CPS 101 Introduction to Computational Science

Wensheng Shen

Department of Computational Science
SUNY Brockport
Chapter 10: Program in Python

- Introduction to Python Programming
- Control Structures
- Functions
- Lists, Tuples and Dictionaries
- Introduction to the Common Gateway Interface (CGI)
- Object-Based Programming
- Customizing Classes
Computer languages

- **Machine languages**
  - Streams of 0’s and 1’s
  - Defined by the computer’s hardware design
  - Machine-dependent

- **Assembly languages**
  - Need translator programs called assemblers
  - Assemblers convert assembly language programs to machine language

- **High-level languages**
  - Enable programmers to write instructions that look almost like everyday English and contain common mathematical notations.
High-level languages

- Compiled languages
  - Programming languages that are implemented by compilers, which generate machine code from source code.

- Interpreted languages
  - Programming languages that are implemented by interpreters, which translates the source code to another form to interpret.
High-level languages

- **FORTRAN (FORmula TRANslator)**
  - Developed by IBM between 1954 and 1957.

- **COBOL (Common Business Oriented Language)**
  - Developed in 1959 by a group of computer manufacturers and government and industrial computer users.

- **C**
  - Developed at Bell Laboratory. Language for writing operating-systems software and compilers.

- **C++**
  - Developed by Bjarne Stroustrup at AT&T in the early 1980s.

- **JAVA**
  - Developed by researchers at Sun Microsystems in the early 1990s. Purely object-oriented language.
The origin of C language

(1) C language was developed by Dennis M. Ritchie between 1972 to 1973 at Bell Laboratory.
(2) It was used to develop the famous Unix operating system in 1973 by D.M. Ritchie and K. Thompson.
(3) They were awarded Turing Award in 1983.
(4) They were awarded National Medal of technology from President of Clinton in 1999.
The Origin of Python Language

- It was developed by Guido van Rossum in 1989.
- An open-source programming language
- A powerful general-purpose programming language
- Effective for developing Internet and Web-based applications.
- Effective for database intensive applications
- Effective for client/server systems.
10.1 Introduction to Python

- First program in Python
- Modifying the first program
- Another Python program: adding integers
- Memory concepts
- Arithmetic
- String formatting
- Equality and relational operators
- Indentation
- Thinking about objects
Python in Windows

- Install Python on your own computer.
  - Add python.exe to system path variable
- Two ways to execute Python statements
  - File mode: Create a program, save it to a file with a .py extension, and use the Python interpreter to execute the program in the file:
    - python file.py
  - Interactive mode: Execute the Python statements interactively:
    - the shell command line runs the Python interpreter in interactive mode.
    - the programmer types statements directly to the interpreter
    - the interpreter executes the statements one at a time
First Program in Python

# printing a line of text in Python.

print ("Welcome to Python!")

• Lines beginning with the ‘#’ symbol are comments.
• Comments are used to document a program and to improve the readability of the program.
• The entire line is called a statement.
• A string of characters are contained by a pair of quotation marks.
• Output and input in Python are accomplished with streams of characters.

Save the file as example1.py, and run it using file mode.
The first three lines display information about the version of Python being used.
“>>>” is the Python prompt (The interpreter executes the statement when a programmer types a statement at the Python prompt and presses the Enter key.)
After printing the text to the screen, the interpreter waits for the user to enter the next statement.
The interpreter mode can be terminated by typing the Ctrl-Z, end-of-file character on Microsoft Windows systems, and pressing the Enter key.
Modifying the First Python Program

- Displaying a single line of text with multiple statements

```python
# printing a line with multiple statements.
print ("Welcome", end = " ")
print ("to Python!")
```

The comma (,) and (end = " ") at the end of a print statement tells Python not to begin a new line but instead to add a space after the string.
Modifying the First Python Program

Displaying multiple lines of text with a single statement

```python
# printing multiple lines with a single statement.
print("Welcome\nto\n
Python!")
```

A single statement can display multiple lines using *newline characters*. Newline characters are special characters that position the screen cursor to the beginning of the next line. Special characters are formed using the backslash (\) character.
### Special Characters in Python

<table>
<thead>
<tr>
<th>Special character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>Newline. Move the cursor to the beginning of the next line</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal tab. Move the cursor to the next tab stop</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return. Move the cursor to the beginning of the current line; do not advance to the next line</td>
</tr>
<tr>
<td>\b</td>
<td>Back space. Move the cursor back one space</td>
</tr>
<tr>
<td>\a</td>
<td>Alert. Sound the system bell</td>
</tr>
<tr>
<td>\</td>
<td>Backslash. Print a backslash character</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote. Print a double quote character</td>
</tr>
<tr>
<td>'</td>
<td>Single quote. Print a single quote character</td>
</tr>
</tbody>
</table>

How to print a blank line?
Another Python Program: Adding Integers

Inputs two integers by a user from the keyboard, computes the sum of the values and displays the result.

```python
#simple addition.

#prompt user for input
integer1 = input("Enter first integer:\n") #read string
integer1 = int(integer1) #convert string to integer

integer2 = input("Enter second integer:\n:")
integer2 = int(integer2)

sum = integer1 + integer2 #compute and assign sum

print ("Sum is", sum) #print sum
```
Explanation of the Program

raw_input():

Python’s built-in function takes the argument “Enter first integer:\n” reads a string

variables:

- integer1 and integer2 are variables
- variable names include letters, numbers, underscore, but **cannot** begin with numbers

Python uses dynamic typing --- determining an object’s type during program execution

int(integer1):
- convert string to integer

“=“: assignment symbol
“+”: binary operator
Example --- Dynamic Typing

What is the output of this code?

```python
#Dynamics typing
value1 = input("Enter an integer: \n")
value2 = input("Enter an integer: \n")

sum = value1 + value2
print (sum)

print (value1 + value2)
```
Memory Concepts

- Variable names such as integer1, integer2, and sum correspond to Python objects.
- Every object has a type, a size, a value, and a location in the computer’s memory.
- A program cannot change an object’s type or location.

```python
integer1 = input("Enter first integer:\n")
integer1 = int(integer1)
integer2 = input("Enter second integer:\n")
integer2 = int(integer2)
sum = integer1 + integer2
```
#simple addition.

#prompt user for input
ingeger1 = input("Enter first integer:\n") #read string
print ("integer1: ", id(integer1), type(integer1), integer1)
ingeger1 = int(integer1) #convert string to integer
print ("integer1: ", id(integer1), type(integer1), integer1)

ingeger2 = input("Enter second integer:\n:"
print ("integer2: ", id(integer2), type(integer2), integer2)
ingeger2 = int(integer2)
print ("integer2: ", id(integer2), type(integer2), integer2)

sum = integer1 + integer2 #compute and assign sum
print ("sum: ", id(sum), type(sum), sum) #print sum
# Arithmetic

<table>
<thead>
<tr>
<th>Python operation</th>
<th>Arithmetic operator</th>
<th>Algebraic expression</th>
<th>Python expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>+</td>
<td>$a+3$</td>
<td>a+3</td>
</tr>
<tr>
<td>subtraction</td>
<td>-</td>
<td>$f-g$</td>
<td>f-g</td>
</tr>
<tr>
<td>multiplication</td>
<td>*</td>
<td>$ab$</td>
<td>a*b</td>
</tr>
<tr>
<td>exponentiation</td>
<td>**</td>
<td>$x^y$</td>
<td>x**y</td>
</tr>
<tr>
<td>division</td>
<td>/</td>
<td>$x/y, x\div y$</td>
<td>x/y</td>
</tr>
<tr>
<td></td>
<td>// (new in Python)</td>
<td>$x/y, x\div y$</td>
<td>x//y</td>
</tr>
<tr>
<td>modulus</td>
<td>%</td>
<td>$x \ mod \ y$</td>
<td>x%y</td>
</tr>
</tbody>
</table>
Arithmetic

In default, “/”

--- floor division for integers
--- true division for non integers

If “from __future__ import division” is execute,
“/” for true division for integers
“//” for floor division

Python 2.6.3 (r263rc1:75186, Oct 2 2009, 20:40:30) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>3/4 #floor division
0
>>>3.0/4.0 #true division
0.75
>>>3//4 #floor division
0
>>>3.0/4.0 #floating-point floor division
0
>>>from __future__ import division
>>>3/4 #true division
0.75
>>>3.0/4.0 #true division
0.75
Arithmetic

“%” (the modulus operator):
--- it yields the remainder after integer division
--- a%b yields the remainder after a is divided by b
--- it can be used to determine if one number is a multiple of another.

What is 7%5?

What is 13%4?

Arithmetic expressions in Python must be entered into the computer in straight-line form. The algebraic notation
\[
\frac{x}{y}
\]
must by written as “x/y” in Python
# Arithmetic

<table>
<thead>
<tr>
<th>Operators</th>
<th>Operations</th>
<th>Order of evaluation (precedence)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>Parentheses</td>
<td>Evaluated first. If they are nested, the expression in the innermost pair is evaluated first. If there are multiple pairs at the same level, they are evaluated from left to right</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
<td>Evaluated second. If there are several, they are evaluated right to left</td>
</tr>
<tr>
<td>* / // %</td>
<td>Multiplication division modulus</td>
<td>Evaluated third. If there several, they are evaluated left to right</td>
</tr>
<tr>
<td>+ -</td>
<td>Addition subtraction</td>
<td>Evaluated last. If there are several, they are evaluated left to right</td>
</tr>
</tbody>
</table>
Example: How the second-degree polynomial is evaluated?

\[ y = a \cdot x^2 + b \cdot x + c \]
String Formatting

- Python provides string as a built-in data type.
- Python strings can be created by double quotes, single quotes, and even triple-quoted strings.

```python
# create strings using quotation marks

print("This is a string with "double quotes."\"")
print('This is another string with “double quotes.”\')
print('This is a string with ‘single quotes.\’")
print("This is another string with ‘single quotes.’")
print("""This string has “double quotes” and ‘single quotes’.
    You can even do multiple lines. """")
print("""This string also has “double” and ‘single’ quotes."")
```
## String Formatting

<table>
<thead>
<tr>
<th>Formatting symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Single character</td>
</tr>
<tr>
<td>s</td>
<td>String</td>
</tr>
<tr>
<td>d</td>
<td>Signed decimal integer</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned decimal integer</td>
</tr>
<tr>
<td>o</td>
<td>Unsigned octal integer</td>
</tr>
<tr>
<td>x</td>
<td>Unsigned hexadecimal integer (lower case)</td>
</tr>
<tr>
<td>X</td>
<td>Unsigned hexadecimal integer (upper case)</td>
</tr>
<tr>
<td>f</td>
<td>Floating-point number</td>
</tr>
<tr>
<td>e, E</td>
<td>Floating-point number (scientific notation)</td>
</tr>
<tr>
<td>g, G</td>
<td>Floating-point number (using least-significant digits)</td>
</tr>
</tbody>
</table>

\ is the line-continuation character
#string formatting

integerValue = 4237
print ("Integer ", integerValue)
print ("Decimal integer %d" %integerValue)
print ("Hexadecimal integer %x\n" %integerValue)

floatValue = 123456.789
print ("Float", floatValue)
print ("Default float %f" %floatValue)
print ("Default exponential %e\n" %floatValue)

print ("Right justify integer (%8d)" %integerValue)
print ("Left justify integer (%-8d)\n" %integerValue)

stringValue = "String formatting"
print ("Force eight digits in integer %.8d" %integerValue)
print ("Five digits after decimal in float %.5f" %floatValue)
print ("Fifteen and five characters allowed in string:")
print ("(%.15s) (%.5s) %s" %(stringValue, stringValue))
### Decision Making: Equality and Relational Operators

<table>
<thead>
<tr>
<th>Standard algebraic equality operator or relational operator</th>
<th>Python equality or relational operator</th>
<th>Example of Python condition</th>
<th>Meaning of Python condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational operators</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td>a&gt;b</td>
<td>a is greater than b</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>a&lt;b</td>
<td>a is less than b</td>
</tr>
<tr>
<td>≥</td>
<td>≥</td>
<td>a&gt;=b</td>
<td>a is greater than or equal to b</td>
</tr>
<tr>
<td>≤</td>
<td>≤</td>
<td>a&lt;=b</td>
<td>a is less than or equal to b</td>
</tr>
<tr>
<td>Equality operators</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>==</td>
<td>a==b</td>
<td>a is equal to b</td>
</tr>
<tr>
<td>≠</td>
<td>!=, &lt;&gt;</td>
<td>a&lt;&gt;b, a!=b</td>
<td>a is not equal to b</td>
</tr>
</tbody>
</table>
#compare integers using if structures, relational operators and equality operators.

Print ("Enter two integers, and I will tell you")
Print ("the relationships they have.")
number1 = input("Enter the first integer: ")
number1=int(number1)
number2=input("Enter the second integer: ")
number2=int(number2)
if number1==number2:
    print (%d is equal to %d % (number1, number2))
if number1 != number2:
    print (%d is not equal to %d % (number1, number2))
if number1<number2:
    print (%d is less than %d % (number1, number2))
if number1>number2:
    print (%d is greater than %d % (number1, number2))
if number1<=number2:
    print (%d is less than or equal to %d % (number1, number2))
if number1>=number2:
    print (%d is greater than or equal to %d % (number1, number2))
# Decision Making: Equality and Relational Operators

```python
# compare integers using if structures, relational operators and equality operators.

Print ("Enter two integers, and I will tell you")
Print ("the relationships they have.")
number1 = input("Enter the first integer: ")
number1=int(number1)
number2=input("Enter the second integer:"")
number2=int(number2)
if number1==number2:
    print ("%d is equal to %d" %(number1, number2))
if number1 != number2:
    print ("%d is not equal to %d" %(number1, number2))
if number1<number2:
    print ("%d is less than %d" %(number1, number2))
if number1>number2:
    print ("%d is greater than %d" %(number1, number2))
if number1<=number2:
    print ("%d is less than or equal to %d" %(number1, number2))
if number1>=number:
    print ("%d is greater than or equal to %d" %(number1, number2))
```
### Indentation

- Python uses indentation to distinguish sections of code.
- Lines of code should be uniformly indented in a single section.
- Indentation in Python should be ensured consistent and proper.

```python
# compare integers using if structures, relational operators and equality operators.

print(“Enter two integers, and I will tell you”)  
print(“the relationships they have.”) 
number1 = input(“Enter the first integer: ”)  
number1=int(number1) 
number2=input(“Enter the second integer:”) 
number2=int(number2) 
if number1==number2: 
    print(“%d is equal to %d”%(number1, number2))  
if number1 != number2: 
    print(“%d is not equal to %d”%(number1, number2)) 
if number1<number2: 
    print(“%d is less than %d”%(number1, number2)) 
if number1>number2: 
    print(“%d is greater than %d”%(number1, number2)) 
if number1<=number2: 
    print(“%d is less than or equal to %d”%(number1, number2)) 
if number1>=number: 
    print(“%d is greater than or equal to %d”%(number1, number2))
```
Thinking About Objects: Introduction to Object Technology

- **Structured program**
  - Functions are the units of program
  - Concentrated on writing functions

- **Object-oriented program**
  - Classes are the units of program
  - Concentrated on building classes

- **Advantages of object-oriented program**
  - A natural way of thinking
  - The ability of abstraction
  - Inheritance relationships or multiple inheritance relationships
  - Classes are reused in future software systems

- **Basic components of object-oriented program**
  - Attributes (data members)
  - Methods (behavior)
10.2 Control Structures

- Algorithms
- Pseudocode
- Control structures
- If selection structures
- If/else and if/elif/else selection structures
- While repetition structure
- Formulating algorithms
  - counter-controlled repetition
  - Sentinel-controlled repetition
  - Nested control structures
- Augmented assignment symbols
- Essentials of counter-controlled repetition
- For repetition structure
  - Using the for repetition structure
  - Break and continue statements
- Logical operators
- Structured-programming summary
Algorithms

- An algorithm is a procedure for solving a problem in terms of:
  - Actions to be taken
  - The order in which these actions are to be executed
- The “rise-and-shine” algorithm

(a) Get out of bed  (a) Get out of bed
(b) Take off pajamas  (b) Take off pajamas
(c) Take a shower  (c) Get dressed
(d) Get dressed  (d) Take a shower
(e) Eat breakfast  (e) Eat breakfast
(f) Carpool to work (f) Carpool to work
Pseudocode is an artificial and informal language. Pseudocode helps programmers develop algorithms. Pseudocode is similar to every English. It is convenient and user-friendly. Pseudocode statements will be converted to Python computer language.

```python
if student’s grade is greater than or equal to 60
    print ("passed")
else
    print ("failed")
```
Control Structures

- Structured programs
  - Do not contain “goto” statements.

- All programs can be written in terms of three control structures
  - The sequence structure
  - The selection structure
    - There are three types of selection structures
  - The repetition structure
    - There are two kinds of repetition structures

- A flowchart can also be used to provide graphical representation of an algorithm

```
add grade to total  total = total + grade
add 1 to counter  counter = counter + 1
```
Python Keywords

- There are 28 keywords in Python
- Keywords are reserved by the language to implement control structures
- Keywords cannot be used as identifies (i.e., variable names)

### Python keywords

<table>
<thead>
<tr>
<th>and</th>
<th>continue</th>
<th>else</th>
<th>for</th>
<th>import</th>
<th>not</th>
<th>raise</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert</td>
<td>def</td>
<td>except</td>
<td>from</td>
<td>in</td>
<td>or</td>
<td>return</td>
</tr>
<tr>
<td>break</td>
<td>del</td>
<td>exec</td>
<td>global</td>
<td>is</td>
<td>pass</td>
<td>try</td>
</tr>
<tr>
<td>class</td>
<td>elif</td>
<td>finally</td>
<td>if</td>
<td>lambda</td>
<td>print</td>
<td>while</td>
</tr>
</tbody>
</table>
if Selection Structure

- Pseudocode
  ```
  if student’s grade is greater than or equal to 60
  print ("passed")
  ```

- Flowchart
  ```
  grade >= 60
  
  true
  
  print "passed"
  ```

- Python program
  ```python
  if grade >= 60:
    print ("passed")
  ```
if / else Selection Structure

- Pseudocode

```python
if student's grade is greater than or equal to 60
    print ("passed")
else
    print ("failed")
```

- Flowchart

- Python program

```python
if grade >= 60:
    print ("passed")
else:
    print ("failed")
```
if /else Selection Structure

if student’s grade is greater than or equal to 90
    print ("A")
else
    if student’s grade is greater than or equal to 80
        print ("B")
    else
        if student’s grade is greater than or equal to 70
            print ("C")
        else
            if student’s grade is greater than or equal to 60
                print ("D")
            else:
                print ("E")

if grade >= 90:
    print ("A")
else:
    if grade >= 80:
        print ("B")
    else:
        if grade >= 70:
            print ("C")
        else:
            if grade >= 60
                print ("D")
            else:
                print ("E")
if /elif/else Selection Structure

if student’s grade is greater than or equal to 90
    print ("A")
elif student’s grade is greater than or equal to 80
    print ("B")
elif student’s grade is greater than or equal to 70
    print ("C")
elif student’s grade is greater than or equal to 60
    print ("D")
else
    print ("E")

if grade >= 90:
    print ("A")
elif grade >= 80:
    print ("B")
elif grade >= 70:
    print ("C")
elif grade >= 60:
    print ("D")
else:
    print ("E")
if /else Selection Structure

- A suite can contain more than one statement: compound statement

```python
If grade >= 60:
    print ("passed.")
else:
    print ("failed")
    print "You must take this course again"
```

```python
If grade >= 60:
    print ("passed.")
else:
    print ("failed")
    print ("You must take this course again")
```
while repetition

- A repetition structure allows a computer program to repeat an action for many times

```plaintext
While there are more items on my shopping list
purchase next item and cross it off my list
```

- The condition “there are more items on my shopping list” is either true or false.
- If it is true, the program performs the action “purchase next item and cross it off my list”
- If the condition remains true, the action is performed repeatedly.
- If the condition is false, the repetition terminates, and the program executes the first statement after the repetition structure.
- If the condition is never false, the repetition will run forever.
while repetition

$\text{product} = 2$

While $\text{product} \leq 1000$:
  $\text{product} = \text{product} \times 2$

• What are the values of product?
• What is the final value of product?
Counter-controlled repetition ---
definite repetition

**problem:** A class of ten students took a quiz. The grades (integers in the range 0 – 100) for this quiz are available. Determine the class average on the quiz.

Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
    input the next grade
    add the grade to the total
    add one to the grade counter

Set the class average to the total divided by ten
Print the class average

```python
total = 0
counter = 1

while counter <= 10:
    grade = raw_input("Enter grade:\n")
    grade = int(grade)
    total = total + grade
    counter = counter + 1

average = total / 10;
print ("Class average is ", average)
```
Sentinel-controlled repetition --- indefinite repetition

**Problem:** Develop a class-averaging program that processes an arbitrary number of grades each time the program is executed.

One way to solve this problem is to use a special value called a sentinel value, a signal value, a dummy value, or a flag value.

1. Initialize total to zero
2. Initialize counter to zero
3. Input the first grade (possibly the sentinel)
4. While the input is not the sentinel
   - Add this grade into total
   - Add one to the counter
   - Input next grade (possibly the sentinel)
5. If the counter is not equal to zero
   - Set the average to the total divided by the counter
   - Print the average
6. Else
   - Print “No grades were entered”
total = 0
counter = 0

grade = input("Enter grade (-1 to end):")
grade = int(grade)

while grade != -1:
    total = total + grade
    counter = counter + 1
    grade = raw_input("Enter grade (-1 to end):")
    grade = int(grade)

if counter != 0:
    average = float(total) / counter
    print ("Class average is: ", average)
else
    print ("No grades were entered")
Problem: A college offers a course that prepares students for the state licensing exam for real estate broker. Last year, several of the students who completed this course took the licensing examination. Naturally, the college wants to know how well its students did on the exam. You have been asked to write a program to summarize the results. You have been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam and a 2 if the student failed.

Your program should analyze the results of the exam as follows:
1. input each test results (i.e., a 1 or a 2). Display the message “Enter result” on the screen each time the program requests another test result.
2. Count the number test results of each type.
3. Display a summary of the test results indicating the number of students who passed and the number students who failed.
4. If more than 8 students passed the exam, print the message “Raise tuition”.
Nested Control Structures

Initialize pass to zero
Initialize failure to zero
Initialize counter to one

While counter is less than or equal to ten
  input the next exam result
  if the student passed
    add one to pass
  else
    add one to failure

Print the number of passes
Print the number of failures

If more than eight students passed
  print “Raise tuition”
Nested Control Structures

```python
pass = 0
failure = 0
counter = 1

while counter <= 10:
    result = raw_input("Enter result (1=pass, 2=fail): \n")
    result = int(result)
    if result == 1:
        pass = pass + 1;
    else:
        failure = failure + 1;

print ("Passed", pass)
print ("Failed", failure)

if pass >= 8:
    print ("Raise tuition")
```
Augmented Assignment Symbols

\[
\text{variable} = \text{variable operator expression} \\
\text{variable operator=} \text{ expression}
\]

<table>
<thead>
<tr>
<th>Assignment symbol</th>
<th>Sample expression</th>
<th>Explanation</th>
<th>Assigns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assumption: c = 3, d = 5, e = 4, f = 2, g = 6, h = 12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+=</td>
<td>c += 7</td>
<td>c = c + 7</td>
<td>10 to c</td>
</tr>
<tr>
<td>-=</td>
<td>d -= 4</td>
<td>d = d - 4</td>
<td>1 to d</td>
</tr>
<tr>
<td>*=</td>
<td>e *= 5</td>
<td>e = e*5</td>
<td>20 to e</td>
</tr>
<tr>
<td>**=</td>
<td>f **= 3</td>
<td>f = f**3</td>
<td>8 to f</td>
</tr>
<tr>
<td>/=</td>
<td>g /= 3</td>
<td>g = f/3</td>
<td>2 to g</td>
</tr>
<tr>
<td>%=</td>
<td>h %= 9</td>
<td>h = h%9</td>
<td>3 to h</td>
</tr>
</tbody>
</table>
Essentials of Counter-Controlled Repetition

1. The name of the control variable
2. The initial value of the control variable
3. The amount of increment/decrement by which the control variable is modified each time through the loop, and
4. The condition that tests for the final value of the control variable (i.e., whether looping should continue)

counter = 0

while counter < 10:
    print counter
counter += 1

print ("the value of counter after the loop is: ", counter)
for Repetition Structure

1. Keywords: for and in
2. Function \texttt{range()}
   1. It can take one argument \texttt{range(end)}
   2. It can take two arguments \texttt{range(start, end)}
   3. It can take three arguments \texttt{range(start, end, increment)}

\texttt{range(end)} returns a sequence in the range:
   \( 0 \sim (end - 1) \)
\texttt{range(start, end)} return a sequence in the range:
   \( (start) \sim (end - 1) \)
\texttt{range(start, end, increment)} returns a sequence in the range:
   \( (start) \sim (end - 1) \) incremented \textit{by increment}

counter = 0
while counter < 10:
    print (counter)
    counter += 1

for counter in range(10):
    print (counter)

Are they the same?
\texttt{range(10)}
\texttt{range(0,10)}
\texttt{range(0,10,1)}
for Repetition Structure

for x in y:
    print x

x = first item in y

More items to process

false

true

print x

x = next item in y
Using the for Repetition Structure

(a) Vary the control variable from 1 to 100 in increments of 1
   for counter in range (1, 101):

(b) Vary the control variable from 100 to 1 in increments of -1 (decrement of 1)
   for counter in range(100, 0, -1):

(c) Vary the control from 7 to 77 in steps of 7
    for counter in range(7, 78, 7):

(d) Vary the control variable from 20 to 2 in steps of -2
    for counter in range(20, 1, -2):

(e) Vary the control variable over the following sequence of values: 2, 5, 8, 11, 14, 17, 20.
    for counter in range(2, 21, 3):

(f) Vary the control variable over the following sequence of values: 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.
    for counter in range(99, -1, -11):
Using the for Repetition Structure

# summation with for.
sum = 0
for number in range(2, 101, 2):
    sum += number
print ("Sum in", sum)
A person invests $1,000 in a savings account yielding 5 percent interest. Assuming that all interest is left on deposit in the account, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula for determining these amounts:

\[ a = p(1 + r)^n \]

Where
- \( p \) is the original amount invested (i.e., the principal),
- \( r \) is the annual interest rate,
- \( n \) is the number of years, and
- \( a \) is the amount on deposit on the end of the \( n \)th year.

```python
# calculating compound interest.
principal = 1000.0
rate = 0.05

print ("Year %21s" % "Amount on deposit")
for year in range(1,11):
    amount = principal*(1.0 + rate)**year
    print ("%4d%21.2f" %(year, amount))
```
break and continue Statements

- Both break and continue statements can be used in for and while loops.
- The break statement causes immediate exit from a loop structure.
- The continue statement skips the remaining statements in the body of that structure and proceeds with the next iteration of the loop.

```python
# Using the break statement in a for structure.
for x in range(1,11):
    if x == 5:
        break
    print (x),
print ("\nBroke out of loop at x = ", x)
```
# Using the break statement to avoid repeating code in the
# class-average program.

total = 0
counter = 0

while 1:
    grade = input("Enter grade (-1 to end):")
    grade = int(grade)

    if grade == -1:
        break
    break
    total += grade
    counter += counter

if counter != 0:
    average = float(total)/counter
    print ("Class average is ", average)
else:
    print ("No grades were entered")
break and continue Statements

# Using the continue statement in a for/in structure
for x in range(1, 11):
    if x == 5:
        continue
    print(x),
print("\nUsed continue to skip printing the value 5")
Logical Operators

(a) Simple conditions contain one condition
(b) Complex conditions contain more than one condition grouped using logical operators:
   (a) and (logical AND),
   (b) or (logical OR),
   (c) not (logical NOT)

if gender == “Female” and age >= 65:
   seniorFemales += 1

<table>
<thead>
<tr>
<th>expression1</th>
<th>expression2</th>
<th>expression1 and expression2</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>
## Logical Operators

<table>
<thead>
<tr>
<th>expression1</th>
<th>expression2</th>
<th>expression1 or expression2</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>expression</th>
<th>not expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>
10.3 Functions

- Program components in Python
- Functions
- Module *math* functions
- Function definitions
- Random-number generation
- Example: a game of chance
- Scope rules
- Keyword *import* and *namespaces*
- Recursion
- Example using recursion: the Fibonacci series
- Recursion vs. iteration
- Default arguments
- Keyword arguments
Program components in Python are:
- functions
- classes
- modules
- packages

Functions can be defined by programmers to perform specific tasks.
A module is a file that contains definitions of functions and classes.
A package is a collection of modules.
Functions

Functions are invoked by a function call
    calling function --- caller
    called function --- callee
    called function should return
All variables created in function definitions are local variables

Why functions?
(1) The divide-and-conquer approach makes program development more manageable
(2) Functions makes software reusable
(3) Functions are used to avoid repeating code in a program
Module **math** Functions

The math module contains functions that allow programmers to perform certain mathematical calculations.

A program must import the module in order to use it

```
* import math
```

Functions are invoked by
- writing the name of the function,
- followed by a left parenthesis,
- followed by the argument (or a comma-separated list of arguments) being passed to the function,
- followed by a right parenthesis

Functions in math module are invoked by
- the module’s name
- a dot
- the function call

Example:
```
print math.sqrt(100)
```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos(x)</td>
<td>arc cosine of x (result in radians)</td>
<td>acos(1.0) is 0.0</td>
</tr>
<tr>
<td>asin(x)</td>
<td>arc sine of x (result in radians)</td>
<td>asin(0.0) is 0.0</td>
</tr>
<tr>
<td>atan(x)</td>
<td>arc tangent of x (result in radians)</td>
<td>Atan(0.0) is 0.0</td>
</tr>
<tr>
<td>ceil(x)</td>
<td>rounds x to the smallest integer not less than x</td>
<td>Ceil(9.2) is 10.0, ceil(-9.8) is -9.0</td>
</tr>
<tr>
<td>cos(x)</td>
<td>cosine of x (x in radians)</td>
<td>cos(0.0) is 1.0</td>
</tr>
<tr>
<td>exp(x)</td>
<td>exponential function $e^x$</td>
<td>exp(1.0) is 2.71828</td>
</tr>
<tr>
<td>fabs(x)</td>
<td>absolute value of x</td>
<td>fabs(-5.0) is 5.0</td>
</tr>
<tr>
<td>floor(x)</td>
<td>rounds x to the largest integer not greater than x</td>
<td>floor(-9.8) is -10.0</td>
</tr>
<tr>
<td>fmod(x)</td>
<td>remainder of $x/y$ as a floating point number</td>
<td>fmod(9.8, 4.0) is 1.8</td>
</tr>
<tr>
<td>hypot(x, y)</td>
<td>Hypotenuse of a triangle with sides of length x and y: $\sqrt{x^2+y^2}$</td>
<td>hypot(3.0, 4.0) is 5.0</td>
</tr>
<tr>
<td>log(x)</td>
<td>natural logarithm of x (base e)</td>
<td>log(2.718282) is 1.0</td>
</tr>
<tr>
<td>log10(x)</td>
<td>logarithm of x (base 10)</td>
<td>log10(10.0) is 1.0</td>
</tr>
<tr>
<td>pow(x, y)</td>
<td>x raised to power y ($x^y$)</td>
<td>Pow(2.0, 7.0) is 128.0</td>
</tr>
<tr>
<td>sin(x)</td>
<td>sine of x (x in radians)</td>
<td>sin(0.0) is 0.0</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>square root of x</td>
<td>sqrt(100.0) is 10.0</td>
</tr>
<tr>
<td>tan(x)</td>
<td>tangent of x (x in radians)</td>
<td>tan(0.0) is 0.0</td>
</tr>
</tbody>
</table>
Function Definitions

def function-name( parameter-list ):  
  statements

# creating and using a programmer-defined function

def square(y):
  return y*y

for x in range (1, 11):
  print (square(x),)

Print()
# find the maximum of three integers.
def max(x, y, z):
    max = x
    if y > max:
        max = y
    if z > max:
        max = z
    return max

a = int(input("Enter first integer: \n"))
b = int(input("Enter second integer: \n"))
c = int(input("Enter third integer: \n"))

d = float(input("Enter first float: \n"))
e = float(input("Enter second float: \n"))
f = float(input("Enter third float: \n"))

print ("Maximum integer is: ", max(a, b, c))

print ("Maximum float is: ", max(a, b, c))

print ("Enter first string: \n")
g = input()
h = input()
i = float(input("Enter third string: \n"))

print ("Maximum string is: ", max(g, h, i))
Random Number Generation

- The element of chance can be introduced into computer applications through module random
- Function random.randrange generates an integer in the range of its first argument upto, but not including, its second argument.

```python
# Random integers produced by randrange.
import random
for i in range(1,21):  #simulate 20 die rolls
    (print "%10d" % (random.randrange(1,7)), end = '')
    if i%5 == 0:  #print newline every 5 rolls
        print()
```
# Roll a six-sided die 6000 times.
import random

frequency1 = 0
frequency2 = 0
frequency3 = 0
frequency4 = 0
frequency5 = 0
frequency6 = 0

for roll in range(1, 6001):
    face = random.randrange(1, 7)
    if face == 1:
        frequency1 += 1
    elif face == 2:
        frequency2 += 1
    elif face == 3:
        frequency3 += 1
    elif face == 4:
        frequency4 += 1
    elif face == 5:
        frequency5 += 1
    elif face == 6:
        frequency6 += 1
    else:
        print("Something is wrong!")

print ("Face %13s" % "Frequency")
print (" 1 %13d" % frequency1)
print (" 2 %13d" % frequency2)
print (" 3 %13d" % frequency3)
print (" 4 %13d" % frequency4)
print (" 5 %13d" % frequency5)
print (" 6 %13d" % frequency6)
A player rolls two dice. Each die has six faces. These faces contain 1, 2, 3, 4, 5 and 6 spots. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, the player wins. If the sum is 2, 3 or 12 on the first throw (called “craps”), the player loses (i.e., the “house” wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, then that sum becomes the player’s “point.” To win, you must continue rolling the dice until you “make your point.” The player loses by rolling a 7 before making the point.
```python
# Craps.
import random

def rollDice():
    die1 = random.randrange(1, 7)
    die2 = random.randrange(1, 7)
    workSum = die1 + die2
    print("Player rolled %d + %d = %d" % (die1, die2, workSum))
    return workSum

sum = rollDice() # first dice roll

if sum == 7 or sum == 11:
    gameStatus = "WON"
elif sum == 2 or sum == 3 or sum == 12:
    gameStatus = "LOST"
else:
    gameStatus = "CONTINUE"
    myPoint = sum
    print("Point is", myPoint)
while gameStatus == "CONTINUE":
    sum = rollDice()
    if sum == myPoint:
        gameStatus = "WON"
    elif sum == 7:
        gameStatus = "LOST"
if gameStatus == "WON":
    print("Player wins")
else:
    print("Player loses")
```
Scope Rules

- The scope of the value of a variable is defined by certain rules.
- The rules are described in terms of namespaces and scopes.

```python
print x
```

- Python must first find the identifier named x and determine the value associated with that identifier.
- Namespaces store information about an identifier and the value to which it is bound.
- Python defines three namespaces: local, global, and built-in.
- Python searches the namespaces in a certain order --- local, global, and built-in namespaces.
- Each function has a unique local namespace.
- One function cannot access the local namespace of another function.

- If the function’s local namespace does not contain an identifier names x, Python searches the next outer namespace --- the global namespace.
Scope Rules

• The global namespace contains the bindings for all identifiers, function names and class names defined within a module or file.
• Each module or file’s global namespace contains an identifier called __name__ that states the module’s name (e.g., “math” or “random”)
• When a Python interpreter session starts or when the Python interpreter begins executing a program stored in a file, the value of __name__ is “__main__”.

• In the previous example, Python searches for an identifier named x in the global namespace.
• If the global namespace contains the identifier, Python stops searching for the identifier and the function prints the value of x to the screen.
• If the global namespace does not contain an identifier named x, Python searches the next outer namespace --- the built-in namespace.

• The built-in namespace contains identifiers that correspond to many Python functions and error messages. For example, functions raw_input, int and range belong to the built-in namespace.
• Python creates the built-in namespace when the interpreter starts, and programs normally do not modify the namespace.
• In the previous example, the built-in namespace does not contain an identifier named x, so Python stops searching and prints an error message stating that the identifier could not be found.
Scope Rules

- An identifier’s scope describes the region of a program that can access the identifier’s value.
- If an identifier is defined in the local namespace (e.g., in a function), all statements in the block may access the identifier.
- Statements that reside outside the block (e.g., in the main portion of a program or in another function) cannot access the identifier.
- Once the code block terminates (e.g., after a return statement), all identifiers in that block’s local namespace “go out of scope” and are inaccessible.

- If an identifier is defined in the global namespace, the identifier has global scope.
- A global identifier is known to all code that executes, from the point at which the identifier is created until the end of the file.
- Identifiers contained in built-in namespaces may be accessed by code in programs, modules or functions.

- When a function creates a local identifier with the same name as an identifier in the module or built-in namespaces, the local identifier shadows the global or built-in identifier.
- A logic error can occur if the programmer references the local variable when meaning to reference the global or built-in identifier.
Scope Rules

- Python provides a way for programmers to determine what identifiers are available from the current namespace.

- Built-in function `dir()` returns a list of these identifiers.
Scope Rules

# Scoping example.
x = 1  #global variable

#alters the local variable x, shadows the global variable
def a():
    x = 25
    print "\nlocal x in a is", x, "after entering a"
    x += 1
    print "local x in a is", x, "before exiting a"

#alters the global variable x
def b():
    global x
    print "\nglobal x is", x, "on entering b"
    x *= 10
    print ("global x is", x, "on exiting b")
print ("global x is", x)

x = 7
print ("global x is", x)
a()
b()
a()
b()

print ("\nglobal x is", x)
Keyword Import and Namespaces

Importing one or more modules
* import math
math.sqrt(9.0)

Importing identifiers from a module
from math import sqrt
from math import sin, cos, tan
from math import *

Binding names for modules and module identifiers
import random
import random as randomModule
from math import sqrt as squareRoot
Recursion

A recursive function is a function that calls itself, either directly or indirectly. A recursive function knows how to solve only the simplest cases, base cases. A recursive function divides the problem into two conceptual pieces:
* a piece that knows how to solve a base case
* a piece that does not know how to solve
A recursive function call can result in many more such recursive calls, and each of which divides the subproblem into two conceptual pieces. The recursion eventually terminates when the sequence of smaller and smaller problems converge on a base case.

The factorial of a nonnegative integer $n$, written $n!$, is the product $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

Recursive solution
$n! = n(n-1)!$
Recursion

A recursive function is a function that calls itself, either directly or indirectly. A recursive function knows how to solve only the simplest cases, base cases. A recursive function divides the problem into two conceptual pieces:
* a piece that knows how to solve a base case
* a piece that does not know how to solve
A recursive function call can result in many more such recursive calls, and each of which divides the subproblem into two conceptual pieces. The recursion eventually terminates when the sequence of smaller and smaller problems converge on a base case.

The factorial of a nonnegative integer $n$, written $n!$, is the product $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

Recursive solution
$n! = n \cdot (n-1)!$

$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$
$5! = 5 \cdot (4!)$
Recursion

factorial = 1

for counter in range(1, number + 1):
    factorial *= counter

# recursive factorial function.
# recursive definition of function factorial
def factorial (number):
    if number <= 1:  # base case
        return 1
    else:
        return number*factorial(number-1)  # recursive call

for i in range(11):
    print ("%2d! = %d" % (i, factorial(i)))
Example Using Recursion: The Fibonacci Series

The Fibonacci series
0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Fibonacci series can be defined recursively as follows:

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
```

#recursive fibonacci function
def fibonacci(n):
    if n<0:
        print "Cannot find the fibonacci of a negative number."
    if n == 0 or n == 1: #base case
        return n
    else:
        #two recursive calls
        return fibonacci(n-1) + fibonacci(n-2)

number = int(raw_input("Enter an integer: "))
result = fibonacci(number)
print ("Fibonacci(%d) = %d" % (number, result))
Recursive vs. Iteration

Repetitive calculations can be completed by either iteration or recursion. Both iteration and recursion are based on a control structure:
- iteration uses for or while loop
- recursive uses if and if/else structure
Both iteration and recursion can occur infinitely:
- an infinite loop occurs if the loop-continuation test never becomes false
- an infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case
Recursion is very expensive in both processor time and memory space.
Reasons for recursion:
- the program is easier to understand and debug
- the program can be implemented with few lines of code
- an iterative solution may not be apparent
Default Arguments

When defining a function, a programmer can specify an argument as a default argument. The programmer can provide a default value for that argument.

* Default arguments allow a programmer to specify fewer arguments when calling a function.
* When a default argument is omitted in a function call, the interpreter inserts the default value of that argument and passes the argument in the call.

```python
# Using default arguments.
# Function definition with default arguments.
def boxVolume(length = 1, width = 1, height = 1):
    return length * width * height
print("The default box volume is:", boxVolume())
print("\nThe volume of a box with length 10,"
print("width 1 and height 1 is:", boxVolume(10))
print("\nThe volume of a box with length 10,"
print("width 5 and height 1 is:", boxVolume(10, 5))
print("\nThe volume of a box with length 10,"
print("width 5 and height 2 is:", boxVolume(10, 5, 2))
```
Keyword Arguments

The programmer can specify that a function receives one or more keyword arguments. The function definition assigns a default value to each keyword. A function may use a default value for a keyword or a function call may assign a new value to the keyword using the format keyword = value.

#keyword arguments example.

def generateWebsite(name, url = "www.deitel.com", Flash="no", CGI="yes"):
    print ("Generating site requested by", name, "using url", url)
    if Flash == "yes":
        print ("Flash is enabled")
    if CGI == "yes":
        print ("CGI scripts are enabled")
    print() # prints a new line

generateWebsite(" Deitel ")
generateWebsite("Deitel", Flash = "yes", url = "www.deitel.com/new")
generateWebsite(CGI="no", name="Prentice Hall")
Lists, Tuples and Dictionaries

Sequences
Creating Sequences
Using Lists and Tuples
  Using Lists
  Using Tuples
  Sequence Unpacking
  Sequence Slicing
Dictionaries
List and Dictionary Methods
References and Reference Parameters
Passing Lists to Functions
Sorting and Searching Lists
Multiple-Subscripted Sequences
Sequences

A sequence is a series of contiguous values that often are related.
Python strings are sequences, as is the value returned by function range.
range --- a Python built-in function that returns a list of integers
A particular element in a sequence can be referenced by writing the sequence name followed by the element’s position number in sequence brackets([])
In general, the ith element of sequence c is c[i-1].
In sequence c, the first element is c[0], the second element is c[1], the sixth element is c[5].
Sequences can also be accessed from the end. the last element is c[-1] the second to last element is c[-2] the ith-from-the-end is c[-i]
Python lists and dictionaries are mutable --- they can be altered.
c[11] = 0

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C[0]</td>
<td>-45</td>
<td>c[-12]</td>
</tr>
<tr>
<td>c[1]</td>
<td>6</td>
<td>c[-11]</td>
</tr>
<tr>
<td>c[2]</td>
<td>0</td>
<td>c[-10]</td>
</tr>
<tr>
<td>c[3]</td>
<td>72</td>
<td>c[-9]</td>
</tr>
<tr>
<td>c[4]</td>
<td>1543</td>
<td>c[-8]</td>
</tr>
<tr>
<td>c[5]</td>
<td>-89</td>
<td>c[-7]</td>
</tr>
<tr>
<td>c[6]</td>
<td>0</td>
<td>c[-6]</td>
</tr>
<tr>
<td>c[7]</td>
<td>62</td>
<td>c[-5]</td>
</tr>
<tr>
<td>c[8]</td>
<td>-3</td>
<td>c[-4]</td>
</tr>
<tr>
<td>c[9]</td>
<td>1</td>
<td>c[-3]</td>
</tr>
<tr>
<td>c[10]</td>
<td>6453</td>
<td>c[-2]</td>
</tr>
<tr>
<td>c[11]</td>
<td>78</td>
<td>c[-1]</td>
</tr>
</tbody>
</table>
Creating Sequences

To create an empty string
aString = ""

To create an empty list
aList = []

To create a list that contains a sequence of values
aList = [1, 2, 3]

To create an empty tuple
aTuple = ()

To create a tuple that contains a sequence of values
aTuple = 1, 2, 3
    aTuple = (1, 2, 3)

To create a one-element tuple --- singleton
aSingleton = 1,
Using Lists and Tuples

```python
aList = [1, 2, 3, 4, 5]
ATuple = (1, 2, 3, 4, 5)

# creating, accessing and changing a list.
aList = []
# add values to list
for number in range(1,11):
    aList += [number]
print "The value of aList is: ", aList
# access list values by iteration
print ("\nAccessing values by iteration:")
for item in aList:
    print item,

# access list values by index
print ("\nAccessing values by index:")
print "Subscript Value"
for i in range (len(aList)):
    print ("%9d %7d" % (i, aList[i]))
print ("\nModifying a list value")
print ("Value of aList before modification: ", aList)
aList[0] = -100
aList[-3] = 19
print ("Value of aList after modification: ", aList)
```
Using Lists and Tuples

```python
# creating a histogram from a list of values.
values = []
# input 10 values from user
print("Enter 10 integers:")
for i in range(10):
    newValue = int(input("Enter integer %d: " % (i+1)))
    values += [newValue]
# create a histogram
print("Creating a histogram from values:")
print("%s %10s %10s" (%"Element", "Value", "Histogram"))
for i in range(len(values)):
    print("%7d %10d %s" (i, values[i], "*"*values[i]))
```
# Creating and accessing tuples.

# retrieve hour, minute and second from user
hour = int(input("Enter hour: "))
minute = int(input("Enter minutes: "))
second = int(input("Enter second: "))

currentTime = hour, minute, second  # create tuple

print ("The value of currentTime is:", currentTime)

# access tuple
print ("The number of seconds since midnight is", 
  (currentTime[0]*3600+currentTime[1]*60 + currentTime[2]))
Sequence Unpacking

Packing a tuple:

aTuple = 1, 2, 3 or aTuple = (1, 2, 3)

Unpacking a tuple:
--- the values stored in the sequence are assigned to various identifiers
--- assigning values to multiple variables in a single statement

Unpacking sequences

#Unpacking sequences

# create sequences
aString = “abc”
aList = [1, 2, 3]
aTuple = “a”, “A”, 1

# unpack sequences to variables
print (“Unpacking string”)
first, second, third = aString
print (“String values:”, first, second, third)

print (“Unpacking list”)
first, second, third = aList
print (“List values:”, first, second, third)

print (“Unpacking tuple”)
first, second, third, = aTuple
print (“Tuple value:”, first, second, third)

# swapping two values
x = 3
y = 4
print (“Before swapping: x = %d, y = %d” % (x, y))
x, y = y, x
print (“After swapping: x = %d, y = %d” % (x, y))
Sequence Slicing

How to create and access sequences:
--- through the [] operator
--- the for statement
--- slicing: accessing a series of sequential values, i.e., the characters of a person’s last name in a string that stores the person’s full name

```python
# create sequences
sliceString = “abcdefghij”
sliceTuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

# print strings
print (“sliceString: “, sliceString)
print (“sliceTuple: “, sliceTuple)
print (“sliceList: “, sliceList)
print

# get slices
start = int(raw_input(“Enter start: “))
end = int(raw_input(“Enter end: “))

# print slices
print (“ \nsliceString[“, start, “:”, end, “] = “, \n    sliceString[start:end])
print (“sliceTuple[“, start, “:”, end, “] = “, \n    sliceTuple[start:end])
print (“sliceList[“, start, “:”, end, “] = “, \n    sliceList[start:end])
```

---

How to create and access sequences:
--- through the [] operator
--- the for statement
--- slicing: accessing a series of sequential values, i.e., the characters of a person’s last name in a string that stores the person’s full name
Dictionaries:

--- mapping constructs of key-value pairs, **key: value**
--- unordered collections of values where each value is referenced through its corresponding key.
--- dictionary keys must be immutable values, such as strings, numbers or tuples
--- dictionary values can be of any Python data type
--- dictionary values are accessed with the expression `dictionaryName[key]`
--- the values associated with key can be modified `dictionaryName[key] = value`
--- deleting an entry from the dictionary `del dictionaryName[key]`

# creating, accessing and modifying a dictionary

# create and print an empty dictionary

```python
emptyDictionary = {}
print("The value of emptyDictionary is:", emptyDictionary)
```

# create and print a dictionary with initial values

```python
grades = {"John": 87, "Steve": 76, "Laura": 92, "Edwin": 89}
print("All grades:", grades)
```

# access and modify an existing dictionary

```python
print("Steve's current grade:", grades["Steve"])
grades["Steve"] = 90
print("Steve's new grade:", grades["Steve"])
```

# add to an existing dictionary

```python
grades["Michael"] = 93
print("Dictionary grades after modification:")
print(grades)
```

# delete entry from dictionary

```python
del grades["John"]
print("Dictionary grades after deletion:")
print(grades)
```
List and Dictionary Methods

A method is a function that performs the behavior (task) of an object.

--- list methods can append a value to the end of a list or determine the index of a particular element in the list.

--- to invoke the list method, specify the name of the list, followed by the dot (.) access operator, followed by the method call.

```python
# Appending items to a list
playList = [] #list of favorite plays

print ("Enter your 5 favorite Shakespearean) plays.\n"

def add_favorite_play(plays):
    num_plays = len(plays)
    print(f"You have added {num_plays} plays.")

# student poll program
responses = [ 1, 2, 6, 4, 8, 5, 9, 7, 8, 10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 ]

print ("Rating Frequency")
for i in range(1, 11):
    print ("%6d %13d" % (i, responses.count(i)))
```
## List Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>append(item)</td>
<td>Inserts item at the end of the list</td>
</tr>
<tr>
<td>count(element)</td>
<td>Returns the number of occurrence of element in the list</td>
</tr>
<tr>
<td>extend(newList)</td>
<td>Inserts the elements of newList at the end of the list</td>
</tr>
<tr>
<td>index(element)</td>
<td>Returns the index of the first occurrence of element in the list</td>
</tr>
<tr>
<td>insert(index, item)</td>
<td>Inserts item at position index</td>
</tr>
<tr>
<td>pop([index])</td>
<td>It removes and returns the element at position index</td>
</tr>
<tr>
<td>remove(element)</td>
<td>Removes the first occurrence of element from the list</td>
</tr>
<tr>
<td>reverse()</td>
<td>Reverses the contents of the list in place</td>
</tr>
<tr>
<td>sort([compare-function])</td>
<td>Sorts the content of the list in place.</td>
</tr>
</tbody>
</table>
# Dictionary Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear()</td>
<td>Deletes all items from the dictionary</td>
</tr>
<tr>
<td>copy()</td>
<td>Creates and returns a shallow copy of the dictionary</td>
</tr>
<tr>
<td>get(key[, return Value])</td>
<td>Returns the value associated with key.</td>
</tr>
<tr>
<td>hash_key(key)</td>
<td>Returns 1 if key is in the dictionary, and 0 if key is not in it</td>
</tr>
<tr>
<td>items()</td>
<td>Returns a list of tuples that are key-value pairs</td>
</tr>
<tr>
<td>keys()</td>
<td>Returns a list of keys in the dictionary</td>
</tr>
<tr>
<td>popitem()</td>
<td>Removes and returns an arbitrary key-value pair as a tuple of two elements</td>
</tr>
<tr>
<td>setdefault(key[, dummyValue])</td>
<td>Behaves similarly to method get.</td>
</tr>
<tr>
<td>update(newDictionary)</td>
<td>Adds all key-value pairs from newDictionary to the current dictionary and overrides the values for keys that already exist.</td>
</tr>
<tr>
<td>values()</td>
<td>returns a list of values in the dictionary</td>
</tr>
<tr>
<td>iterkeys()</td>
<td>Returns an iterator of dictionary keys.</td>
</tr>
<tr>
<td>iteritems()</td>
<td>Returns an iterator of key-value pairs</td>
</tr>
<tr>
<td>itervalues()</td>
<td>Returns an iterator of dictionary values</td>
</tr>
</tbody>
</table>
# Dictionary methods

```python
                   8: "August", 9: "September", 10: "October",
                   11: "November", 12: "December"}

print ("The dictionary items are:")
print (monthsDictionary.items())

print ("\nThe dictionary keys are:")
print (monthsDictionary.keys())

print ("\nThe dictionary values are:")
print (monthsDictionary.values())

print ("\nUsing a for loop to get dictionary items:")

for key in monthsDictionary.keys():
  print ("monthsDictionary[", key, "] =", monthsDictionary[key])
```
Values or arguments have to be passed to functions through a certain protocol.

Pass-by-value: a copy of the argument’s value is made and passed to the called function.

Pass-by-reference: the caller allows the called function to access the caller’s data directly and to modify that data.

Python arguments are always passed by object reference
--- the function receives references to the values passed as arguments.
--- if a function receives a reference to a mutable object (a dictionary or a list), the function can modify the original value of the object
--- if a function receives a reference to an immutable object (a number, a string, or a tuple), the function cannot modify the original object directly
Passing Lists to Functions

To pass a list argument to a function, specify the name of the list.

```python
# passing lists and individual list elements to functions.
def modifyList(aList):
    for i in range(len(aList)):
        aList[i] *= 2
def modifyElement(element)
    element *= 2
aList = [1, 2, 3, 4, 5]
print ("Effects of passing entire list:")
print ("The values of the original list are:")
for item in aList:
    print (item),
modifyList(aList)
print ("\n\nThe values of the modified list are:")
for item in aList:
    print (item),
print ("\n\nEffects of passing list element:")
print ("aList[3] before modifyElement:", aList[3])
modifyElement(aList[3])
print ("aList[3] after modifyElement:", aList[3])
print ("\n\nEffects of passing slices of list:")
modifyList(aList[2:4])
```
Sorting and Searching Lists

Sorting is a very important application
--- a bank sorts checks by account number
--- telephone companies sort accounts by last name, then by first name, to simplify the
  search for phone numbers

# sorting a list.

```python
aList = [2, 6, 4, 8, 10, 12, 89, 68, 45, 37]
print("Data items in original order")
for item in aList:
    print(item),

aList.sort()
print("Data items after sorting")
for item in aList:
    print(item),
```

# searching a list for an integer.

```python
# create a list of even integers 0 to 198
aList = range(0, 199, 2)

searchKey = int(input("Enter integer search key: "))

if searchKey in aList:
    print("Found at index: ", aList.index(searchKey))
else:
    print("Value not found")
```
Sequences that require two subscripts to identify a particular element are called double-subscripted sequences or two-dimensional sequences. Multiple-subscripted sequences can have more than two subscripts.  
--- Python does not support multiple-subscripted sequence directly, but allows programmers to specify single-subscripted tuples and lists whose elements are also single-subscripted tuples and lists, thus achieving the same effect.  
--- a sequence with m rows and n columns is called an m-by-n sequence.

<table>
<thead>
<tr>
<th>Row 0</th>
<th>Column 0</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0][0]</td>
<td>a[0][1]</td>
<td>a[0][2]</td>
<td>a[0][3]</td>
<td></td>
</tr>
<tr>
<td>a[1][0]</td>
<td>a[1][1]</td>
<td>a[1][2]</td>
<td>a[1][3]</td>
<td></td>
</tr>
</tbody>
</table>
Multiple-Subscripted Sequences

\[ b = \begin{bmatrix} [1, 2], [3, 4] \end{bmatrix} \]
\[ b[0][0] = 1; b[0][1] = 2; b[1][0] = 3, b[1][1] = 4 \]
\[ c = ( (1, 2), (3, 4, 5) ) \]
The first row has two elements (1, 2)
The second row has three elements (3, 4, 5)

```python
# making tables using lists of lists and tuples of tuples

table1 = [ [1, 2, 3], [4, 5, 6] ]
table2 = ( (1, 2), (3), (4, 5, 6) )
print("Values in table1 by row are")
for row in table1:
    for item in row:
        print(item),
    print()
print("Values in table2 by row are")
for row in table2:
    for item in row:
        print(item),
    print()
```
Multiple-Subscripted Sequences

```python
for column in range(len(a[2])):
    a[2][column] = 0

a[2][0] = 0
a[2][1] = 0
a[2][2] = 0
a[2][3] = 0

total = 0
for row in a:
    for column in row:
        total += column
```
def printGrades(grades):
    students = len(grades)
    exams = len(grades[0])
    print(“The list is:”)
    print(“”),
    for i in range(exams):
        print(“[%d]” % i),
    print()
    for I in range(students):
        print(“grades[%d] “ % I),
        for j in range(exams)
            print(grades[i][j], “”),
        print()

def minimum(grades):
    lowScore = 100
    for studentExams in grades:
        for score in studentExams:
            if score<lowScore:
                lowScore=score
    return lowScore

def maximum(grades):
    highScore = 0
    for studentExams in grades:
        for score in studentExams:
            if score>highScore:
                highScore=score
    return highScore

def average(setOfGrades):
    total = 0.0
    for grade in setOfGrades:
        total += grade
    return total/len(setOfGrades)

#main program
grades = [ [77, 68, 86, 73],
           [96, 87, 89, 81],
           [70, 90, 86, 81] ]
printGrades(grades)
print (“\n\nLowest grades:”, minimum(grades))
print (“Highest grade:”, maximum(grades))
print (“\n”)

#print average for each student
for i in range(len(grades))
    print (“Average for student”, i, “is”,
           average(grades[i]))
What is leap year: (1) divisible by 4 but not divisible by 100; (2) divisible by 400.

`year % 4 == 0; year % 100 != 0; year % 400 == 0.`
Program examples

- Write a program to decide if a year is a leap year
- Program design:
  - Input any year from the keyboard;
  - Output the user if the year is a leap year or not;
  - The algorithm to determine if a year is a leap year;
  - Variables, if else statements;
```c
#include "stdio.h"

int main(int argc, char* argv[]) {
    int year, leap;
    printf("This program decide if a year is a leap year!!!\n");
    printf("Enter a year:\n");
    scanf("%d", &year);
    if( (year%4 == 0) && (year%100 != 0) || (year%400 == 0) )
    {
        leap = 1;
    }
    else
    {
        leap = 0;
    }
    if(leap)
    {
        printf("Year %d is a leap year!\n");
    }
    else
    {
        printf("Year %d is NOT a leap year!\n");
    }
    return 0;
}
```
Loop examples

- To calculate the summation of n integers
  \[ \text{sum} = 1 + 2 + 3 + \ldots + n; \]
Loop examples

- The Fibonacci sequence
  
  \[ \begin{align*}
  f_1 &= 1 \\
  f_2 &= 1 \\
  f(n) &= f(n-1) + f(n-2)
  \end{align*} \]

  Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...,
```c
int fibo, fibo_1, fibo_2;
int i, n;
printf(“Enter an integer:\n”)
scanf(“%d”, &n);
for(i=1; i<=n; i++)
{
    if(i == 1)
    {
        fibo_1 = 1;
        printf(“The %dst number in Fibonacci sequence is: %d\n”, i,fibo_1);
    }
    else if(i == 2)
    {
        fibo_2 = 1;
        printf(“The %dnd number in Fibonacci sequence is: %d\n”, i,fibo_2);
    }
    else
    {
        fibo = fibo_1 + fibo_2;
        fibo_2 = fibo_1;
        fibo_1 = fibo;
        printf(“The %dth number in Fibonacci sequence is: %d\n”, i,fibo);
    }
}
```
include "stdio.h"

int main(int argc, char *argv[]) {
    int n;

    for(n=1; n<=200; n++) {
        if(n%3 == 0) {
            continue;
        }
        printf("%d", n);
    }
    return 0;
}

include "stdio.h"

int main(int argc, char *argv[]) {
    int n;

    for(n=1; n<=200; n++) {
        if(n%3 == 0) {
            continue;
        }
        printf("%d", n);
    }
    return 0;
}
Nested loops

int i, j;
for(i=0; i<10; i=i+1)
{
    printf(" ******\n");
    for(j=0; j<10; j=j+1)
    {
        printf(" $$$$$$$\n");
    }
}

How many times “******” should be printed?
How many times “$$$$$$$$” should be printed?
What are i and j values after the loop?
int i, j;
for(i=0; i<10; i=i+1)
{
    printf("*****\n");
    for(j=0; j<10; j=j+2)
    {
        printf("$$$$$$\n");
    }
}

How many times “*****” should be printed?
How many times “$$$$$$” should be printed?
What are i and j values after the loop?
Loop example

- Print the follow:

  *                                      if n = 1
  *
  *                                      if n = 2
  ***
  *
  *                                      if n = 3
  ***
  ***
  *******
Bubble sorting

Compare two neighboring numbers, move the smaller one ahead of the larger one.