = 0 n! = 1
  - If n > 0 n! = n * n-1 * n-2 * … * 1

  - Examples:
    - 0! = 1
    - 1! = 1
    - 3! = 3 * 2 * 1 = 6
    - 6! = 6 * 5 * 4 * 3 * 2 * 1 = 720

Using Recursion to Solve a Factorial

- A function that uses a repetitive algorithm to return the factorial of a number.
  - 3! = 3 * 2 * 1 = 6

```
def main() :
  answer = factorial(3)
  print("3! =", number))

def factorial(n) :
  result = 1
  for number in range(n, 0, -1) :
    result *= number
  return result

main()
```

```
3! = 6
```

Using Recursion to Solve a Factorial

- Let's rewrite n! so we think of it as a function call than a mathematical expression:
  - The **base case** is when n = 0
  - The **recursive case** is when n > 0
  - If n = 0 factorial(n) = 1
  - If n > 0 factorial(n) = n * n-1 * n-2 * … * 1
    = **n * factorial(n-1)**

  - Examples:
    - factorial(0) = 1
    - factorial(1) = 1 * factorial(0) = 1 * 1 = 1
    - factorial(2) = 2 * factorial(1) = 2 * 1 = 2
    - factorial(3) = 3 * factorial(2) = 3 * 2 = 6

Using Recursion to Solve a Factorial

- A recursive function that returns the factorial of a number
  - Base Case: n = 0

```
def main() :
  answer = factorial(0)
  print("0! = " + str(answer))

def factorial(n) :
  if n > 0 :
    #Recursive Case
    return n * factorial(n - 1)
  else :
    #Base Case
    return 1

main()
```

```
0! = 1
```

Using Recursion to Solve a Factorial

- Recursive Case: n > 0

```
def main() :
  answer = factorial(3)
  print("3! =", answer)

def factorial(n) :
  if n > 0 :
    #Recursive Case
    return n * factorial(n - 1)
  else :
    #Base Case
    return 1

main()
```

3

```
3! = 6
```

2

```
def factorial(n) :
  if n > 0 :
    #Recursive Case
    return n * factorial(n - 1)
  else :
    #Base Case
    return 1
```

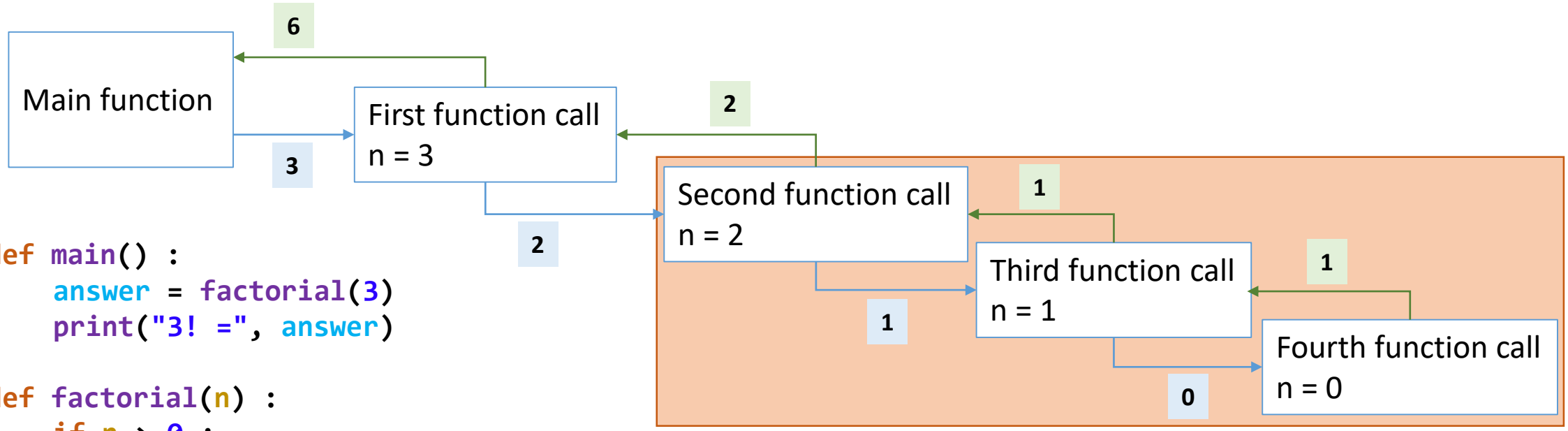
1

```
def factorial(n) :
  if n > 0 :
    #Recursive Case
    return n * factorial(n - 1)
  else :
    #Base Case
    return 1
```

0

```
def factorial(n) :
  if n > 0 :
    #Recursive Case
    return n * factorial(n - 1)
  else :
    #Base Case
    return 1
```

Depth




```
def main() :
    answer = factorial(3)
    print("3! =", answer)

def factorial(n) :
    if n > 0 :
        #Recursive Case
        return n * factorial(n - 1)
    else :
        #Base Case
        return 1

main()
```

Modules

- A ***module*** is a Python source code file that contains a collection of functions.
  - Modules can include statements outside of any particular function.
  - These statements are only executed once, when the module is imported.

- When the module is **imported** in another Python program, the module's functions can then be used in that program.

- This is useful for not having to rewrite the same functions across multiple programs.

Modules

- Python provides some standard modules, like the math module seen earlier in the course.

- When we import the math module into a Python program, the functions contained in the module are now available for use.

```
import math
original_number = 15.1
rounded_number = math.ceil(original_number)
print(rounded_number)
```

Creating a Module

- Not much needs to be done to create a module.

- Create a Python source code file that contains functions.
  - The functions can return data or be void.
  - The functions can take arguments.

- The source code file's name is the module's name.
  - If your module's filename is examplemodule.py, the module's name is examplemodule.

Creating a Module

rectangletools.py

```
def calculate_area(length_in, width_in) :
  area = length_in * width_in
  return area

def calculate_perimeter(length_in, width_in) :
  perimeter = 2 * length_in + 2 * width_in
  return perimeter
```

rectangletest.py

```
import rectangletools
def main() :
  length, width = 10, 20
  print("The area is", rectangletools.calculate_area(length, width))
  print("The perimeter is", rectangletools.calculate_perimeter(length, width))

main()
```

Importing Functions from a Module

- When we import a module, we gain access to all of the module's functions.
- Sometimes, we may only want access to a few of the module's functions.
  - To do this, we modify the import statement to include the **from** keyword.

```
from rectangletools import calculate_area
```

- This statement imports only the calculate_area function from the rectangletools module.

Importing Functions from a Module

rectangletools.py

```
def calculate_area(length_in, width_in) :
  area = length_in * width_in
  return area

def calculate_perimeter(length_in, width_in) :
  perimeter = 2 * length_in + 2 * width_in
  return perimeter
```

rectangletest.py

```
from rectangletools import calculate_area
def main() :
  length, width = 10, 20
  print("The area is", calculate_area(length, width))
  print("The perimeter is", rectangletools.calculate_perimeter(length, width))

main()
```

Importing Functions from a Module

rectangletools.py

```
def calculate_area(length_in, width_in) :
  area = length_in * width_in
  return area

def calculate_perimeter(length_in, width_in) :
  perimeter = 2 * length_in + 2 * width_in
  return perimeter
```

rectangletest.py

```
from rectangletools import calculate_area, calculate_perimeter
def main() :
  length, width = 10, 20
  print("The area is", calculate_area(length, width))
  print("The perimeter is", calculate_perimeter(length, width))

main()
```

Random Number Generators

- A ***random number*** is number chosen from a set of possible values, each with the same probability of being selected.

- A ***random number generator*** is software or hardware that produces a random number.

- A ***seed*** is a number provided to an algorithm to produce strings of random numbers.

Types of Random Number Generators

- A Pseudo-Random Number Generator (PRNG) uses a mathematical algorithm to generate random numbers.
  - Software RNGs.

- A True Random Number Generator (TRNG) uses an unpredictable physical means to generate random numbers.
  - Hardware RNGs.

Random Module

- Python's random module is a standard module that must be imported.

```
import random
```

- The random module's randint function gives us a random number from a specified range.

randint Function

```
import random

def main() :
   random_number = random.randint(1, 10)
   print("The randomly selected number is", random_number)

main()
```

```
The randomly selected number is (1-10)
```

- The randint function takes two arguments, the start of the range and the end of the range.
  - Both the start and end value is included in the range.

Setting the random module's seed

- The random module can accept a user-specified seed.

```
random.seed(10)
```

- The random module uses a PRNG.
  - Using the same seed will always return the same string of numbers.

Setting the random module's seed

```
import random
random.seed(10)

def main() :
  random_number = random.randint(1, 100)
  print("The randomly selected number is", random_number)

main()
```

```
The randomly selected number is 58
```