

FACULTY OF COMPUTING  
& INFORMATION TECHNOLOGY

KING ABDULAZIZ UNIVERSITY



FCIT  
KAU

كلية الحاسبات  
وتقنية المعلومات

جامعة الملك عبدالعزيز

## Chapter 3

# Mathematical Functions, and Strings

---

CPIT 110 (Problem-Solving and Programming)

Introduction to Programming Using Python, By: Y. Daniel Liang

# Sections

- 3.1. Motivations
- 3.2. Common Python Functions
- 3.3. Strings and Characters
- 3.4. Case Study: Minimum Number of Coins
- 3.6. Formatting Numbers and Strings



# Programs

- Program 1: Compute Angles
- Program 2: Compute Change
- Problem 3: Print a Table

# Check Points

- Section 3.2
  - #1
- Section 3.3
  - #2
  - #3
- Section 3.6
  - #4
  - #5



# Objectives

- To solve mathematics problems by using the functions in the math module ([3.2](#)).
- To represent and process strings and characters ([3.3-3.4](#)).
- To represent special characters using the escape sequence ([3.3.4](#)).
- To invoke the print function with the end argument ([3.3.5](#)).
- To convert numbers to a string using the str function ([3.3.6](#)).
- To use the + operator to concatenate strings ([3.3.7](#)).
- To read strings from the console ([3.3.8](#)).
- To format numbers and strings using the format function ([3.6](#)).

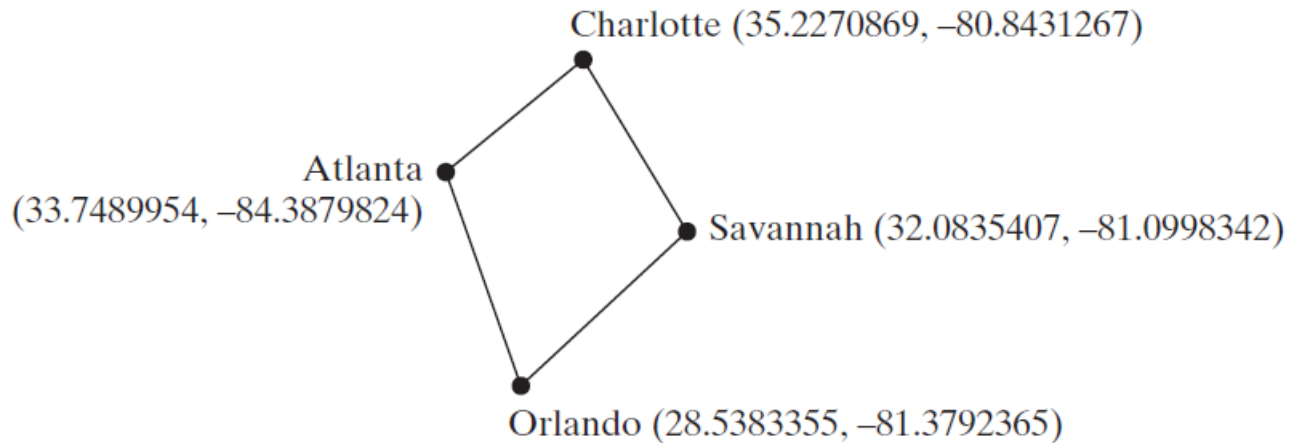




# 3.1. Motivations

# Motivations

- Suppose you need to estimate the area enclosed by four cities, given the GPS locations (latitude and longitude) of these cities, as shown in the following diagram. How would you write a program to solve this problem? You will be able to write such a program after completing this chapter.





## 3.2. Common Python Functions

- Python Built-in Functions
- Simple Python Built-in Functions
- math Module
- Problem 1: Compute Angles
- Check Point #1





# Python Built-in Functions

- A **function** is a **group of statements** that performs a specific task.
- Python, as well as other programming languages, provides a **library of functions**.
- You have already used the functions **eval**, **input**, **print**, and **int**.
- These are **built-in functions** and they are **always available** in the Python interpreter. You don't have to **import** any **modules** to use these functions.
- Additionally, you can use the built-in functions **abs**, **max**, **min**, **pow**, and **round**, as shown in the following slide.

# Simple Python Built-in Functions

**TABLE 3.1** Simple Python Built-in Functions

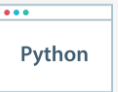
<i>Function</i>	<i>Description</i>	<i>Example</i>
<code>abs(x)</code>	Returns the absolute value for <code>x</code> .	<code>abs(-2)</code> is 2
<code>max(x1, x2, ...)</code>	Returns the largest among <code>x1</code> , <code>x2</code> , ...	<code>max(1, 5, 2)</code> is 5
<code>min(x1, x2, ...)</code>	Returns the smallest among <code>x1</code> , <code>x2</code> , ...	<code>min(1, 5, 2)</code> is 1
<code>pow(a, b)</code>	Returns <code>a<sup>b</sup></code> . Same as <code>a ** b</code> .	<code>pow(2, 3)</code> is 8
<code>round(x)</code>	Returns an integer nearest to <code>x</code> . If <code>x</code> is equally close to two integers, the even one is returned.	<code>round(5.4)</code> is 5 <code>round(5.5)</code> is 6 <code>round(4.5)</code> is 4
<code>round(x, n)</code>	Returns the float value rounded to <code>n</code> digits after the decimal point.	<code>round(5.466, 2)</code> is 5.47 <code>round(5.463, 2)</code> is 5.46



# Simple Python Built-in Functions

## Example

```
>>> abs(-3) # Returns the absolute value
3
>>> abs(-3.5) # Returns the absolute value
3.5
>>> max(2, 3, 4, 6) # Returns the maximum number
6
>>> min(2, 3, 4) # Returns the minimum number
2
>>> pow(2, 3) # Same as 2 ** 3
8
>>> pow(2.5, 3.5) # Same as 2.5 ** 3.5
24.705294220065465
>>> round(3.51) # Rounds to its nearest integer
4
>>> round(3.4) # Rounds to its nearest integer
3
>>> round(3.1456, 3) # Rounds to 3 digits after the decimal point
3.146
```



# math Module

- Many programs are created to solve mathematical problems.
- Some of the most popular mathematical functions are defined in the Python math module. These include trigonometric functions, representation functions, logarithmic functions, angle conversion functions, etc.
- In addition, two mathematical constants (pi and e) are also defined in this module.

# math Module

- The Python **math module** provides the mathematical functions listed in the **following slide**.
- To **use functions in a module**, you have to **import it first** as the following syntax:

```
import module_name
```

- For example, we have to import the **math module** before using its functions or constants :

```
>>> import math
>>> math.pi # pi is a constant
3.141592653589793
>>> math.cos(5.89) # cos is a function
0.92369335287311
```



**TABLE 3.2** Mathematical Functions

<i>Function</i>	<i>Description</i>	<i>Example</i>
<code>fabs(x)</code>	Returns the absolute value for <code>x</code> as a float.	<code>fabs(-2)</code> is 2.0
<code>ceil(x)</code>	Rounds <code>x</code> up to its nearest integer and returns that integer.	<code>ceil(2.1)</code> is 3 <code>ceil(-2.1)</code> is -2
<code>floor(x)</code>	Rounds <code>x</code> down to its nearest integer and returns that integer.	<code>floor(2.1)</code> is 2 <code>floor(-2.1)</code> is -3
<code>exp(x)</code>	Returns the exponential function of <code>x</code> ( $e^x$ ).	<code>exp(1)</code> is 2.71828
<code>log(x)</code>	Returns the natural logarithm of <code>x</code> .	<code>log(2.71828)</code> is 1.0
<code>log(x, base)</code>	Returns the logarithm of <code>x</code> for the specified base.	<code>log(100, 10)</code> is 2.0
<code>sqrt(x)</code>	Returns the square root of <code>x</code> .	<code>sqrt(4.0)</code> is 2
<code>sin(x)</code>	Returns the sine of <code>x</code> . <code>x</code> represents an angle in radians.	<code>sin(3.14159 / 2)</code> is 1 <code>sin(3.14159)</code> is 0
<code>asin(x)</code>	Returns the angle in radians for the inverse of sine.	<code>asin(1.0)</code> is 1.57 <code>asin(0.5)</code> is 0.523599
<code>cos(x)</code>	Returns the cosine of <code>x</code> . <code>x</code> represents an angle in radians.	<code>cos(3.14159 / 2)</code> is 0 <code>cos(3.14159)</code> is -1
<code>acos(x)</code>	Returns the angle in radians for the inverse of cosine.	<code>acos(1.0)</code> is 0 <code>acos(0.5)</code> is 1.0472
<code>tan(x)</code>	Returns the tangent of <code>x</code> . <code>x</code> represents an angle in radians.	<code>tan(3.14159 / 4)</code> is 1 <code>tan(0.0)</code> is 0
<code>degrees(x)</code>	Converts angle <code>x</code> from radians to degrees.	<code>degrees(1.57)</code> is 90
<code>radians(x)</code>	Converts angle <code>x</code> from degrees to radians.	<code>radians(90)</code> is 1.57

# math Module


## Example

LISTING 3.1 MathFunctions.py

```
1 import math # import Math module to use the math functions
2
3 # Test algebraic functions
4 print("exp(1.0) =", math.exp(1))
5 print("log(math.e) =", math.log(math.e))
6 print("log10(10, 10) =", math.log(10, 10))
7 print("sqrt(4.0) =", math.sqrt(4.0))
8
9 # Test trigonometric functions
10 print("sin(PI / 2) =", math.sin(math.pi / 2))
11 print("cos(PI / 2) =", math.cos(math.pi / 2))
12 print("tan(PI / 2) =", math.tan(math.pi / 2))
13 print("degrees(1.57) =", math.degrees(1.57))
14 print("radians(90) =", math.radians(90))
```

# math Module

## The Output of The Example



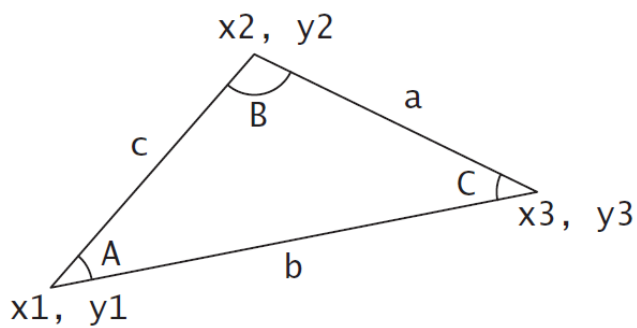
```
exp(1.0) = 2.718281828459045
log(math.e) = 1.0
log10(10, 10) = 1.0
sqrt(4.0) = 2.0
sin(PI / 2) = 1.0
cos(PI / 2) = 6.123233995736766e-17
tan(PI / 2) = 1.633123935319537e+16
degrees(1.57) = 89.95437383553924
radians(90) = 1.5707963267948966
```



# Compute Angles

## Program 1

Write a program to **get three points of a triangle** from the user. Then your program should **compute and display the angles in degrees** using the following formula:



$$A = \text{acos}((a * a - b * b - c * c) / (-2 * b * c))$$

$$B = \text{acos}((b * b - a * a - c * c) / (-2 * a * c))$$

$$C = \text{acos}((c * c - b * b - a * a) / (-2 * a * b))$$

Note:  $\text{acos}(x)$  returns the arc cosine of  $x$ , in **radians**

$$\text{distance}((p1x, p1y), (p2x, p2y)) = \sqrt{(p2x - p1x)^2 + (p2y - p1y)^2}$$

Enter three points: 1, 1, 6.5, 1, 6.5, 2.5 <Enter>

The three angles are 15.26 90.0 74.74



# Remember

- Don't be intimidated by the mathematical formula.
- As we discussed early in **Chapter 2, Program 9** (Computing Loan Payments), You don't have to know how the mathematical formula is derived in order to write a program for computing the loan payments.
- Here in this example (Program 1):
  - Given the length of three sides, you can use the given formula to write a program to compute the angles without having to know how the formula is derived.
  - In order to compute the lengths of the sides, we need to know the coordinates of three corner points and compute the distances between the points.

# Compute Angles

## Phase 1: Problem-solving

- The problem didn't give us specific formulas to calculate **a**, **b**, and **c**. But it gives us a general formula to calculate the distance of two points ( $p_1, p_2$ ).

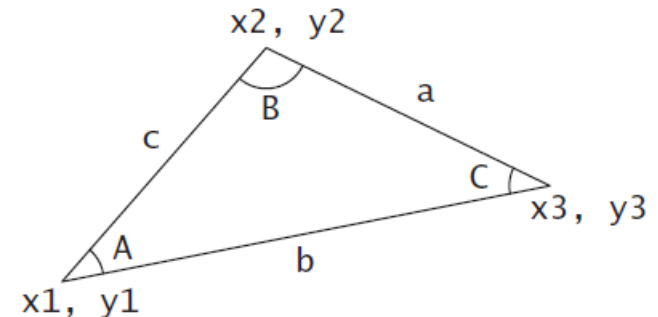
$$\text{distance}((p_1x, p_1y), (p_2x, p_2y)) = \sqrt{(p_2x - p_1x)^2 + (p_2y - p_1y)^2}$$

- So, we can use this formula to generate formulas for calculating **a**, **b**, and **c** as the following:

$$a = \sqrt{(x_2 - x_3)^2 + (y_2 - y_3)^2}$$

$$b = \sqrt{(x_1 - x_3)^2 + (y_1 - y_3)^2}$$

$$c = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$



# Compute Angles

## Phase 1: Problem-solving

- Design your algorithm:

1. Get three points.

- Use `input` function
- 1 point = `x` and `y` (two inputs)
- 3 points =  $3 * 2 = 6$  **inputs**

2. Compute `a`:  $a = \sqrt{(x_2 - x_3)^2 + (y_2 - y_3)^2}$

3. Compute `b`:  $b = \sqrt{(x_1 - x_3)^2 + (y_1 - y_3)^2}$

4. Compute `c`:  $c = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

5. Compute `A`:  $A = \text{acos}((a * a - b * b - c * c) / (-2 * b * c))$

6. Compute `B`:  $B = \text{acos}((b * b - a * a - c * c) / (-2 * a * c))$

7. Compute `C`:  $C = \text{acos}((c * c - b * b - a * a) / (-2 * a * b))$

8. Convert angles (`A`, `B`, `C`) in radians to degrees

- Use `math.degrees` function


9. Display the results (`A`, `B`, `C`) with two digits after the decimal point



# Compute Angles

## Phase 2: Implementation


- Remember:  $\sqrt{a} = a^{0.5}$   $a^2 = a \times a$
- In Python, you can use the `sqrt` function in the `math` module to calculate a square root. For example:



```
>>> import math
>>> math.sqrt(50)
7.0710678118654755
>>> 50 ** 0.5
7.0710678118654755
```

$$\sqrt{a} = \text{math.sqrt}(a)$$
$$\sqrt{a} = a ** 0.5$$

- Also, you can use the `pow` function to calculate the power of a number. You don't need to import any module because this function is built-in Python interpreter. For example:



```
>>> pow(2, 6)
64
>>> 2 ** 6
64
```

$$a^b = \text{pow}(a, b)$$
$$a^b = a ** b$$

# Compute Angles

## Phase 2: Implementation

- Note:
  - The function `math.acos(x)` returns the arc cosine of `x`, in **radians**.
  - In the solution of the problem, we need to display the angles in **degrees** not in **radians**.
  - So, we can use the function `math.degrees(x)` to convert angle `x` from **radians** to **degrees**.

```
>>> import math
>>> math.acos(0.5)
1.0471975511965979
>>> math.degrees(1.0471975511965979)
60.000000000000001
>>> math.radians(60)
1.0471975511965976
```

Python

# Compute Angles

## Phase 2: Implementation

LISTING 3.2 ComputeAngles.py

```
1 import math
2
3 x1, y1, x2, y2, x3, y3 = eval(input("Enter three points: "))
4
5 a = math.sqrt((x2 - x3) * (x2 - x3) + (y2 - y3) * (y2 - y3))
6 b = math.sqrt((x1 - x3) * (x1 - x3) + (y1 - y3) * (y1 - y3))
7 c = math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))
8
9 A = math.degrees(math.acos((a * a - b * b - c * c) / (-2 * b * c)))
10 B = math.degrees(math.acos((b * b - a * a - c * c) / (-2 * a * c)))
11 C = math.degrees(math.acos((c * c - b * b - a * a) / (-2 * a * b)))
12
13 print("The three angles are ", round(A * 100) / 100.0,
14       round(B * 100) / 100.0, round(C * 100) / 100.0)
```



```
Enter three points: 1, 1, 6.5, 1, 6.5, 2.5 <Enter>
The three angles are 15.26 90.0 74.74
```



# Compute Angles

## Discussion

- In **line 3**, the program prompts the user to enter three points.
- This prompting message is **not clear**. You should give the user **explicit instructions** on how to enter these points as follows:

```
input("Enter six coordinates of three points separated by commas \
like x1, y1, x2, y2, x3, y3: ")
```

- In **lines 5–7**, the program computes the distances between the points.
- In **lines 9–11**, it applies the formula to compute the angles.
- In **lines 13–14**, the angles are **rounded** to display **up to two digits** after the decimal point.
- In the following slide, we have **simplified** the implementation.



# Compute Angles

## Simplified Implementation

SimplifiedComputeAngles.py

```
1 import math
2
3 x1, y1, x2, y2, x3, y3 = eval(input("Enter six coordinates \
4 of three points separated by commas like x1, y1, x2, y2, x3, y3: "))
5
6 a = math.sqrt(pow(x2 - x3, 2) + pow(y2 - y3, 2))
7 b = math.sqrt(pow(x1 - x3, 2) + pow(y1 - y3, 2))
8 c = math.sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2))
9
10 # Compute angles in radians
11 A = math.acos((a * a - b * b - c * c) / (-2 * b * c))
12 B = math.acos((b * b - a * a - c * c) / (-2 * a * c))
13 C = math.acos((c * c - b * b - a * a) / (-2 * a * b))
14
15 # Convert angles to degrees
16 A = math.degrees(A)
17 B = math.degrees(B)
18 C = math.degrees(C)
19
20 # Get two digits after the decimal point
21 A = round(A, 2)
22 B = round(B, 2)
23 C = round(C, 2)
24
25 # Display results
26 print("The three angles are ", A, B, C)
```



# Compute Angles

## Simplified Implementation

Output:



```
Enter six coordinates of three points separated by commas  
like x1, y1, x2, y2, x3, y3: 1, 1, 6.5, 1, 6.5, 2.5 <Enter>  
The three angles are 15.26 90.0 74.74
```





# Remember

- The following statement is wrong (**syntax error**):

```
1 input("Enter six coordinates of three points separated by commas  
2 like x1, y1, x2, y2, x3, y3: ")
```



- This because Python interpreter doesn't see the **closing quotation mark** of the string in **line 1**.
- So, you have to tell Python interpreter that the string is continued on the next line by place **the line continuation symbol (\)** at the **end of a line**.
- To fix the previous example:

```
1 input("Enter six coordinates of three points separated by commas \  
2 like x1, y1, x2, y2, x3, y3: ")
```





# Check Point #1

Evaluate the following functions:

- `min(2, 2, 1)` → 1
- `max(2, 3, 4)` → 4
- `abs(-2.5)` → 2.5
- `math.ceil(-2.5)` → -2
- `math.floor(-2.5)` → -3
- `round(3.5)` → 4
- `round(3.4)` → 3
- `math.ceil(2.5)` → 3
- `math.floor(2.5)` → 2
- `round(-2.5)` → -2
- `round(-2.6)` → -3





## 3.3. Strings and Characters

- Escape Sequences for Special Characters
- Python Escape Sequences
- Printing Without the Newline
- The str Function
- The String Concatenation Operator
- Reading Strings From the Console
- Check Point #2 - #3



# Strings and Characters

- In addition to processing numeric values, you can **process strings** in Python.
- A string is a **sequence of characters** and can **include text and numbers**.
- String values must be **enclosed in matching single quotes (')** or **double quotes (")**.
- Python **does not have a data type for characters**.
- A **single-character string** represents a **character**.

# Strings and Characters

## Example

```
1 letter = 'A' # Same as letter = "A"  
2 numChar = '4' # Same as numChar = "4"  
3 message = "Good morning" # Same as message = 'Good morning'
```

- The first statement assigns a string with the character **A** to the variable **letter**.
- The second statement assigns a string with the digit character **4** to the variable **numChar**.
- The third statement assigns the string **Good morning** to the variable **message**.



# Note

- For **consistency**, this book uses:
  - **double quotes** (") for a string with **more than one character**.
  - **single quotes** (') for a string with a **single character** or an **empty string**.
- This convention is **consistent with other programming languages**, so it will be easy for you to convert a Python program to a program written in other languages.





# Escape Sequences for Special Characters

- Suppose you want to print a **message with quotation marks** in the output. Can you write a statement like this?

```
1 print("He said, "John's program is easy to read")
```



- **No, this statement** has an **error**. Python thinks the second quotation mark is the end of the string and does not know what to do with the rest of the characters.
- To **overcome** this problem, Python uses a **special notation** to represent special characters.
- This special notation, which consists of a **backslash** (**\**) followed by a **letter** or a **combination of digits**, is called an **escape sequence**.



# Python Escape Sequences

**TABLE 3.3** Python Escape Sequences

<i>Character Escape Sequence</i>	<i>Name</i>	<i>Numeric Value</i>
<code>\b</code>	Backspace	8
<code>\t</code>	Tab	9
<code>\n</code>	Linefeed	10
<code>\f</code>	Formfeed	12
<code>\r</code>	Carriage Return	13
<code>\\</code>	Backslash	92
<code>\'</code>	Single Quote	39
<code>\"</code>	Double Quote	34

- The `\n` character is also known as a **newline**, **line break** or **end-of-line (EOL)** character, which signifies the end of a line.
- The `\f` character forces the printer to print from the next page.
- The `\r` character is used to move the cursor to the first position on the same line.
- The `\f` and `\r` characters are **rarely used** in this book.



# Python Escape Sequences

- Now you can print the previous quoted message using the following statement:

```
1 print("He said, \"John's program is easy to read\"")
```

- The output:



```
He said, "John's program is easy to read"
```

- Note that the symbols `\` and `"` together **represent one character**.

# Python Escape Sequences Example

Newline (`\n`)

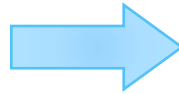
```
print("ABC\nDEF")
```



```
ABC  
DEF
```

Tab (`\t`)

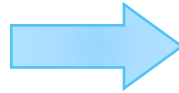
```
print("ABC\tDEF")
```



```
ABC    DEF
```

Carriage return (`\r`)

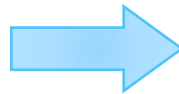
```
print("ABC\rDEF")
```



```
DEF
```

Backspace (`\b`)

```
print("ABC\bDEF")
```



```
ABDEF
```

# Python Escape Sequences Example

Double Quote (\")

```
print("He said \"OK\"")
```



```
He said "OK"
```

Single Quote (\')

```
print('He said \'OK\'')
```



```
He said 'OK'
```

Backslash (\\)

```
print("ABC\\DEF")
```



```
ABC\DEF
```

Formfeed (\f)

```
print("ABC\fDEF")
```



```
ABC  
DEF
```



# Don't Get Confused

```
print("nnn\nnnn\n\n\t\nnn\tt\n\n\r * \nTest\b\b\b\bAhmad")
```



```
nnn
nnn
      nnn    t
*
Ahmad
```

```
print("") # Empty String
print("Line 2")
print('\b\b\b')
print("Line 4")
```



```
Line 2
Line 4
```



# Printing Without The Newline

- When you use the `print` function, it automatically prints a `linefeed (\n)` to cause the output to advance to the next line.
- If you don't want this to happen after the `print` function is finished, you can invoke the `print` function by passing a special argument `end = "anyEndingString"` using the following syntax:

```
print(item, end = "anyEndingString")
```

- For example:

```
1 print("AAA", end = ' ')
2 print("BBB", end = '')
3 print("CCC", end = '***')
4 print("DDD", end = '***')
```



```
AAA BBBCCC***DDD***
```

# Printing Without The Newline

- Also, You can also use the end argument for printing multiple items using the following syntax:

```
1 import math
2 radius = 3
3 print("The area is", radius * radius * math.pi, end = ' ')
4 print("and the perimeter is", 2 * radius)
```



```
The area is 28.274333882308138 and the perimeter is 6
```



# The str Function

- The `str` function can be used to convert a **number** into a **string**.
- For example:

```
>>> s = str(3.4) # Convert a float to string
>>> s
'3.4'
>>> s = str(3) # Convert an integer to string
>>> s
'3'
>>>
```

Python

# The String Concatenation Operator

- You can use the `+` operator to **add two numbers**.
- Also, the `+` operator can be used to **concatenate two strings**.
- Here are some examples:

```
>>> message = "Welcome " + "to " + "Python"
>>> message
'Welcome to Python'
>>> chapterNo = 3
>>> s = "Chapter " + str(chapterNo)
>>> s
'Chapter 3'
>>>
```



# The String Concatenation Operator

- The **augmented assignment +=** operator can also be used for **string concatenation**.
- For example:


```
>>> message = "Welcome to Python"
>>> message
'Welcome to Python'
>>> message += " and Python is fun"
>>> message
'Welcome to Python and Python is fun'
>>>
```

Python

# Reading Strings From the Console

- To read a string from the console, use the `input` function.
- For example, the following code reads three strings from the keyboard:

```
1 s1 = input("Enter a string: ")
2 s2 = input("Enter a string: ")
3 s3 = input("Enter a string: ")
4 print("s1 is " + s1)
5 print("s2 is " + s2)
6 print("s3 is " + s3)
```



```
Enter a string: Welcome <Enter>
Enter a string: to <Enter>
Enter a string: Python <Enter>
s1 is Welcome
s2 is to
s3 is Python
```



# Check Point #2

Write a one statement that is equivalent to the following code:

```
1 print("Line 1")
2 print("Line 2")
3 print("Line 3")
```

➤ **Solution:**

```
print("Line 1" + "\n" + "Line 2" + "\n" + "Line 3")
```

Or

```
print("Line 1\n" + "Line 2\n" + "Line 3")
```

Or

```
print("Line 1\nLine 2\nLine 3")
```



# Check Point #3

Show the result of the following code:

```
1 sum = 2 + 3
2 print(sum)
3 s = '2' + '3'
4 print(s)
```

➤ Solution:



```
5
23
```






## 3.4. Case Study: Minimum Number of Coins

- Program 2: Compute Change

# Compute Change Program 2

Write a program that lets the user enter the **amount in decimal** representing **dollars and cents** and output a report listing the monetary equivalent in single **dollars, quarters, dimes, nickels,** and **pennies** as **shown in the sample run**. Your program should report maximum number of dollars, then the maximum number of quarters, and so on, in this order.



```
Enter an amount in double, e.g., 11.56: 11.56 <Enter>
Your amount 11.56 consists of
    11 dollars
    2 quarters
    0 dimes
    1 nickels
    1 pennies
```



# Compute Change


## Phase 1: Problem-solving

- A reminder about U.S. monetary units:
  - 1 dollar = 100 cents (or pennies)
  - 1 quarter = 25 cents
  - 1 dime = 10 cents
  - 1 nickel = 5 cents
- So if you need to give someone **42 cents** in change, you should give: **0 dollar, 1 quarter, 1 dime, 1 nickel, and 2 pennies.**

# Compute Change

## Phase 1: Problem-solving

- First step: UNDERSTAND the problem!
- So let us look at an example run:



```
Enter an amount in double, e.g., 11.56: 11.56 <Enter>
Your amount 11.56 consists of
    11 dollars
    2 quarters
    0 dimes
    1 nickels
    1 pennies
```

- Is it clear what the problem is asking of us?
  - Make sure you understand the question before starting

# Compute Change

## Phase 1: Problem-solving

Design your algorithm:

1. Prompt the user to enter the amount as a decimal number.
  - `amount = eval(input("Message..."))`
    - Example: 11.56
2. Convert this `amount` into `cents` (multiply by 100)
  - `totalCents = int(amount * 100)`
    - Example:  $11.56 * 100 = 1156$

# Compute Change

## Phase 1: Problem-solving

Design your algorithm:

3. Get the total number of **dollars** by now dividing by 100. And get **remaining cents** by using `totalCents % 100`.
  - `totalDollars = totalCents // 100`
    - Example:  $1156 // 100 = 11$
  - `remainingCents = totalCents % 100`
    - Example:  $1156 \% 100 = 56$

# Compute Change

## Phase 1: Problem-solving

Design your algorithm:

4. Get the total # of **quarters** by dividing **remainingCents** by 25. And then recalculate **remainingCents**.
  - `totalQuarters = remainingCents // 25`
    - Example:  $56 // 25 = 2$
  - `remainingCents = remainingCents % 25`
    - Example:  $56 \% 25 = 6$

# Compute Change

## Phase 1: Problem-solving

Design your algorithm:

5. Get the total # of **dimes** by dividing `remainingCents` by 10. And then recalculate `remainingCents`.
  - `totalDimes = remainingCents // 10`
    - Example:  $6 // 10 = 0$
  - `remainingCents = remainingCents % 10`
    - Example:  $6 \% 10 = 6$
    - So nothing changed at this step. `remainingCents` is still 6.

# Compute Change

## Phase 1: Problem-solving

Design your algorithm:

6. Get the total # of **nickels** by dividing `remainingCents` by 5. And then recalculate `remainingCents`.
  - `totalDimes = remainingCents // 5`
    - Example:  $6 // 5 = 1$
  - `remainingCents = remainingCents % 5`
    - Example:  $6 \% 5 = 1$

# Compute Change

## Phase 1: Problem-solving

Design your algorithm:

7. The value stored in `remainingCents` is the number of **pennies** left over.
  - Example: `remainingCents = 1`
8. Display the result.
  - Example:  
Dollars = 11, Quarters = 2, Dimes = 0, Nickels = 1, Pennies = 1



# Compute Change

## Phase 2: Implementation

LISTING 3.4 ComputeChange.py

```
1 # Receive the amount
2 amount = eval(input("Enter an amount in double, e.g., 11.56: "))
3
4 # Convert the amount to cents
5 remainingAmount = int(amount * 100)
6
7 # Find the number of one dollars
8 numberOfOneDollars = remainingAmount // 100
9 remainingAmount = remainingAmount % 100
10
11 # Find the number of quarters in the remaining amount
12 numberOfQuarters = remainingAmount // 25
13 remainingAmount = remainingAmount % 25
14
15 # Find the number of dimes in the remaining amount
16 numberOfDimes = remainingAmount // 10
17 remainingAmount = remainingAmount % 10
18
```



# Compute Change

## Phase 2: Implementation

LISTING 3.4 ComputeChange.py

```
19 # Find the number of nickels in the remaining amount
20 numberOfNickels = remainingAmount // 5
21 remainingAmount = remainingAmount % 5
22
23 # Find the number of pennies in the remaining amount
24 numberOfPennies = remainingAmount
25
26 # Display results
27 print("Your amount", amount, "consists of\n",
28       "\t", numberOfOneDollars, "dollars\n",
29       "\t", numberOfQuarters, "quarters\n",
30       "\t", numberOfDimes, "dimes\n",
31       "\t", numberOfNickels, "nickels\n",
32       "\t", numberOfPennies, "pennies")
```



# Compute Change

## Trace The Program Execution

Enter an amount in double, e.g., 11.56: 11.56

Your amount 11.56 consists of

11 dollars  
 2 quarters  
 0 dimes  
 1 nickels  
 1 pennies



	line#	2	5	8	9	12	13	16	17	20	21	24
<b>variables</b>												
amount		11.56										
remainingAmount			1156		56		6		6		1	
numberOfOneDollars				11								
numberOfQuarters						2						
numberOfDimes								0				
numberOfNickels										1		
numberOfPennies												1



# Compute Change

## Discussion

- The variable `amount` stores the amount entered from the console (line 2). This variable is **not changed**, because the `amount` has to be **used at the end of the program** to display the results.
- The program introduces the variable `remainingAmount` (line 5) to store the changing `remainingAmount`.
- The variable `amount` is a **float** representing dollars and cents.
- It is converted **to an integer** variable `remainingAmount`, which represents all the cents. For instance, if `amount` is 11.56, then the initial `remainingAmount` is 1156. `1156 // 100` is 11 (line 8).
- The remainder operator obtains the remainder of the division. So, `1156 % 100` is 56 (line 9).

# Compute Change

## Discussion

- The program extracts the maximum number of quarters from `remainingAmount` and obtains a new `remainingAmount` (lines 12–13).
- Continuing the same process, the program finds the maximum number of dimes, nickels, and pennies in the remaining amount.
- As shown in the sample run, 0 dimes, 1 nickels, and 1 pennies are displayed in the result. **It would be better not to display 0 dimes**, and to display 1 nickel and 1 penny using the singular forms of the words. You will **learn how to use selection statements** to modify this program in the **next chapter**.



# Caution

- One serious problem with this example is the possible loss of precision when converting a float amount to the integer remainingAmount.
- This could lead to an inaccurate result.
- If you try to enter the amount 10.03,  $10.03 * 100$  might be 1002.9999999999999999.
- You will find that the program displays 10 dollars and 2 pennies.
- To fix the problem, enter the amount as an integer value representing cents (see Exercise 3.8).





## 3.6. Formatting Numbers and Strings

- Formatting Floating-Point Numbers
- Formatting as a Percentage
- Justifying Format
- Formatting Integers
- Formatting Strings
- Frequently Used Specifiers
- Problem 3: Print a Table
- Check Point #4 - #5



# Formatting Numbers and Strings

- When printing float values, often we **do not need** or want **all the decimals**.
- In fact, often we want **only two (for money)!**
- In **Chapter 2** and this chapter, we have learned that you can get two digits after the decimal as follows:

```
1 x = 16.404674
2 x = int(x * 100) / 100 # x => 1640 / 100 = 16.4
3 print("x is", x) # output: x is 16.4
4 # Or we can use the round(num, digit) function
5 print("x is", round(x, 2)) # output: x is 16.4
```

- However, the **format is still not correct**. There should be **two digits after the decimal point** like 16.40 rather than 16.4.



# Formatting Numbers and Strings

- You can use the `format` function to `format` a `number` or a `string` and `return the result as a string`.
- How to print `16.404674` with only `two decimals`?

```
1 x = 16.404674
2 print("x is", format(x, ".2f")) # print: x is 16.40
```

- The syntax to invoke `format` function is

```
format(item, format-specifier)
```

- where `item` is a `number` or a `string` and `format-specifier` is a `string` that specifies `how the item is formatted`.
- The function `returns a string`.

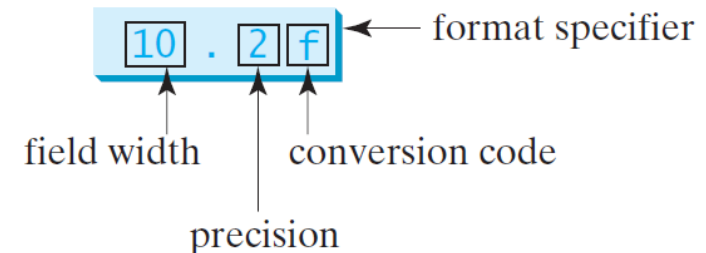
# Formatting Floating-Point Numbers

- If the item is a float value, you can use the specifier to give the width and precision of the format in the form of `width.precisionf`. Here, `width` specifies the width of the resulting string, `precision` specifies the number of digits after the decimal point, and `f` is called the conversion code, which sets the formatting for floating point numbers. For example:

```
1 print(format(57.467657, "10.2f"))
2 print(format(123456782.923, "10.2f"))
3 print(format(57.4, "10.2f"))
4 print(format(57, "10.2f"))
```

← 10 →  
□□□□ 57.47  
123456782.92  
□□□□ 57.40  
□□□□ 57.00

- A square box □ denotes a blank space.
- Note that the decimal point is counted as one space.



# Formatting Floating-Point Numbers

- The `format("10.2f")` function formats the number into a string whose width is **10**, including a decimal point and two digits after the point.
  - The number is rounded to two decimal places.
  - Thus there are seven digits allocated before the decimal point.
  - If there are fewer than seven digits before the decimal point, spaces are inserted before the number.
  - If there are more than seven digits before the decimal point, the number's width is automatically increased.
  - For example, `format(12345678.923, "10.2f")` returns **12345678.92**, which has a width of **11**.

# Formatting Floating-Point Numbers

- You can **omit** the **width specifier**. If so, it **defaults to 0**.
- In this case, the width is **automatically set to the size needed** for formatting the number.
- For example:

```
1 print(format(57.467657, "10.2f"))  
2 print(format(57.467657, ".2f"))
```

|← 10 →|  
□□□□ 57.47  
57.47

# Formatting as a Percentage

- You can use the conversion code `%` to format a number as a percentage.
- For example:

```
1 print(format(0.53457, "10.2%"))
2 print(format(0.0033923, "10.2%"))
3 print(format(7.4, "10.2%"))
4 print(format(57, "10.2%"))
```

← 10 →  
□□□□ 53.46%  
□□□□□ 0.34%  
□□□ 740.00%  
□□ 5700.00%

- The format `10.2%` causes the number to be multiplied by 100 and displayed with a `%` sign following it.
- The total width includes the `%` sign counted as one space.

# Justifying Format

- By default, the format of a number is **right justified**.
- You can put the symbol **<** in the **format specifier** to specify that the item be **left-justified** in the resulting format within the **specified width**.
- For example:

```
1 print(format(57.467657, "10.2f"))  
2 print(format(57.467657, "<10.2f"))
```

Diagram illustrating string formatting. A horizontal line with arrows at both ends is labeled "10", indicating a width of 10 characters. Below this, the number "57.47" is shown. The first four characters of the width are represented by empty boxes, showing that the number is right-aligned within the 10-character field.

# Formatting Integers

- The conversion code `d` can be used to format an integer in **decimal**.
- You can specify a width for the conversion.
- For example:

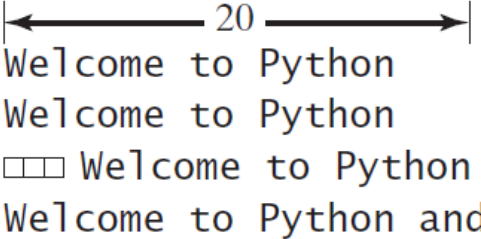
```
1 print(format(59832, "10d"))  
2 print(format(59832, "<10d"))
```

Diagram illustrating integer formatting. A double-headed arrow above the number 59832 indicates a width of 10 characters. Below the arrow, five empty boxes represent the padding characters, followed by the digits 59832.

# Formatting Strings

- You can use the conversion code `s` to format a string with a specified width.
- For example:

```
1 print(format("Welcome to Python", "20s"))
2 print(format("Welcome to Python", "<20s"))
3 print(format("Welcome to Python", ">20s"))
4 print(format("Welcome to Python and Java", ">20s"))
```



← 20 →  
Welcome to Python  
Welcome to Python  
Welcome to Python  
Welcome to Python and Java

- The format specifier `20s` specifies that the string is formatted within a width of 20.
- By default, a string is **left justified**.
- To **right-justify** it, put the symbol `>` in the format **specifier**. If the string is longer than the specified width, the width is **automatically increased to fit the string**.



# Frequently Used Specifiers

**TABLE 3.4** Frequently Used Specifiers

<i>Specifier</i>	<i>Format</i>
"10.2f"	Format the float item with width 10 and precision 2.
"5d"	Format the integer item in decimal with width 5.
"10.2%"	Format the number in decimal.
"50s"	Format the string item with width 50.
"<10.2f"	Left-justify the formatted item.
">10.2f"	Right-justify the formatted item.



# Note

- The format function uses the built-in round function when dealing with floating-point numbers.

```
print(format(123.426, "10.2f"))
```

→ 123.43

```
print(round(123.426, 2))
```

→ 123.43

```
print(format(123.424, "10.2f"))
```

→ 123.42

```
print(round(123.424, 2))
```

→ 123.42

```
print(format(12.678, ".0f"))
```

→ 13

```
print(round(12.678))
```

→ 13

```
print(format(12.378, ".0f"))
```

→ 12

```
print(round(12.378))
```

→ 12





# Note

- The behavior of `round()` for floats can be surprising.
- For example: `round(2.675, 2)` gives `2.67` instead of the expected `2.68`.
- This is **not** a **bug**: it's a result of the fact that **most decimal fractions can't be represented exactly as a float**.

```
Python 3.7.4 Shell
File Edit Shell Debug Options Window Help
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 19:29:22) [MSC v.1916 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> round(2.675, 2)
2.67
>>> round(2.677)
3
>>>
```

Ln: 7 Col: 4



# Print a Table

## Program 3

Write a program that displays the following table as shown in the sample run:



```
-----  
    x | x^2           sqrt(x)  
-----  
    5 | 25.00          2.24  
  500 | 250000.00      22.36  
   20 | 400.00          4.47  
 3000 | 9000000.00     54.77  
-----
```

# Print a Table

## Implementation A

PrintTableA.py

```
1 import math
2
3 # The header of the table
4 print("-----")
5 print(format("x", ">5s") + '|', format("x^2", "12s"), format("√x", "10s"))
6 print("-----")
7
8 print(format(5, ">5d") + '|', format(5 ** 2, "<12.2f"), format(math.sqrt(5), "<10.2f"))
9 print(format(500, ">5d") + '|', format(500 ** 2, "<12.2f"), format(math.sqrt(500), "<10.2f"))
10 print(format(20, ">5d") + '|', format(20 ** 2, "<12.2f"), format(math.sqrt(20), "<10.2f"))
11 print(format(3000, ">5d") + '|', format(3000 ** 2, "<12.2f"), format(math.sqrt(3000), "<10.2f"))
12
13 # The footer of the table
14 print("-----")
```



- Let us see another implementation of the same problem in the next slide. Then we will discuss which one is the best.



# Print a Table

## Implementation B

PrintTableB.py

```
1 import math
2
3 SEPARATOR = '|' # a constant to represent the separator between the first and second columns
4 FORMAT_1 = ">5d" # a constant to represent the Specifier for the first column format
5 FORMAT_2 = "<12.2f" # a constant to represent the Specifier for the second column format
6 FORMAT_3 = "<10.2f" # a constant to represent the Specifier for the first third format
7
8 # The header of the table
9 print("-----")
10 print(format("x", ">5s") + SEPARATOR, format("x^2", "12s"), format("√x", "10s"))
11 print("-----")
12
13 x = 5 # this variable stores the value of x that will be included with the next calculations
14 print(format(x, FORMAT_1) + SEPARATOR, format(x ** 2, FORMAT_2), format(math.sqrt(x), FORMAT_3))
15
16 x = 500 # change the value, so we don't need to change the previous statement to apply the change
17 print(format(x, FORMAT_1) + SEPARATOR, format(x ** 2, FORMAT_2), format(math.sqrt(x), FORMAT_3))
18
19 x = 20 # change the value, so we don't need to change the previous statement to apply the change
20 print(format(x, FORMAT_1) + SEPARATOR, format(x ** 2, FORMAT_2), format(math.sqrt(x), FORMAT_3))
21
22 x = 3000 # change the value, so we don't need to change the previous statement to apply the change
23 print(format(x, FORMAT_1) + SEPARATOR, format(x ** 2, FORMAT_2), format(math.sqrt(x), FORMAT_3))
24
25 # The footer of the table
26 print("-----")
```



# Print a Table

## Discussion

- Implementation A and Implementation B are both correct.
- However, Implementation B is better than Implementation A because it **requires fewer modifications on the code** when you want to change the formats of the columns and the values of x.
- In other words, Implementation B is **flexible**.



# Check Point #4

Show the printout of the following statements:

- `print(format(57.467657, "9.3f"))` → 57.468
- `print(format(12345678.923, "9.1f"))` → 12345678.9
- `print(format(57.4, ".2f"))` → 57.40
- `print(format(57.4, "10.2f"))` → 57.40







# Check Point #5

Show the printout of the following statements:

- `print(format("Programming is fun", "25s"))`



Programming is fun

- `print(format("Programming is fun", "<25s"))`



Programming is fun

- `print(format("Programming is fun", ">25s"))`



Programming is fun



# End

- Test Questions
- Programming Exercises

# Test Questions

- Do the test questions for this chapter online at <https://liveexample-ppe.pearsoncmg.com/selftest/selftestpy?chapter=3>

**Introduction to Programming Using Python, Y. Daniel Liang**

This quiz is for students to practice. A large number of additional quiz is available for instructors from the Instructor's Resource Website.

**Chapter 3 Mathematical Functions, Strings, and Objects**

[Check Answer for All Questions](#)

**Section 3.2 Common Python Functions**

3.1 What is `max(3, 5, 1, 7, 4)`?

A. 1  
 B. 3  
 C. 5  
 D. 7  
 E. 4

[Check Answer for Question 1](#)

3.2 What is `min(3, 5, 1, 7, 4)`?

A. 1  
 B. 3  
 C. 5  
 D. 7  
 E. 4

[Check Answer for Question 2](#)

3.3 What is `round(3.52)`?

A. 3.5  
 B. 3  
 C. 5  
 D. 4  
 E. 3.0

[Check Answer for Question 3](#)

3.4 What is `round(6.5)`?

A. 4  
 B. 5  
 C. 6  
 D. 7



# Programming Exercises

- Page 85 – 88:
  - 3.1 - 3.5
  - 3.8 - 3.9
  - 3.11
- Lab #5

