CN101

Lecture 6

Functions

Topics

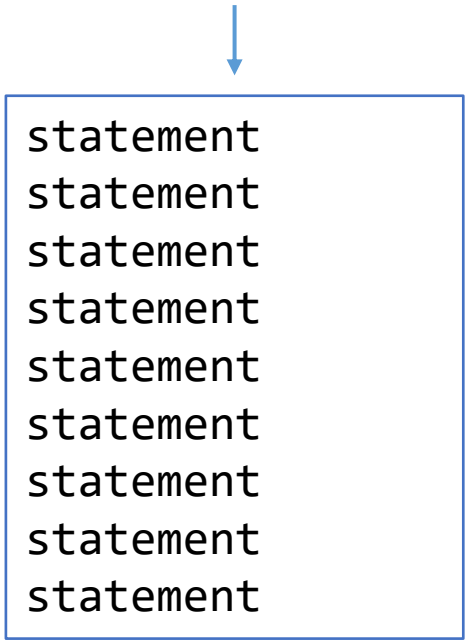
- Introduction to Functions
- Defining and Calling a Void Function
- Designing a Program to Use Functions
- Local Variables
- Passing Arguments to Functions
- Global Variables and Global Constants
- Introduction to Value-Returning Functions: Generating Random Numbers
- Writing Your Own Value-Returning Functions
- The `math` Module

Introduction to Functions

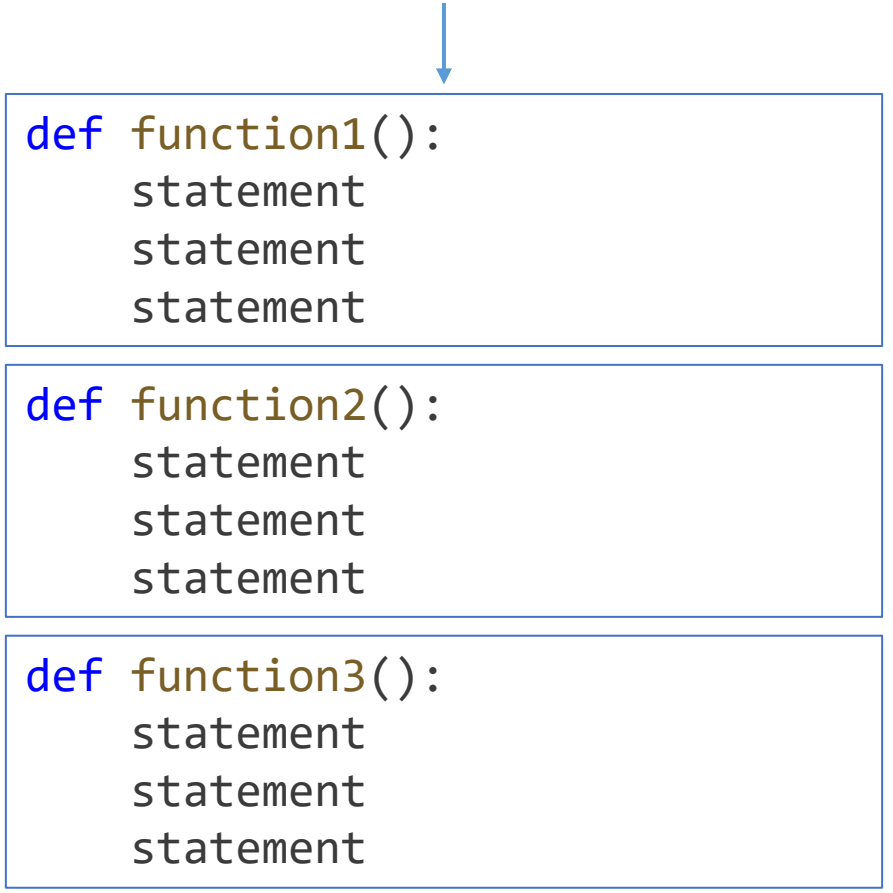
- Function: a group of statements within a program that perform a specific task
  - Usually one task of a large program
    - Functions can be executed in order to perform overall program task
  - Known as *divide and conquer* approach
- Modularized program: a program where each task within the program is in its own function

Using functions to divide and conquer a large task

This program is one long sequence of statements.



In this program, the task has been divided into smaller tasks. Each smaller task is performed by a separate function



Benefits of Modularizing a Program with Functions

- The benefits of using functions include:
  - Simpler code
  - Code reuse
    - write the code once and call it multiple times
  - Better testing and debugging
    - Can test and debug each function individually
  - Faster development
  - Easier facilitation of teamwork
    - Different team members can write different functions

Defining a Function

- Consist of 2 parts
  - a function header and a function body (block)

```
def function_name(optional_list_of_parameters):
    statement
    statement
    …
```

- Function header: first line of function
  - Includes keyword `def` and `function_name`, followed by `optional_list_of_parameters` within parentheses, and a colon
- Function body (Block):
  - indented set of statements that belong together as a group

Function Names

- Function naming rules:
  - Cannot use key words as a function name
  - Cannot contain spaces
  - First character must be a letter or underscore
  - All other characters must be a letter, number or underscore
  - Uppercase and lowercase characters are distinct, i.e. case sensitive

Defining a Function (cont'd)

- Function name should be descriptive of the task carried out by the function
  - Often includes a **verb**
- Function definition: specifies what function does

```
def function_name():
    statement
    statement
    …
```

function definition

```
def message():
    print('I am Arthur.')
    print('King of the Britons.')
```

function definition

Void Functions and Value-Returning Functions

- A void function:
  - Executes the statements it contains
  - then terminates.
  - Technically returns `None`
- A value-returning function:
  - Executes the statements it contains
  - then returns a value back to the statement that called it.
    - Examples of value-returning functions: `input`, `int`, `float`
  - Always contains at least one or more `return` statements
  - A return statement
    - a statement with `return` keyword with an optional expression
  - if `return` is without any expression, default to return `None`

Example: void functions (1)

void_function_1.py

```
1  # examples of void functions
2  def print_name():
3      print("You name is: unknown")
4
5  def print_len():
6      print("Length of your name: 7")
7
8  def do_something():
9      a, b = 20, 30
10     print(f"{a} + {b} = {a + b}")
11
12 def do_nothing():
13     # pass statement – does nothing when executed
14     pass
```

Example: void functions (2)

void_function_2.py

```
 1  # examples of void functions
 2  def func_xxx():
 3      print("This is a function.")
 4
 5  def func_yyy():
 6      print("This is a function.")
 7
 8  # Both func_xxx and func_yyy are essentially the same
 9  functions
10  # since they both
11  # - are void functions
12  # - same parameters (no parameters)
13  # - have the same body that give the same result
```

Example: value-returning functions (1)

value_returning_function_1.py

```
 1  # examples of value-returning functions
 2  def get_name():
 3      name = "unknown"
 4      return name
 5
 6  def get_len():
 7      return 7
 8
 9  def add_something():
10      a, b = 20, 30
11      return a + b
12
13  def do_nothing():
14      # default to return None
15      return
```

Example: value-returning functions (2)

value_returning_function_2.py

```
1  # examples of value-returning functions
2  def get_xxx():
3      a, b = 10, 20
4      return a + b
5
6  def get_yyy():
7      x, y = 10, 20
8      return x + y
9
10 print(f"{get_xxx()}")
11 print(f"{get_yyy()}")
12
13 # Both get_xxx and get_yyy are essentially the same functions
14 # since they both
15 # - are void functions
16 # - same parameters (no parameters)
```

Calling a Function

- Call a function to execute it
  - When a function is called:
    - Interpreter jumps to the function and executes statements in the block (body of the function)
    - Interpreter jumps back to the part of the program where the function is called
      - Known as function return

function_demo.py

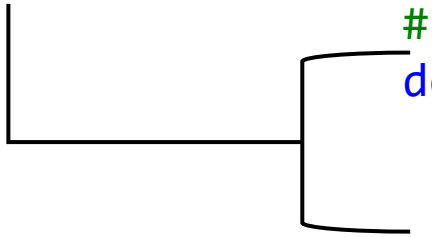
```
1  # This program demonstrates a function.
2  # First, we define a function named message.
3  def message():
4      print('I am Arthur')
5      print('King of the Britons')
6
7  # Call the message function.
8  message()
```

Program output

```
I am Arthur
King of the Britons
```

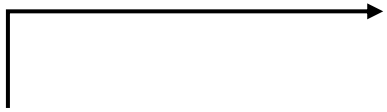
Function Definition and Function Call

These statements cause the message function to be created.



```
# This program demonstrates a function.
# First, we define a function named message.
def message():
    print('I am Arthur')
    print('King of the Britons')

# Call the message function.
message()
```



This statement calls the message function causing it to execute.

Defining and Calling a Function (cont'd)

- function `main`:
  - Is only a convention to indicate
    - the *mainline logic* of a program
  - Has no special meaning in Python
  - Is normally used to call other functions when they are needed

two_functions.py

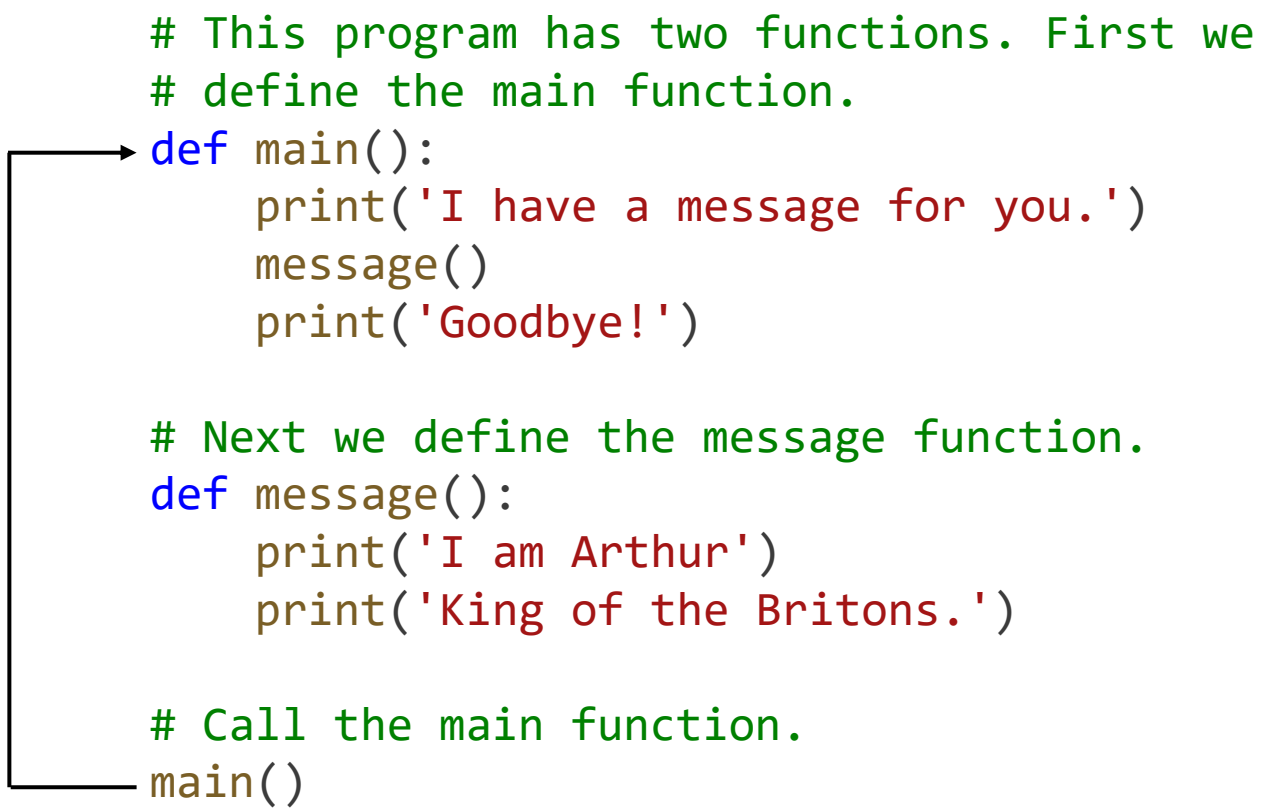
```
 1  # This program has two functions. First we
 2  # define the main function.
 3  def main():
 4      print('I have a message for you.')
 5      message()
 6      print('Goodbye!')
 7
 8  # Next we define the message function.
 9  def message():
10      print('I am Arthur')
11      print('King of the Britons.')
12
13  # Call the main function.
14  main()
```

Program output

```
I have a message for you.
I am Arthur
King of the Britons.
Goodbye!
```

Calling the `main` function

The interpreter jumps to the `main` function and begins executing the statements in its block.



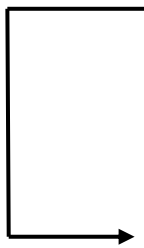
```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur')
    print('King of the Britons.')

# Call the main function.
main()
```

Calling the `message` function

The interpreter jumps to the `message` function and begins executing the statements in its block.



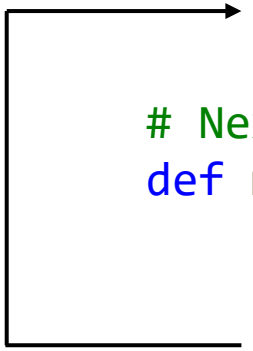
```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur')
    print('King of the Britons.')

# Call the main function.
main()
```

The `message` function returns

When the `message` function ends, the interpreter jumps back to the part of the program that called it and resumes execution from that point.



```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur')
    print('King of the Britons.')

# Call the main function.
main()
```

The `main` function returns

When the `main` function ends, the interpreter jumps back to the part of the program that called it. There are no more statements, so the program ends.

```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur')
    print('King of the Britons.')

# Call the main function.
main()
```

Indentation in Python

- Each block must be indented
  - Lines in block must begin with the same number of spaces
    - Use tabs or spaces (prefer spaces) to indent lines in a block, but not both as this can confuse the Python interpreter
    - IDLE automatically indents the lines in a block
  - Blank lines that appear in a block are ignored

The last indented line is the last line in the block.

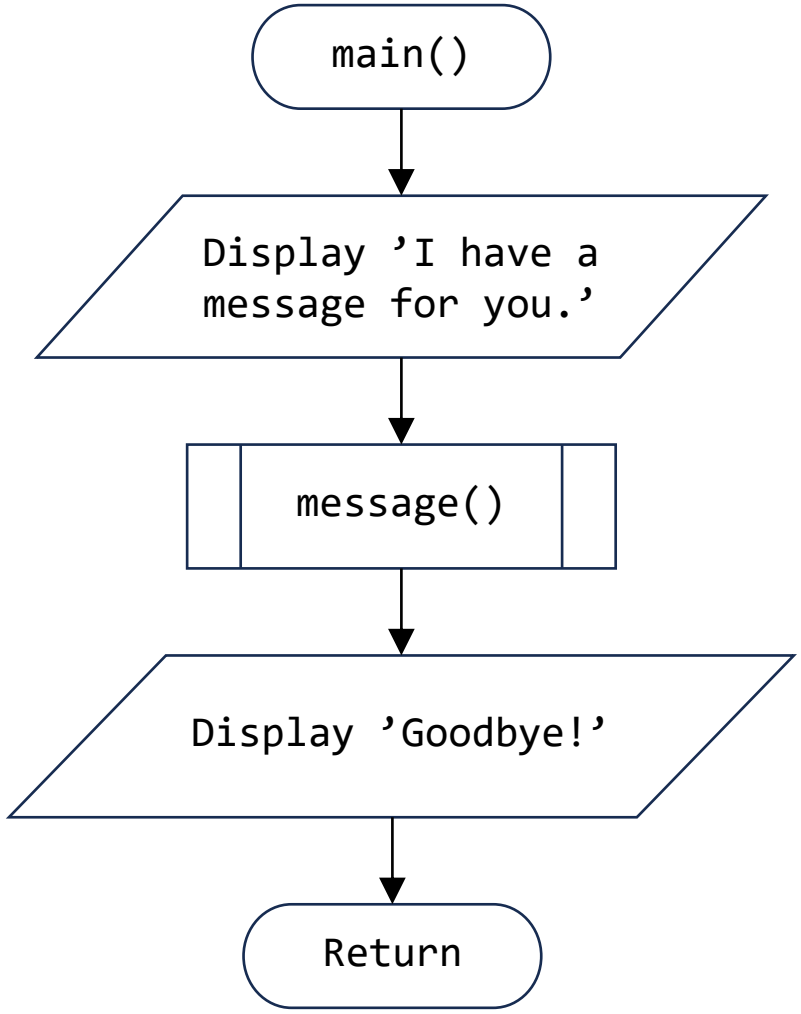
These statements are not in the block.

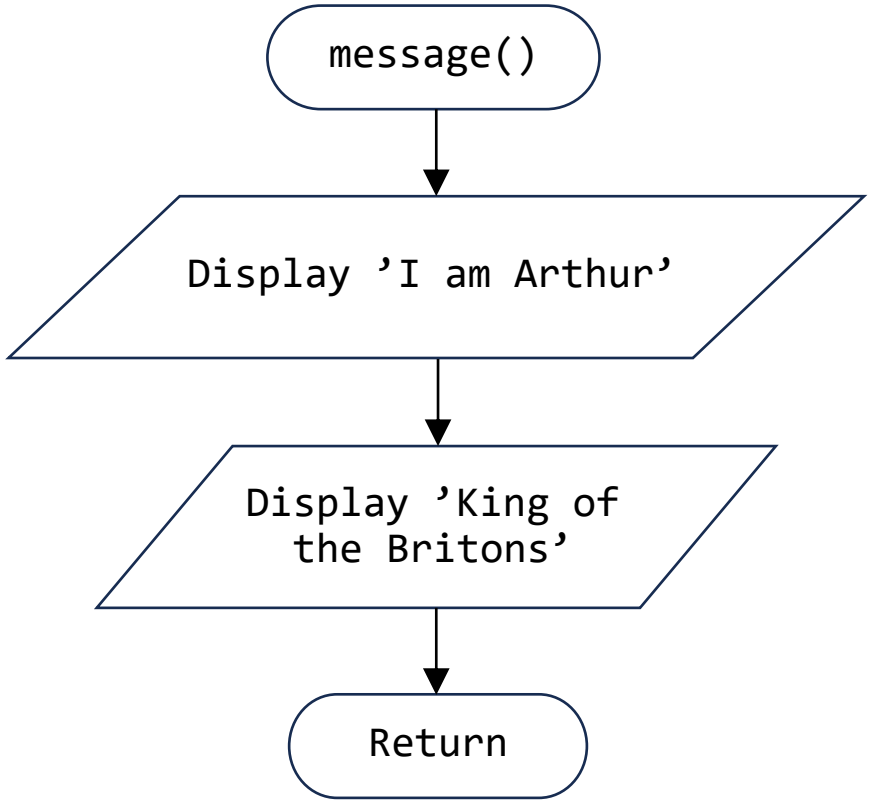
```
def greeting():
    print('Good morning!')
    print('Today we will learn about functions')

print('I will call the greeting function.')
greeting()
```

Designing a Program to Use Functions

- In a flowchart, function call is shown as rectangle with vertical bars at each side
  - Function name written in the symbol
  - Typically draw separate flow chart for each function in the program
    - End terminal symbol usually reads `Return`

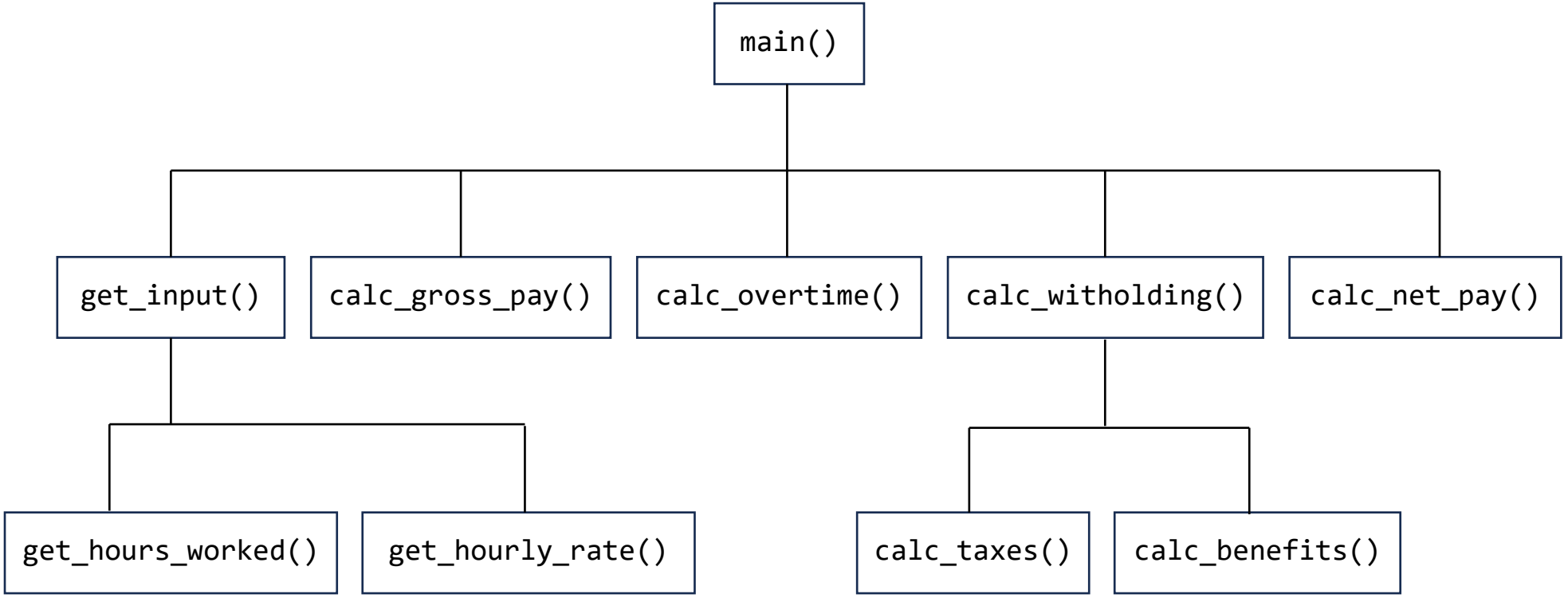

main()
Display ’I have a message for you.’
message()
Display ’Goodbye!’
Return


message()
Display ’I am Arthur’
Display ’King of the Britons’
Return

Designing a Program to Use Functions (cont'd)

- Top-down design: technique for breaking algorithm into functions
- Hierarchy chart: depicts relationship between functions
  - AKA structure chart
  - Box for each function in the program, Lines connecting boxes illustrate the functions called by each function
  - Does not show steps taken inside a function
- Use `input` function to have program wait for user to press enter

A hierarchy chart



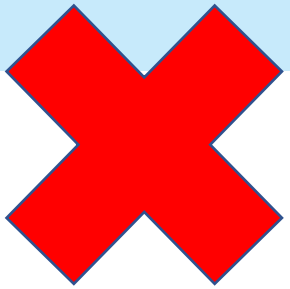
Local Variables

- Local variable: variable that is assigned a value inside a function
  - Belongs to the function in which it was created
    - only statements inside that function can access it
    - error will occur if another function tries to access the variable
- Scope: the part of a program in which a name may be accessed
  - For local variable:
    - within the function that the variable is created

Local Variables (cont'd)

- Local variable can only be accessed by
  - statements inside its function, after the variable is created
- Different functions may have local variables with the same name
  - Each function does not see the other function's local variables

bad_local.py



```
1  # Definition of the main function.
2  def main():
3      get_name()
4      print('Hello', name)     # This causes an error!
5
6  # Definition of the get_name function.
7  def get_name():
8      name = input('Enter your name: ')
9
10 # Call the main function.
11 main()
```

- Error in function `main` because variable `name` is not defined before use.
- Variable `name` in function `get_name` is defined in function `get_name`.
- Another variable `name` in function `main` is used in function `main`.
- Both variables `name` are not the same variables.

birds.py

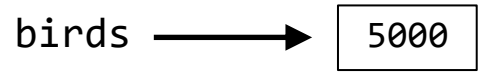
```
1   # This program demonstrates two functions that
2   # have local variables with the same name.
3
4   def main():
5       # Call the texas function.
6       texas()
7       # Call the california function.
8       california()
9
10  # Definition of the texas function. It creates
11  # a local variable named birds.
12  def texas():
13      birds = 5000
14      print('texas has', birds, 'birds.')
15
16  # Definition of the california function. It also
17  # creates a local variable named birds.
18  def california():
19      birds = 8000
20      print('california has', birds, 'birds.')
21
22  # Call the main function.
23  main()
```

Program output

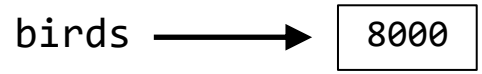
```
texas has 5000 birds.
california has 8000 birds.
```

Each function has its own `birds` variable

```
def texas():
    birds = 5000
    print('texas has', birds, 'birds. ')
```




```
def california():
    birds = 8000
    print('california has', birds, 'birds.')
```



Example: Local variables (1)

local_variable_1.py

```
1  # each function has its own local variable "x"
2  def a1():
3      # a local variable "x" in function a1()
4      x = 10
5      print(f"In function a1: x = {x}")
6
7  def b1():
8      # another local variable "x" in function b1()
9      x = 20
10     print(f"In function b1: x = {x}")
11
12 def main():
13     a1()
14     b1()
15     a1()
16     b1()
17
18 main()
```

Example: Local variables (2)

local_variable_2.py

```
1  # variable can only be accessed after it is defined
2  def a2():
3      x = 10
4      # can use variable x after it is defined
5      print(f"In function a2: x = {x}")
6
7  def b2():
8      # this will cause an error because variable x is not defined yet
9      # take note of the error message
10     print(f"In function b2: x = {x}")
11     x = 20
12
13 def main():
14     a2()
15     b2() # this function call will cause an error
16
17 main()
```

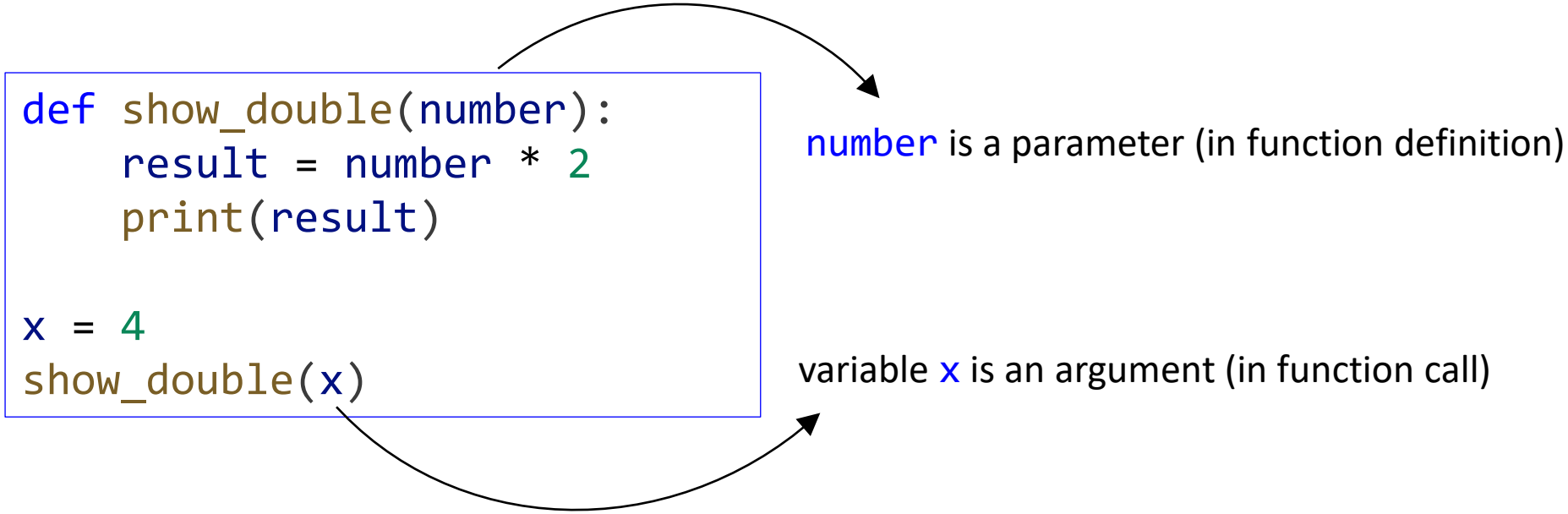
Example: Local variables (3)

local_variable_3.py

```
1  # a function can define any number of local variables
2  def a3():
3      x = 10          # define a local variable x
4      print(f"In function a3: x = {x}")
5
6      y = 4.5         # define another local variable y
7      print(f"In function a3: y = {y}")
8
9      z = [1,2,3]     # define another local variable z
10     print(f"In function a3: z = {z}")
11
12 def main():
13     a3()
14
15 main()
```

Passing Arguments to Functions

- Argument: piece of data that is sent into a function
  - Function can use argument in calculations
  - When calling the function, the argument is placed in parentheses following the function name



Passing Arguments to Functions (cont'd)

- Parameter variable: variable that is assigned the value of an argument when the function is called
  - The parameter and the argument reference the same value
  - General format:

```
def function_name(list_of_parameters):
```

  - Scope of a parameter: within the function in which the parameter is used

pass_arg.py (Chapter 5)

```
1   # This program demonstrates an argument being
2   # passed to a function.
3
4   def main():
5       value = 5
6       show_double(value)
7
8   # The show_double function accepts an argument
9   # and displays double its value.
10  def show_double(number):
11      result = number * 2
12      print(result)
13
14  # Call the main function.
15  main()
```

Program output

```
10
```

The `value` variable is passed as an argument

```
def main():
    value = 5
    show_double(value)
                  ↓
def show_double(number):
    result = number * 2
    print(result)
```

The `value` variable and the `number` parameter reference the same value

```
def main():
    value = 5
    show_double(value)   value → 5
                                 ↑
def show_double(number): number ─┘
    result = number * 2
    print(result)
```

Example: Single parameter (1)

single_parameter_1.py

```
1  # void functions with a single parameter
2  def c1(x):
3      # x is a parameter
4      print(f"In function c1: x = {x}")
5
6  def main():
7      c1(1)       # argument is 1
8
9      m = 2
10     c1(m)       # argument is m
11
12     n = m*m
13     c1(n)       # argument is n
14
15     c1(1+m+n)   # argument is 1+m+n
16
17 main()
```

Example: Single parameter (2)

single_parameter_2.py

```
1  # void functions with a single parameter
2  def c2(x):
3      # x is a parameter
4      print(f"In function c2: x = {x}")
5
6  def main():
7      c2([1,2,3])  # argument is [1,2,3]
8
9      m = [4,5,6,7]
10     c2(m)        # argument is m
11
12     n = (7,8,9,10,11)
13     c2(n)        # argument is n
14
15 main()
```

Example: Single parameter (3)

single_parameter_3.py

```
1  # void functions with a single parameter
2  def c3(x):
3      # x is a parameter
4      y = len(x)
5      print(f"Length is => {y}")
6
7  def main():
8      c3("Message") # argument is "Message"
9
10     m = [1,2,3]
11     c3(m)         # argument is m
12
13     n = (4,5,6,7)
14     c3(n)         # argument is n
15
16 main()
```

Example: Single parameter (4)

single_parameter_4.py

```
1  # void functions with a single parameter
2  def c4(x):
3      # x is a parameter
4      y = len(x)  # x must have value that can be used with function len
5      print(f"Length is => {y}")
6
7  def main():
8      c4(1)       # argument is 1, this function call will cause an error
9
10     m = True
11     c4(m)       # argument is m, this function call will cause an error
12
13     n = 3.14159
14     c4(n)       # argument is n, this function call will cause an error
15
16 main()
```

Example: Single parameter (5)

single_parameter_5.py

```
1  # void functions with a single parameter
2  def c5(x):
3      # x is a parameter
4      y = sorted(x)
5      print(f"Min = {y[0]}, Max = {y[-1]}, Length is => {len(y)}")
6
7  def main():
8      c5([1, 3, 5, 7, 3, 5, 1])
9
10     c5([2020, -5, -10, 8, 12])
11
12     m = (3.5, 1.5, -2.7, -9, 11, 20.2)
13
14     c5(m)
15
16 main()
```

Exercise: Single parameter (1)

Write a function `print_digit()` which

- takes a single string parameter
- prints all characters that are decimal digits, i.e. 0 - 9

```
def print_digit(a_string):
    # complete the body of this function
    pass

print_digit('24 hours in 1 day, 7 days in a 1 week.')
message = 'I met a man with 7 wives. Each wife had 7 sacks.'
print_digit(message)
```

```
# output
>2 4 1 7 1
>7 7
```

Exercise: Single parameter (2)

Write a function `print_even()` which

- takes a list of integer numbers as a parameter
- prints all numbers that are even

```
def print_even(a_list):
    # complete the body of this function
    pass

print_even([1, 2, 3, 4, 5])
list_1 = [1, 3, 5, 7, 9, 11, 13, 15, 17, 20, 21, 20]
list_2 = list(range(20, 0, -3))
print_even(list_1)
print_even(list_2)
```

```
# output
>2 4
>20 20
>20 14 8 2
```

Passing Multiple Arguments

- Python allows writing a function that accepts multiple arguments
  - Parameter list replaces single parameter
    - Parameter list items are separated by comma
- Positional arguments are passed *by position* to corresponding parameters
  - First parameter receives value of first argument
  - Second parameter receives value of second argument
  - etc.

multiple_args.py

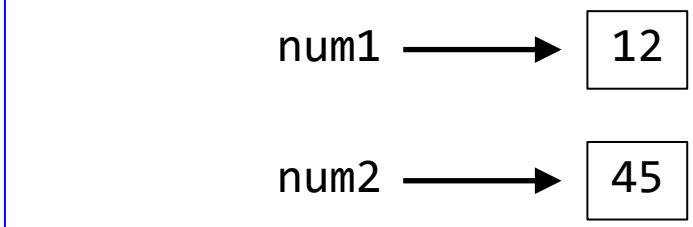
```
1   # This program demonstrates a function that accepts
2   # two arguments.
3
4   def main():
5       print('The sum of 12 and 45 is')
6       show_sum(12, 45)
7
8   # The show_sum function accepts two arguments
9   # and displays their sum.
10  def show_sum(num1, num2):
11      result = num1 + num2
12      print(result)
13
14  # Call the main function.
15  main()
```

Program output

```
The sum of 12 and 45 is
57
```

```
def main():
    print('The sum of 12 and 45 is')
    show_sum(12, 45)

def show_sum(num1, num2):
    result = num1 + num2
    print(result)
```



string_args.py

```
1  # This program demonstrates passing two string
2  # arguments to a function.
3
4  def main():
5      first_name = input('Enter your first name: ')
6      last_name = input('Enter your last name: ')
7      print('Your name reversed is')
8      reverse_name(first_name, last_name)
9
10 def reverse_name(first, last):
11     print(last, first)
12
13 # Call the main function.
14 main()
```

Program output (with input shown underlined)

```
Enter your first name: Matt
Enter your last name: Hoyle
Your name reversed is
Hoyle Matt
```

Example: Multiple parameters (1)

multiple_parameters_1.py

```
1  # void functions with multiple parameters
2  def d1(x, y):
3      # x and y are parameters
4      print(f"{x} + {y} => {x + y}")
5
6  def main():
7      d1(1, 2)      # arguments are 1 and 2
8
9      m = 3
10     n = 4
11     d1(m, n)      # arguments are m and n
12
13     d1(m*m, n*n)  # arguments are m*m and n*n
14
15 main()
```

Example: Multiple parameters (2)

multiple_parameters_2.py

```
1   # void functions with multiple parameters
2   def d2(x, y):
3       # x and y are parameters
4       print(f"{x} + {y} => {x + y}")
5
6   def main():
7       d2(1, True)        # arguments are 1 and True
8
9       m = 3.14159
10      d2(m, 5)           # arguments are m and 5
11
12      m = 1
13      d2((m+1)*2, 2*m)   # arguments are (m+1)*2 and 2*m
14
15  main()
```

Example: Multiple parameters (3)

multiple_parameters_3.py

```
 1  # void functions with multiple parameters
 2  def d3(x, y):
 3      # x and y are parameters
 4      print(f"{x} + {y} => {x + y}")
 5
 6  def main():
 7      d3([1, 2], [3, 4, 5])  # arguments are [1, 2] and [3, 4, 5]
 8
 9      m = [6, 7]
10      n = [8, 9, 10]
11      d3(m, n)    # arguments are m and n
12
13      d3(n, m)    # arguments are n and m (note the order of n and m)
14
15  main()
```

Example: Multiple parameters (4)

multiple_parameters_4.py

```
1  # void functions with multiple parameters
2  def d4(x, y):
3      # x and y are parameters
4      print(f"{x} + {y} => {x + y}")
5
6  def main():
7      m = [1, 2]
8      n = [3, 4, 5]
9      d4(m, n)            # arguments are m and n
10
11     d4(len(m), len(n)) # arguments are len(m) and len(n)
12
13     d4(min(m), max(n)) # arguments are min(m) and max(n)
14
15 main()
```

Exercise: Multiple parameters (1)

Write a function `print_sqare()` which

- takes 2 integers, `x` and `y` as parameters
- prints the sum of the squares of the 2 integers

```
def print_square(x, y):
    # complete the body of this function
    pass

print_square(3, 4)
a, b = 7, 9
print_square(a, b)
```

```
# output
>9 + 16 = 25
>49 + 81 = 130
```

Exercise: Multiple parameters (2)

Write a function `print_less()` which

- takes an integer, `x` and a list on integers, `a_list` as parameters
- prints all numbers in the list `a_list` that are less than `x`

```
def print_less(x, a_list):
    # complete the body of this function
    pass

print_less(50, [50, 51, 99, 79, 47, 83, 90, 39, 90, 25])
a, list_1 = 55, list(range(30, 100, 15))
print_less(a, list_1)
```

```
# output
>47 39 25
>30 45
```

Making Changes to Parameters

- Changes made to a parameter value within the function do not affect the argument
  - Known as pass by value
  - Provides a way for unidirectional communication between one function and another function
    - Calling function can communicate with called function

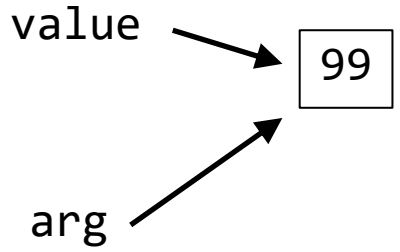
change_me.py

```
1  # This program demonstrates what happens when you
2  # change the value of a parameter.
3
4  def main():
5      value = 99
6      print('The value is', value)
7      change_me(value)
8      print('Back in main the value is', value)
9
10 def change_me(arg):
11     print('I am changing the value.')
12     arg = 0
13     print('Now the value is', arg)
14
15 # Call the main  function.
16 main()
```

Program output

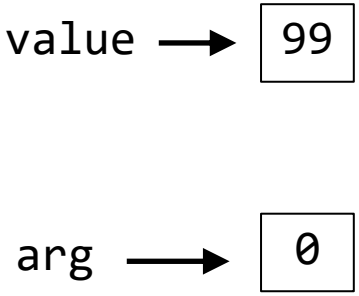
```
The value is 99
I am changing the value.
Now the value is 0
Back in main the value is 99
```

```
4   def main():
5       value = 99
6       print('The value is', value)
7       change_me(value)
8       print('Back in main the value is', value)
9
10  def change_me(arg):
11      print('I am changing the value.')
12      arg = 0
13      print('Now the value is', arg)
```



The `value` variable passed to the `change_me` function cannot be changed by it

```
4   def main():
5       value = 99
6       print('The value is', value)
7       change_me(value)
8       print('Back in main the value is', value)
9
10  def change_me(arg):
11      print('I am changing the value.')
12      arg = 0
13      print('Now the value is', arg)
```



Default Arguments

- Function argument can be given default value
  - In function definition, parameters can be assigned default.
- Default arguments must be given after all positional parameters
- All default arguments must come after any positional argument

Example: Default arguments (1)

default_args_1.py

```python
def production_cost(labor_cost, material_cost=1000, fixed_cost=1000):
    total_cost = material_cost + labor_cost + fixed_cost
    return total_cost

def main():
    total_cost = production_cost(500)
    print(f"Total Production Cost: {total_cost}")

    total_cost = production_cost(500, 500)
    print(f"Total Production Cost: {total_cost}")

    total_cost = production_cost(500, 500, 500)
    print(f"Total Production Cost: {total_cost}")

main()
```

Example: Default arguments (2)

default_args_2.py

```
1  # function with 3 parameters
2  # 2 parameters are assigned default values
3  def ask_ok(prompt, retries=4, reminder='Please try again!'):
4      while True:
5          ok = input(prompt)
6          if ok in ('y', 'ye', 'yes'):
7              return True
8          if ok in ('n', 'no', 'nop', 'nope'):
9              return False
10         retries = retries - 1
11         if retries <= 0:
12             print('invalid user response')
13             break
14         print(reminder)
15
16 ask_ok('Enter yes or no: ')  # call the function
```

Keyword Arguments

- Keyword argument: argument that specifies which parameter the value should be passed to
  - Position when calling function is irrelevant
  - General Format:

```
def function_name(parameter=value)
```

- Possible to mix keyword and positional arguments when calling a function
  - Positional arguments must appear before keyword arguments

keyword_args.py

```
1   # This program demonstrates keyword arguments.
2
3   def main():
4       # Show the amount of simple interest using 0.01 as
5       # interest rate per period, 10 as the number of periods,
6       # and $10,000 as the principal.
7       show_interest(rate=0.01, periods=10, principal=10000.0)
8
9   # The show_interest function displays the amount of
10  # simple interest for a given principal, interest rate
11  # per period, and number of periods.
12
13  def show_interest(principal, rate, periods):
14      interest = principal * rate * periods
15      print(f'The simple interest will be ${interest:,.2f}')
16
17  # Call the main function.
18  main()
```

Program output

```
The simple interest will be $1,000.00
```

keyword_string_args.py

```
1  # This program demonstrates passes two strings as
2  # keyword arguments to a function.
3
4  def main():
5      first_name = input('Enter your first name: ')
6      last_name = input('Enter your last name: ')
7      print('Your name reversed is')
8      reverse_name(last=last_name, first=first_name)
9
10 def reverse_name(first, last):
11     print(last, first)
12
13 # Call the main function.
14 main()
```

Program output (with input shown underlined)

```
Enter your first name: Matt
Enter your last name: Hoyle
Your name reversed is
Hoyle Matt
```

Example: Keyword Arguments (1)

keyword_args_1.py

```
1   # void functions with 6 parameters
2   def e1(a, b, c, d, e, f):
3       # a, b, c, d, e, f are parameters
4       print(f"a = {a}, b = {b}, c = {c}, d = {d}, e = {e}, f = {f}")
5
6   def main():
7       e1(1, 2, 3, 4, 5, 6)
8       e1(1, 2, 3, 4, e=5, f=6)        # keyword arguments e, f
9       e1(1, 2, c=3, d=4, e=5, f=6)    # keyword arguments c, d, e, f
10      e1(1, 2, d=4, e=5, f=6, c=3)    # keyword arguments c, d, e, f
11      e1(1, 2, e=5, f=6, c=3, d=4)    # keyword arguments c, d, e, f
12      e1(1, 2, d=4, f=6, e=5, c=3)    # keyword arguments c, d, e, f
13      e1(1, e=5, f=6, c=3, b=2, d=4) # keyword arguments b, c, d, e, f
14
15  main()
```

Example: Keyword Arguments (2)

keyword_args_2.py

```
1   def e2(a, b, c, d=4, e=5, f=6):
2       # a, b, c, d, e, f are parameters
3       # d, e, f have default values
4       print(f"a = {a}, b = {b}, c = {c}, d = {d}, e = {e}, f = {f}")
5
6   def main():
7       e2(1, 2, 3)                     # used default value for d, e, f
8       e2(1, 2, 3, 4)                  # used default value for e, f
9       e2(1, 2, 3, 4, 5)               # used default value for f
10      e2(1, 2, 3, 4, 5, 6)
11      e2(1, 2, d=4, e=5, f=6, c=3)    # keyword arguments c, d, e, f
12      e2(1, 2, e=5, f=6, c=3, d=4)    # keyword arguments c, d, e, f
13      e2(1, 2, d=4, f=6, e=5, c=3)    # keyword arguments c, d, e, f
14      e2(1, e=5, f=6, c=3, b=2, d=4) # keyword arguments b, c, d, e, f
15
16  main()
```

Global Variables and Global Constants

- Global variable: created by assignment statement written outside all functions
  - Can be accessed by any statement in the program file, including from within a function
  - If a function needs to assign a value to the global variable
    - the global variable must be redeclared within the function
    - general format: `global variable_name`

global1.py

```
1  # Create a global variable.
2  my_value = 10
3
4  # The show_value function prints
5  # the value of the global variable.
6  def show_value():
7      print(my_value)
8
9  # Call the show_value function.
10 show_value()
```

Program output

```
10
```

global2.py

```
1  # Create a global variable.
2  number = 0
3
4  def main():
5      global number
6      number = int(input('Enter a number: '))
7      show_number()
8
9  def show_number():
10     print('The number you entered is', number)
11
12 # Call the main function.
13 main()
```

Program output (with input shown underlined)

```
Enter a number: 55
The number you entered is 55
```

Global Variables and Global Constants (cont'd)

- Reasons to avoid using global variables:
  - Global variables making debugging difficult
    - Many locations in the code could be causing a wrong variable value
  - Functions that use global variables are usually dependent on those variables
    - Makes function hard to transfer to another program
  - Global variables make a program hard to understand

Global Constants

- Global constant: global name that references a value that cannot be changed
  - Permissible to use global constants in a program
  - To simulate global constant in Python, create global variable and do not re-declare it within functions
  - By convention, use all uppercases for constant names

retirement.py

```
1  # The following is used as a global constant to represent
2  # the contribution rate.
3  CONTRIBUTION_RATE = 0.05
4
5  def main():
6      gross_pay = float(input('Enter the gross pay: '))
7      bonus = float(input('Enter the amount of bonuses: '))
8      show_pay_contrib(gross_pay)
9      show_bonus_contrib(bonus)
10
11 # The show_pay_contrib function accepts the gross
12 # pay as an argument and displays the retirement
13 # contribution for that amount of pay.
14 def show_pay_contrib(gross):
15     contrib = gross * CONTRIBUTION_RATE
16     print(f'Contribution for gross pay: ${contrib:,.2f}')
17
```

```
18  # The show_bonus_contrib function accepts the
19  # bonus amount as an argument and displays the
20  # retirement contribution for that amount of pay.
21  def show_bonus_contrib(bonus):
22      contrib = bonus * CONTRIBUTION_RATE
23      print(f'Contribution for bonuses: ${contrib:,.2f}')
24
25  # Call the main function.
26  main()
```

Program output (with input shown underlined)

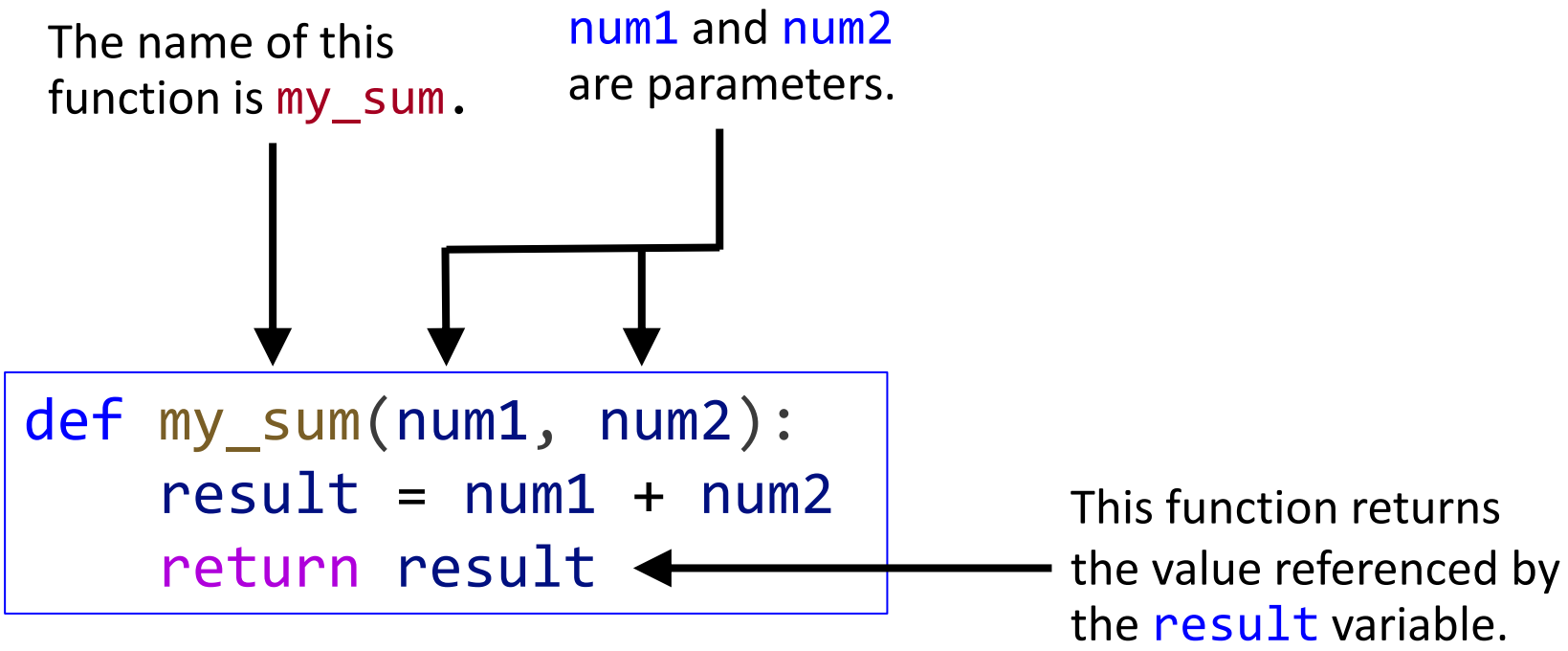
```
Enter the gross pay: 80000
Enter the amount of bonuses: 20000
Contribution for gross pay: $4,000.00
Contribution for bonuses: $1,000.00
```

Writing Your Own Value-Returning Functions

- To write a value-returning function, you write a simple function and add one or more `return` statements
  - Format: `return` *`expression`*
    - The value for *`expression`* will be returned to the part of the program that called the function
  - The expression in the `return` statement can be a complex expression, such as a sum of two variables or the result of another value- returning function

```
def function_name():
    statement
    statement
    …
    return expression
```

Writing Your Own Value-Returning Functions (cont'd)

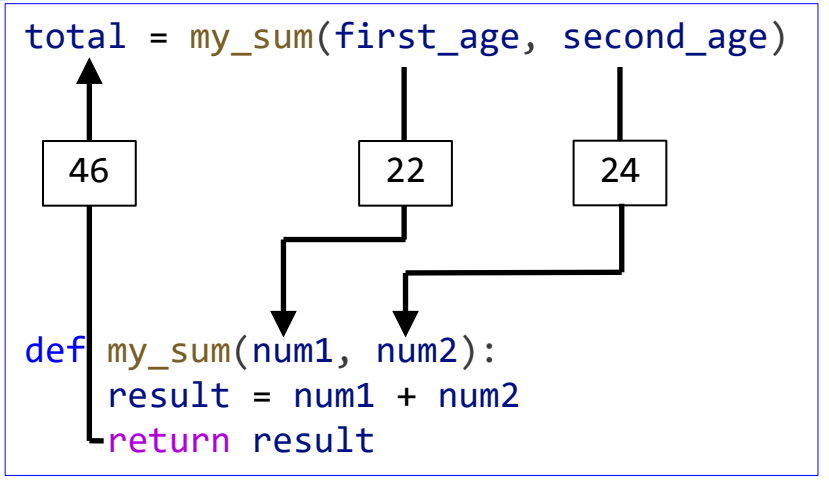


total_ages.py

```
1   # This program uses the return value of a function.
2
3   def main():
4       # Get the user's age.
5       first_age = int(input('Enter your age: '))
6
7       # Get the user's best friend's age.
8       second_age = int(input("Enter your best friend's age: "))
9
10      # Get the sum of both ages.
11      total = my_sum(first_age, second_age)
12
13      # Display the total age.
14      print('Together you are', total, 'years old.')
15
16  # The my_sum function accepts two numeric arguments and
17  # returns the sum of those arguments.
18  def my_sum(num1, num2):
19      result = num1 + num2
20      return result
21
22  # Call the main function.
23  main()
```

Program output (with input shown underlined)

```
Enter your age: 22
Enter your best friend's age: 24
Together you are 46 years old.
```



Because the return statement can return the value of an expression, you can eliminate the result variable and rewrite the function as:

```
def my_sum(num1, num2):
    return num1 + num2
```

sale_price.py

```
1  # This program calculates a retail item's
2  # sale price.
3
4  # DISCOUNT_PERCENTAGE is used as a global
5  # constant for the discount percentage.
6  DISCOUNT_PERCENTAGE = 0.20
7
8  # The main function.
9  def main():
10     # Get the item's regular price.
11     reg_price = get_regular_price()
12
13     # Calculate the sale price.
14     sale_price = reg_price - discount(reg_price)
15
16     # Display the sale price.
17     print(f'The sale price is ${sale_price:.2f}')
18
```

```
19  # The get_regular_price function prompts the
20  # user to enter an item's regular price and it
21  # returns that value.
22  def get_regular_price():
23      price = float(input("Enter the item's regular price: "))
24      return price
25
26  # The discount function accepts an item's price
27  # as an argument and returns the amount of the
28  # discount, specified by DISCOUNT_PERCENTAGE.
29  def discount(price):
30      return price * DISCOUNT_PERCENTAGE
31
32  # Call the main function.
33  main()
```

Program output (with input shown underlined)

```
Enter the item's regular price: 100.00
The sale price is $80.00
```

Returning a String

- You can write functions that return a string
- For example:

```
def get_name():
    # Get the user's first name and last name.
    full_name = input('Enter your full name: ')
    # Return the name.
    return full_name
```

Example: Returning a String

return_string.py

```
1   def get_name():
2       # Get the user's first name and last name.
3       full_name = input('Enter your full name: ')
4       # Return the name.
5       return full_name
6
7   def main():
8       fisrt_name, last_name = get_name().split()
9       print(f"Hello: {fisrt_name} {last_name}")
10
11  main()
```

Returning a Boolean Value

- Boolean function: returns either `True` or `False`
  - Use to test a condition such as for decision and repetition structures
    - Common calculations, such as whether a number is even, can be easily repeated by calling a function
  - Use to simplify complex input validation code

Example: Returning a Boolean Value

return_boolean.py

```
1  def is_even(number):
2      # Determine whether number is even.
3      # If it is, set status to true.
4      # Otherwise, set status to false.
5      if (number % 2) == 0:
6          status = True
7      else:
8          status = False
9      # Return the value of the status variable.
10     return status
11
12 def main():
13     x = 10
14     y = 25
15     print(f"{x} is even: {is_even(x)}")
16     print(f"{y} is even: {is_even(y)}")
17
18 main()
```

Returning a List

- You can write functions that return a list
- For example:

```
# zero out all values below a given threshold in a list
def zero_below(a_list, threshold):
    res = a_list[:]  # create a new list from a_list
    for i in range(len(a_list)):
        if res[i] < threshold:
            res[i] = 0
    return res
```

Example: Returning a List

return_string.py

```
1  # zero out all values below a given threshold in a list
2  def zero_below(a_list, threshold):
3      res = a_list[:]  # create a new list from a_list
4      for i in range(len(a_list)):
5          if res[i] < threshold:
6              res[i] = 0
7      return res
8
9  def main():
10     data = [10 , 20, 30, 40, 50, 60, 70, 80, 90, 100]
11     print(f"Before: {data}")
12     res = zero_below(data, 50)
13     print(f"After: {data}")
14     print(f"After: {res}")
15
16 main()
```

Example: Sum of First n Integers

Write a function `sum_n` to find summation of the first n integers from 1

$$S(n) = \sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n$$

sum_n.py

```
def sum_n(n):
    s = 0
    for i in range(1, n+1):
        s = s + i
    return s
```

sum_n.py (cont'd)

```
x = sum_n(5)
y = sum_n(10)

print(f"x = {x}")
print(f"y = {y}")
```

Example: Factorial

Write a function `factorial` to find factorial of n

$$F(n) = n! = n\,(n-1)(n-2)\ldots(2)(1)$$

factorial.py

```
1   import math
2   def factorial(n):
3       fact = 1
4       for i in range(1, n+1):
5           fact = fact * i
6       return fact
7
8   x = factorial(5)       # using user-defined function
9   y = math.factorial(5) # using function factorial in math module
10
11  print(f"x = {x}, y = {y}")
```

Example: Sum of Numbers in a List

Write a function `sum_list` to find summation of all numbers in a list

sum_list.py

```
def sum_list(a_list):
    s = 0
    for i in a_list:
        s = s + i
    return s

list_1 = [1,3,5,7,9]
list_2 = list( range(1,10,2) )
```

sum_list.py (cont'd)

```
# using user-defined function
x = sum_list(list_1)
y = sum_list(list_2)

# using built-in function sum
a = sum(list_1)
b = sum(list_2)

print(f"x = {x}, y = {y}")
print(f"a = {a}, b = {b}")
```

Returning Multiple Values

- In Python, a function can return multiple values
  - Specified after the `return` statement separated by commas
    - Format: `return` *`expression1, expression2, etc.`*
  - When you call such a function in an assignment statement, you need a separate variable on the left side of the = operator to receive each returned value

```
def get_name():
    # Get the user's first and last names.
    first = input("Enter your first name: ")
    last = input("Enter your last name: ")
    return first, last  # Return both names.

first_name, last_name = get_name()
print(f"First name: {first_name}\nLast name: {last_name}")
```

Example: Returning Multiple Values (1)

return_multiple_values_1.py

```
 1  def stat(data):
 2      minimum = min(data)
 3      maximum = max(data)
 4      mean = sum(data) / len(data)
 5      return minimum, maximum, mean
 6
 7  def main():
 8      input_list = input("Enter data: ").split()
 9      data = []
10      for item in input_list:
11          data.append(float(item))
12
13      mn, mx, mean = stat(data)
14      print(f"min = {mn}, max = {mx}, mean = {mean}")
15
16  main()
```

Example: Returning Multiple Values (2)

return_multiple_values_2.py

```
1   def stat(data):
2       minimum = min(data)
3       maximum = max(data)
4       mean = sum(data) / len(data)
5       return minimum, maximum, mean
6
7   def main():
8       input_list = input("Enter data: ").split()
9       data = []
10      for item in input_list:
11          data.append(float(item))
12
13      res = stat(data)
14      print(f"result = {res}")
15      print(f"min = {res[0]}, max = {res[1]}, mean = {res[2]}")
16
17  main()
```

Standard Library Functions and the `import` Statement

- Standard library: library of pre-written functions that comes with Python
  - *Library functions* perform tasks that programmers commonly need
    - Example: `print`, `input`, `range`
    - Viewed by programmers as a "black box"
- Some library functions built into Python interpreter
  - To use, just call the function

Standard Library Functions and the `import` Statement (cont'd)

- Modules: files that stores functions of the standard library
  - Help organize library functions not built into the interpreter
  - Copied to computer when you install Python
- To call a function stored in a module, need to write an `import` statement
  - Written at the top of the program
  - Format: `import` *`module_name`*

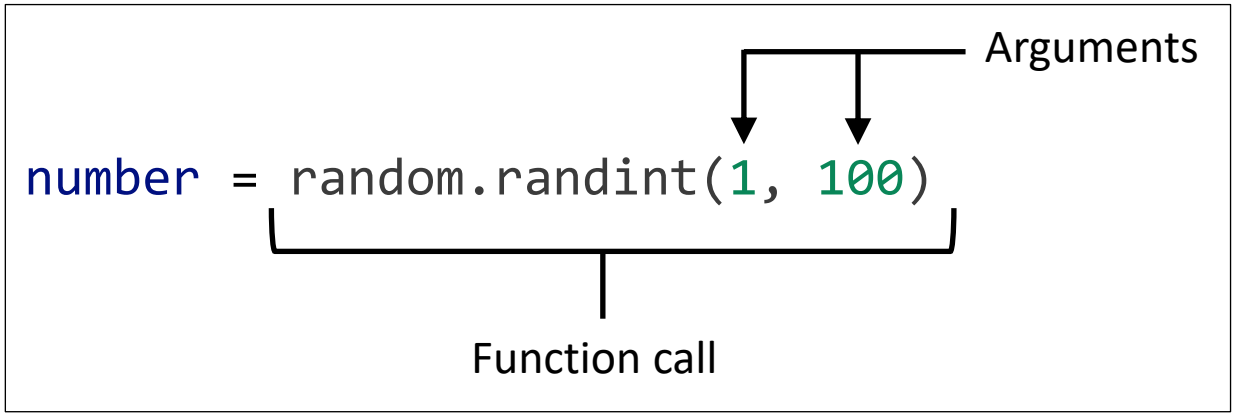
The `random` Module

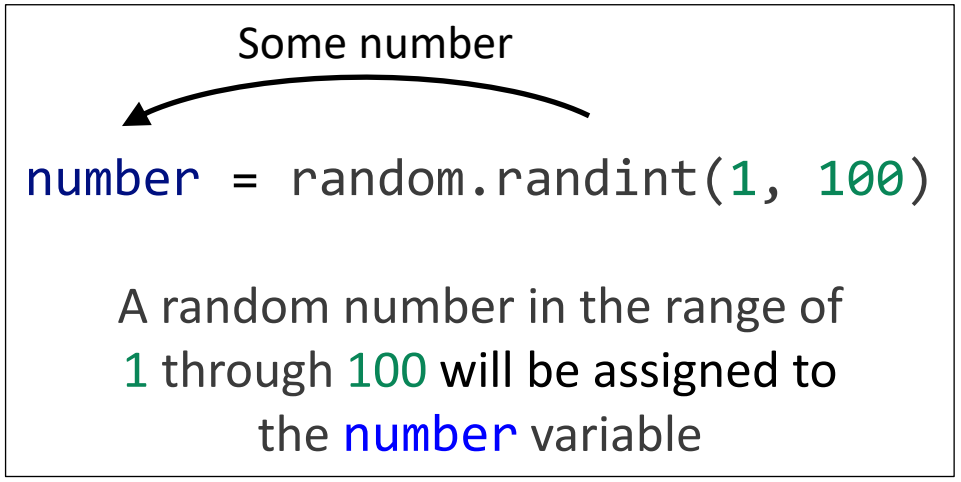
- module `random`: includes library functions for working with random numbers
- Random number are useful in a lot of programming tasks
- Dot notation: notation for calling a function belonging to a module
  - Format: `module_name.function_name()`

Generating Random Numbers

- function `randint`: generates a random number in the range provided by the arguments
  - Returns the random number to part of program that called the function
  - Returned integer can be used anywhere that an integer would be used
  - You can experiment with the function in interactive mode

A statement that calls the random function





random_numbers.py

```
 1  # This program displays a random number
 2  # in the range of 1 through 10.
 3  import random
 4
 5  def main():
 6      # Get a random number.
 7      number = random.randint(1, 10)
 8      # Display the number.
 9      print('The number is', number)
10
11  # Call the main function.
12  main()
```

Program output

```
The number is 7
```

random_number2.py

```
1   # This program displays five random
2   # numbers in the range of 1 through 100.
3   import random
4
5   def main():
6       for count in range(5):
7           # Get a random number.
8           number = random.randint(1, 100)
9           # Display the number.
10          print(number)
11
12  # Call the main function.
13  main()
```

Program output

```
89
7
16
41
12
```

random_number3.py

```
1   # This program displays five random
2   # numbers in the range of 1 through 100.
3   import random
4
5   def main():
6       for count in range(5):
7           print(random.randint(1, 100))
8
9   # Call the main function.
10  main()
```

dice.py

```
1   # This program simulates the rolling of dice.
2   import random
3
4   # Constants for the minimum and maximum random numbers
5   MIN = 1
6   MAX = 6
7
8   def main():
9       # Create a variable to control the loop.
10      again = 'y'
11
12      # Simulate rolling the dice.
13      while again == 'y' or again == 'Y':
14          print('Rolling the dice...')
15          print('Their values are:')
16          print(random.randint(MIN, MAX))
17          print(random.randint(MIN, MAX))
18
19          # Do another roll of the dice?
20          again = input('Roll them again? (y = yes): ')
21
22  # Call the main function.
23  main()
```

Program output (with input shown underlined)

```
Rolling the dice...
Their values are:
3
1
Roll them again? (y = yes): y
Rolling the dice...
Their values are:
1
1
Roll them again? (y = yes): y
Rolling the dice...
Their values are:
5
6
Roll them again? (y = yes): n
```

coin_toss.py

```
1   # This program simulates 10 tosses of a coin.
2   import random
3
4   # Constants
5   HEADS = 1
6   TAILS = 2
7   TOSSES = 10
8
9   def main():
10      for toss in range(TOSSES):
11          # Simulate the coin toss.
12          if random.randint(HEADS, TAILS) == HEADS:
13              print('Heads')
14          else:
15              print('Tails')
16
17  # Call the main function.
18  main()
```

Program output

```
Tails
Tails
Heads
Tails
Heads
Heads
Tails
Heads
Heads
Tails
```

Generating Random Numbers (cont'd)

- function `randrange` : similar to `range` function, but returns randomly selected integer from the resulting sequence

```
number = random.randrange(0, 101, 10)
```

  - Same arguments as for the `range` function
- function `random` : returns a random float in the range of 0.0 and 1.0

```
number = random.random()
```

  - Does not receive arguments
- function `uniform` : returns a random float but allows user to specify range

```
number = random.uniform(1.0, 10.0)
```

Random Number Seeds

- Random number created by functions in random module are actually pseudo-random numbers
- Seed value: initializes the formula that generates random numbers
  - Need to use different seeds in order to get different series of random numbers
    - By default uses system time for seed
    - Can use `random.seed()` function to specify desired seed value

If we start a new interactive session and repeat these statements, we get the same sequence of pseudorandom numbers, as shown here:

```
>>> import random
>>> random.seed(10)
>>> random.randint(1, 100)
58
>>> random.randint(1, 100)
43
>>> random.randint(1, 100)
58
>>> random.randint(1, 100)
21
>>>
```


```
>>> import random
>>> random.seed(10)
>>> random.randint(1, 100)
58
>>> random.randint(1, 100)
43
>>> random.randint(1, 100)
58
>>> random.randint(1, 100)
21
>>>
```

The `math` Module

- module `math`: part of standard library that contains functions that are useful for performing mathematical calculations
  - Typically accept one or more values as arguments, perform mathematical operation, and return the result
  - Use of module requires an `import math` statement

math Module Function	Description
acos(x)	Return the arc cosine of x, in radians.
asin(x)	Return the arc sine of x, in radians.
atan(x)	Return the arc tangent of x, in radians.
ceil(x)	Return the smallest integer that is greater than or equal to x.
cos(x)	Return the cosine of x in radians.
degrees(x)	Assuming x is an angle in radians, the fuction returns the angle converted to degrees.
exp(x)	Return e^x
floor(x)	Return the largest integer that is less than or equal to x.
hypot(x)	Returns the length of a hypotenuse that extends from (0, 0) to (x, y).
log(x)	Returns the natural logarithm of x.
log10(x)	Returns the base-10 logarithm of x.
radians(x)	Assuming x is an angle in degrees, the function returns the angle converted to radians.
sin(x)	Returns the sine of x in radians.
sqrt(x)	Returns the square root of x.
tan(x)	Returns the tangent of x in radians.

The `math` Module (cont'd)

- The `math` module defines variables `pi` and `e`, which are assigned the mathematical values for *`pi`* and *`e`*
  - Can be used in equations that require these values, to get more accurate results
- Variables must also be called using the dot notation
  - Example:

```
circle_area = math.pi * radius**2
```

hypotenuse.py

```
1   # This program calculates the length of a right
2   # triangle's hypotenuse.
3   import math
4
5   def main():
6       # Get the length of the triangle's two sides.
7       a = float(input('Enter the length of side A: '))
8       b = float(input('Enter the length of side B: '))
9
10      # Calculate the length of the hypotenuse.
11      c = math.hypot(a, b)
12
13      # Display the length of the hypotenuse.
14      print('The length of the hypotenuse is', c)
15
16  # Call the main function.
17  main()
```

Program output (with input shown underlined)

```
Enter the length of side A: 5.0
Enter the length of side B: 12.0
The length of the hypotenuse is 13.0
```

Example: Area of a Triangle

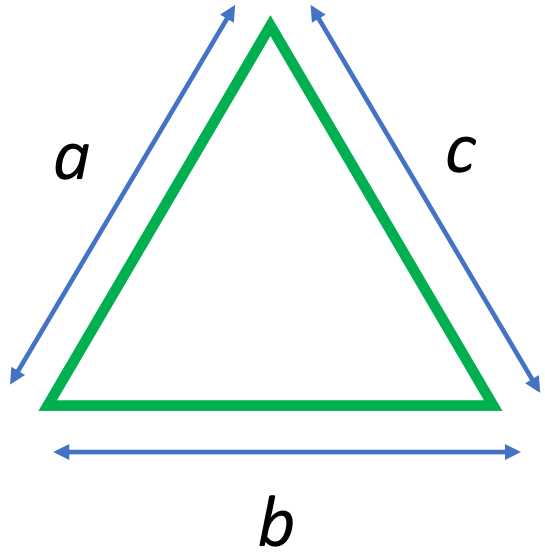
Heron's formula states that

- the area (A) of a triangle whose sides have lengths a, b and c is:

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

Where s is the semi-perimeter (half perimeter) of the triangle:

$$s = \frac{a+b+c}{2}$$



Example: Area of a Triangle (cont'd.)

area_of_triangle.py

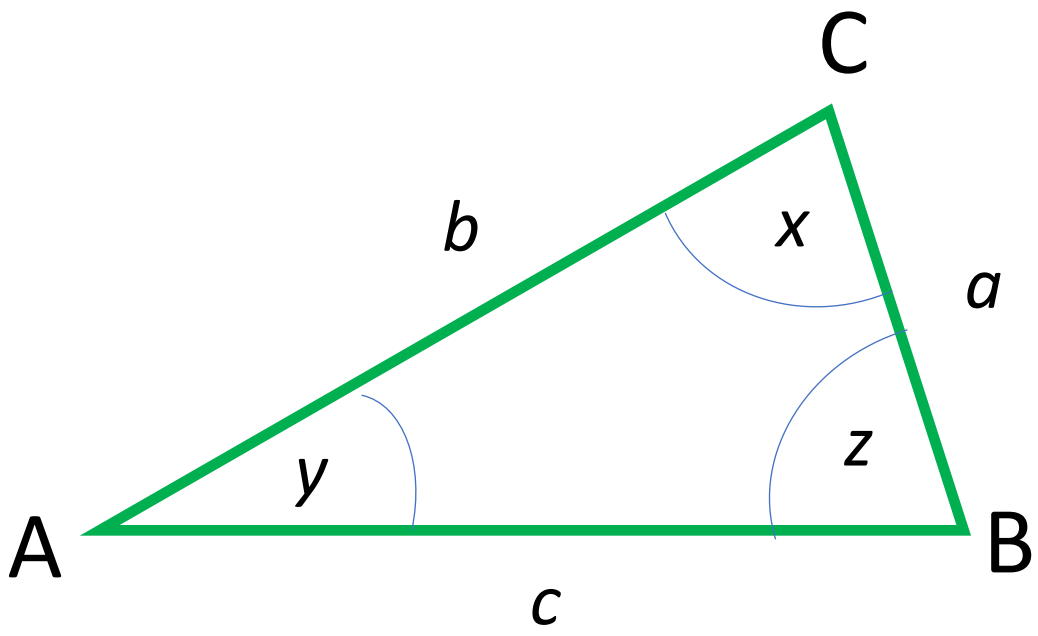
```
1   import math
2
3   def area_of_triangle(a, b, c):
4       s = 0.5 * (a + b + c)
5       area = math.sqrt(s*(s-a)*(s-b)*(s-c))
6       return area
7
8   def main():
9       a, b, c = input("Enter lengths of 3 sides: ").split()
10      a = float(a); b = float(b); c = float(c)
11      area = area_of_triangle(a, b, c)
12      print(f"Area = {area}")
13
14  main()
```

Example: Law of Cosines

The law of cosines states

$$c^2 = a^2 + b^2 - 2ab\cos x$$

where *a, b, c* are the lengths of 3 sides of a triangle



Example: Law of Cosines (cont'd)

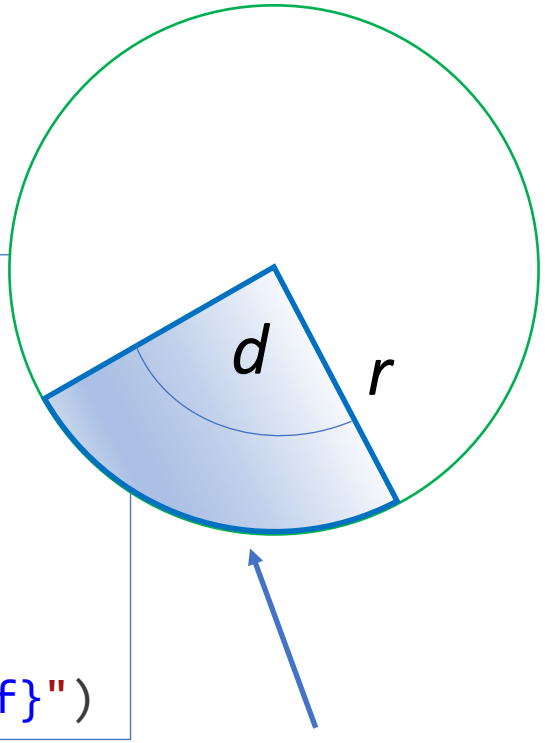
law_of_cosine.py

```
1   import math
2
3   def length_of_triangle(a, b, x):
4       # x is angle in radians, between two sides a and b
5       radian = math.radians(x)
6       c = math.sqrt(a * a + b * b - a * b * math.cos(radian))
7       return c
8
9   def main():
10      a, b = input("Enter lengths of 2 sides: ").split()
11      x = input("Enter angle between the 2 sides (degree): ")
12      a, b = float(a), float(b)
13      x = float(x)
14      c = length_of_triangle(a, b, x)
15      print(f"Length of remaining side = {c}")
16
17  main()
```

Exercise: Area of a Circular Sector

Write a function `area_of_circular_sector()` which

- takes 2 numbers as parameters
  - `r` - radius of the circle
  - `d` - angle of the circular sector, in degrees
- return the area of the circular sector



circular sector

```
import math
def area_of_circular_sector(r, d):
    # complete the body of this function
    pass

area_1 = area_of_circular_sector(10, 90)
area_2 = area_of_circular_sector(10, 180)
print(f"Area 1 = {area_1:.2f}, Area 2 = {area_2:.3f}")
```

```
# output
>Area 1 = 78.54, Area 2 = 157.080
```

Exercise: logarithm $y = e^{n \ln x}$

Write a function `my_power()` (not using `math.pow()`)

- takes 2 numbers as parameters
  - `x` - base
  - `n` - exponent
- return value of $e^{n \ln x}$

```
import math
def my_power(x, n):
    # complete the body of this function
    pass

a1 = my_power(4, 0.5); a2 = my_power(5.0625, 0.25)
b1 = math.pow(4, 0.5); b2 = math.pow(5.0625, 0.25)
print(f"a1 = {a1}, a2 = {a2}, b1 = {b1}, b2 = {b2}")
```

```
# output
>a1 = 2.0, a2 = 1.5, b1 = 2.0, b2 = 1.5
```

Summary

- This chapter covered:
  - The advantages of using functions
  - The syntax for defining and calling a function
  - Methods for designing a program to use functions
  - Use of local variables and their scope
  - Syntax and limitations of passing arguments to functions
  - Global variables, global constants, and their advantages and disadvantages

Summary (cont'd)

- Value-returning functions, including:
  - Writing value-returning functions
  - Using value-returning functions
  - Functions returning multiple values
- Using library functions and the `import` statement
- Modules, including the **`random`** and **`math`** modules