CHAPTER 1: PYTHON BASICS

1. Entering expressions into the interactive shell
2. The integer, floating-point and String Data Types
3. String concatenation and replication
4. Storing values in variables
5. Your first program
6. Dissecting your program

1.1. Entering expressions into the interactive shell

- Run the interactive shell by launching IDLE, which is installed with Python. On Windows, open the Start menu, select **All Programs** ▸ **Python 3.3**, and then select **IDLE (Python GUI)**. On OS X, select **Applications** ▸ **MacPython 3.3** ▸ **IDLE**. On Ubuntu, open a new Terminal window and enter **idle3**.
- A window with the >>> prompt should appear; that‘s the interactive shell.

```
>>> 2+2
4
```

- The IDLE window should now show some text like this: Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53) [MSC v.1600 64 bit (AMD64)] on win32
- Type "copyright", "credits" or "license()" for more information.

```
>>> 2+2
4
```

- In Python, 2 + 2 is called an *expression*, which is the most basic kind of programming instruction in the language. Expressions consist of *values* (such as 2) and *operators* (such as +), and they can always *evaluate* (that is, reduce) down to a single value. That means you can use expressions anywhere in Python code that you could also use a value.

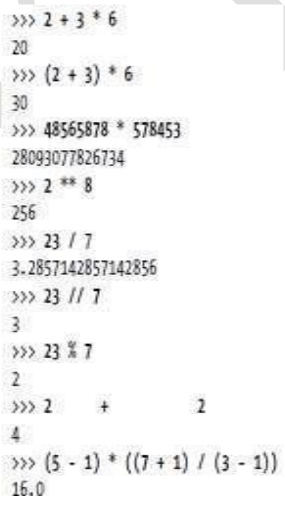
- In the previous example, 2 + 2 is evaluated down to a single value, 4. A single value with no operators is also considered an expression, though it evaluates only to itself, as shown here:

```
>>> 2+2

4
```

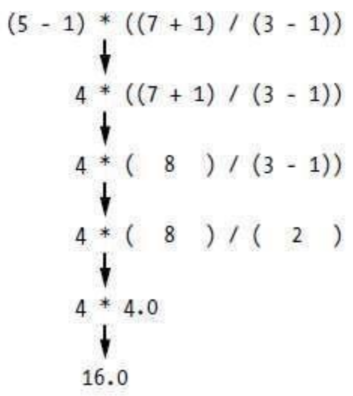
- The other operators which can be used are:

Operator	Operation	Example	Evaluates to...
**	Exponent	2 ** 3	8
%	Modulus/remainder	22 % 8	6
//	Integer division/floored quotient	22 // 8	2
/	Division	22 / 8	2.75
*	Multiplication	3 * 5	15
-	Subtraction	5 - 2	3
+	Addition	2 + 2	4

- The order of operations (also called precedence) of Python math operators is similar to that of mathematics. The ** operator is evaluated first; the *, /, //, and % operators are evaluated next, from left to right; and the + and - operators are evaluated last (also from left to right). We can use parentheses to override the usual precedence if you need to.



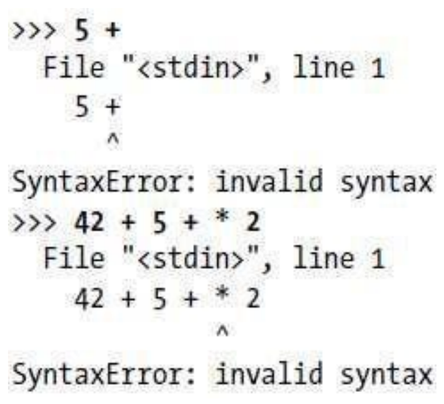
```
>>> 2 + 3 * 6
20
>>> (2 + 3) * 6
30
>>> 48565878 * 578453
28093077826734
>>> 2 ** 8
256
>>> 23 / 7
3.2857142857142856
>>> 23 // 7
3
>>> 23 % 7
2
>>> 2      +          2
4
>>> (5 - 1) * ((7 + 1) / (3 - 1))
16.0
```



```
(5 - 1) * ((7 + 1) / (3 - 1))
        ↓
    4 * ((7 + 1) / (3 - 1))
        ↓
    4 * (  8   ) / (3 - 1))
        ↓
    4 * (  8   ) / (  2  )
        ↓
    4 * 4.0
        ↓
    16.0
```

Figure 1-1: Evaluating an expression reduces it to a single value.

- Due to wrong instructions errors occurs as shown below:



```
>>> 5 +
  File "<stdin>", line 1
    5 +
      ^
SyntaxError: invalid syntax
>>> 42 + 5 + * 2
  File "<stdin>", line 1
    42 + 5 + * 2
             ^
SyntaxError: invalid syntax
```

1.2 The integer, floating-point and String Data Types

- The expressions are just values combined with operators, and they always evaluate down to a single value.
- A data type is a category for values, and every value belongs to exactly one data type.

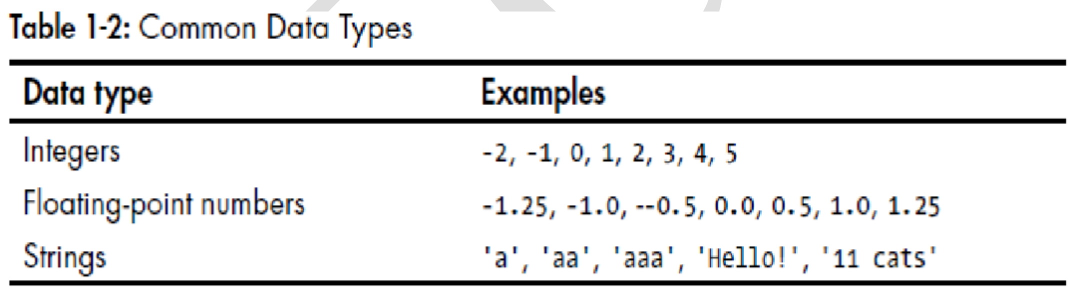
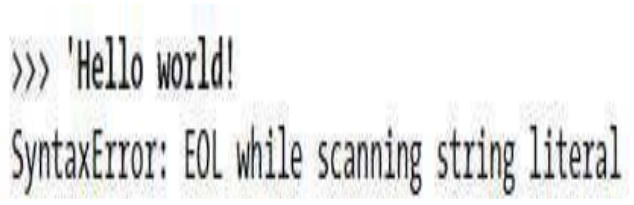


Table 1-2: Common Data Types

Data type	Examples
Integers	-2, -1, 0, 1, 2, 3, 4, 5
Floating-point numbers	-1.25, -1.0, --0.5, 0.0, 0.5, 1.0, 1.25
Strings	'a', 'aa', 'aaa', 'Hello!', '11 cats'

- The integer (or int) data type indicates values that are whole numbers.
- Numbers with a decimal point, such as 3.14, are called floating-point numbers (or floats).
- Note that even though the value 42 is an integer, the value 42.0 would be a floating-point number.
- Python programs can also have text values called strings, or strs and surrounded in single quote.
- The string with no characters, '', called a blank string.

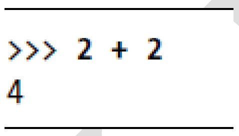
- If the error message SyntaxError: EOL while scanning string literal, then probably the final single quote character at the end of the string is missing.



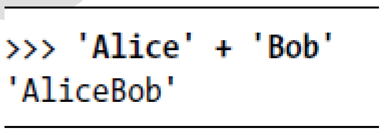
```
>>> 'Hello world!
SyntaxError: EOL while scanning string literal
```

1.3 String concatenation and replication

- The meaning of an operator may change based on the data types of the values next to it
- For example, + is the addition operator when it operates on two integers or floating-point values.
- However, when + is used on two string values, it joins the strings as the string concatenation operator.

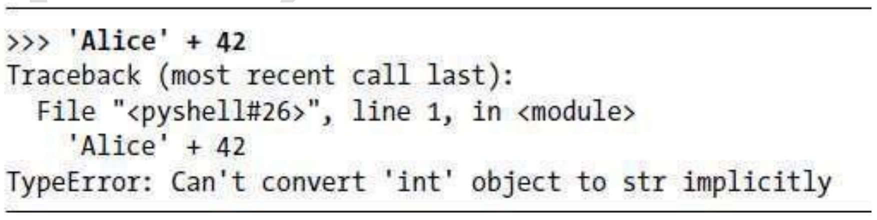


```
>>> 2 + 2
4
```



```
>>> 'Alice' + 'Bob'
'AliceBob'
```

- If we try to use the + operator on a string and an integer value, Python will not know how to handle this, and it will display an error message.



```
>>> 'Alice' + 42
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    'Alice' + 42
TypeError: Can't convert 'int' object to str implicitly
```

- The * operator is used for multiplication when it operates on two integer or floating-point values.
- But, when the * operator is used on one string value and one integer value; it becomes the string replication operator.

```
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
```

- The * operator can be used with only two numeric values (for multiplication) or one string value and one integer value (for string replication). Otherwise, Python will just display an error message.

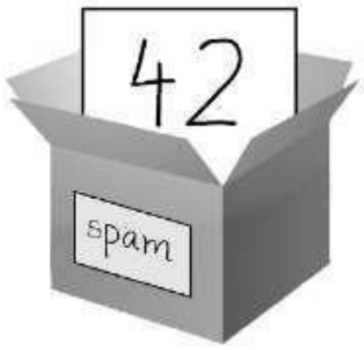
```
>>> 'Alice' * 'Bob'
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    'Alice' * 'Bob'
TypeError: can't multiply sequence by non-int of type 'str'
>>> 'Alice' * 5.0
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    'Alice' * 5.0
TypeError: can't multiply sequence by non-int of type 'float'
```

<u>1.4 Storing Values in Variables</u>

- A variable is like a box in the computer‘s memory where you can store a single value.
- If we need to use variables later, then the result must be stored in variable.

<u>Assignment Statements</u>

- You‘ll store values in variables with an assignment statement.
- An assignment statement consists of a variable name, an equal sign (called the assignment operator), and the value to be stored.
- Ex: spam = 42



Overwriting the variable

- A variable is initialized (or created) the first time a value is stored in it ❶.
- After that, you can use it in expressions with other variables and values ❷.
- When a variable is assigned a new value ③, the old value is forgotten, which is why spam evaluated to 42 instead of 40 at the end of the example.

```
❶ >>> spam = 40
  >>> spam
  40
  >>> eggs = 2
❷ >>> spam + eggs
  42
  >>> spam + eggs + spam
  82
❸ >>> spam = spam + 2
  >>> spam
  42
```

One more example

```
>>> spam = 'Hello'
>>> spam
'Hello'
>>> spam = 'Goodbye'
>>> spam
'Goodbye'
```

'Goodbye'
'Hello'
spam

Variable names

We can name a variable anything as long as it obeys the following three rules:

1. It can be only one word.

2. It can use only letters, numbers, and the underscore (_) character.

3. It can‘t begin with a number.

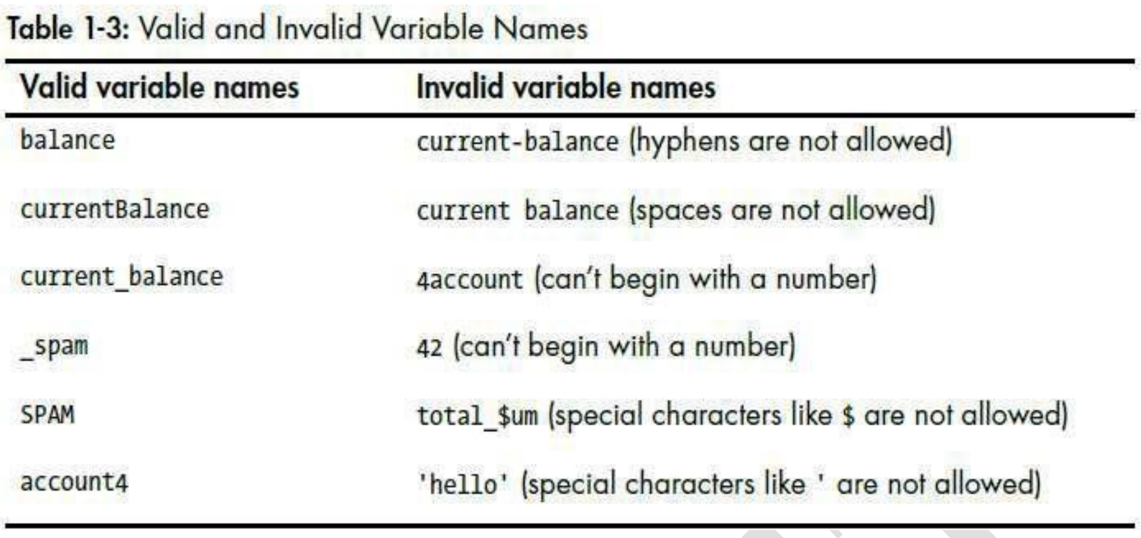


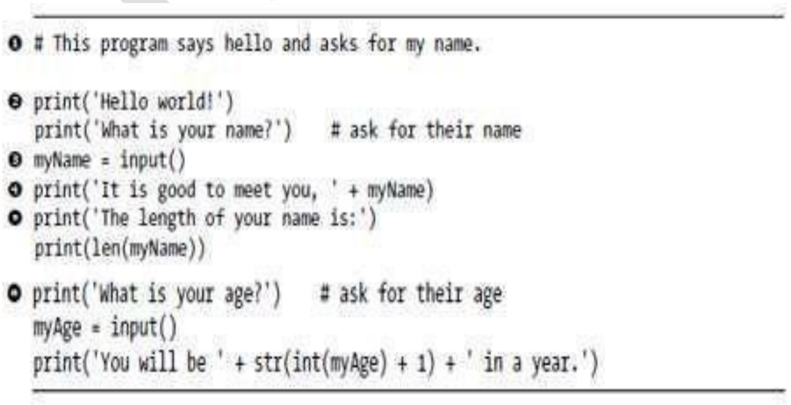
Table 1-3: Valid and Invalid Variable Names

Valid variable names	Invalid variable names
balance	current-balance (hyphens are not allowed)
currentBalance	current balance (spaces are not allowed)
current_balance	4account (can't begin with a number)
_spam	42 (can't begin with a number)
SPAM	total_$um (special characters like $ are not allowed)
account4	'hello' (special characters like ' are not allowed)

- Variable names are case-sensitive, meaning that spam, SPAM, Spam, and sPaM are four different variables.
- This book uses camelcase for variable names instead of underscores; that is, variables look LikeThis instead of looking_like_this.
- A good variable name describes the data it contains.

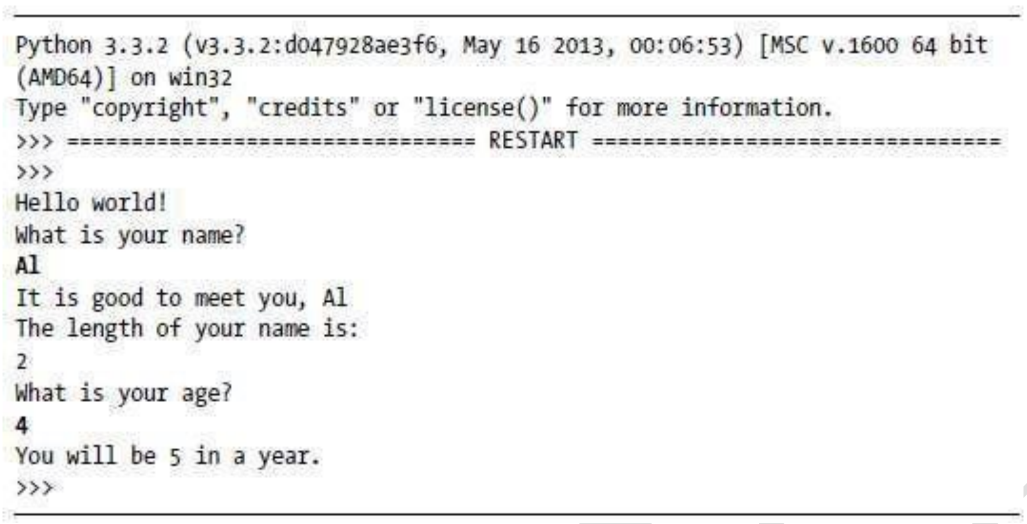
1.5 Your First Program

- The file editor is similar to text editors such as Notepad or TextMate, but it has some specific features for typing in source code.
- The interactive shell window will always be the one with the >>> prompt.
- The file editor window will not have the >>> prompt.
- The extension for python program is .py Example program:



```
❶ # This program says hello and asks for my name.

❷ print('Hello world!')
  print('What is your name?')    # ask for their name
❸ myName = input()
❹ print('It is good to meet you, ' + myName)
❺ print('The length of your name is:')
  print(len(myName))
❻ print('What is your age?')    # ask for their age
  myAge = input()
  print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

➢ The output looks like:



```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53) [MSC v.1600 64 bit
(AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Hello world!
What is your name?
Al
It is good to meet you, Al
The length of your name is:
2
What is your age?
4
You will be 5 in a year.
>>>
```

1.6 Dissecting Your Program

Comments

➢ The following line is called a *comment*.

```
❶ # This program says hello and asks for my name.
```

➢ Python ignores comments, and we can use them to write notes or remind ourselves what the code is trying to do.

➢ Any text for the rest of the line following a hash mark (#) is part of a comment.

➢ Sometimes, programmers will put a # in front of a line of code to temporarily remove it while testing a program. This is called *commenting out* code, and it can be useful when you‘re trying to figure out why a program doesn‘t work.

➢ Python also ignores the blank line after the comment.

The print() Function

➢ The print() function displays the string value inside the parentheses on the screen.

```
❷ print('Hello world!')
  print('What is your name?') # ask for their name
```

- The line print('Hello world!') means ―Print out the text in the string 'Hello world!'.‖
- When Python executes this line, you say that Python is *calling* the print() function and the string value is being *passed* to the function.
- A value that is passed to a function call is an *argument.*
- The quotes are not printed to the screen. They just mark where the string begins and ends; they are not part of the string value.

The Input Function

- The input() function waits for the user to type some text on the keyboard and press ENTER

```
❸ myName = input()
```

- This function call evaluates to a string equal to the user's text, and the previous line of code assigns the myName variable to this string value.
- We can think of the input() function call as an expression that evaluates to whatever string the user typed in. If the user entered 'Al', then the expression would evaluate to myName = 'Al'.

Printing the User's Name

- The following call to print() actually contains the expression 'It is good to meet you, ' + myName between the parentheses.

```
❹ print('It is good to meet you, ' + myName)
```

- Remember that expressions can always evaluate to a single value.
- If 'Al' is the value stored in myName on the previous line, then this expression evaluates to 'It is good to meet you, Al'.
- This single string value is then passed to print(), which prints it on the screen.

The len() Function

- We can pass the len() function a string value (or a variable containing a string), and the function evaluates to the integer value of the number of characters in that string.

```
❺ print('The length of your name is:')
  print(len(myName))
```

- In the interactive shell:

```
>>> len('hello')
5
>>> len('My very energetic monster just scarfed nachos.')
46
>>> len('')
0
```

- len(myName) evaluates to an integer. It is then passed to print() to be displayed on the screen.
- Possible errors: The print() function isn‘t causing that error, but rather it‘s the expression you tried to pass to print().

```
 >>> print('I am ' + 29 + ' years old.')
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    print('I am ' + 29 + ' years old.')
TypeError: Can't convert 'int' object to str implicitly
```

- Python gives an error because we can use the + operator only to add two integers together or concatenate two strings. We can‘t add an integer to a string because this is ungrammatical in Python.

```
>>> 'I am ' + 29 + ' years old.'
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    'I am ' + 29 + ' years old.'
TypeError: Can't convert 'int' object to str implicitly
```

The str(), int() and float() Functions

- If we want to concatenate an integer such as 29 with a string to pass to print(), we‘ll need to get the value '29', which is the string form of 29.
- The str() function can be passed an integer value and will evaluate to a string value version of it, as follows:

```
>>> str(29)
'29'
>>> print('I am ' + str(29) + ' years old.')
I am 29 years old.
```

- Because str(29) evaluates to '29', the expression 'I am ' + str(29) + ' years old.' evaluates to 'I am ' + '29' + ' years old.', which in turn evaluates to 'I am 29 years old.'. This is the value that is passed to the print() function.
- The str(), int(), and float() functions will evaluate to the string, integer, and floating-point forms of the value you pass, respectively.
- Converting some values in the interactive shell with these functions:

```
>>> str(0)
'0'
>>> str(-3.14)
'-3.14'
>>> int('42')
42
>>> int('-99')
-99
```

```
>>> int(1.25)
1
>>> int(1.99)
1
>>> float('3.14')
3.14
>>> float(10)
10.0
```

- The previous examples call the str(), int(), and float() functions and pass them values of the other data types to obtain a string, integer, or floating-point form of those values.
- The str() function is handy when you have an integer or float that you want to concatenate to a string.
- The int() function is also helpful if we have a number as a string value that you want to use in some mathematics.
- For example, the input() function always returns a string, even if the user enters a number.
- Enter **spam = input()** into the interactive shell and enter **101** when it waits for your text.

```
>>> spam = input()
101
>>> spam
'101'
```

- The value stored inside spam isn‘t the integer 101 but the string '101'.
- If we want to do math using the value in spam, use the int() function to get the integer form of spam and then store this as the new value in spam.

```
>>> spam = int(spam)
>>> spam
101
```

- Now we should be able to treat the spam variable as an integer instead of a string.

```
>>> spam * 10 / 5
202.0
```

- Note that if we pass a value to int() that it cannot evaluate as an integer, Python will display an error message.

```
>>> int('99.99')
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    int('99.99')
ValueError: invalid literal for int() with base 10: '99.99'
>>> int('twelve')
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    int('twelve')
ValueError: invalid literal for int() with base 10: 'twelve'
```

- The int() function is also useful if we need to round a floating-point number down. If we want to round a floating-point number up, just add 1 to it afterward.

```
>>> int(7.7)
7
>>> int(7.7) + 1
8
```

- In your program, we used the int() and str() functions in the last three lines to get a value of the appropriate data type for the code.

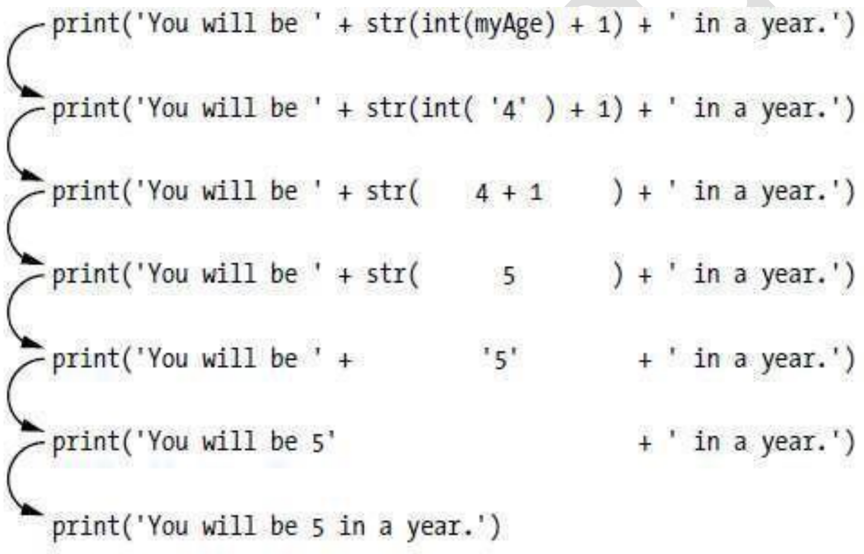
```
❸ print('What is your age?') # ask for their age
  myAge = input()
  print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

- The myAge variable contains the value returned from input().
- Because the input() function always returns a string (even if the user typed in a number), we can use the int(myAge) code to return an integer value of the string in myAge.
- This integer value is then added to 1 in the expression int(myAge) + 1.

- The result of this addition is passed to the str() function: str(int(myAge) + 1).
- The string value returned is then concatenated with the strings 'You will be ' and ' in a year.' to evaluate to one large string value.
- This large string is finally passed to print() to be displayed on the screen.

Another input:

- Let‘s say the user enters the string '4' for myAge.
- The string '4' is converted to an integer, so you can add one to it. The result is 5.
- The str() function converts the result back to a string, so we can concatenate it with the second string, 'in a year.', to create the final message. These evaluation steps would look something like below:

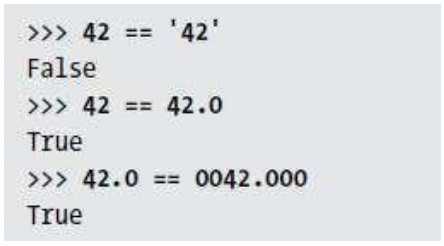


```
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
print('You will be ' + str(int( '4' ) + 1) + ' in a year.')
print('You will be ' + str(   4 + 1    ) + ' in a year.')
print('You will be ' + str(     5      ) + ' in a year.')
print('You will be ' +         '5'       + ' in a year.')
print('You will be 5'                    + ' in a year.')
print('You will be 5 in a year.')
```

Figure 1-4: The evaluation steps, if 4 was stored in myAge

Text and Number Equivalence

- Although the string value of a number is considered a completely different value from the integer or floating-point version, an integer can be equal to a floating point.



```
>>> 42 == '42'
False
>>> 42 == 42.0
True
>>> 42.0 == 0042.000
True
```

CHAPTER 2: FLOW CONTROL

1. Boolean Values
2. Comparison Operators
3. Boolean Operators
4. Mixing Boolean and Comparison Operators
5. Elements of Flow Control
6. Program Execution
7. Flow Control Statements
8. Importing Modules
9. Ending a Program Early with sys.exit()

Introduction

- Flow control statements can decide which Python instructions to execute under which conditions.
- These flow control statements directly correspond to the symbols in a flowchart
- In a flowchart, there is usually more than one way to go from the start to the end.
- Flowcharts represent these branching points with diamonds, while the other steps are represented with rectangles.
- The starting and ending steps are represented with rounded rectangles.

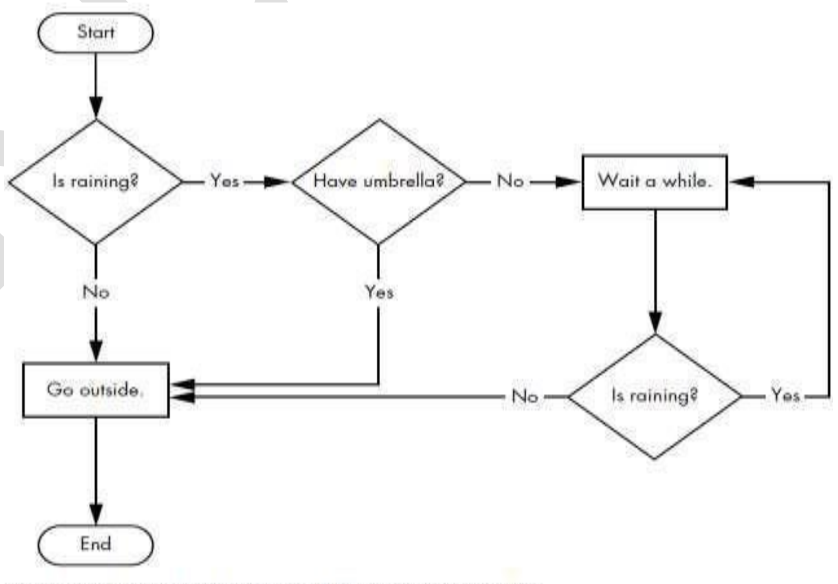
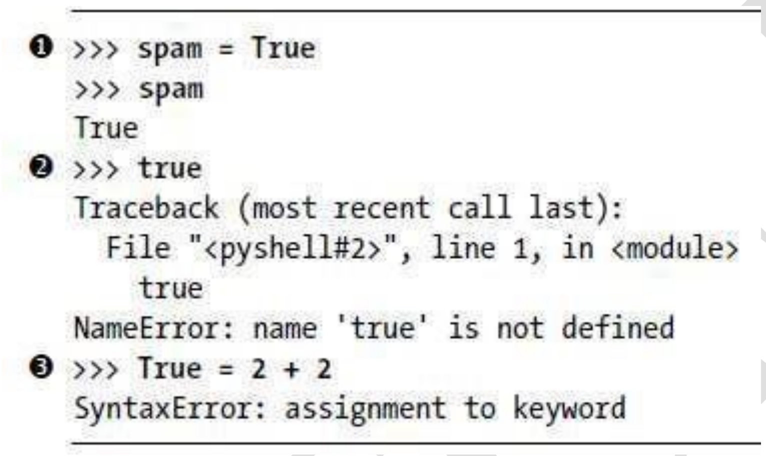


Figure 2-1: A flowchart to tell you what to do if it is raining

2.1 Boolean Values

- The Boolean data type has only two values: True and False.
- When typed as Python code, the Boolean values True and False lack the quotes you place around strings, and they always start with a capital T or F, with the rest of the word in lowercase.
- Examples:

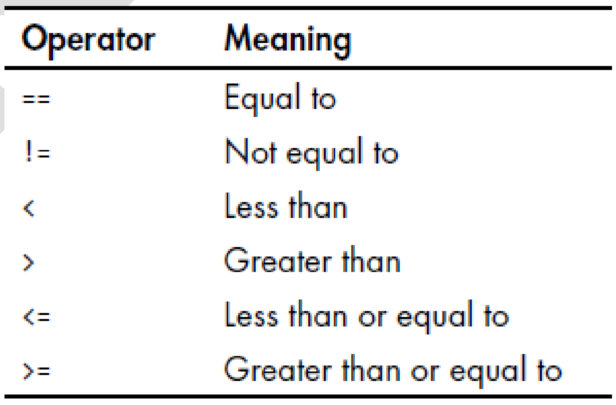


```
❶ >>> spam = True
   >>> spam
   True
❷ >>> true
   Traceback (most recent call last):
     File "<pyshell#2>", line 1, in <module>
       true
   NameError: name 'true' is not defined
❸ >>> True = 2 + 2
   SyntaxError: assignment to keyword
```

- Like any other value, Boolean values are used in expressions and can be stored in variables ❶. If we don‘t use the proper case ❷ or we try to use True and False for variable names ❸, Python will give you an error message.

2.2 Comparison Operators

- Comparison operators compare two values and evaluate down to a single Boolean value. Table 2-1 lists the comparison operators.



Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

➢ These operators evaluate to True or False depending on the values we give them.

```
>>> 42 == 42
True
>>> 42 == 99
False
>>> 2 != 3
True
>>> 2 != 2
False
```

➢ The == and != operators can actually work with values of any data type.

```
  >>> 'hello' == 'hello'
  True
  >>> 'hello' == 'Hello'
  False
  >>> 'dog' != 'cat'
  True
  >>> True == True
  True
  >>> True != False
  True
  >>> 42 == 42.0
  True
❶ >>> 42 == '42'
  False
```

➢ Note that an integer or floating-point value will always be unequal to a string value. The expression 42 == '42' ❶ evaluates to False because Python considers the integer 42 to be different from the string '42'.

➢ ➢ The <, >, <=, and >= operators, on the other hand, work properly only with integer and floating-point values.

```
  >>> 42 < 100
  True
  >>> 42 > 100
  False
  >>> 42 < 42
  False
  >>> eggCount = 42
❶ >>> eggCount <= 42
  True
  >>> myAge = 29
❷ >>> myAge >= 10
  True
```

The Difference Between the == and = Operators

- The == operator (equal to) asks whether two values are the same as each other.
- ➢ The = operator (assignment) puts the value on the right into the variable on the left.
- ➢ We often use comparison operators to compare a variable‘s value to some other value, like in the eggCount <= 42 ❶ and myAge >= 10 ❷ examples.

2.3 Boolean Operators

- The three Boolean operators (and, or, and not) are used to compare Boolean values.

Binary Boolean Operators

- The and and or operators always take two Boolean values (or expressions), so they‘re considered binary Operators.

and operator: The and operator evaluates an expression to True if both Boolean values are True; otherwise, it evaluates to False.

Table 2-2: The and Operator's Truth Table

Expression	Evaluates to...
True and True	True
True and False	False
False and True	False
False and False	False

```
>>> True and True
True
>>> True and False
False
```

or operator: The or operator valuates an expression to True if either of the two Boolean values is True. If both are False, it evaluates to False.

Table 2-3: The or Operator's Truth Table

Expression	Evaluates to...
True or True	True
True or False	True
False or True	True
False or False	False

```
>>> False or True
True
>>> False or False
False
```

not operator: The not operator operates on only one Boolean value (or expression). The not operator simply evaluates to the opposite Boolean value. Much like using double negatives in speech and writing, you can nest not operators ❶, though there‘s never not no reason to do this in real programs.

Table 2-4: The not Operator's Truth Table

Expression	Evaluates to...
`not True`	`False`
`not False`	`True`

```
  >>> not True
  False
❶ >>> not not not not True
  True
```

2.4 Mixing Boolean and Comparison Operators

- Since the comparison operators evaluate to Boolean values, we can use them in expressions with the Boolean operators. Ex:

```
>>> (4 < 5) and (5 < 6)
True
>>> (4 < 5) and (9 < 6)
False
>>> (1 == 2) or (2 == 2)
True
```

- The computer will evaluate the left expression first, and then it will evaluate the right expression. When it knows the Boolean value for each, it will then evaluate the whole expression down to one Boolean value. You can think of the computer‘s evaluation process for (4 < 5) and (5 < 6) as shown in Figure below:

$$\begin{tikzcd}
\texttt{(4 < 5) and (5 < 6)} \arrow[d] \\
\texttt{True and (5 < 6)} \arrow[d] \\
\texttt{True and True} \arrow[d] \\
\texttt{True}
\end{tikzcd}$$

- We can also use multiple Boolean operators in an expression, along with the comparison operators.

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
```

- The Boolean operators have an order of operations just like the math operators do. After any math and comparison operators evaluate, Python evaluates the not operators first, then the and operators, and then the or operators.

<u>2.4 Elements of Flow Control</u>

- Flow control statements often start with a part called the condition, and all are followed by a block of code called the clause.

<u>Conditions:</u>

- The Boolean expressions you‘ve seen so far could all be considered conditions, which are the same thing as expressions; condition is just a more specific name in the context of flow control statements.
- Conditions always evaluate down to a Boolean value, True or False.
- A flow control statement decides what to do based on whether its condition is True or False, and almost every flow control statement uses a condition.

<u>Blocks of Code:</u>

- Lines of Python code can be grouped together in blocks. There are three rules for blocks.

1. Blocks begin when the indentation increases.
2. Blocks can contain other blocks.
3. Blocks end when the indentation decreases to zero or to a containing block‘s indentation.

```
  if name == 'Mary':
❶     print('Hello Mary')
  if password == 'swordfish':
❷     print('Access granted.')
  else:
❸     print('Wrong password.')
```

- The first block of code ❶ starts at the line print('Hello Mary') and contains all the lines after it. Inside this block is another block ❷, which has only a single line in it: print('Access Granted.'). The third block ❸ is also one line long: print('Wrong password.').

Program Execution:

- The program execution (or simply, execution) is a term for the current instruction being executed.

Flow Control Statements:

1. if Statements:

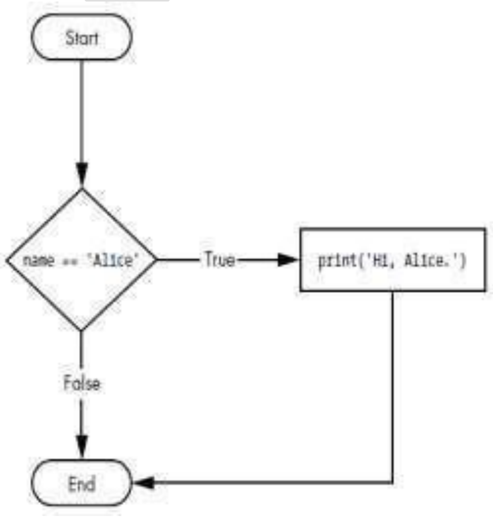
- The most common type of flow control statement is the if statement.
- An if statement‛s clause (that is, the block following the if statement) will execute if the statement‛s condition is True. The clause is skipped if the condition is False.
- In plain English, an if statement could be read as, ―If this condition is true, execute the code in the clause.‖ In Python, an if statement consists of the following:

1. The if keyword
2. A condition (that is, an expression that evaluates to True or False)
3. A colon
4. Starting on the next line, an indented block of code (called the if clause)

- Example:

```
if name == 'Alice':
    print('Hi, Alice.')
```

- Flowchart:



2. else Statements:

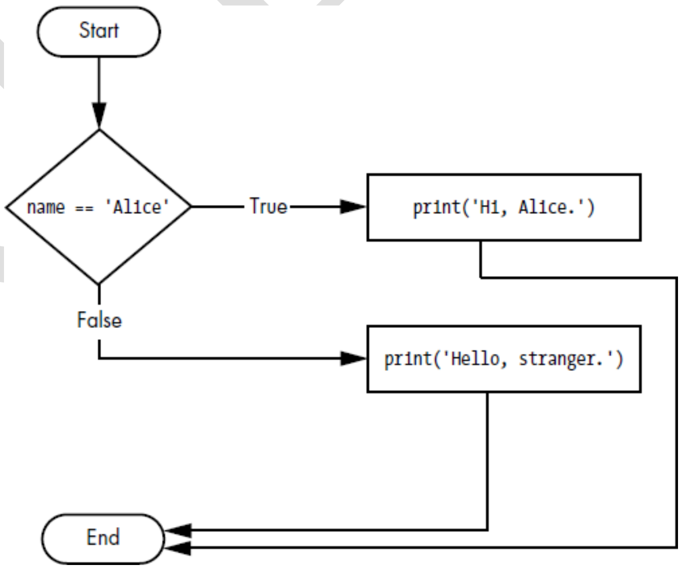
- ➢ An if clause can optionally be followed by an else statement. The else clause is executed only when the if statement‘s condition is False.
- ➢ In plain English, an else statement could be read as, ―If this condition is true, execute this code. Or else, execute that code.‖
- ➢ An else statement doesn‘t have a condition, and in code, an else statement always consists of the following:

1. The else keyword
2. A colon
3. Starting on the next line, an indented block of code (called the else clause)

➢ Example:

```
if name == 'Alice':
    print('Hi, Alice.')
else:
    print('Hello, stranger.')
```

➢ Flowchart:



3. elif Statements:

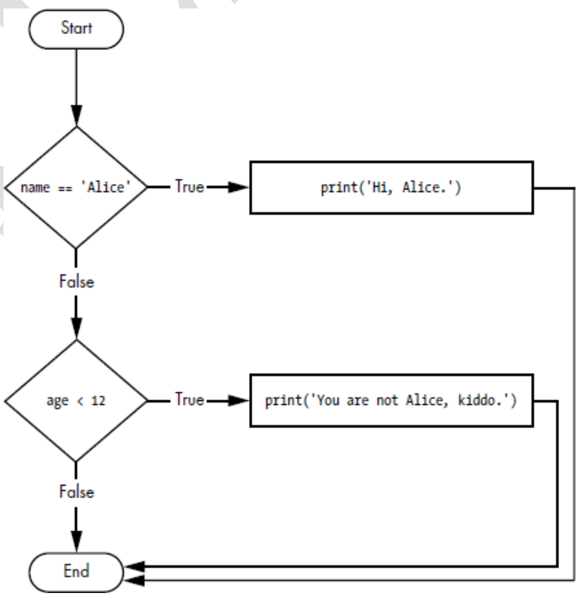
- ➢ While only one of the if or else clauses will execute, we may have a case where we want one of many possible clauses to execute.
- ➢ The elif statement is an –else if‖ statement that always follows an if or another elif statement.
- ➢ It provides another condition that is checked only if all of the previous conditions were False.
- ➢ In code, an elif statement always consists of the following:

1. The elif keyword
2. A condition (that is, an expression that evaluates to True or False)
3. A colon
4. Starting on the next line, an indented block of code (called the elif clause)

➢ Example:

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
```

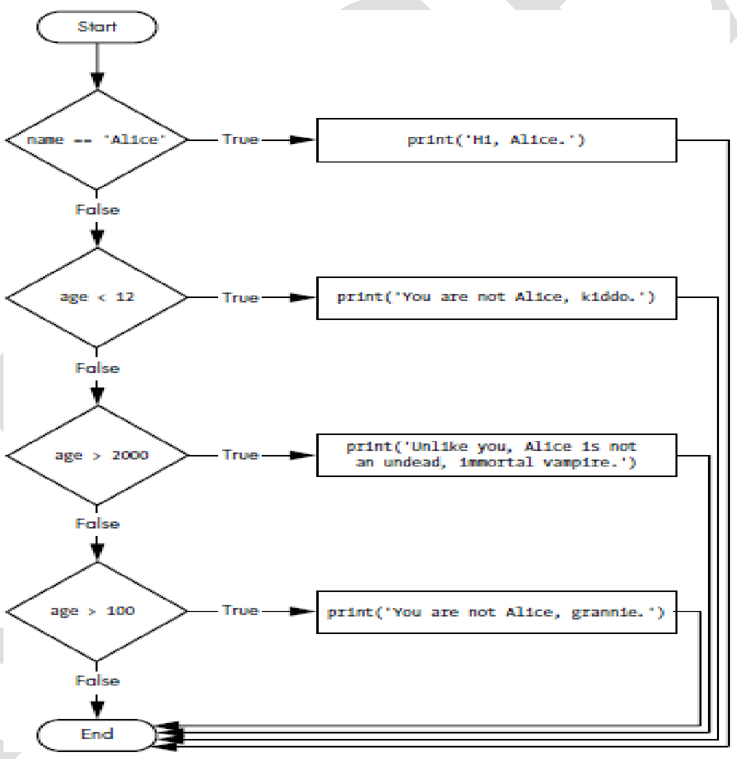
➢ Flowchart:



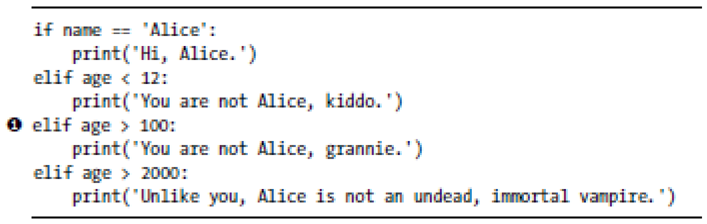
- When there is a chain of elif statements, only one or none of the clauses will be executed.
- Example:

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
elif age > 100:
    print('You are not Alice, grannie.')
```

- Flowchart:

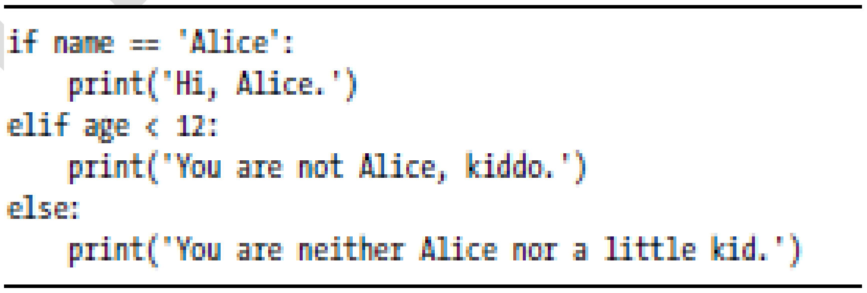

Start
name == 'Alice'
True
print('Hi, Alice.')
False
age < 12
True
print('You are not Alice, kiddo.')
False
age > 2000
True
print('Unlike you, Alice is not an undead, immortal vampire.')
False
age > 100
True
print('You are not Alice, grannie.')
False
End

- The order of the elif statements does matter, however. Let‘s see by rearranging the previous code.
- Say the age variable contains the value 3000 before this code is executed.



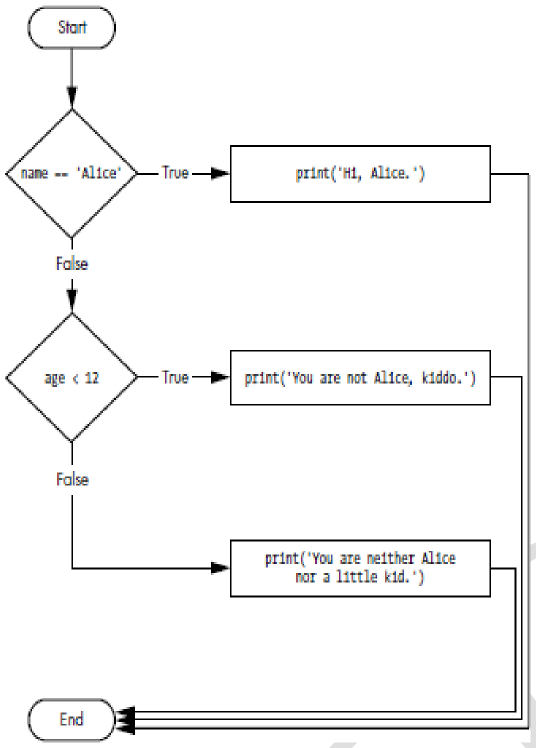
```
  if name == 'Alice':
      print('Hi, Alice.')
  elif age < 12:
      print('You are not Alice, kiddo.')
❶ elif age > 100:
      print('You are not Alice, grannie.')
  elif age > 2000:
      print('Unlike you, Alice is not an undead, immortal vampire.')
```

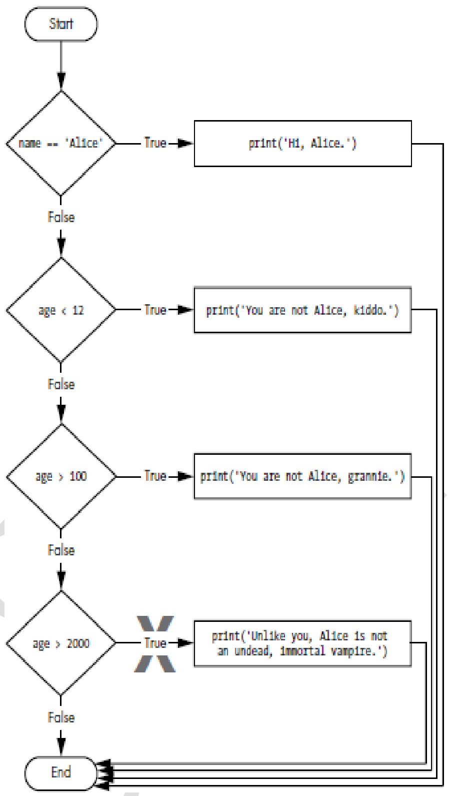
- We might expect the code to print the string 'Unlike you, Alice is not an undead, immortal vampire.'.
- However, because the age > 100 condition is True (after all, 3000 is greater than 100) ❶, the string 'You are not Alice, grannie.' is printed, and the rest of the elif statements are automatically skipped.
- Remember, at most only one of the clauses will be executed, and for elif statements, the order matters!
- Flowchart → (1)
- Optionally, we can have an else statement after the last elif statement.
- In that case, it is guaranteed that at least one (and only one) of the clauses will be executed.
- If the conditions in every if and elif statement are False, then the else clause is executed.
- In plain English, this type of flow control structure would be, ―If the first condition is true, do this. Else, if the second condition is true, do that. Otherwise, do something else.‖
- Example:



```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')
```

➢ Flowchart → (2)


Start
name == 'Alice'
True
print('Hi, Alice.')
False
age < 12
True
print('You are not Alice, kiddo.')
False
print('You are neither Alice nor a little kid.')
End


Start
name == 'Alice'
True
print('Hi, Alice.')
False
age < 12
True
print('You are not Alice, kiddo.')
False
age > 100
True
print('You are not Alice, grannie.')
False
age > 2000
True
print('Unlike you, Alice is not an undead, immortal vampire.')
False
End

4. while loop Statements:

➢ We can make a block of code execute over and over again with a while statement

➢ The code in a while clause will be executed as long as the while statement‘s condition is True.

➢ In code, a while statement always consists of the following:

1. The while keyword
2. A condition (that is, an expression that evaluates to True or False.
3. A colon
4. Starting on the next line, an indented block of code (called the while clause)

➢ We can see that a while statement looks similar to an if statement. The difference is in how they behave. At the end of an if clause, the program execution continues after the if statement.

➢ But, at the end of a while clause, the program execution jumps back to the start of the while statement. The while clause is often called the while loop or just the loop.

➢ Example:

Using if statement	Using while statement

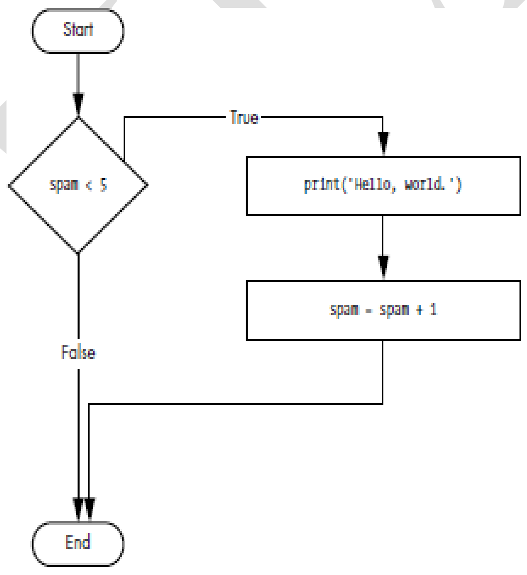
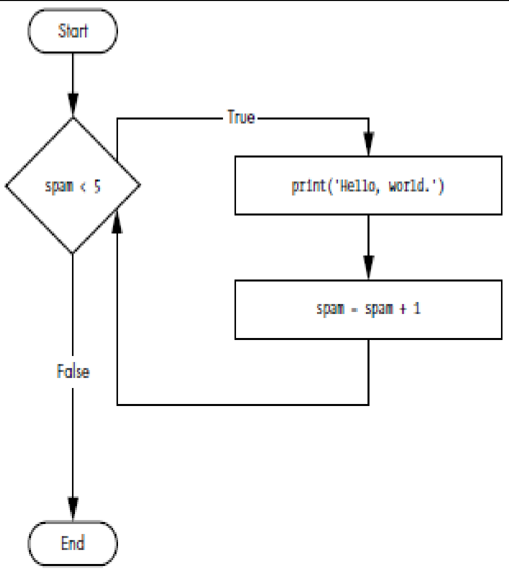
`spam = 0`
`if spam < 5:`
`    print('Hello, world.')`
`    spam = spam + 1`
	`spam = 0`
	`while spam < 5:`
	`    print('Hello, world.')`
	`    spam = spam + 1`

➢ These statements are similar—both if and while check the value of spam, and if it‘s less than five, they print a message.

➢ But when we run these two code snippets, for the if statement, the output is simply "Hello, world."

➢ But for the while statement, it‘s "Hello, world." repeated five times!

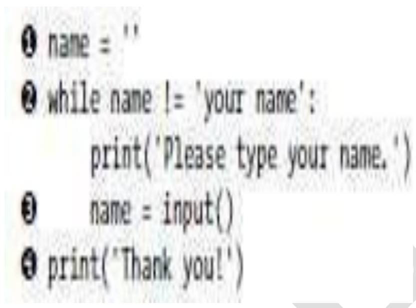
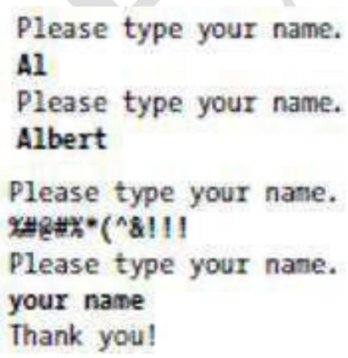
➢ Flowchart:

Using if Statement	Using While Statement
 Start spam < 5 True print('Hello, world.') spam = spam + 1 False End	 Start spam < 5 True print('Hello, world.') spam = spam + 1 False End

- In the while loop, the condition is always checked at the start of each iteration (that is, each time the loop is executed).
- If the condition is True, then the clause is executed, and afterward, the condition is checked again.
- The first time the condition is found to be False, the while clause is skipped.

An annoying while loop:

- Here's a small example program that will keep asking to type, literally, your name.

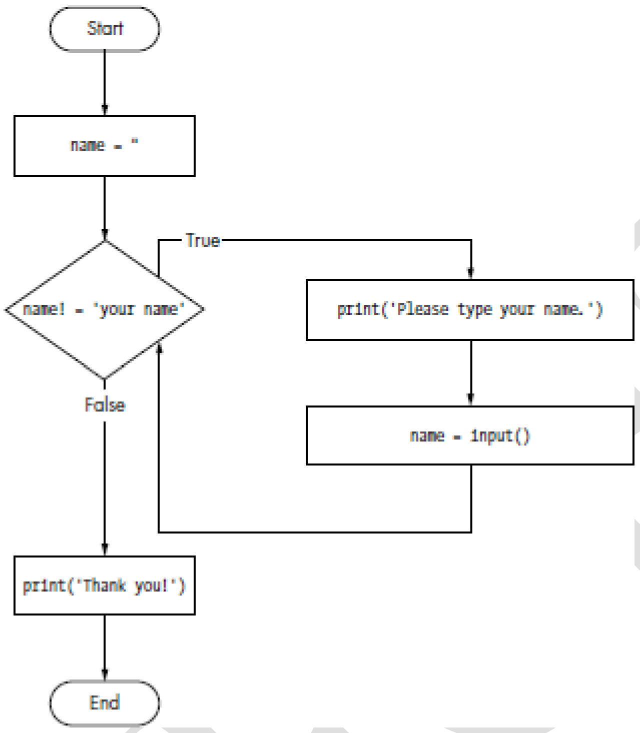
Example Program	Output
 `❶ name = ''` `❷ while name != 'your name':` `print('Please type your name.')` `❸ name = input()` `❹ print('Thank you!')`	 `Please type your name.` `Al` `Please type your name.` `Albert` `Please type your name.` `%#@#%*(^&!!!` `Please type your name.` `your name` `Thank you!`

```python
❶ name = ''
❷ while name != 'your name':
      print('Please type your name.')
❸     name = input()
❹ print('Thank you!')
```

```
Please type your name.
Al
Please type your name.
Albert
Please type your name.
%#@#%*(^&!!!
Please type your name.
your name
Thank you!
```

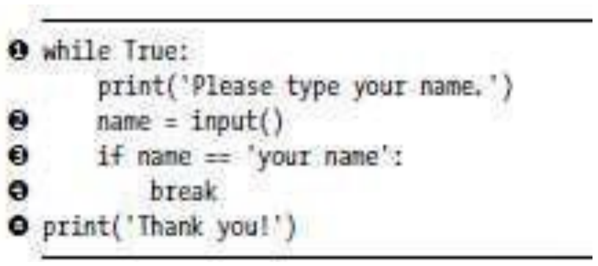
- First, the program sets the name variable ❶ to an empty string.
- This is so that the name != 'your name' condition will evaluate to True and the program execution will enter the while loop's clause ❷.
- The code inside this clause asks the user to type their name, which is assigned to the name variable ❸.
- Since this is the last line of the block, the execution moves back to the start of the while loop and reevaluates the condition.
- If the value in name is not equal to the string 'your name', then the condition is True, and the execution enters the while clause again.
- But once the user types your name, the condition of the while loop will be 'your name' != 'your name', which evaluates to False.

- The condition is now False, and instead of the program execution reentering the while loop's clause, it skips past it and continues running the rest of the program ❹.
- Flowchart:



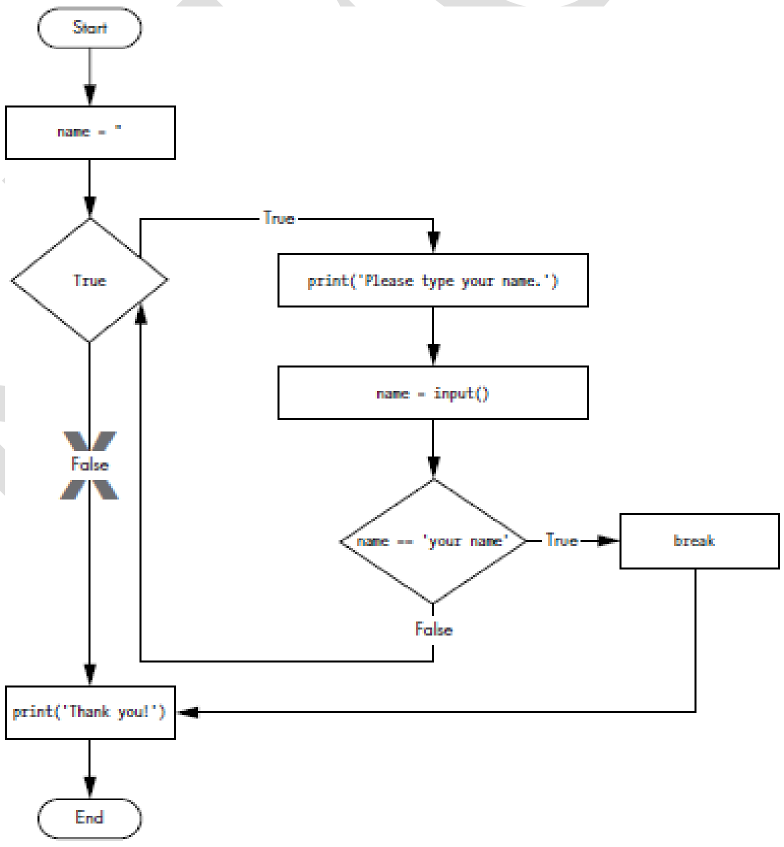
5. ***break Statements:***

- There is a shortcut to getting the program execution to break out of a while loop's clause early.
- If the execution reaches a break statement, it immediately exits the while loop's clause
- In code, a break statement simply contains the break keyword.
- Example:



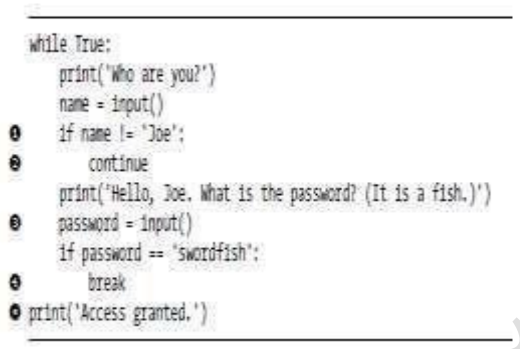
```
❶ while True:
      print('Please type your name.')
❷     name = input()
❸     if name == 'your name':
❹         break
❺ print('Thank you!')
```

- The first line ❶ creates an infinite loop; it is a while loop whose condition is always True. (The expression True, after all, always evaluates down to the value True.)
- The program execution will always enter the loop and will exit it only when a break statement is executed. (An infinite loop that never exits is a common programming bug.)
- Just like before, this program asks the user to type your name ❷.
- Now, however, while the execution is still inside the while loop, an if statement gets executed ❸ to check whether name is equal to your name.
- If this condition is True, the break statement is run ❹, and the execution moves out of the loop to print('Thank you!') ❺.
- Otherwise, the if statement‘s clause with the break statement is skipped, which puts the execution at the end of the while loop.
- At this point, the program execution jumps back to the start of the while statement ❶ to recheck the condition. Since this condition is merely the True Boolean value, the execution enters the loop to ask the user to type your name again.
- Flowchart:

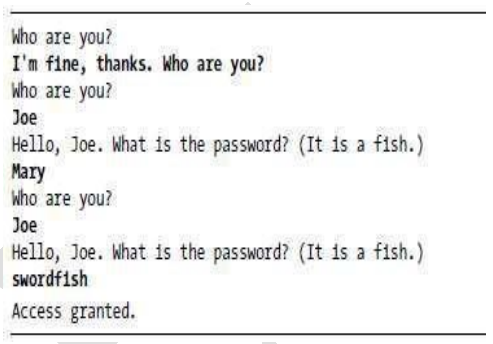


6. *Continue statement*

- Like break statements, continue statements are used inside loops.
- When the program execution reaches a continue statement, the program execution immediately jumps back to the start of the loop and reevaluates the loop's condition.
- Example and Output:

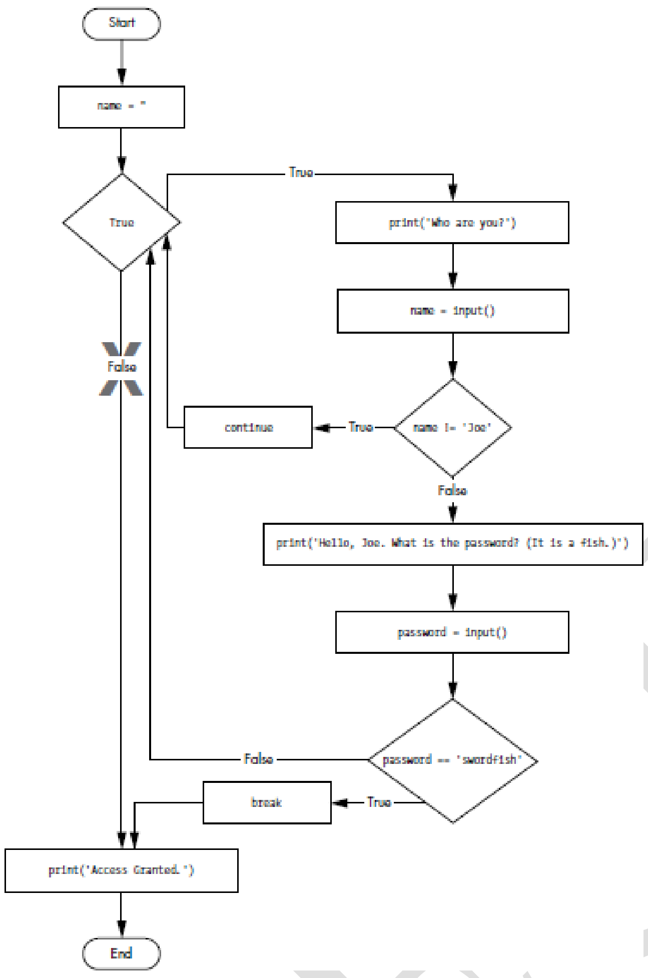


```
while True:
    print('Who are you?')
    name = input()
❶   if name != 'Joe':
❷       continue
    print('Hello, Joe. What is the password? (It is a fish.)')
❸   password = input()
    if password == 'swordfish':
❹       break
❺ print('Access granted.')
```



```
Who are you?
I'm fine, thanks. Who are you?
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
Mary
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
swordfish
Access granted.
```

- If the user enters any name besides Joe ❶, the continue statement ❷ causes the program execution to jump back to the start of the loop.
- When it reevaluates the condition, the execution will always enter the loop, since the condition is simply the value True. Once they make it past that if statement, the user is asked for a password ❸.
- If the password entered is swordfish, then the break statement ❹ is run, and the execution jumps out of the while loop to print Access granted ❺.
- Otherwise, the execution continues to the end of the while loop, where it then jumps back to the start of the loop.
- Flowchart:



- There are some values in other data types that conditions will consider equivalent to True and False.
- When used in conditions, 0, 0.0, and '' (the empty string) are considered False, while all other values are considered True.
- Example:

```
name = ''
while not name:❶
    print('Enter your name:')
    name = input()
print('How many guests will you have?')
numOfGuests = int(input())
if numOfGuests:❷
    print('Be sure to have enough room for all your guests.')❸
print('Done')
```

7. for loops and the range() function:

- If we want to execute a block of code only a certain number of times then we can do this with a for loop statement and the range() function.
- In code, a for statement looks something like for i in range(5): and always includes the following:

1. The for keyword
2. A variable name
3. The in keyword
4. A call to the range() method with up to three integers passed to it
5. A colon
6. Starting on the next line, an indented block of code (called the for clause)

- Example and output:

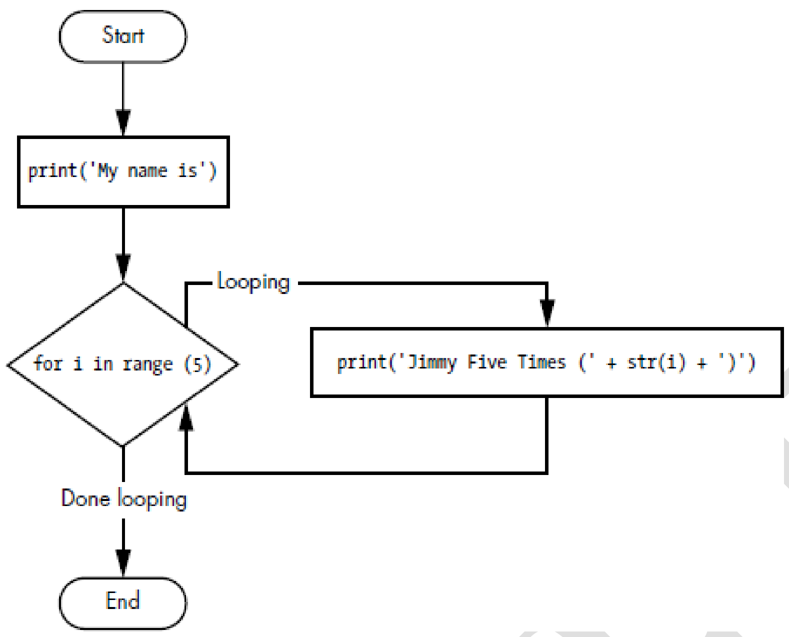
Example

```
print('My name is')
for i in range(5):
    print('Jimmy Five Times (' + str(i) + ')')
```

Output

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

- The code in the for loop‘s clause is run five times.
- The first time it is run, the variable i is set to 0.
- The print() call in the clause will print Jimmy Five Times (0).
- After Python finishes an iteration through all the code inside the for loop‘s clause, the execution goes back to the top of the loop, and the for statement increments i by one.
- This is why range(5) results in five iterations through the clause, with i being set to 0, then 1, then 2, then 3, and then 4.
- The variable i will go up to, but will not include, the integer passed to range().
- Flowchart:



- Example 2:

```
❶ total = 0
❷ for num in range(101):
❸     total = total + num
❹ print(total)
```

- The result should be 5,050. When the program first starts, the total variable is set to 0 ❶.
- The for loop ❷ then executes total = total + num ❸ 100 times.
- By the time the loop has finished all of its 100 iterations, every integer from 0 to 100 will have been added to total. At this point, total is printed to the screen ❹.

An equivalent while loop: For the first example of for loop.

```
print('My name is')
i = 0
while i < 5:
    print('Jimmy Five Times (' + str(i) + ')')
    i = i + 1
```

8. <u>The Starting, Stopping, and Stepping Arguments to range()</u>

- Some functions can be called with multiple arguments separated by a comma, and range() is one of them.
- This lets us change the integer passed to range() to follow any sequence of integers, including starting at a number other than zero.

```
for i in range(12, 16):
    print(i)
```

- The first argument will be where the for loop‘s variable starts, and the second argument will be up to, but not including, the number to stop at.

```
12
13
14
15
```

- The range() function can also be called with three arguments. The first two arguments will be the start and stop values, and the third will be the step argument. The step is the amount that the variable is increased by after each iteration.

```
for i in range(0, 10, 2):
    print(i)
```

- So calling range(0, 10, 2) will count from zero to eight by intervals of two.

```
0
2
4
6
8
```

- The range() function is flexible in the sequence of numbers it produces for for loops. We can even use a negative number for the step argument to make the for loop count down instead of up.

```
for i in range(5, -1, -1):
    print(i)
```

- Running a for loop to print i with range(5, -1, -1) should print from five down to zero.

```
5
4
3
2
1
0
```

2.5 Importing Modules

- All Python programs can call a basic set of functions called built-in functions, including the print(), input(), and len() functions.
- Python also comes with a set of modules called the standard library.
- Each module is a Python program that contains a related group of functions that can be embedded in your programs.
- For example, the math module has mathematics-related functions, the random module has random number–related functions, and so on.
- Before we can use the functions in a module, we must import the module with an import statement. In code, an import statement consists of the following:
  1. The import keyword
  2. The name of the module
  3. Optionally, more module names, as long as they are separated by commas
- Once we import a module, we can use all the functions of that module.

➢ Example with output:

```
import random
for i in range(5):
    print(random.randint(1, 10))
```

```
4
1
8
4
1
```

➢ The random.randint() function call evaluates to a random integer value between the two integers that you pass it.

➢ Since randint() is in the random module, we must first type random. in front of the function name to tell Python to look for this function inside the random module.

➢ Here‘s an example of an import statement that imports four different modules:

```
import random, sys, os, math
```

➢ Now we can use any of the functions in these four modules.

<u>from import Statements</u>

➢ An alternative form of the import statement is composed of the from keyword, followed by the module name, the import keyword, and a star; for example, from random import *.

➢ With this form of import statement, calls to functions in random will not need the random prefix.

➢ However, using the full name makes for more readable code, so it is better to use the normal form of the import statement.

2.6 Ending a Program Early with sys.exit()

- The last flow control concept is how to terminate the program. This always happens if the program execution reaches the bottom of the instructions.
- However, we can cause the program to terminate, or exit, by calling the sys.exit() function. Since this function is in the sys module, we have to import sys before your program can use it.

```
import sys

while True:
    print('Type exit to exit.')
    response = input()
    if response == 'exit':
        sys.exit()
    print('You typed ' + response + '.')
```

- This program has an infinite loop with no break statement inside. The only way this program will end is if the user enters exit, causing sys.exit() to be called.
- When response is equal to exit, the program ends.
- Since the response variable is set by the input() function, the user must enter exit in order to stop the program.

CHAPTER 3: FUNCTIONS

1. def Statements with Parameters
2. Return Values and return Statements
3. The None Value
4. Keyword Arguments and print()
5. Local and Global Scope
6. The global Statement
7. Exception Handling
8. A Short Program: Guess the Number

Introduction

- A function is like a mini-program within a program.
- Example:

```
❶ def hello():
❷     print('Howdy!')
      print('Howdy!!!')
      print('Hello there.')

❸ hello()
  hello()
  hello()
```

- The first line is a def statement ❶, which defines a function named hello().
- The code in the block that follows the def statement ❷ is the body of the function. This code is executed when the function is called, not when the function is first defined.
- The hello() lines after the function ❸ are function calls.
- In code, a function call is just the function's name followed by parentheses, possibly with some number of arguments in between the parentheses.
- When the program execution reaches these calls, it will jump to the top line in the function and begin executing the code there.
- When it reaches the end of the function, the execution returns to the line that called the function and continues moving through the code as before.
- Since this program calls hello() three times, the code in the hello() function is executed three times. When we run this program, the output looks like this:

```
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
```

- A major purpose of functions is to group code that gets executed multiple times. Without a function defined, we would have to copy and paste this code each time, and the program would look like this:

```
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
```

3.1 def Statements with Parameters

- When we call the print() or len() function, we pass in values, called arguments in this context, by typing them between the parentheses.
- We can also define our own functions that accept arguments.
- Example with output:

```
❶ def hello(name):
❷     print('Hello ' + name)

❸ hello('Alice')
  hello('Bob')
```

```
Hello Alice
Hello Bob
```

- The definition of the hello() function in this program has a parameter called name ❶.
- A parameter is a variable that an argument is stored in when a function is called. The first time the hello() function is called, it‘s with the argument 'Alice' ❸.

- The program execution enters the function, and the variable name is automatically set to 'Alice', which is what gets printed by the print() statement ❷.
- One special thing to note about parameters is that the value stored in a parameter is forgotten when the function returns.

3.2 Return Values and Return Statements

- The value that a function call evaluates to is called the return value of the function.
- Ex: len(‗Hello‘) → Return values is 5
- When creating a function using the def statement, we can specify what the return value should be with a return statement.
- A return statement consists of the following:
  1. The return keyword
  2. The value or expression that the function should return.
- When an expression is used with a return statement, the return value is what this expression evaluates to.
- For example, the following program defines a function that returns a different string depending on what number it is passed as an argument.

```
❶ import random

❷ def getAnswer(answerNumber):
❸     if answerNumber == 1:
          return 'It is certain'
      elif answerNumber == 2:
          return 'It is decidedly so'
      elif answerNumber == 3:
          return 'Yes'
      elif answerNumber == 4:
          return 'Reply hazy try again'
      elif answerNumber == 5:
          return 'Ask again later'
      elif answerNumber == 6:
          return 'Concentrate and ask again'
      elif answerNumber == 7:
          return 'My reply is no'
      elif answerNumber == 8:
          return 'Outlook not so good'
      elif answerNumber == 9:
          return 'Very doubtful'

❹ r = random.randint(1, 9)
❺ fortune = getAnswer(r)
❻ print(fortune)
```

- When this program starts, Python first imports the random module ❶.
- Then the getAnswer() function is defined ❷. Because the function is being defined (and not called), the execution skips over the code in it.
- Next, the random.randint() function is called with two arguments, 1 and 9 ❹.
- It evaluates to a random integer between 1 and 9 (including 1 and 9 themselves), and this value is stored in a variable named r.
- The getAnswer() function is called with r as the argument ❺.
- The program execution moves to the top of the getAnswer() function ❸, and the value r is stored in a parameter named answerNumber.
- Then, depending on this value in answerNumber, the function returns one of many possible string values. The program execution returns to the line at the bottom of the program that originally called getAnswer() ❺.
- The returned string is assigned to a variable named fortune, which then gets passed to a print() call ❻ and is printed to the screen.
- Note that since we can pass return values as an argument to another function call, we could shorten these three lines into single line as follows:

```
r = random.randint(1, 9)
fortune = getAnswer(r)
print(fortune)
```

```
print(getAnswer(random.randint(1, 9)))
```

3.3 The None Value

- In Python there is a value called None, which represents the absence of a value.
- None is the only value of the NoneType data type.
- This value-without-a-value can be helpful when we need to store something that won't be confused for a real value in a variable.
- One place where None is used is as the return value of print().
- The print() function displays text on the screen, but it doesn't need to return anything in the same way len() or input() does. But since all function calls need to evaluate to a return value, print() returns None.

```
>>> spam = print('Hello!')
Hello!
>>> None == spam
True
```

3.3 Keyword Arguments and print()

- Most arguments are identified by their position in the function call.
- For example, random.randint(1, 10) is different from random.randint(10, 1).
- The function call random.randint(1, 10) will return a random integer between 1 and 10, because the first argument is the low end of the range and the second argument is the high end while random.randint(10, 1) causes an error.
- However, keyword arguments are identified by the keyword put before them in the function call.
- Keyword arguments are often used for optional parameters.
- For example, the print() function has the optional parameters end and sep to specify what should be printed at the end of its arguments and between its arguments (separating them), respectively.

```
print('Hello')
print('World')
```

```
Hello
World
```

- The two strings appear on separate lines because the print() function automatically adds a newline character to the end of the string it is passed.
- However, we can set the end keyword argument to change this to a different string.
- For example, if the program were this:

```
print('Hello', end='')
print('World')
```

```
HelloWorld
```

- The output is printed on a single line because there is no longer a new-line printed after 'Hello'. Instead, the blank string is printed. This is useful if we need to disable the newline that gets added to the end of every print() function call.
- Similarly, when we pass multiple string values to print(), the function will automatically separate them with a single space.

```
>>> print('cats', 'dogs', 'mice')
cats dogs mice
```

- But we could replace the default separating string by passing the sep keyword argument

```
>>> print('cats', 'dogs', 'mice', sep=',')
cats,dogs,mice
```
.

3.4 Local and Global Scope

- Parameters and variables that are assigned in a called function are said to exist in that function's *local scope.*
- Variables that are assigned outside all functions are said to exist in the *global scope*.
- A variable that exists in a local scope is called a *local variable*, while a variable that exists in the global scope is called a *global variable.*
- A variable must be one or the other; it cannot be both local and global.
- When a scope is destroyed, all the values stored in the scope's variables are forgotten.
- There is only one global scope, and it is created when your program begins. When your program terminates, the global scope is destroyed, and all its variables are forgotten.
- A local scope is created whenever a function is called. Any variables assigned in this function exist within the local scope. When the function returns, the local scope is destroyed, and these variables are forgotten.
- Scopes matter for several reasons:
  1. Code in the global scope cannot use any local variables.
  2. However, a local scope can access global variables.
  3. Code in a function's local scope cannot use variables in any other local scope.
  1. We can use the same name for different variables if they are in different scopes. That is, there can be a local variable named spam and a global variable also named spam.

Local Variables Cannot Be Used in the Global Scope

- Consider this program, which will cause an error when you run it:

Example

```
def spam():
    eggs = 31337
spam()
print(eggs)
```

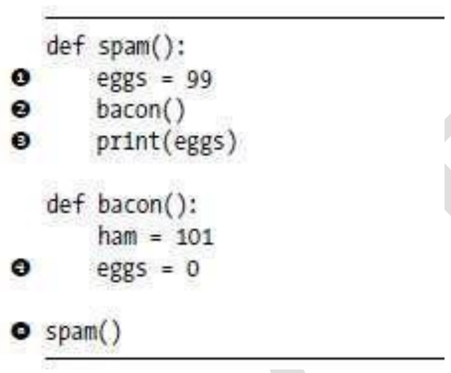
Output

```
Traceback (most recent call last):
  File "C:/test3784.py", line 4, in <module>
    print(eggs)
NameError: name 'eggs' is not defined
```

- The error happens because the eggs variable exists only in the local scope created when spam() is called.
- Once the program execution returns from spam, that local scope is destroyed, and there is no longer a variable named eggs.

<u>Local Scopes Cannot Use Variables in Other Local Scopes</u>

- A new local scope is created whenever a function is called, including when a function is called from another function. Consider this program:



```
  def spam():
❶     eggs = 99
❷     bacon()
❸     print(eggs)

  def bacon():
      ham = 101
❹     eggs = 0

❺ spam()
```

- When the program starts, the spam() function is called ❺, and a local scope is created.
- The local variable eggs ❶ is set to 99.
- Then the bacon() function is called ❷, and a second local scope is created. Multiple local scopes can exist at the same time.
- In this new local scope, the local variable ham is set to 101, and a local variable eggs—which is different from the one in spam()'s local scope—is also created ❹ and set to 0.
- When bacon() returns, the local scope for that call is destroyed. The program execution continues in the spam() function to print the value of eggs ❸, and since the local scope for the call to spam() still exists here, the eggs variable is set to 99.

Global Variables Can Be Read from a Local Scope

- Consider the following program:

```
def spam():
    print(eggs)
eggs = 42
spam()
print(eggs)
```

- Since there is no parameter named eggs or any code that assigns eggs a value in the spam() function, when eggs is used in spam(), Python considers it a reference to the global variable eggs. This is why 42 is printed when the previous program is run.

Local and Global Variables with the Same Name

- To simplify, avoid using local variables that have the same name as a global variable or another local variable.
- But technically, it‘s perfectly legal to do so.

Example

```
  def spam():
❶     eggs = 'spam local'
      print(eggs)    # prints 'spam local'

  def bacon():
❷     eggs = 'bacon local'
      print(eggs)    # prints 'bacon local'
      spam()
      print(eggs)    # prints 'bacon local'

❸ eggs = 'global'
  bacon()
  print(eggs)        # prints 'global'
```

Output

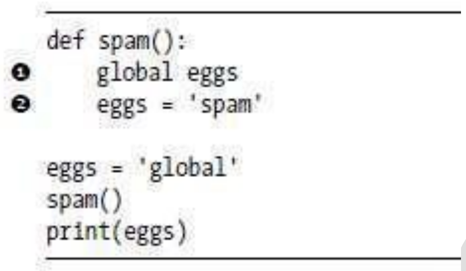
```
bacon local
spam local
bacon local
global
```

- There are actually three different variables in this program, but confusingly they are all named eggs. The variables are as follows:
- ❶ A variable named eggs that exists in a local scope when spam() is called.
- ❷ A variable named eggs that exists in a local scope when bacon() is called.
- ❸ A variable named eggs that exists in the global scope.
- Since these three separate variables all have the same name, it can be confusing to keep track of which one is being used at any given time. This is why we should avoid using the same variable name in different scopes.

3.5 The Global Statement

- If we need to modify a global variable from within a function, use the global statement.
- If we have a line such as global eggs at the top of a function, it tells Python, ―In this function, eggs refers to the global variable, so don‘t create a local variable with this name.‖
- For example:

Program **Output**



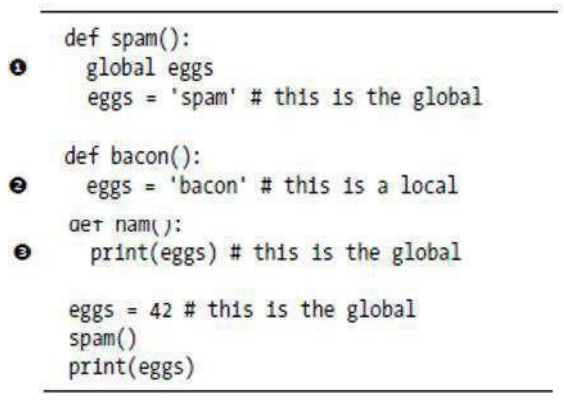
```
  def spam():
❶     global eggs
❷     eggs = 'spam'

  eggs = 'global'
  spam()
  print(eggs)
```

```
spam
```

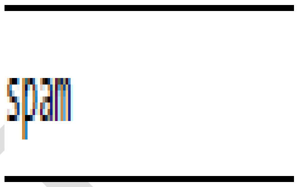
- Because eggs is declared global at the top of spam() ❶, when eggs is set to 'spam' ❷, this assignment is done to the globally scoped eggs. No local eggs variable is created.
- There are four rules to tell whether a variable is in a local scope or global scope:
- If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
- If there is a global statement for that variable in a function, it is a global variable.
- Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
- But if the variable is not used in an assignment statement, it is a global variable.

- Example:

Program **Output**



```
    def spam():
❶     global eggs
      eggs = 'spam' # this is the global

    def bacon():
❷     eggs = 'bacon' # this is a local
    def ham():
❸     print(eggs) # this is the global

    eggs = 42 # this is the global
    spam()
    print(eggs)
```



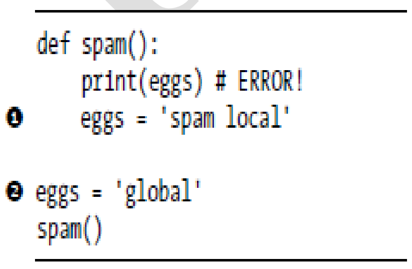
```
spam
```

- In the spam() function, eggs is the global eggs variable, because there's a global statement for eggs at the beginning of the function ❶.
- In bacon(), eggs is a local variable, because there's an assignment statement for it in that function ❷.
- In ham() ❸, eggs is the global variable, because there is no assignment statement or global statement for it in that function
- In a function, a variable will either always be global or always be local. There's no way that the code in a function can use a local variable named eggs and then later in that same function use the global eggs variable.

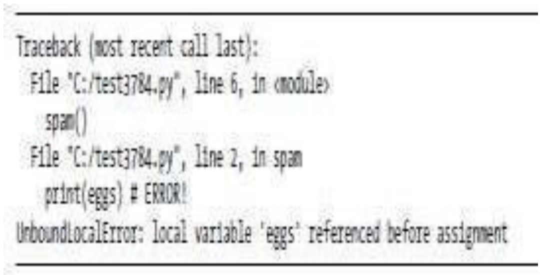
Note

- If we ever want to modify the value stored in a global variable from in a function, we must use a global statement on that variable.
- If we try to use a local variable in a function before we assign a value to it, as in the following program, Python will give you an error.

Program **Output**



```
  def spam():
      print(eggs) # ERROR!
❶     eggs = 'spam local'

❷ eggs = 'global'
  spam()
```



```
Traceback (most recent call last):
  File "C:/test3784.py", line 6, in <module>
    spam()
  File "C:/test3784.py", line 2, in spam
    print(eggs) # ERROR!
UnboundLocalError: local variable 'eggs' referenced before assignment
```

- This error happens because Python sees that there is an assignment statement for eggs in the spam() function ❶ and therefore considers eggs to be local.
- But because print(eggs) is executed before eggs is assigned anything, the local variable eggs doesn‘t exist. Python will not fall back to using global eggs variable ❷.

3.6 Exception Handling

- If we don‘t want to crash the program due to errors instead we want the program to detect errors, handle them, and then continue to run.
- For example,

Program

```
def spam(divideBy):
    return 42 / divideBy

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

Output

```
21.0
3.5
Traceback (most recent call last):
  File "C:/zeroDivide.py", line 6, in <module>
    print(spam(0))
  File "C:/zeroDivide.py", line 2, in spam
    return 42 / divideBy
ZeroDivisionError: division by zero
```

- A ZeroDivisionError happens whenever we try to divide a number by zero. From the line number given in the error message, we know that the return statement in spam() is causing an error.
- Errors can be handled with try and except statements.
- The code that could potentially have an error is put in a try clause. The program execution moves to the start of a following except clause if an error happens.
- We can put the previous divide-by-zero code in a try clause and have an except clause contain code to handle what happens when this error occurs.

Program

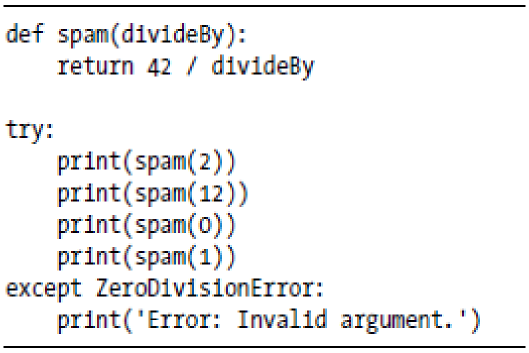
```
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        print('Error: Invalid argument.')

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

Output

```
21.0
3.5
Error: Invalid argument.
None
42.0
```

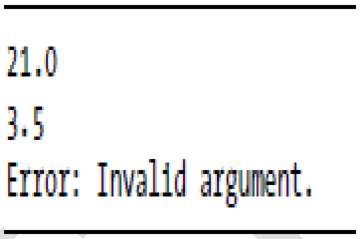
- Note that any errors that occur in function calls in a try block will also be caught. Consider the following program, which instead has the spam() calls in the try block:

Program



```
def spam(divideBy):
    return 42 / divideBy

try:
    print(spam(2))
    print(spam(12))
    print(spam(0))
    print(spam(1))
except ZeroDivisionError:
    print('Error: Invalid argument.')
```

Output

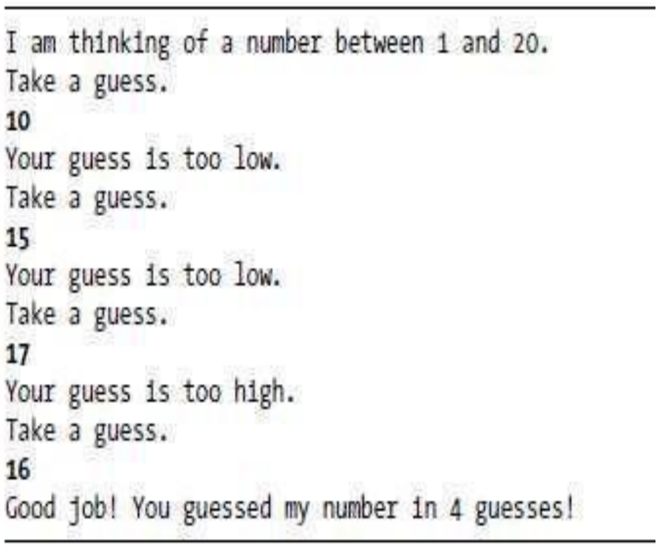


```
21.0
3.5
Error: Invalid argument.
```

- The reason print(spam(1)) is never executed is because once the execution jumps to the code in the except clause, it does not return to the try clause. Instead, it just continues moving down as normal.

3.7 A Short program: Guess the Number

- This is a simple ―guess the number‖ game. When we run this program, the output will look something like this:



```
I am thinking of a number between 1 and 20.
Take a guess.
10
Your guess is too low.
Take a guess.
15
Your guess is too low.
Take a guess.
17
Your guess is too high.
Take a guess.
16
Good job! You guessed my number in 4 guesses!
```

- Code for the above program is:

```
# This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
print('I am thinking of a number between 1 and 20.')

# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
    print('Take a guess.')
    guess = int(input())

    if guess < secretNumber:
        print('Your guess is too low.')
    elif guess > secretNumber:
        print('Your guess is too high.')
    else:
        break    # This condition is the correct guess!

if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + ' guesses!')
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

- Let‘s look at this code line by line, starting at the top.

```
# This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
```

- First, a comment at the top of the code explains what the program does.
- Then, the program imports the random module so that it can use the random.randint() function to generate a number for the user to guess.
- The return value, a random integer between 1 and 20, is stored in the variable secretNumber.

```
print('I am thinking of a number between 1 and 20.')

# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
    print('Take a guess.')
    guess = int(input())
```

- The program tells the player that it has come up with a secret number and will give the player six chances to guess it.
- The code that lets the player enter a guess and checks that guess is in a for loop that will loop at most six times.
- The first thing that happens in the loop is that the player types in a guess.
- Since input() returns a string, its return value is passed straight into int(), which translates the string into an integer value. This gets stored in a variable named guess.

```
    if guess < secretNumber:
        print('Your guess is too low.')
    elif guess > secretNumber:
        print('Your guess is too high.')
```

- These few lines of code check to see whether the guess is less than or greater than the secret number. In either case, a hint is printed to the screen.

```
    else:
        break    # This condition is the correct guess!
```

- If the guess is neither higher nor lower than the secret number, then it must be equal to the secret number, in which case you want the program execution to break out of the for loop.

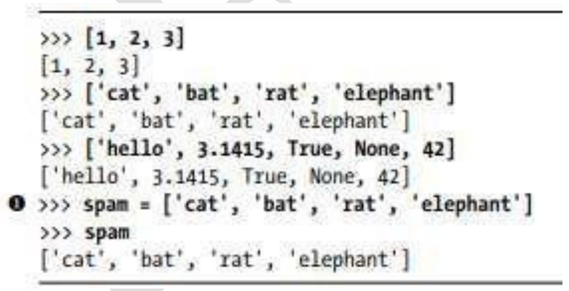
```
if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + ' guesses!')
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

- After the for loop, the previous if...else statement checks whether the player has correctly guessed the number and prints an appropriate message to the screen.
- In both cases, the program displays a variable that contains an integer value (guessesTaken and secretNumber).
- Since it must concatenate these integer values to strings, it passes these variables to the str() function, which returns the string value form of these integers.
- Now these strings can be concatenated with the + operators before finally being passed to the print() function call.

CHAPTER 1: LISTS

1.1 The List Data Type

- A list is a value that contains multiple values in an ordered sequence.
- A list value looks like this: ['cat', 'bat', 'rat', 'elephant'].
- A list begins with an opening square bracket and ends with a closing square bracket, [].
- Values inside the list are also called items and are separated with commas.

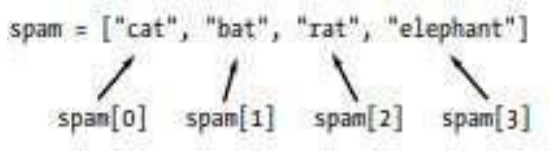


```
>>> [1, 2, 3]
[1, 2, 3]
>>> ['cat', 'bat', 'rat', 'elephant']
['cat', 'bat', 'rat', 'elephant']
>>> ['hello', 3.1415, True, None, 42]
['hello', 3.1415, True, None, 42]
❶ >>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam
['cat', 'bat', 'rat', 'elephant']
```

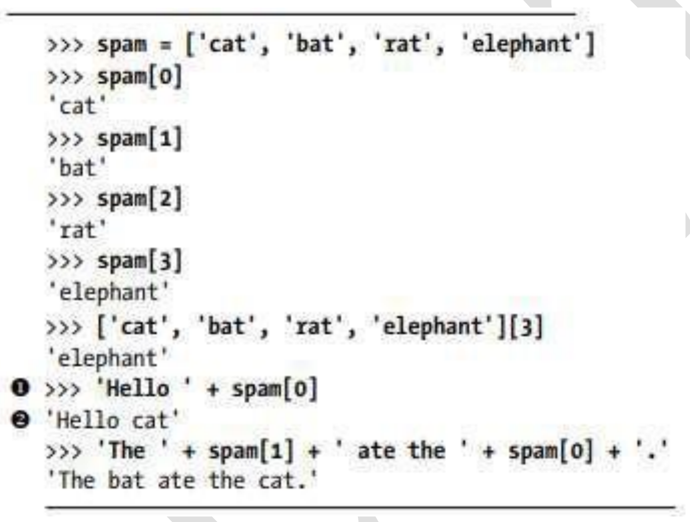
- The spam variable ❶ is still assigned only one value: the list value(contains multiple values).
- The value [] is an empty list that contains no values, similar to '', the empty string.

Getting Individual Values in a List with Indexes

- Say you have the list ['cat', 'bat', 'rat', 'elephant'] stored in a variable named spam.
- The Python code spam[0] would evaluate to 'cat', and spam[1] would evaluate to 'bat', and so on.

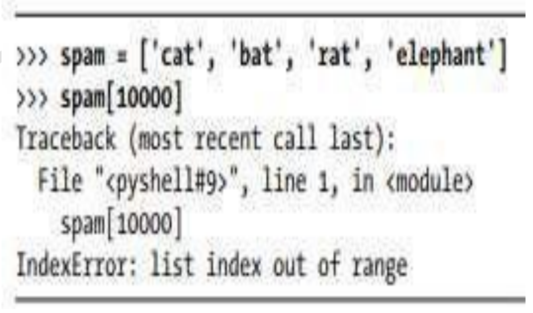


- The first value in the list is at index 0, the second value is at index 1, and the third value is at index 2, and so on.
- For example, type the following expressions into the interactive shell.



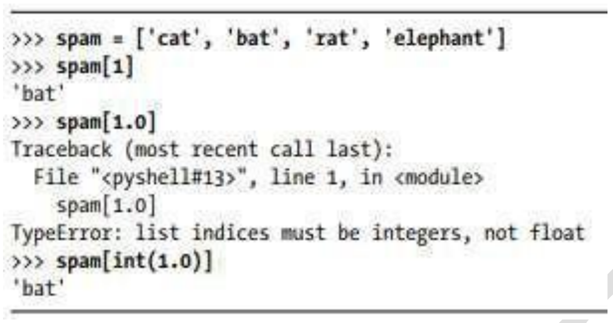
```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0]
'cat'
>>> spam[1]
'bat'
>>> spam[2]
'rat'
>>> spam[3]
'elephant'
>>> ['cat', 'bat', 'rat', 'elephant'][3]
'elephant'
❶ >>> 'Hello ' + spam[0]
❷ 'Hello cat'
>>> 'The ' + spam[1] + ' ate the ' + spam[0] + '.'
'The bat ate the cat.'
```

- The expression 'Hello ' + spam[0] evaluates to 'Hello ' + 'cat' because spam[0] evaluates to the string 'cat'. This expression in turn evaluates to the string value 'Hello cat'.
- If we use an index that exceeds the number of values in the list value then, python gives IndexError.



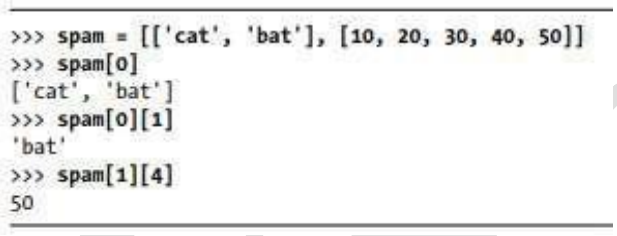
```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[10000]
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    spam[10000]
IndexError: list index out of range
```

- Indexes can be only integer values, not floats. The following example will cause a TypeError error:



```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1]
'bat'
>>> spam[1.0]
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    spam[1.0]
TypeError: list indices must be integers, not float
>>> spam[int(1.0)]
'bat'
```

- Lists can also contain other list values. The values in these lists of lists can be accessed using multiple indexes.

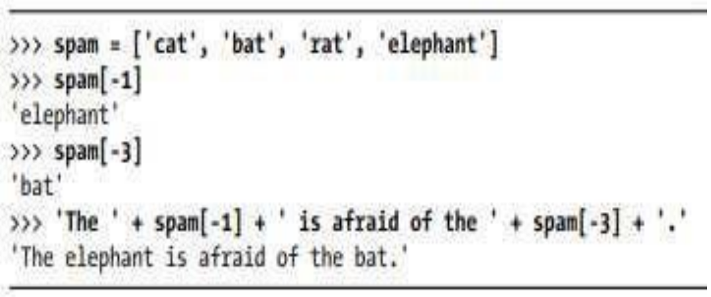


```
>>> spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
>>> spam[0]
['cat', 'bat']
>>> spam[0][1]
'bat'
>>> spam[1][4]
50
```

- The first index dictates which list value to use, and the second indicates the value within the list value. **Ex**, spam[0][1] prints 'bat', the second value in the first list.

<u>Negative Indexes</u>

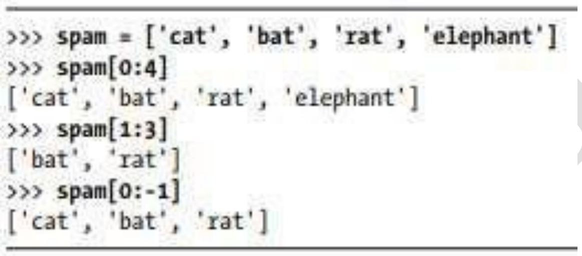
- We can also use negative integers for the index. The integer value -1 refers to the last index in a list, the value -2 refers to the second-to-last index in a list, and so on.



```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[-1]
'elephant'
>>> spam[-3]
'bat'
>>> 'The ' + spam[-1] + ' is afraid of the ' + spam[-3] + '.'
'The elephant is afraid of the bat.'
```

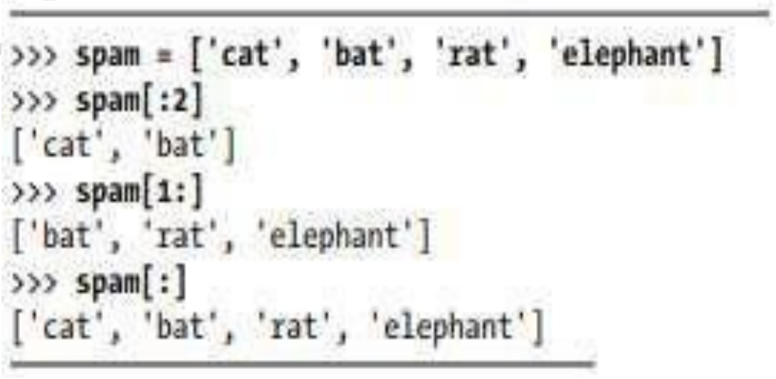
Getting Sublists with Slices

- An index will get a single value from a list, a slice can get several values from a list, in the form of a new list.
- A slice is typed between square brackets, like an index, but it has two integers separated by a colon.
- **Difference between indexes and slices.**
    - spam[2] is a list with an index (one integer).
    - spam[1:4] is a list with a slice (two integers).
- In a slice, the first integer is the index where the slice starts. The second integer is the index where the slice ends (but will not include the value at the second index).



```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0:4]
['cat', 'bat', 'rat', 'elephant']
>>> spam[1:3]
['bat', 'rat']
>>> spam[0:-1]
['cat', 'bat', 'rat']
```

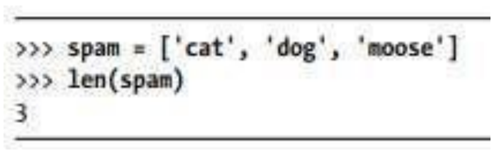
- ➢ As a shortcut, we can leave out one or both of the indexes on either side of the colon in the slice.
    - Leaving out the first index is the same as using 0, or the beginning of the list.
    - Leaving out the second index is the same as using the length of the list, which will slice to the end of the list.



```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[:2]
['cat', 'bat']
>>> spam[1:]
['bat', 'rat', 'elephant']
>>> spam[:]
['cat', 'bat', 'rat', 'elephant']
```

Getting a List’s Length with len()

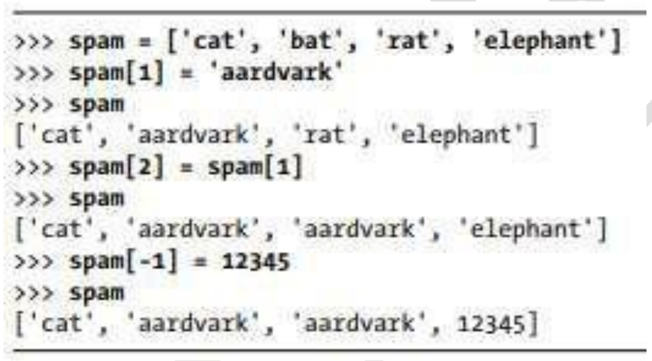
- The len() function will return the number of values that are in a list value.



```
>>> spam = ['cat', 'dog', 'moose']
>>> len(spam)
3
```

Changing Values in a List with Indexes

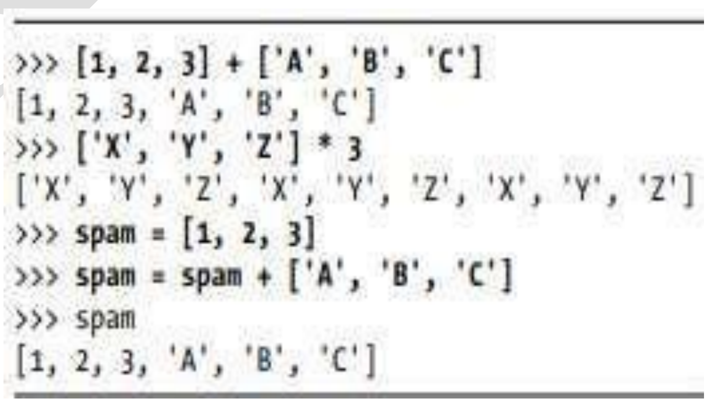
- We can also use an index of a list to change the value at that index.
- **Ex:** spam[1] = 'aardvark' means “Assign the value at index 1 in the list spam to the string 'aardvark'.”



```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1] = 'aardvark'
>>> spam
['cat', 'aardvark', 'rat', 'elephant']
>>> spam[2] = spam[1]
>>> spam
['cat', 'aardvark', 'aardvark', 'elephant']
>>> spam[-1] = 12345
>>> spam
['cat', 'aardvark', 'aardvark', 12345]
```

List Concatenation and List Replication

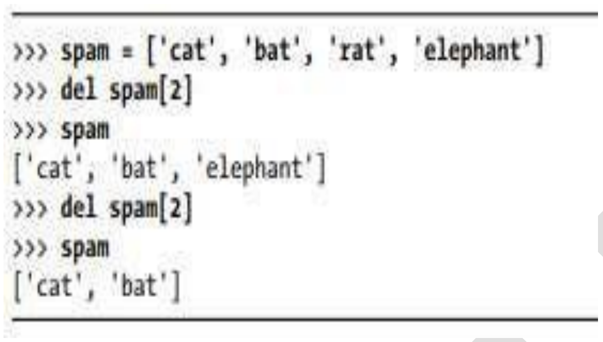
- The + operator can combine two lists to create a new list value in the same way it combines two strings into a new string value.
- The * operator can also be used with a list and an integer value to replicate the list.



```
>>> [1, 2, 3] + ['A', 'B', 'C']
[1, 2, 3, 'A', 'B', 'C']
>>> ['X', 'Y', 'Z'] * 3
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
>>> spam = [1, 2, 3]
>>> spam = spam + ['A', 'B', 'C']
>>> spam
[1, 2, 3, 'A', 'B', 'C']
```

Removing Values from Lists with del Statements

- The del statement will delete values at an index in a list.

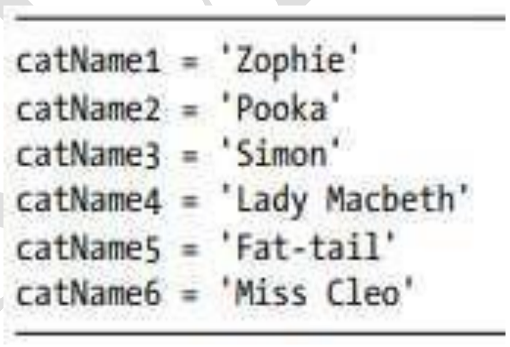


```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat']
```

- The del statement can also be used to delete a variable After deleting if we try to use the variable, we will get a NameError error because the variable no longer exists.
- In practice, you almost never need to delete simple variables.
- The del statement is mostly used to delete values from lists.

1.2 Working with Lists

- When we first begin writing programs, it's tempting to create many individual variables to store a group of similar values.



```
catName1 = 'Zophie'
catName2 = 'Pooka'
catName3 = 'Simon'
catName4 = 'Lady Macbeth'
catName5 = 'Fat-tail'
catName6 = 'Miss Cleo'
```

- Which is bad way to write code because it leads to have a duplicate code in the program.

```
print('Enter the name of cat 1:')
catName1 = input()
print('Enter the name of cat 2:')
catName2 = input()
print('Enter the name of cat 3:')
catName3 = input()
print('Enter the name of cat 4:')
catName4 = input()
print('Enter the name of cat 5:')
catName5 = input()
print('Enter the name of cat 6:')
catName6 = input()
print('The cat names are:')
print(catName1 + ' ' + catName2 + ' ' + catName3 + ' ' + catName4 + ' ' +
catName5 + ' ' + catName6)
```

- Instead of using multiple, repetitive variables, we can use a single variable that contains a list value.
- **For Ex:** The following program uses a single list and it can store any number of cats that the user types in.
- Program:

```
catNames = []
while True:
    print('Enter the name of cat ' + str(len(catNames) + 1) +
      ' (Or enter nothing to stop.):')
    name = input()
    if name == '':
        break
    catNames = catNames + [name]  # list concatenation
print('The cat names are:')
for name in catNames:
    print('  ' + name)
```

Output:

```
Enter the name of cat 1 (Or enter nothing to stop.):
Zophie
Enter the name of cat 2 (Or enter nothing to stop.):
Pooka
Enter the name of cat 3 (Or enter nothing to stop.):
Simon
Enter the name of cat 4 (Or enter nothing to stop.):
Lady Macbeth
Enter the name of cat 5 (Or enter nothing to stop.):
Fat-tail
Enter the name of cat 6 (Or enter nothing to stop.):
Miss Cleo
Enter the name of cat 7 (Or enter nothing to stop.):

The cat names are:
  Zophie
  Pooka
  Simon
  Lady Macbeth
  Fat-tail
  Miss Cleo
```

Using for Loops with Lists

- A for loop repeats the code block once for each value in a list or list-like value.

Program

```
for i in range(4):
    print(i)
```

Output:

```
0
1
2
3
```

- A common Python technique is to use range (len(someList)) with a for loop to iterate over the indexes of a list.

```
>>> supplies = ['pens', 'staplers', 'flame-throwers', 'binders']
>>> for i in range(len(supplies)):
    print('Index ' + str(i) + ' in supplies is: ' + supplies[i])

Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flame-throwers
Index 3 in supplies is: binders
```

- The code in the loop will access the index (as the variable i), the value at that index (as supplies[i]) and range(len(supplies)) will iterate through all the indexes of supplies, no matter how many items it contains.

The in and not in Operators

- We can determine whether a value is or isn't in a list with the in and not in operators.
- **in** and **not in** are used in expressions and connect two values: a value to look for in a list and the list where it may be found and these expressions will evaluate to a Boolean value.

```
>>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']
True
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> 'cat' in spam
False
>>> 'howdy' not in spam
False
>>> 'cat' not in spam
True
```

- The following program lets the user type in a pet name and then checks to see whether the name is in a list of pets.

Program

```
myPets = ['Zophie', 'Pooka', 'Fat-tail']
print('Enter a pet name:')
name = input()
if name not in myPets:
    print('I do not have a pet named ' + name)
else:
    print(name + ' is my pet.')
```

Output

```
Enter a pet name:
Footfoot
I do not have a pet named Footfoot
```

The Multiple Assignment Trick

- The multiple assignment trick is a shortcut that lets you assign multiple variables with the values in a list in one line of code.

```
>>> cat = ['fat', 'black', 'loud']
>>> size = cat[0]
>>> color = cat[1]
>>> disposition = cat[2]
```

```
>>> cat = ['fat', 'black', 'loud']
>>> size, color, disposition = cat
```

- Instead of left-side program we could type the right-side program to assignment multiple variables but the number of variables and the length of the list must be exactly equal, or Python will give you a ValueError:

```
>>> cat = ['fat', 'black', 'loud']
>>> size, color, disposition, name = cat
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in <module>
    size, color, disposition, name = cat
ValueError: need more than 3 values to unpack
```

1.3 Augmented Assignment Operators

- When assigning a value to a variable, we will frequently use the variable itself

```
>>> spam = 42
>>> spam = spam + 1
>>> spam
43
```

```
>>> spam = 42
>>> spam += 1
>>> spam
43
```

- Instead of left-side program we could use right-side program i.e., with the augmented assignment operator += to do the same thing as a shortcut.
- The Augmented Assignment Operators are listed in the below table:

Augmented assignment statement	Equivalent assignment statement
spam = spam + 1	spam += 1
spam = spam - 1	spam -= 1
spam = spam * 1	spam *= 1
spam = spam / 1	spam /= 1
spam = spam % 1	spam %= 1

- The += operator can also do string and list concatenation, and the *= operator can do string and list replication.

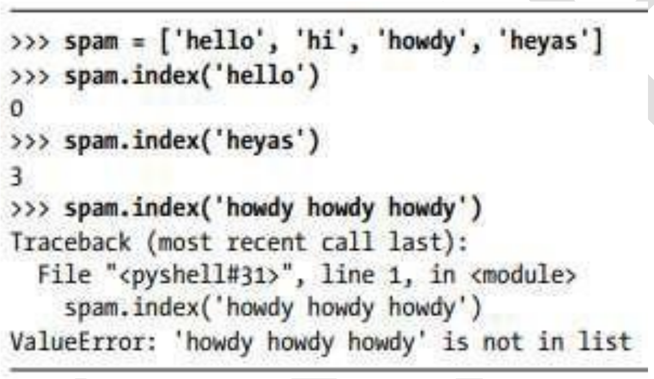
```
>>> spam = 'Hello'
>>> spam += ' world!'
>>> spam
'Hello world!'
>>> bacon = ['Zophie']
>>> bacon *= 3
>>> bacon
['Zophie', 'Zophie', 'Zophie']
```

1.4 Methods

- A method is same as a function, except it is “called on” a value.
- The method part comes after the value, separated by a period.
- Each data type has its own set of methods.
- The list data type has several useful methods for finding, adding, removing, and manipulating values in a list.

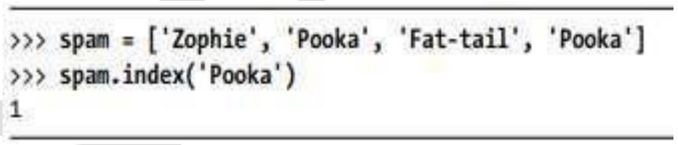
Finding a Value in a List with the index() Method

- List values have an index() method that can be passed a value, and if that value exists in the list, the index of the value is returned. If the value isn’t in the list, then Python produces a ValueError error.



```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> spam.index('hello')
0
>>> spam.index('heyas')
3
>>> spam.index('howdy howdy howdy')
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    spam.index('howdy howdy howdy')
ValueError: 'howdy howdy howdy' is not in list
```

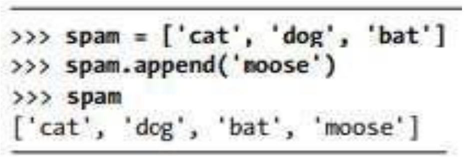
- When there are duplicates of the value in the list, the index of its first appearance is returned.



```
>>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
>>> spam.index('Pooka')
1
```

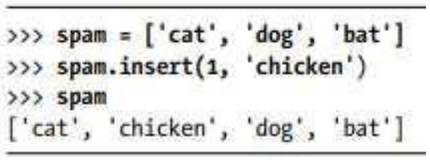
Adding Values to Lists with the append() and insert() Methods

- To add new values to a list, use the append() and insert() methods.
- The append() method call adds the argument to the end of the list.



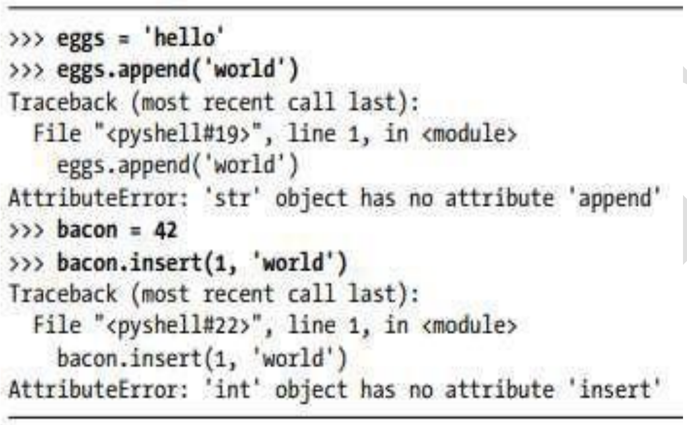
```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.append('moose')
>>> spam
['cat', 'dog', 'bat', 'moose']
```

- The insert() method can insert a value at any index in the list. The first argument to insert() is the index for the new value, and the second argument is the new value to be inserted.



```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.insert(1, 'chicken')
>>> spam
['cat', 'chicken', 'dog', 'bat']
```

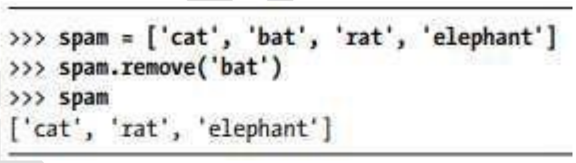
- Methods belong to a single data type.
- The append() and insert() methods are list methods and can be called only on list values, not on other values such as strings or integers.



```
>>> eggs = 'hello'
>>> eggs.append('world')
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    eggs.append('world')
AttributeError: 'str' object has no attribute 'append'
>>> bacon = 42
>>> bacon.insert(1, 'world')
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    bacon.insert(1, 'world')
AttributeError: 'int' object has no attribute 'insert'
```

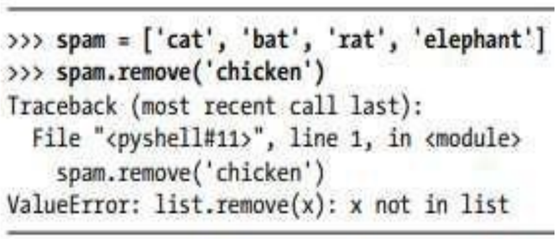
<u>**Removing Values from Lists with remove()**</u>

- The remove() method is passed the value to be removed from the list it is called on.



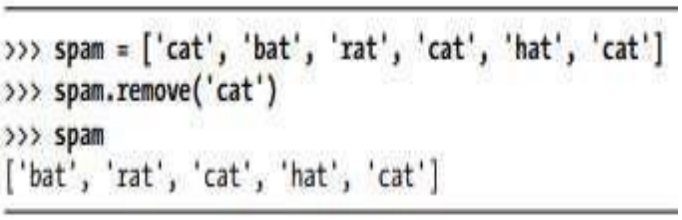
```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('bat')
>>> spam
['cat', 'rat', 'elephant']
```

- Attempting to delete a value that does not exist in the list will result in a ValueError error.



```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('chicken')
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    spam.remove('chicken')
ValueError: list.remove(x): x not in list
```

- If the value appears multiple times in the list, only the first instance of the value will be removed.

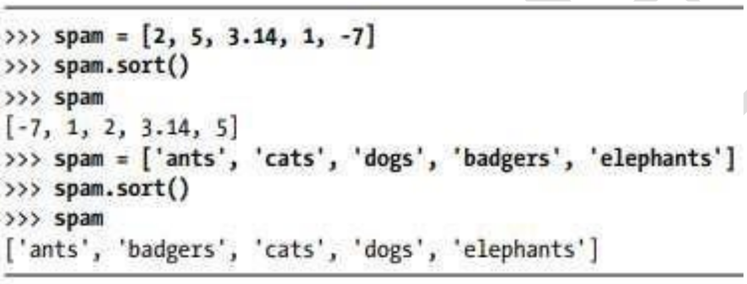


```
>>> spam = ['cat', 'bat', 'rat', 'cat', 'hat', 'cat']
>>> spam.remove('cat')
>>> spam
['bat', 'rat', 'cat', 'hat', 'cat']
```

- The del statement is good to use when you know the index of the value you want to remove from the list. The remove() method is good when you know the value you want to remove from the list.

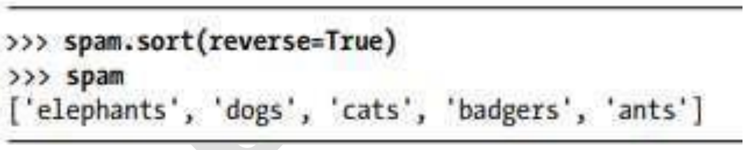
Sorting the Values in a List with the sort() Method

- Lists of number values or lists of strings can be sorted with the sort() method.



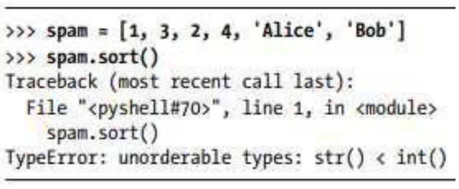
```
>>> spam = [2, 5, 3.14, 1, -7]
>>> spam.sort()
>>> spam
[-7, 1, 2, 3.14, 5]
>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
>>> spam.sort()
>>> spam
['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

- You can also pass True for the reverse keyword argument to have sort() sort the values in reverse order.



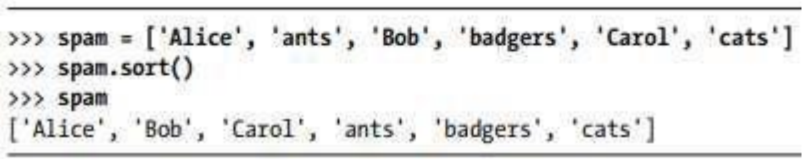
```
>>> spam.sort(reverse=True)
>>> spam
['elephants', 'dogs', 'cats', 'badgers', 'ants']
```

- There are three things you should note about the sort() method.
  - **First,** the sort() method sorts the list in place; don't try to return value by writing code like spam = spam.sort().
  - **Second,** we cannot sort lists that have both number values and string values in them.



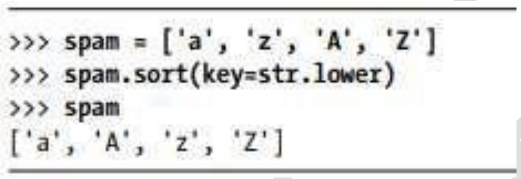
```
>>> spam = [1, 3, 2, 4, 'Alice', 'Bob']
>>> spam.sort()
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    spam.sort()
TypeError: unorderable types: str() < int()
```

○ **Third,** sort() uses "ASCIIbetical order(upper case)" rather than actual alphabetical order(lower case) for sorting strings.



```
>>> spam = ['Alice', 'ants', 'Bob', 'badgers', 'Carol', 'cats']
>>> spam.sort()
>>> spam
['Alice', 'Bob', 'Carol', 'ants', 'badgers', 'cats']
```

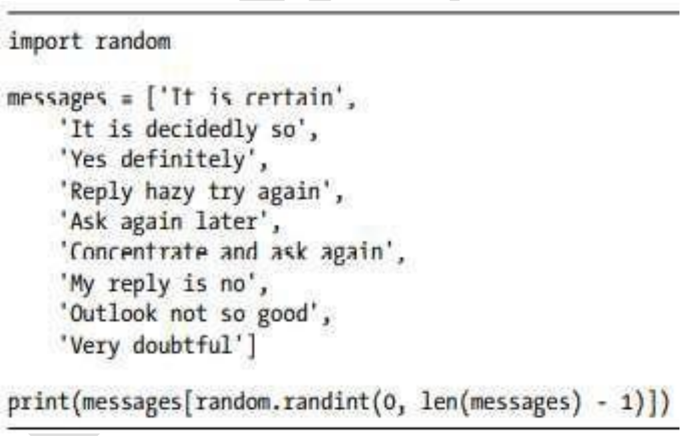
- If we need to sort the values in regular alphabetical order, pass str.lower for the key keyword argument in the sort() method call.



```
>>> spam = ['a', 'z', 'A', 'Z']
>>> spam.sort(key=str.lower)
>>> spam
['a', 'A', 'z', 'Z']
```

1.5 Example Program: Magic 8 Ball with a List

- We can write a much more elegant version of the Magic 8 Ball program. Instead of several lines of nearly identical elif statements, we can create a single list.

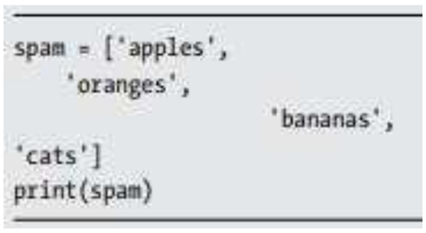


```
import random

messages = ['It is certain',
    'It is decidedly so',
    'Yes definitely',
    'Reply hazy try again',
    'Ask again later',
    'Concentrate and ask again',
    'My reply is no',
    'Outlook not so good',
    'Very doubtful']

print(messages[random.randint(0, len(messages) - 1)])
```

- The expression you use as the index into messages: random .randint (0, len(messages) - 1). This produces a random number to use for the index, regardless of the size of messages. That is, you'll get a random number between 0 and the value of len(messages) - 1.

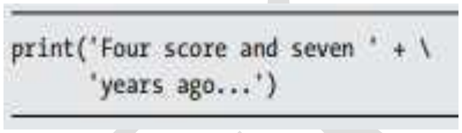
Exceptions to Indentation Rules in Python

- The amount of indentation for a line of code tells Python what block it is in.
- lists can actually span several lines in the source code file. The indentation of these lines do not matter; Python knows that until it sees the ending square bracket, the list is not finished.



```
spam = ['apples',
     'oranges',
                     'bananas',
'cats']
print(spam)
```

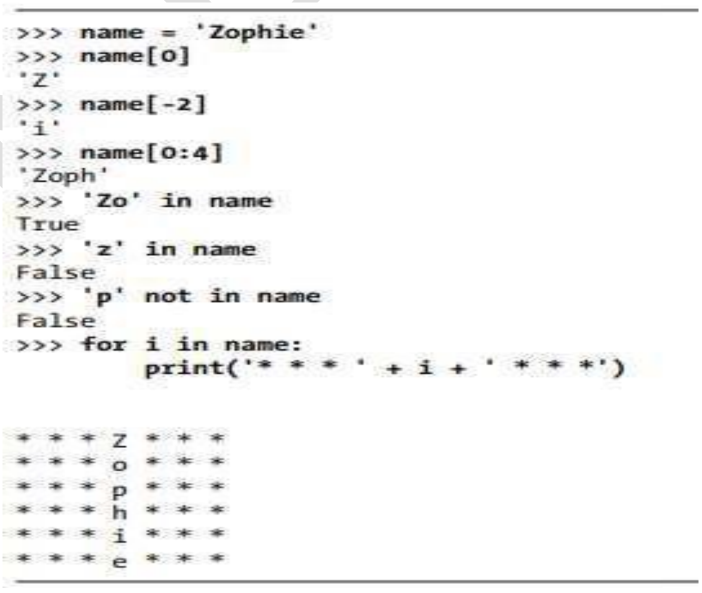
- We can also split up a single instruction across multiple lines using the \ line continuation character at the end.



```
print('Four score and seven ' + \
      'years ago...')
```

1.6 List-like Types: Strings and Tuples

- Lists aren't the only data types that represent ordered sequences of values.
- **Ex,** we can also do these with strings: indexing; slicing; and using them with for loops, with len(), and with the in and not in operators.

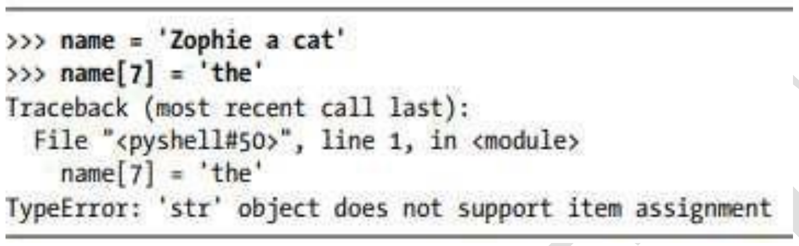


```
>>> name = 'Zophie'
>>> name[0]
'Z'
>>> name[-2]
'i'
>>> name[0:4]
'Zoph'
>>> 'Zo' in name
True
>>> 'z' in name
False
>>> 'p' not in name
False
>>> for i in name:
        print('* * * ' + i + ' * * *')


* * * Z * * *
* * * o * * *
* * * p * * *
* * * h * * *
* * * i * * *
* * * e * * *
```

Mutable and Immutable Data Types

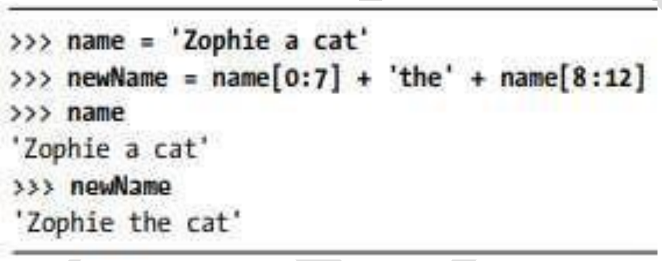
String

- However, a string is immutable: It cannot be changed. Trying to reassign a single character in a string results in a TypeError error.



```
>>> name = 'Zophie a cat'
>>> name[7] = 'the'
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    name[7] = 'the'
TypeError: 'str' object does not support item assignment
```

- The proper way to "mutate" a string is to use slicing and concatenation to build a new string by copying from parts of the old string.

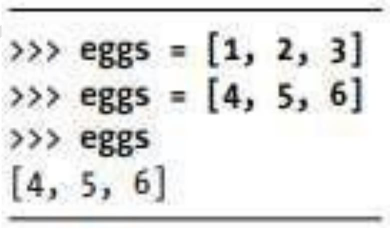


```
>>> name = 'Zophie a cat'
>>> newName = name[0:7] + 'the' + name[8:12]
>>> name
'Zophie a cat'
>>> newName
'Zophie the cat'
```

- We used [0:7] and [8:12] to refer to the characters that we don't wish to replace. Notice that the original 'Zophie a cat' string is not modified because strings are immutable.

List

- A list value is a mutable data type: It can have values added, removed, or changed.



```
>>> eggs = [1, 2, 3]
>>> eggs = [4, 5, 6]
>>> eggs
[4, 5, 6]
```

- The list value in eggs isn't being changed here; rather, an entirely new and different list value ([4, 5, 6]) is overwriting the old list value ([1, 2, 3]).

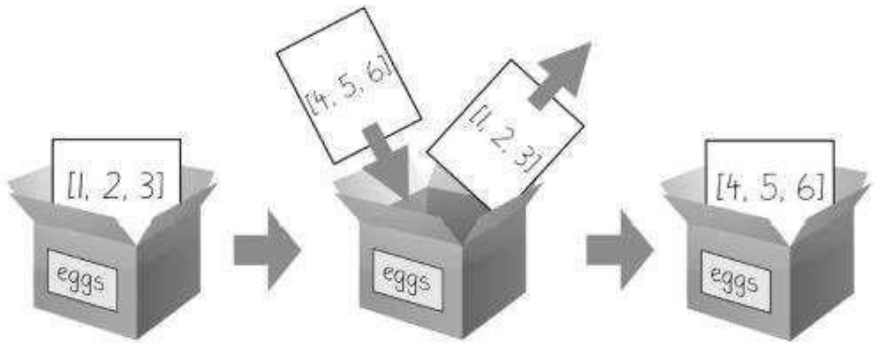
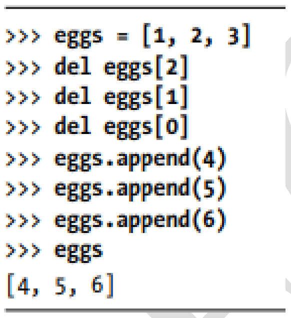


Figure: When eggs = [4, 5, 6] is executed, the contents of eggs are replaced with a new list value.

- If we want to modify the original list in eggs to contain [4, 5, 6], you would have to delete the items in that and then add items to it.



```
>>> eggs = [1, 2, 3]
>>> del eggs[2]
>>> del eggs[1]
>>> del eggs[0]
>>> eggs.append(4)
>>> eggs.append(5)
>>> eggs.append(6)
>>> eggs
[4, 5, 6]
```

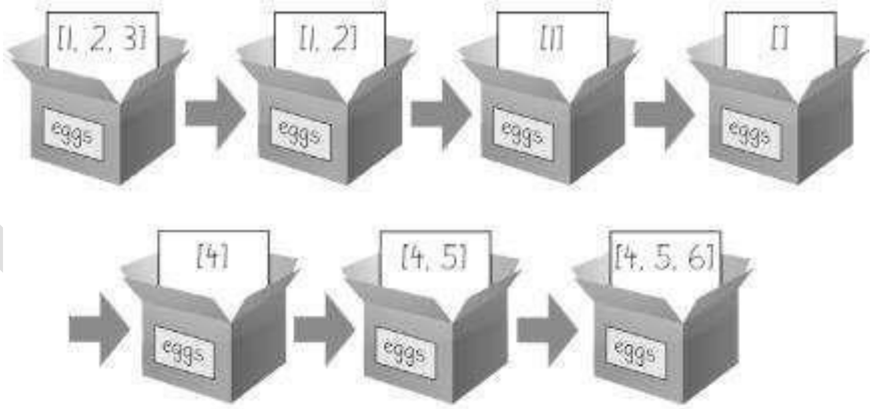
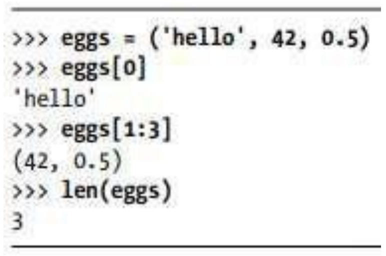


Figure: The del statement and the append() method modify the same list value in place.

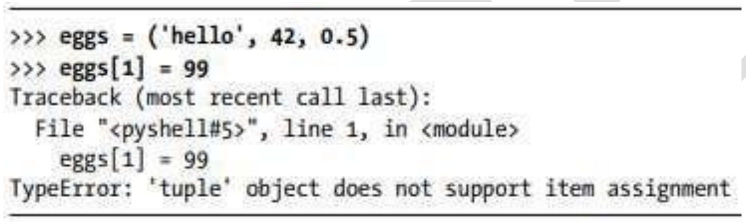
The Tuple Data Type

- The tuple data type is almost identical to the list data type, except in two ways.
- **First**, tuples are typed with parentheses, (and), instead of square brackets, [and].



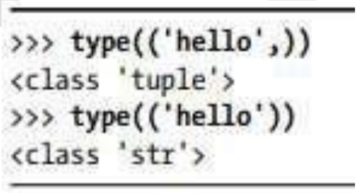
```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[0]
'hello'
>>> eggs[1:3]
(42, 0.5)
>>> len(eggs)
3
```

- **Second**, benefit of using tuples instead of lists is that, because they are immutable and their contents don't change. Tuples cannot have their values modified, appended, or removed.



```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[1] = 99
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    eggs[1] = 99
TypeError: 'tuple' object does not support item assignment
```

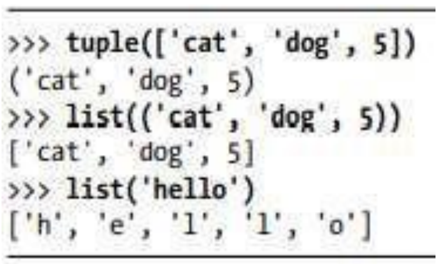
- If you have only one value in your tuple, you can indicate this by placing a trailing comma after the value inside the parentheses.



```
>>> type(('hello',))
<class 'tuple'>
>>> type(('hello'))
<class 'str'>
```

Converting Types with the list() and tuple() Functions

- The functions list() and tuple() will return list and tuple versions of the values passed to them.



```
>>> tuple(['cat', 'dog', 5])
('cat', 'dog', 5)
>>> list(('cat', 'dog', 5))
['cat', 'dog', 5]
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

References

- As , variables store strings and integer values.

```
>>> spam = 42
>>> cheese = spam
>>> spam = 100
>>> spam
100
>>> cheese
42
```

- We assign 42 to the spam variable, and then we copy the value in spam and assign it to the variable cheese. When we later change the value in spam to 100, this doesn't affect the value in cheese. This is because spam and cheese are different variables that store different values.
- But lists works differently. When we assign a list to a variable, we are actually assigning a list reference to the variable. A reference is a value that points to some bit of data, and a list reference is a value that points to a list.

```
❶ >>> spam = [0, 1, 2, 3, 4, 5]
❷ >>> cheese = spam
❸ >>> cheese[1] = 'Hello!'
  >>> spam
  [0, 'Hello!', 2, 3, 4, 5]
  >>> cheese
  [0, 'Hello!', 2, 3, 4, 5]
```

- When we create the list ❶, we assign a reference to it in the spam variable. But the next line copies only the list reference in spam to cheese, not the list value itself. This means the values stored in spam and cheese now both refer to the same list.
- There is only one underlying list because the list itself was never actually copied. So when we modify the first element of cheese, we are modifying the same list that spam refers to.
- List variables don't actually contain lists—they contain references to lists.

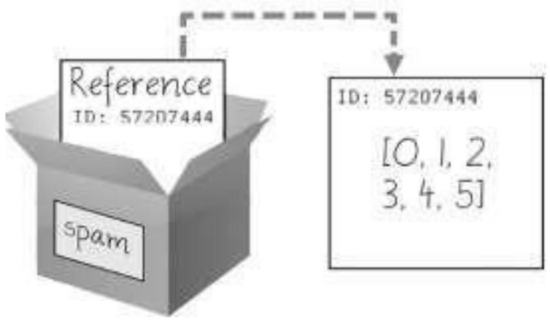


Figure: spam = [0, 1, 2, 3, 4, 5] stores a reference to a list, not the actual list.

- The reference in spam is copied to cheese. Only a new reference was created and stored in cheese, not a new list.

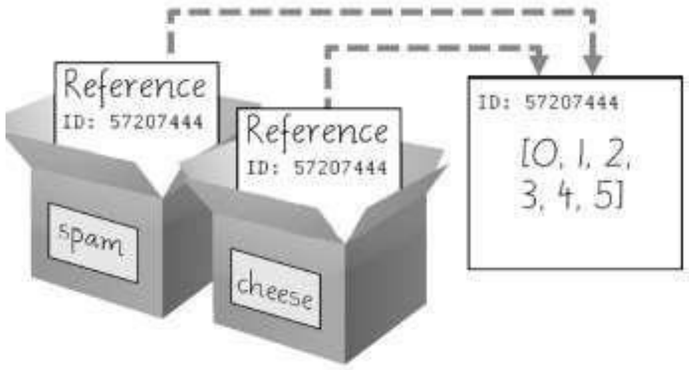


Figure: spam = cheese copies the reference, not the list

- When we alter the list that cheese refers to, the list that spam refers to is also changed, because both cheese and spam refer to the same list.

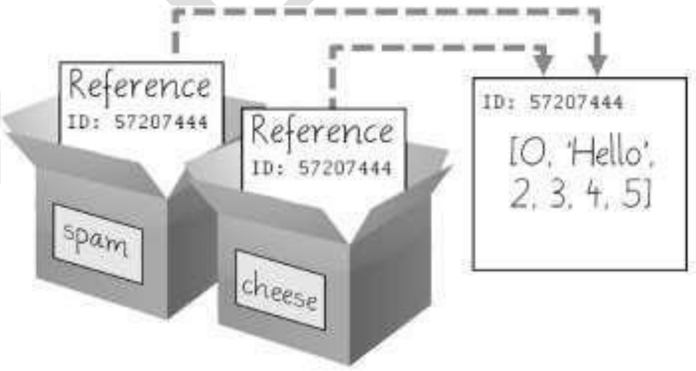


Figure: cheese[1] = 'Hello!' modifies the list that both variables refer to

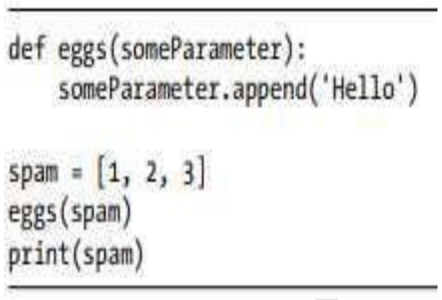
- Variables will contain references to list values rather than list values themselves.

- But for strings and integer values, variables will contain the string or integer value.
- Python uses references whenever variables must store values of mutable data types, such as lists or dictionaries. For values of immutable data types such as strings, integers, or tuples, Python variables will store the value itself.

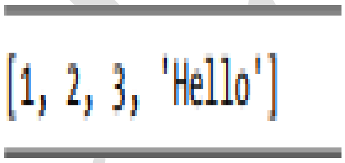
Passing References

- References are particularly important for understanding how arguments get passed to functions.
- When a function is called, the values of the arguments are copied to the parameter variables.

Program **Output**



```
def eggs(someParameter):
    someParameter.append('Hello')

spam = [1, 2, 3]
eggs(spam)
print(spam)
```



```
[1, 2, 3, 'Hello']
```

- when eggs() is called, a return value is not used to assign a new value to spam.
- Even though spam and someParameter contain separate references, they both refer to the same list. This is why the append('Hello') method call inside the function affects the list even after the function call has returned

The copy Module's copy() and deepcopy() Functions

- If the function modifies the list or dictionary that is passed, we may not want these changes in the original list or dictionary value.
- For this, Python provides a module named copy that provides both the copy() and deepcopy() functions.
- **copy(),** can be used to make a duplicate copy of a mutable value like a list or dictionary, not just a copy of a reference.
- Now the spam and cheese variables refer to separate lists, which is why only the list in cheese is modified when you assign 42 at index 1.
- The reference ID numbers are no longer the same for both variables because the variables refer to independent lists

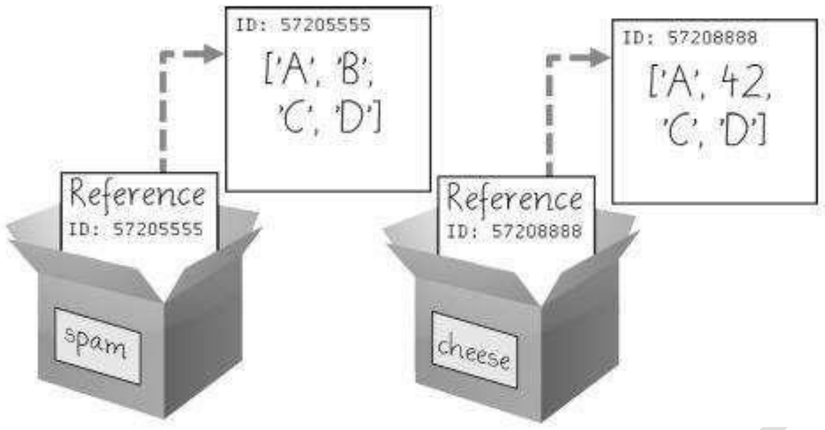


Figure: cheese = copy.copy(spam) creates a second list that can be modified independently of the first.

```
>>> import copy
>>> spam = ['A', 'B', 'C', 'D']
>>> cheese = copy.copy(spam)
>>> cheese[1] = 42
>>> spam
 Output: ['A', 'B', 'C', 'D']
>>> cheese
  Output: ['A', 42, 'C', 'D']
```

copy function

```
>>> import copy
>>> old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
>>> new_list = copy.copy(old_list)
>>> old_list[1][0] = 'BB'
>>> print("Old list:", old_list)
>>> print("New list:", new_list)
>>> print(id(old_list))
>>> print(id(new_list))
```

Output:

```
Old list: [[1, 1, 1],  ['BB', 2, 2],  [3, 3, 3]]
New list: [[1, 1, 1], ['BB', 2, 2],  [3, 3, 3]]
1498111334272
1498110961152
```

deepcopy() Function

```
>>> import copy
>>> old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
>>> new_list = copy.deepcopy(old_list)
>>> old_list[1][0] = 'BB'
>>> print("Old list:", old_list)
>>> print("New list:", new_list)
>>> print(id(old_list))
>>> print(id(new_list))
```

Output

```
Old list:  [[1, 1, 1], ['BB', 2, 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 2, 2],    [3, 3, 3]]
1498111298880
1498111336064
```

CHAPTER2: DICTIONARIES AND STRUCTURING DATA

1. The Dictionary Data Type

2. Pretty Printing

3. Using Data Structures to Model Real-World Things.

2.1 The Dictionary Data Type

- A dictionary is a collection of many values. Indexes for dictionaries can use many different data types, not just integers. Indexes for dictionaries are called keys, and a key with its associated value is called a key-value pair. A dictionary is typed with braces, {}.
- A dictionary is typed with braces, {}.

```
>>> myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

- This assigns a dictionary to the myCat variable. This dictionary's keys are 'size', 'color', and 'disposition'. The values for these keys are 'fat', 'gray', and 'loud', respectively. You can access these values through their keys:

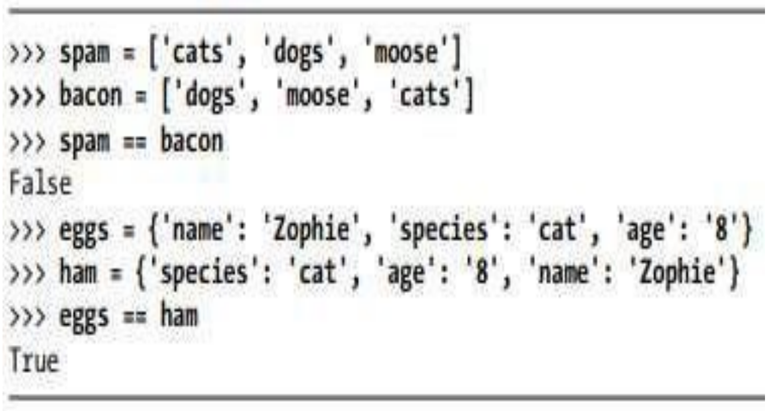
```
>>> myCat['size']
'fat'
>>> 'My cat has ' + myCat['color'] + ' fur.'
'My cat has gray fur.'
```

- Dictionaries can still use integer values as keys, but they do not have to start at 0 and can be any number.

```
>>> spam = {12345: 'Luggage Combination', 42: 'The Answer'}
```

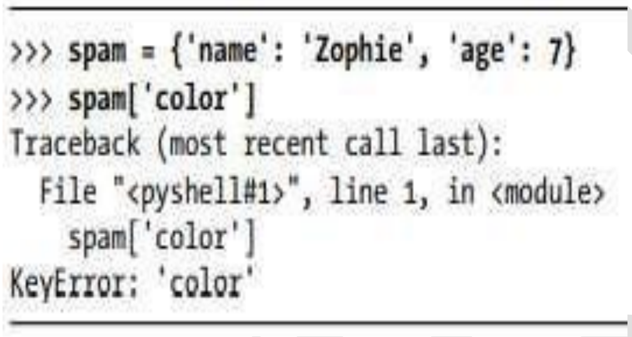
<u>Dictionaries vs. Lists</u>

- Unlike lists, items in dictionaries are unordered.
- The first item in a list named spam would be spam[0]. But there is no "first" item in a dictionary. While the order of items matters for determining whether two lists are the same, it does not matter in what order the key-value pairs are typed in a dictionary.



```
>>> spam = ['cats', 'dogs', 'moose']
>>> bacon = ['dogs', 'moose', 'cats']
>>> spam == bacon
False
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> eggs == ham
True
```

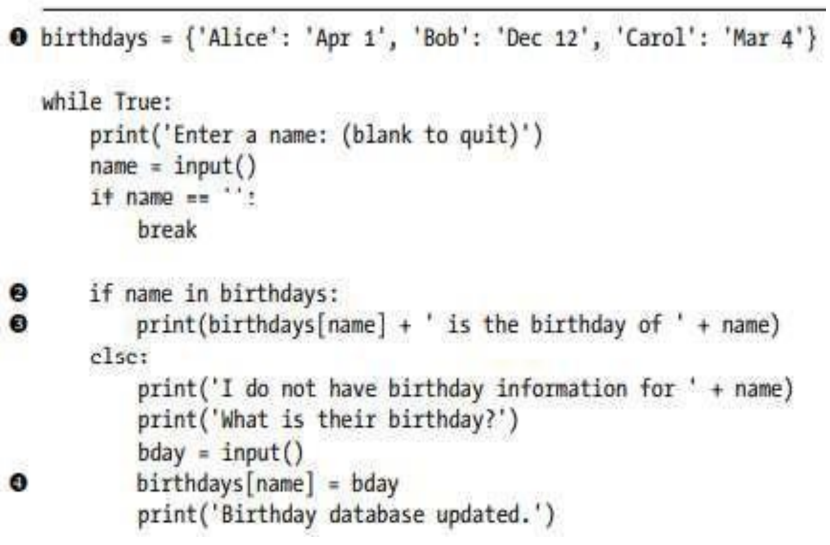
- Trying to access a key that does not exist in a dictionary will result in a KeyError error message, much like a list's "out-of-range" IndexError error message.



```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> spam['color']
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    spam['color']
KeyError: 'color'
```

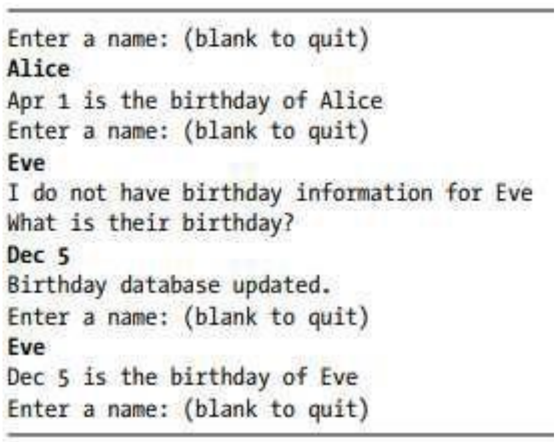
- We can have arbitrary values for the keys that allows us to organize our data in powerful ways.
- **Ex:** we want to store data about our friends' birthdays. We can use a dictionary with the names as keys and the birthdays as values.

Program:



```
❶ birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}

  while True:
      print('Enter a name: (blank to quit)')
      name = input()
      if name == '':
          break

❷     if name in birthdays:
❸         print(birthdays[name] + ' is the birthday of ' + name)
      else:
          print('I do not have birthday information for ' + name)
          print('What is their birthday?')
          bday = input()
❹         birthdays[name] = bday
          print('Birthday database updated.')
```

Output:

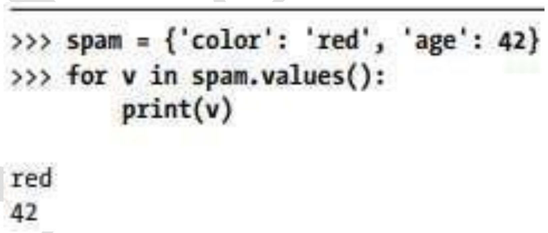


```
Enter a name: (blank to quit)
Alice
Apr 1 is the birthday of Alice
Enter a name: (blank to quit)
Eve
I do not have birthday information for Eve
What is their birthday?
Dec 5
Birthday database updated.
Enter a name: (blank to quit)
Eve
Dec 5 is the birthday of Eve
Enter a name: (blank to quit)
```

- We create an initial dictionary and store it in birthdays **1**.
- We can see if the entered name exists as a key in the dictionary with the in keyword **2**.
- If the name is in the dictionary, we access the associated value using square brackets **3**; if not, we can add it using the same square bracket syntax combined with the assignment operator **4**.

The keys(), values(), and items() Methods

- There are three dictionary methods that will return list-like values of the dictionary's keys, values, or both keys and values: keys(), values(), and items().
- Data types (dict_keys, dict_values, and dict_items, respectively) can be used in for loops



```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
        print(v)

red
42
```

- A for loop can iterate over the keys, values, or key-value pairs in a dictionary by using keys(), values(), and items() methods.
- The values in the dict_items value returned by the items() method are tuples of the key and value.

```
>>> for k in spam.keys():
        print(k)

color
age
>>> for i in spam.items():
        print(i)

('color', 'red')
('age', 42)
```

- If we want a true list from one of these methods, pass its list-like return value to the list() function.

```
>>> spam = {'color': 'red', 'age': 42}
>>> spam.keys()
dict_keys(['color', 'age'])
>>> list(spam.keys())
['color', 'age']
```

- The list(spam.keys()) line takes the dict_keys value returned from keys() and passes it to list(), which then returns a list value of ['color', 'age'].
- We can also use the multiple assignment trick in a for loop to assign the key and value to separate variables.

```
>>> spam = {'color': 'red', 'age': 42}
>>> for k, v in spam.items():
        print('Key: ' + k + ' Value: ' + str(v))

Key: age Value: 42
Key: color Value: red
```

Checking Whether a Key or Value Exists in a Dictionary

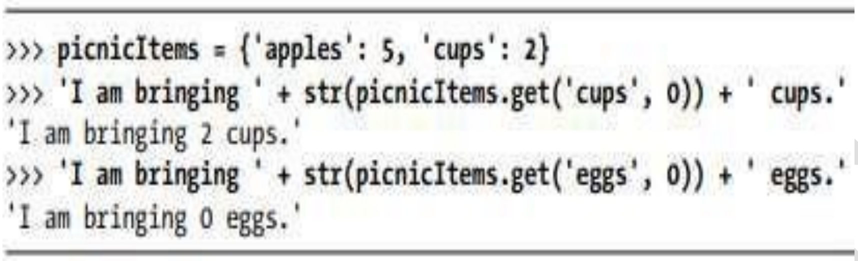
- We can use the **in** and **not in** operators to see whether a certain key or value exists in a dictionary

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> 'name' in spam.keys()
True
>>> 'Zophie' in spam.values()
True
>>> 'color' in spam.keys()
False
>>> 'color' not in spam.keys()
True
>>> 'color' in spam
False
```

<u>The get() Method</u>

Dictionaries have a get() method that takes two arguments:

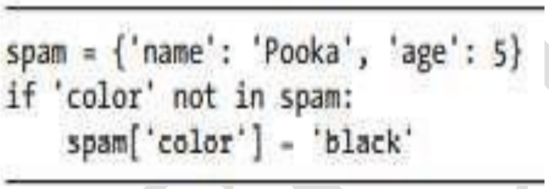
- The key of the value to retrieve and
- A fallback value to return if that key does not exist.



```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'
'I am bringing 2 cups.'
>>> 'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'
'I am bringing 0 eggs.'
```

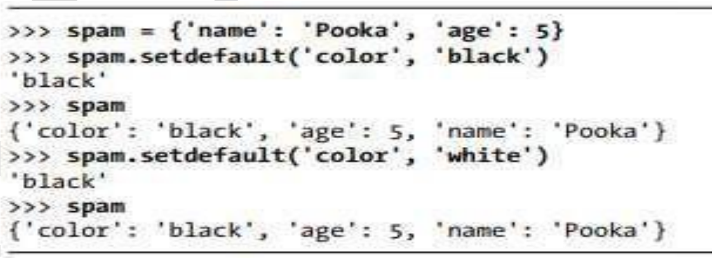
<u>The setdefault() Method</u>

- To set a value in a dictionary for a certain key only if that key does not already have a value



```
spam = {'name': 'Pooka', 'age': 5}
if 'color' not in spam:
    spam['color'] = 'black'
```

- The setdefault() method offers a way to do this in one line of code.
- Setdeafault() takes 2 arguments:
    - The first argument is the key to check for, and
    - The second argument is the value to set at that key if the key does not exist. If the key does exist, the setdefault() method returns the key's value.



```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
>>> spam.setdefault('color', 'white')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

- The first time setdefault() is called, the dictionary in spam changes to {'color': 'black', 'age': 5, 'name': 'Pooka'}. The method returns the value 'black' because this is now the value set for the key 'color'. When spam.setdefault('color', 'white') is called next, the value for that key is not changed to 'white' because spam already has a key named 'color'.

Ex: program that counts the number of occurrences of each letter in a string.

```
message = 'It was a bright cold day in April, and the clocks were striking thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

print(count)
```

- ➢ The program loops over each character in the message variable’s string, counting how often each character appears.
- ➢ The setdefault() method call ensures that the key is in the count dictionary (with a default value of 0), so the program doesn’t throw a KeyError error when count[character] = count[character] + 1 is executed.

Output:

```
{' ': 13, ',': 1, '.': 1, 'A': 1, 'I': 1, 'a': 4, 'c': 3, 'b': 1, 'e': 5, 'd': 3, 'g': 2, 'i':
6, 'h': 3, 'k': 2, 'l': 3, 'o': 2, 'n': 4, 'p': 1, 's': 3, 'r': 5, 't': 6, 'w': 2, 'y': 1}
```

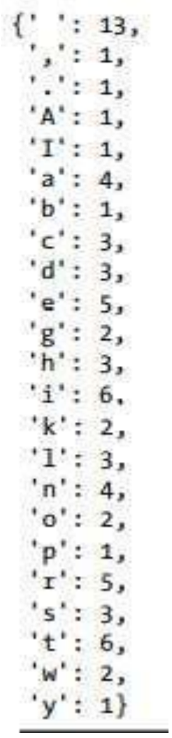
2.2 Pretty Printing

- ➢ Importing pprint module will provide access to the pprint() and pformat() functions that will “pretty print” a dictionary’s values.
- ➢ This is helpful when we want a cleaner display of the items in a dictionary than what print() provides and also it is helpful when the dictionary itself contains nested lists or dictionaries..

Program: counts the number of occurrences of each letter in a string.

```
import pprint
message = 'It was a bright cold day in April, and the clocks were striking
thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

pprint.pprint(count)
```

Output:



```
{' ': 13,
 ',': 1,
 '.': 1,
 'A': 1,
 'I': 1,
 'a': 4,
 'b': 1,
 'c': 3,
 'd': 3,
 'e': 5,
 'g': 2,
 'h': 3,
 'i': 6,
 'k': 2,
 'l': 3,
 'n': 4,
 'o': 2,
 'p': 1,
 'r': 5,
 's': 3,
 't': 6,
 'w': 2,
 'y': 1}
```

2.3 Using Data Structures to Model Real-World Things

A Tic-Tac-Toe Board

- A tic-tac-toe board looks like a large hash symbol (#) with nine slots that can each contain an X, an O, or a blank. To represent the board with a dictionary, we can assign each slot a string-value key as shown in below figure.

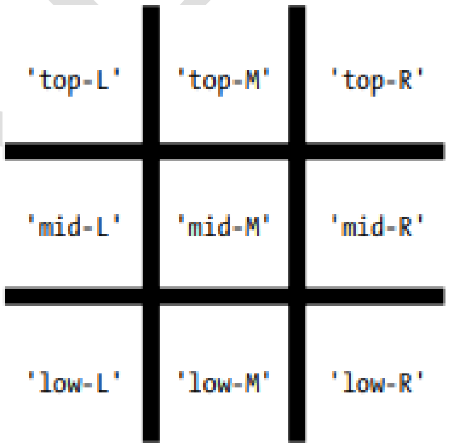
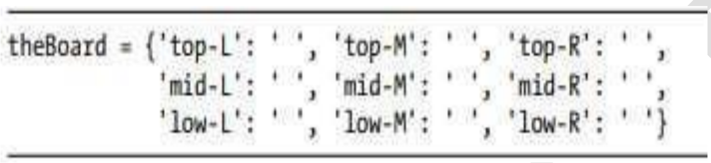


Figure: The slots of a tic-tactoe board with their corresponding keys

- We can use string values to represent what's in each slot on the board: 'X', 'O', or ' ' (a space character).
- To store nine strings. We can use a dictionary of values for this.
  - The string value with the key 'top-R' can represent the top-right corner,
  - The string value with the key 'low-L' can represent the bottom-left corner,
  - The string value with the key 'mid-M' can represent the middle, and so on.
- Store this board-as-a-dictionary in a variable named theBoard.



```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

- The data structure stored in the theBoard variable represents the tic-tactoe board in the below Figure.

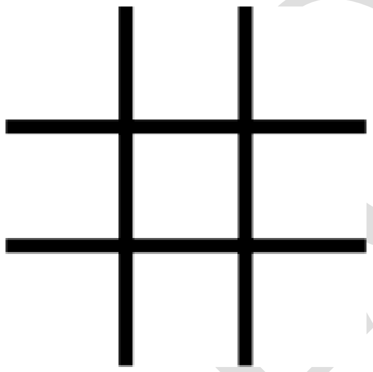


Figure: An empty tic-tac-toe board

- Since the value for every key in theBoard is a single-space string, this dictionary represents a completely clear board. If player X went first and chose the middle space, you could represent that board with this dictionary as shown below:

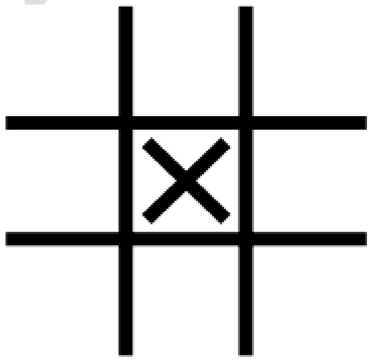


Figure: A first move

- A board where player O has won by placing Os across the top might look like this:

```python
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O',
            'mid-L': 'X', 'mid-M': 'X', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}
```

➢ The data structure in theBoard now represents tic-tac-toe board in the below Figure.

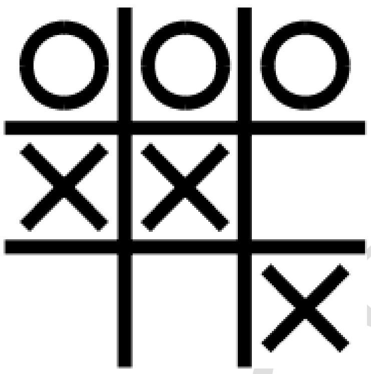


Figure: Player O wins.

➢ The player sees only what is printed to the screen, not the contents of variables.

➢ The tic-tac-toe program is updated as below.

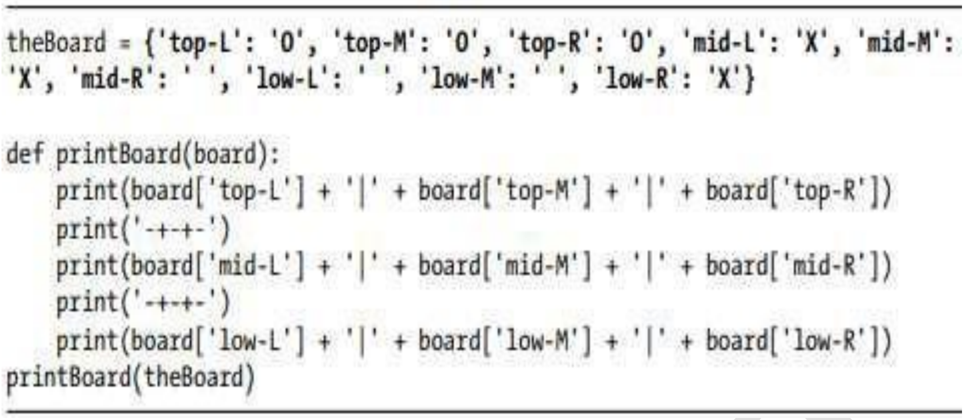
```python
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
printBoard(theBoard)
```

Output:

```
 | |
-+-+-
 | |
-+-+-
 | |
```

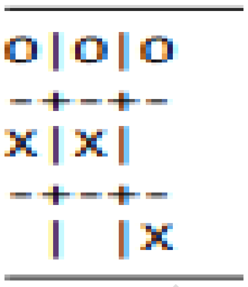
➢ The printBoard() function can handle any tic-tac-toe data structure you pass it.

Program



```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O', 'mid-L': 'X', 'mid-M':
'X', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}

def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
printBoard(theBoard)
```

Output:



```
O|O|O
-+-+-
X|X|
-+-+-
 | |X
```

- Now we created a data structure to represent a tic-tac-toe board and wrote code in printBoard() to interpret that data structure, we now have a program that "models" the tic-tac-toe board.
- **Program:** allows the players to enter their moves.

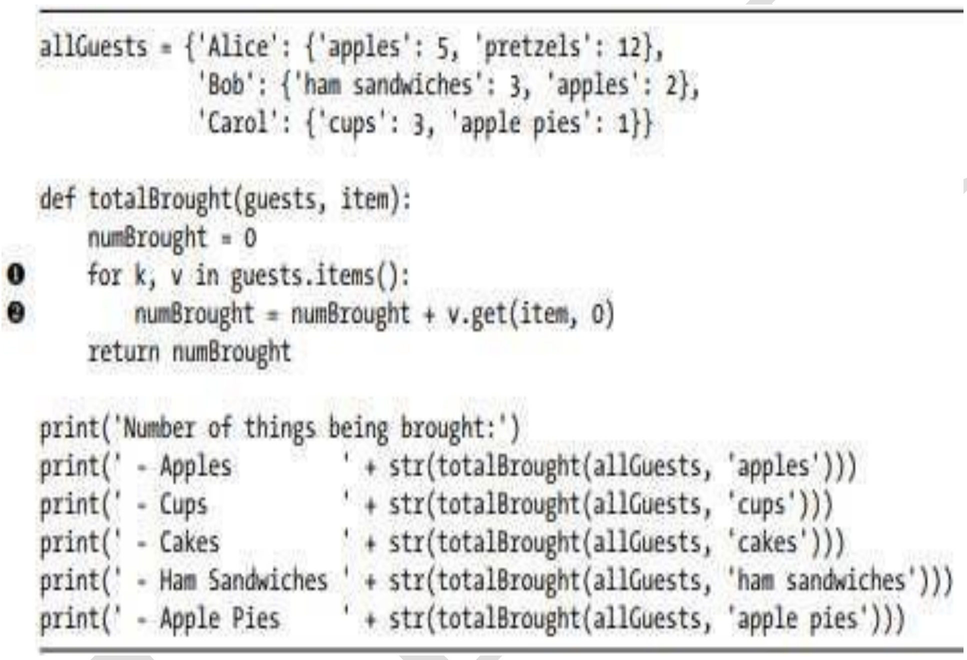
```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ', 'mid-L': ' ', 'mid-M': '
', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': ' '}

def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
turn = 'X'
for i in range(9):
❶    printBoard(theBoard)
     print('Turn for ' + turn + '. Move on which space?')
❷    move = input()
❸    theBoard[move] = turn
❹    if turn == 'X':
         turn = 'O'
     else:
         turn = 'X'
printBoard(theBoard)
```

Output:

```
 | |
-+-+-
 | |
-+-+-
 | |
Turn for X. Move on which space?
mid-M
 | |
-+-+-
 |X|
-+-+-
 | |
Turn for O. Move on which space?
low-L
 | |
-+-+-
 |X|
-+-+-
O| |

--snip--

O|O|X
-+-+-
X|X|O
-+-+-
O| |X
Turn for X. Move on which space?
low-M
O|O|X
-+-+-
X|X|O
-+-+-
O|X|X
```

Nested Dictionaries and Lists

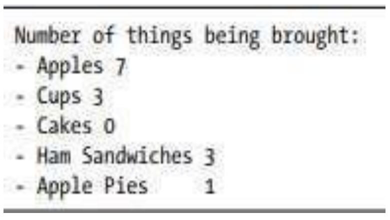
- We can have program that contains dictionaries and lists which in turn contain other dictionaries and lists.
- Lists are useful to contain an ordered series of values, and dictionaries are useful for associating keys with values.
- **Program:** which contains nested dictionaries in order to see who is bringing what to a picnic



```
allGuests = {'Alice': {'apples': 5, 'pretzels': 12},
             'Bob': {'ham sandwiches': 3, 'apples': 2},
             'Carol': {'cups': 3, 'apple pies': 1}}

def totalBrought(guests, item):
    numBrought = 0
❶   for k, v in guests.items():
❷       numBrought = numBrought + v.get(item, 0)
    return numBrought

print('Number of things being brought:')
print(' - Apples         ' + str(totalBrought(allGuests, 'apples')))
print(' - Cups           ' + str(totalBrought(allGuests, 'cups')))
print(' - Cakes          ' + str(totalBrought(allGuests, 'cakes')))
print(' - Ham Sandwiches ' + str(totalBrought(allGuests, 'ham sandwiches')))
print(' - Apple Pies     ' + str(totalBrought(allGuests, 'apple pies')))
```
.

- Inside the totalBrought() function, the for loop iterates over the keyvalue pairs in guests **1.**
- Inside the loop, the string of the guest's name is assigned to k, and the dictionary of picnic items they're bringing is assigned to v.
- If the item parameter exists as a key in this dictionary, it's value (the quantity) is added to numBrought **2**.
- If it does not exist as a key, the get() method returns 0 to be added to numBrought.

Output:



```
Number of things being brought:
- Apples 7
- Cups 3
- Cakes 0
- Ham Sandwiches 3
- Apple Pies     1
```

CHAPTER 1
MANIPULATING STRINGS

1. Working with Strings
2. Useful String Methods
3. Project: Password Locker
4. Project: Adding Bullets to Wiki Markup

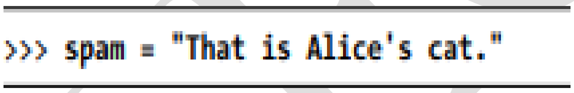
1.1 Working with strings

String Literals

- String values begin and end with a single quote.
- But we want to use either double or single quotes within a string then we have a multiple ways to do it as shown below.

Double Quotes

- One benefit of using double quotes is that the string can have a single quote character in it.

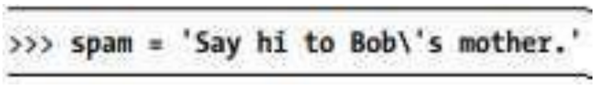


```
>>> spam = "That is Alice's cat."
```

- Since the string begins with a double quote, Python knows that the single quote is part of the string and not marking the end of the string.

Escape Characters

- If you need to use both single quotes and double quotes in the string, you’ll need to use escape characters.
- An escape character consists of a backslash (\) followed by the character you want to add to the string.



```
>>> spam = 'Say hi to Bob\'s mother.'
```

- Python knows that the single quote in Bob\'s has a backslash, it is not a single quote meant to end the string value. The escape characters \' and \" allows to put single quotes and double quotes inside your strings, respectively.

Ex:

```
>>> print("Hello there!\nHow are you?\nI\'m doing fine.")
Hello there!
How are you?
I'm doing fine.
```

- The different special escape characters can be used in a program as listed below in a table.

Escape character	Prints as
\'	Single quote
\"	Double quote
\t	Tab
\n	Newline (line break)
\\	Backslash

Raw Strings

- You can place an r before the beginning quotation mark of a string to make it a raw string. A raw string completely ignores all escape characters and prints any backslash that appears in the string

```
>>> print(r'That is Carol\'s cat.')
That is Carol\'s cat.
```

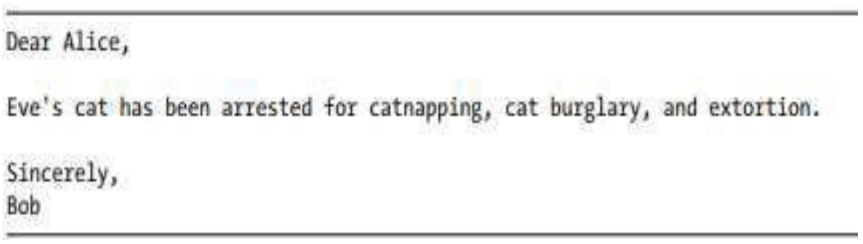
Multiline Strings with Triple Quotes

- A multiline string in Python begins and ends with either three single quotes or three double quotes.
- Any quotes, tabs, or newlines in between the “triple quotes” are considered part of the string.

Program

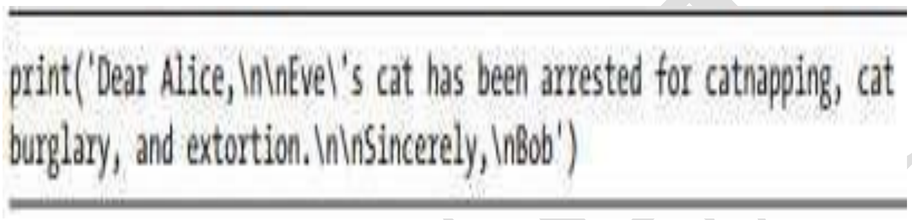
```
print('''Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob''')
```

<u>Output</u>



```
Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob
```

➢ The following print() call would print identical text but doesn't use a multiline string.

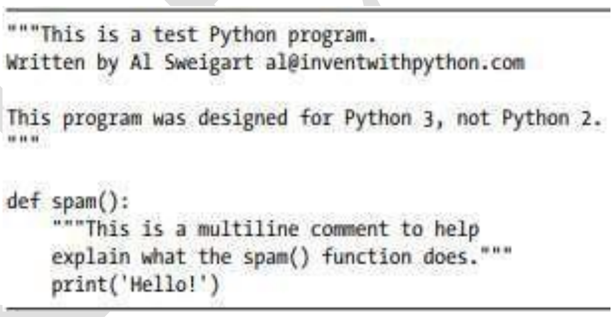


```
print('Dear Alice,\n\nEve\'s cat has been arrested for catnapping, cat
burglary, and extortion.\n\nSincerely,\nBob')
```

<u>Multiline Comments</u>

➢ While the hash character (#) marks the beginning of a comment for the rest of the line.

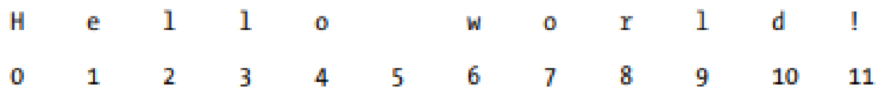
➢ A multiline string is often used for comments that span multiple lines.



```
"""This is a test Python program.
Written by Al Sweigart al@inventwithpython.com

This program was designed for Python 3, not Python 2.
"""

def spam():
    """This is a multiline comment to help
    explain what the spam() function does."""
    print('Hello!')
```

<u>Indexing and Slicing Strings</u>

➢ Strings use indexes and slices the same way lists do. We can think of the string 'Hello world!' as a list and each character in the string as an item with a corresponding index.



```
H   e   l   l   o       w   o   r   l   d   !
0   1   2   3   4   5   6   7   8   9   10  11
```

- The space and exclamation point are included in the character count, so 'Hello world!' is 12 characters long.
- If we specify an index, you'll get the character at that position in the string.

```
>>> spam = 'Hello world!'
>>> spam[0]
'H'
>>> spam[4]
'o'
>>> spam[-1]
'!'
>>> spam[0:5]
'Hello'
>>> spam[:5]
'Hello'
>>> spam[6:]
'world!'
```

- If we specify a range from one index to another, the starting index is included and the ending index is not.

```
>>> spam = 'Hello world!'
>>> fizz = spam[0:5]
>>> fizz
'Hello'
```

- The substring we get from spam[0:5] will include everything from spam[0] to spam[4], leaving out the space at index 5.

The in and not in Operators with Strings

- The **in** and **not in** operators can be used with strings just like with list values.
- An expression with two strings joined using in or not in will evaluate to a Boolean True or False.

```
>>> 'Hello' in 'Hello World'
True
>>> 'Hello' in 'Hello'
True
>>> 'HELLO' in 'Hello World'
False
>>> '' in 'spam'
True
>>> 'cats' not in 'cats and dogs'
False
```

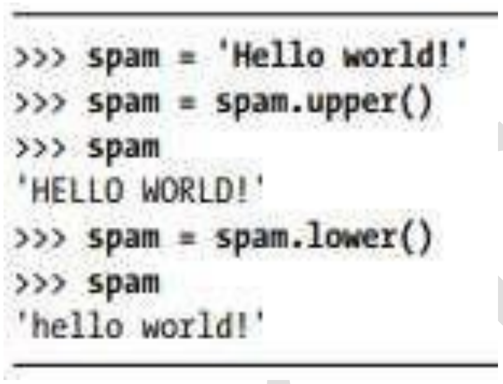
- These expressions test whether the first string (the exact string, case sensitive) can be found within the second string.

1.2 Useful String Methods

- Several string methods analyze strings or create transformed string values.

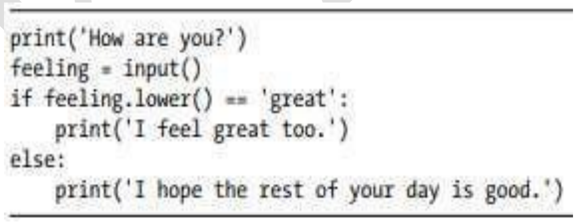
The upper(), lower(), isupper(), and islower() String Methods

- The upper() and lower() string methods return a new string where all the letters in the original string have been converted to uppercase or lowercase, respectively.

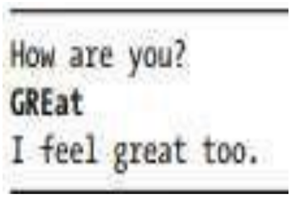


```
>>> spam = 'Hello world!'
>>> spam = spam.upper()
>>> spam
'HELLO WORLD!'
>>> spam = spam.lower()
>>> spam
'hello world!'
```

- These methods do not change the string itself but return new string values.
- If we want to change the original string, we have to call upper() or lower() on the string and then assign the new string to the variable where the original was stored.
- The upper() and lower() methods are helpful if we need to make a case-insensitive comparison.
- In the following small program, it does not matter whether the user types Great, GREAT, or grEAT, because the string is first converted to lowercase.



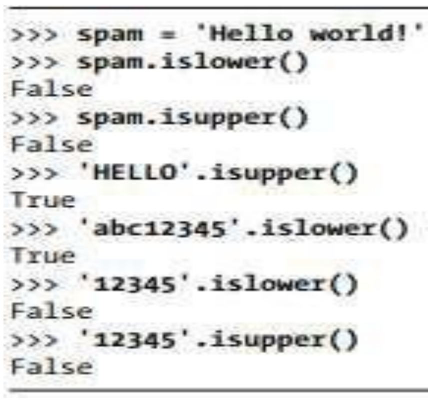
```
print('How are you?')
feeling = input()
if feeling.lower() == 'great':
    print('I feel great too.')
else:
    print('I hope the rest of your day is good.')
```



```
How are you?
GREat
I feel great too.
```

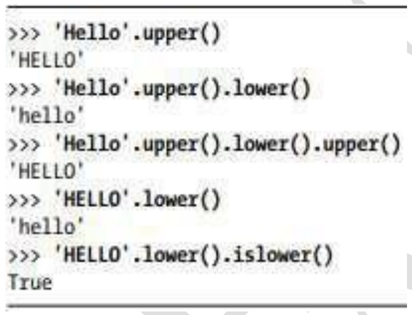
Program **Output**

- The isupper() and islower() methods will return a Boolean True value if the string has at least one letter and all the letters are uppercase or lowercase, respectively. Otherwise, the method returns False.



```
>>> spam = 'Hello world!'
>>> spam.islower()
False
>>> spam.isupper()
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
>>> '12345'.isupper()
False
```

- Since the upper() and lower() string methods themselves return strings, you can call string methods on those returned string values as well. Expressions that do this will look like a chain of method calls.



```
>>> 'Hello'.upper()
'HELLO'
>>> 'Hello'.upper().lower()
'hello'
>>> 'Hello'.upper().lower().upper()
'HELLO'
>>> 'HELLO'.lower()
'hello'
>>> 'HELLO'.lower().islower()
True
```

The isX String Methods

- There are several string methods that have names beginning with the word is. These methods return a Boolean value that describes the nature of the string.
- Here are some common isX string methods:
  - **isalpha()** returns True if the string consists only of letters and is not blank.
  - **isalnum()** returns True if the string consists only of letters and numbers and is not blank.
  - **isdecimal()** returns True if the string consists only of numeric characters and is not blank.
  - **isspace()** returns True if the string consists only of spaces, tabs, and newlines and is not blank.
  - **istitle()** returns True if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

```
>>> 'hello'.isalpha()
True
>>> 'hello123'.isalpha()
False
>>> 'hello123'.isalnum()
True
>>> 'hello'.isalnum()
True
>>> '123'.isdecimal()
True
>>> '    '.isspace()
True
>>> 'This Is Title Case'.istitle()
True
>>> 'This Is Title Case 123'.istitle()
True
>>> 'This Is not Title Case'.istitle()
False
>>> 'This Is NOT Title Case Either'.istitle()
False
```

- The isX string methods are helpful when you need to validate user input.
- For example, the following program repeatedly asks users for their age and a password until they provide valid input.

Program

```
while True:
    print('Enter your age:')
    age = input()
    if age.isdecimal():
        break
    print('Please enter a number for your age.')

while True:
    print('Select a new password (letters and numbers only):')
    password = input()
    if password.isalnum():
        break
    print('Passwords can only have letters and numbers.')
```

output

```
Enter your age:
forty two
Please enter a number for your age.
Enter your age:
42
Select a new password (letters and numbers only):
secr3t!
Passwords can only have letters and numbers.
Select a new password (letters and numbers only):
secr3t
```

The startswith() and endswith() String Methods

- The startswith() and endswith() methods return True if the string value they are called on begins or ends (respectively) with the string passed to the method; otherwise, they return False.

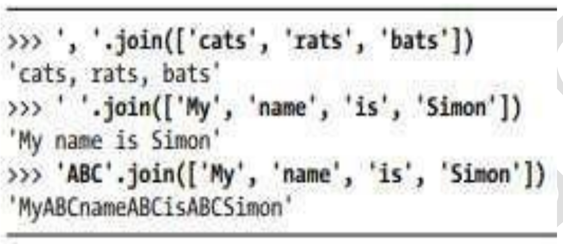
```
>>> 'Hello world!'.startswith('Hello')
True
>>> 'Hello world!'.endswith('world!')
True
>>> 'abc123'.startswith('abcdef')
False
>>> 'abc123'.endswith('12')
False
>>> 'Hello world!'.startswith('Hello world!')
True
>>> 'Hello world!'.endswith('Hello world!')
True
```

- These methods are useful alternatives to the == equals operator if we need to check only whether the first or last part of the string, rather than the whole thing, is equal to another string.

The join() and split() String Methods

Join()

- The join() method is useful when we have a list of strings that need to be joined together into a single string value.
- The join() method is called on a string, gets passed a list of strings, and returns a string. The returned string is the concatenation of each string in the passed-in list.

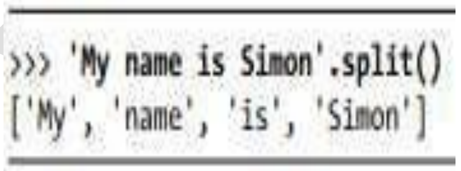


```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

- string join() calls on is inserted between each string of the list argument.
  - **Ex:** when join(['cats', 'rats', 'bats']) is called on the ', ' string, the returned string is 'cats, rats, bats'.
  - join() is called on a string value and is passed a list value.

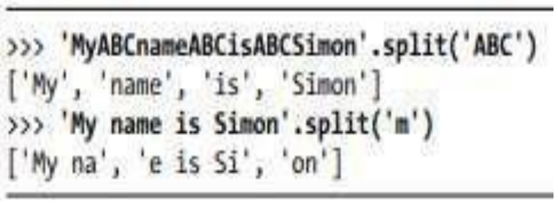
Split()

- The split() method is called on a string value and returns a list of strings.



```
>>> 'My name is Simon'.split()
['My', 'name', 'is', 'Simon']
```

- We can pass a delimiter string to the split() method to specify a different string to split upon.



```
>>> 'MyABCnameABCisABCSimon'.split('ABC')
['My', 'name', 'is', 'Simon']
>>> 'My name is Simon'.split('m')
['My na', 'e is Si', 'on']
```

- common use of split() is to split a multiline string along the newline characters.

```
>>> spam = '''Dear Alice,
How have you been? I am fine.
There is a container in the fridge
that is labeled "Milk Experiment".

Please do not drink it.
Sincerely,
Bob'''
>>> spam.split('\n')
['Dear Alice,', 'How have you been? I am fine.', 'There is a container in the
fridge', 'that is labeled "Milk Experiment".', '', 'Please do not drink it.',
'Sincerely,', 'Bob']
```

- Passing split() the argument '\n' lets us split the multiline string stored in spam along the newlines and return a list in which each item corresponds to one line of the string.

Justifying Text with rjust(), ljust(), and center()

- The rjust() and ljust() string methods return a padded version of the string they are called on, with spaces inserted to justify the text.
- The **first** argument to both methods is an integer length for the justified string.

```
>>> 'Hello'.rjust(10)
'     Hello'
>>> 'Hello'.rjust(20)
'               Hello'
>>> 'Hello World'.rjust(20)
'         Hello World'
>>> 'Hello'.ljust(10)
'Hello     '
```

- 'Hello'.rjust(10) says that we want to right-justify 'Hello' in a string of total length 10. 'Hello' is five characters, so five spaces will be added to its left, giving us a string of 10 characters with 'Hello' justified right.
- An optional **second** argument to rjust() and ljust() will specify a fill character other than a space character.

```
>>> 'Hello'.rjust(20, '*')
'***************Hello'
>>> 'Hello'.ljust(20, '-')
'Hello---------------'
```

- The center() string method works like ljust() and rjust() but centers the text rather than justifying it to the left or right.

```
>>> 'Hello'.center(20)
'       Hello        '
>>> 'Hello'.center(20, '=')
'=======Hello========'
```

- These methods are especially useful when you need to print tabular data that has the correct spacing.
- In the below program, we define a printPicnic() method that will take in a dictionary of information and use center(), ljust(), and rjust() to display that information in a neatly aligned table-like format.
    - The dictionary that we'll pass to printPicnic() is picnicItems.
    - In picnicItems, we have 4 sandwiches, 12 apples, 4 cups, and 8000 cookies. We want to organize this information into two columns, with the name of the item on the left and the quantity on the right.

Program

```
def printPicnic(itemsDict, leftWidth, rightWidth):
    print('PICNIC ITEMS'.center(leftWidth + rightWidth, '-'))
    for k, v in itemsDict.items():
        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))
picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4, 'cookies': 8000}
printPicnic(picnicItems, 12, 5)
printPicnic(picnicItems, 20, 6)
```

output

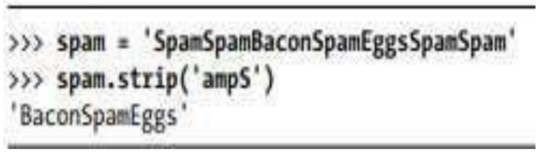
```
---PICNIC ITEMS--
sandwiches..    4
apples......   12
cups........    4
cookies..... 8000
-------PICNIC ITEMS-------
sandwiches..........     4
apples..............    12
cups................     4
cookies.............  8000
```

Removing Whitespace with strip(), rstrip(), and lstrip()

- The strip() string method will return a new string without any whitespace characters at the beginning or end.
- The lstrip() and rstrip() methods will remove whitespace characters from the left and right ends, respectively.

```
>>> spam = '    Hello World    '
>>> spam.strip()
'Hello World'
>>> spam.lstrip()
'Hello World    '
>>> spam.rstrip()
'    Hello World'
```

- Optionally, a string argument will specify which characters on the ends should be stripped.

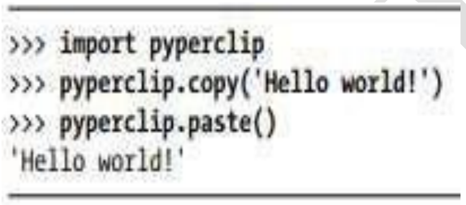


```
>>> spam = 'SpamSpamBaconSpamEggsSpamSpam'
>>> spam.strip('ampS')
'BaconSpamEggs'
```

- Passing strip() the argument 'ampS' will tell it to strip occurences of a, m, p, and capital S from the ends of the string stored in spam.
- The order of the characters in the string passed to strip() does not matter: strip('ampS') will do the same thing as strip('mapS') or strip('Spam').

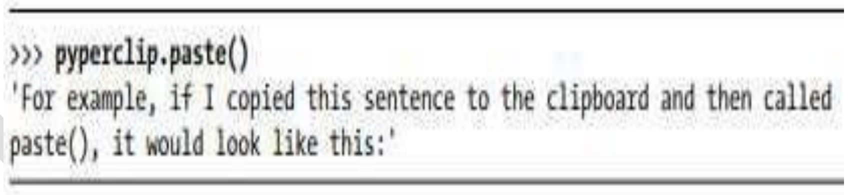
Copying and Pasting Strings with the pyperclip Module

- The pyperclip module has copy() and paste() functions that can send text to and receive text from your computer's clipboard.



```
>>> import pyperclip
>>> pyperclip.copy('Hello world!')
>>> pyperclip.paste()
'Hello world!'
```

- Of course, if something outside of your program changes the clipboard contents, the paste() function will return it.



```
>>> pyperclip.paste()
'For example, if I copied this sentence to the clipboard and then called
paste(), it would look like this:'
```

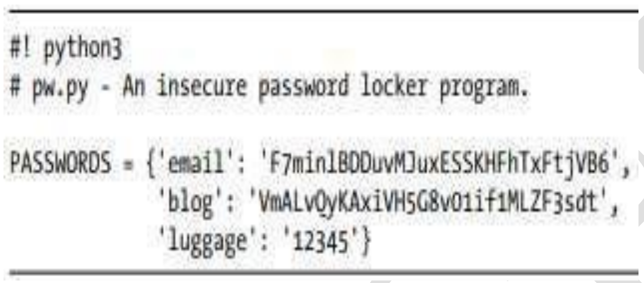
1.3 Project: Password Locker

- We probably have accounts on many different websites.
- It's a bad habit to use the same password for each of them because if any of those sites has a security breach, the hackers will learn the password to all of your other accounts.
- It's best to use password manager software on your computer that uses one master password to unlock the password manager.
- Then you can copy any account password to the clipboard and paste it into the website's Password field

- The password manager program you'll create in this example isn't secure, but it offers a basic demonstration of how such programs work.

Step 1: Program Design and Data Structures

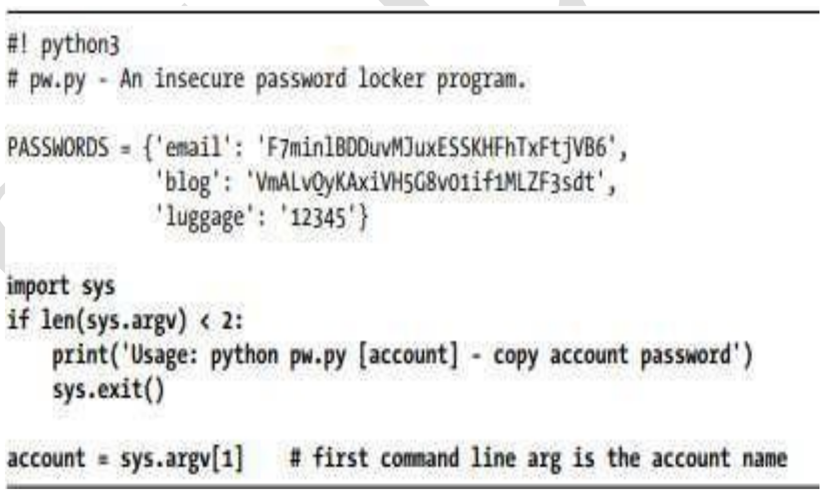
- We have to run this program with a command line argument that is the account’s name--for instance, email or blog. That account’s password will be copied to the clipboard so that the user can paste it into a Password field. The user can have long, complicated passwords without having to memorize them.
- We need to start the program with a #! (shebang) line and should also write a comment that briefly describes the program. Since we want to associate each account’s name with its password, we can store these as strings in a dictionary.



```
#! python3
# pw.py - An insecure password locker program.

PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sdt',
             'luggage': '12345'}
```

Step 2: Handle Command Line Arguments

- The command line arguments will be stored in the variable sys.argv.
- The **first** item in the sys.argv list should always be a string containing the program’s filename ('pw.py'), and the **second** item should be the first command line argument.



```
#! python3
# pw.py - An insecure password locker program.

PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sdt',
             'luggage': '12345'}

import sys
if len(sys.argv) < 2:
    print('Usage: python pw.py [account] - copy account password')
    sys.exit()

account = sys.argv[1]    # first command line arg is the account name
```

Step 3: Copy the Right Password

- The account name is stored as a string in the variable account, you need to see whether it exists in the PASSWORDS dictionary as a key. If so, you want to copy the key’s value to the clipboard using pyperclip.copy().

```
#! python3
# pw.py - An insecure password locker program.
PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sdt',
             'luggage': '12345'}

import sys, pyperclip
if len(sys.argv) < 2:
    print('Usage: py pw.py [account] - copy account password')
    sys.exit()

account = sys.argv[1]    # first command line arg is the account name

if account in PASSWORDS:
    pyperclip.copy(PASSWORDS[account])
    print('Password for ' + account + ' copied to clipboard.')
else:
    print('There is no account named ' + account)
```

- This new code looks in the PASSWORDS dictionary for the account name. If the account name is a key in the dictionary, we get the value corresponding to that key, copy it to the clipboard, and print a message saying that we copied the value. Otherwise, we print a message saying there's no account with that name.
- On Windows, you can create a batch file to run this program with the win-R Run window. Type the following into the file editor and save the file as pw.bat in the C:\Windows folder:

```
@py.exe C:\Python34\pw.py %*
@pause
```

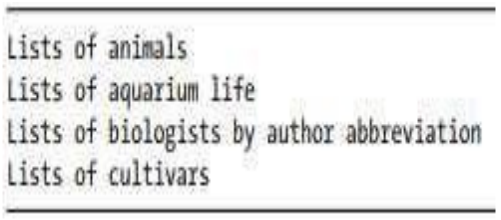
- With this batch file created, running the password-safe program on Windows is just a matter of pressing win-R and typing pw <account name>.

1.4 <u>Project: Adding Bullets to Wiki Markup</u>

- When editing a Wikipedia article, we can create a bulleted list by putting each list item on its own line and placing a star in front.
- But say we have a really large list that we want to add bullet points to. We could just type those stars at the beginning of each line, one by one. Or we could automate this task with a short Python script.
- The bulletPointAdder.py script will get the text from the clipboard, add a star and space to the beginning of each line, and then paste this new text to the clipboard.

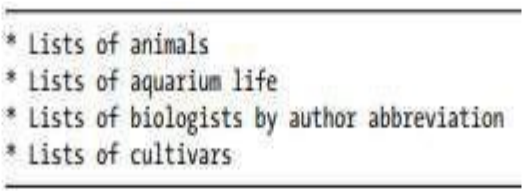
- **Ex:**

Program



```
Lists of animals
Lists of aquarium life
Lists of biologists by author abbreviation
Lists of cultivars
```

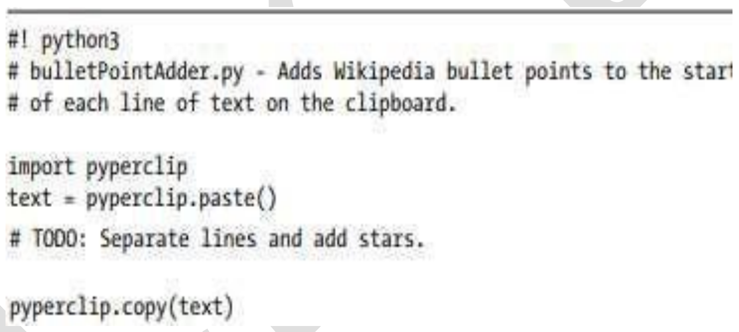
output



```
* Lists of animals
* Lists of aquarium life
* Lists of biologists by author abbreviation
* Lists of cultivars
```

Step 1: Copy and Paste from the Clipboard

- You want the bulletPointAdder.py program to do the following:
  1. Paste text from the clipboard
  2. Do something to it
  3. Copy the new text to the clipboard
- Steps 1 and 3 are pretty straightforward and involve the pyperclip.copy() and pyperclip.paste() functions. saving the following program as bulletPointAdder.py:



```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the star
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()
# TODO: Separate lines and add stars.

pyperclip.copy(text)
```

Step 2: Separate the Lines of Text and Add the Star

- The call to pyperclip.paste() returns all the text on the clipboard as one big string. If we used the "List of Lists of Lists" example, the string stored in text.
- The \n newline characters in this string cause it to be displayed with multiple lines when it is printed or pasted from the clipboard.
- We could write code that searches for each \n newline character in the string and then adds the star just after that. But it would be easier to use the split() method to return a list of strings, one for each line in the original string, and then add the star to the front of each string in the list.

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()

# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)):    # loop through all indexes in the "lines" list
    lines[i] = '* ' + lines[i] # add star to each string in "lines" list

pyperclip.copy(text)
```

- We split the text along its newlines to get a list in which each item is one line of the text. For each line, we add a star and a space to the start of the line. Now each string in lines begins with a star

Step 3: Join the Modified Lines

- The lines list now contains modified lines that start with stars.
- pyperclip.copy() is expecting a single string value, not a list of string values. To make this single string value, pass lines into the join() method to get a single string joined from the list's strings.

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()

# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)):    # loop through all indexes for "lines" list
    lines[i] = '* ' + lines[i] # add star to each string in "lines" list
text = '\n'.join(lines)
pyperclip.copy(text)
```

- When this program is run, it replaces the text on the clipboard with text that has stars at the start of each line.

CHAPTER -2 READ ING AND WRITING FI LES

1. Files and File Paths

- A file has two key properties: a filename (usually written as one word) and a path.
- The part of the filename after the last period is called the file's extension and tells you a file's type. project.docx is a Word document, and Users, asweigart, and Documents all refer to folders
- Folders can contain files and other folders. For example, project.docx s in the Documents folder, which is inside the asweigart folder, which is inside the Users folder.

1.1 Backslash on Windows and Forward Slash on OS X and Linux

- On Windows, paths are written using backslashes (\) as the separator between folder names. OS X and Linux, however, use the forward slash (/) as their path separator.
- Fortunately, this is simple to do with the os.path.join() function. If you os.path.join() will return a string with a file path using the correct path separators.

```
>>> import os
>>> os.path.join('usr', 'bin', 'spam')
'usr\\bin\\spam'
```

- The os.path.join() function is helpful if you need to create strings for filenames.

Program:

```
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in myFiles:
print(os.path.join('C:\\Users\\asweigart', filename))
C:\Users\asweigart\accounts.txt
C:\Users\asweigart\details.csv
C:\Users\asweigart\invite.docx
```

1.2 The Current Working Directory

- Every program that runs on your computer has a *current working directory*, or *cwd*
- Any filenames or paths that do not begin with the root folder are assumed to be under the current working directory.
- can get the current working directory as a string value with the os.getcwd() function and change it with os.chdir().

```
>>> import os
>>> os.getcwd()
'C:\\Python34'
>>> os.chdir('C:\\Windows\\System32')
>>> os.getcwd()
'C:\\Windows\\System32'
```

- The current working directory is set to C:\Python34, so the filename project.docx refers to C:\Python34\project.docx.
- When we change the current working directory to C:\Windows, project.docx is interpreted as C:\Windows\project.docx.
- Python will display an error if you try to change to a directory that does not exist.

```
>>> os.chdir('C:\\ThisFolderDoesNotExist')
Traceback (most recent call last):
File "<pyshell#18>", line 1, in <module>
os.chdir('C:\\ThisFolderDoesNotExist')
FileNotFoundError: [WinError 2] The system cannot find the file specified:
'C:\\ThisFolderDoesNotExist'
```

1.3 <u>Absolute vs. Relative Paths</u>

There are two ways to specify a file path.

- An absolute path, which always begins with the root folder
- A relative path, which is relative to the program’s current working directory

- There are also the *dot* (.) and *dot-dot* (..) folders. These are not real folders but special names that can be used in a path
- A single period ("dot") for a folder name is shorthand for "this directory." Two periods ("dot-dot") means "the parent folder."
- When the current working directory is set to *C:\bacon*, the relative paths for the other folders and files are set as they are in the figure.

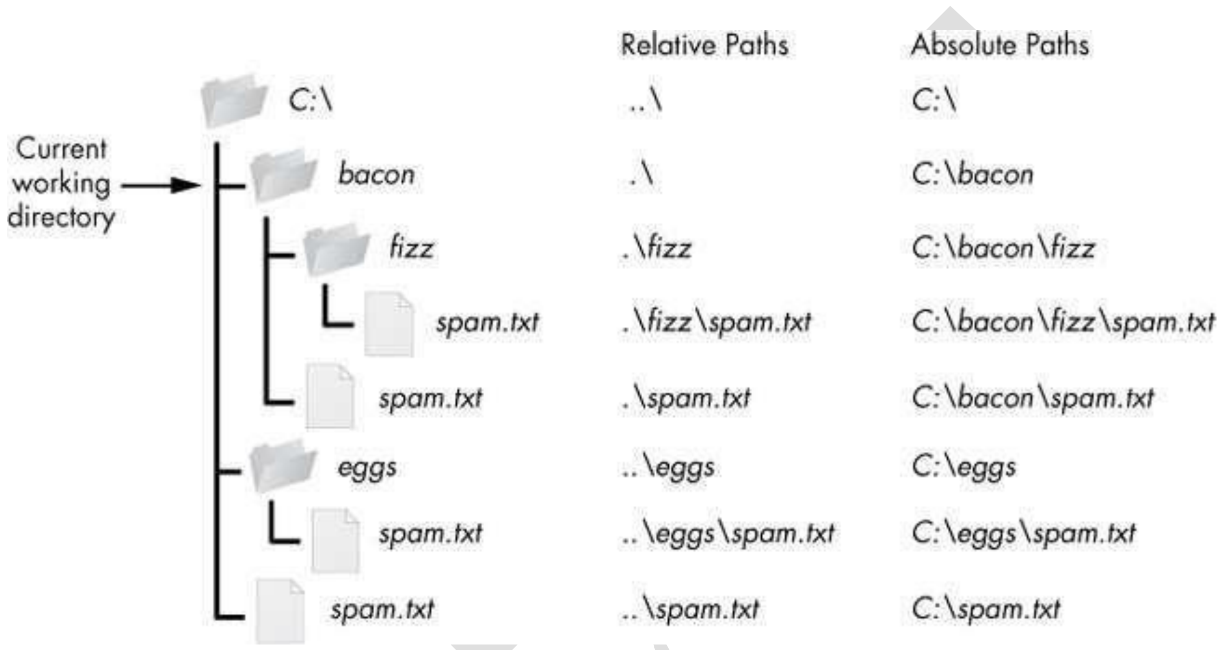


Figure 8-2: The relative paths for folders and files in the working directory C:\bacon

- The .\ at the start of a relative path is optional. For example, *.\spam.txt* and *spam.txt* refer to the same file.

1.4 Creating New Folders with os.makedirs()

- Your programs can create new folders (directories) with the os.makedirs() function.

```
>>> import os
>>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

- This will create not just the C:\delicious folder but also a walnut folder inside C:\delicious and a waffles folder inside C:\delicious\walnut.

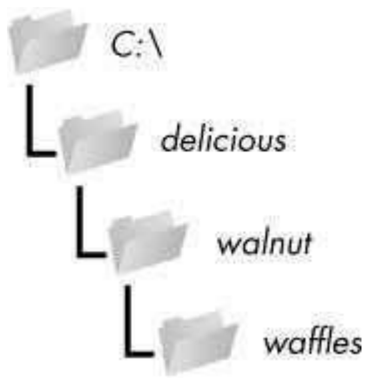


Figure : The result of os.makedirs('C:\\delicious \\walnut\\waffles')

2. The os.path Module

- The os.path module contains many helpful functions related to filenames and file paths
- Since os.path is a module inside the os module, you can import it by simply running import os.

2.1 Handling Absolute and Relative Paths

- The os.path module provides functions for returning the absolute path of a relative path and for checking whether a given path is an absolute path.
- Calling os.path.abspath(path) will return a string of the absolute path of the argument. This is an easy way to convert a relative path into an absolute one.
- Calling os.path.isabs(path) will return True if the argument is an absolute path and False if it is a relative path
- Calling os.path.relpath(path, start) will return a string of a relative path from the start path to path. If start is not provided, the current working directory is used as the start path.

```
>>> os.path.abspath('.')
'C:\\Python34'
>>> os.path.abspath('.\\Scripts')
'C:\\Python34\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
```

- Enter the following calls to os.path.relpath() into the interactive shell

Program:

```
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
>>> os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
'..\\..\\Windows'
>>> os.getcwd()
    'C:\\Python34'
```

- Calling os.path.dirname(*path*) will return a string of everything that comes before the last slash in the path argument
- Calling os.path.basename(*path*) will return a string of everything that comes after the last slash in the path argument.

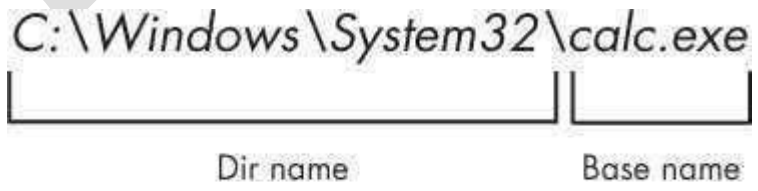


Figure: The base name follows the last slash in a path and is the same as the filename. The dirname is everything before the last slash.

For example

```
>>> path = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.basename(path)
'calc.exe'
>>> os.path.dirname(path)
'C:\\Windows\\System32'
```

- If you need a path's dir name and base name together, you can just call os.path.split() to get a tuple value with these two strings

Program:

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.split(calcFilePath)
('C:\\Windows\\System32', 'calc.exe')
```

- you could create the same tuple by calling os.path.dirname() and os.path.basename() and placing their return values in a tuple.

```
>>> (os.path.dirname(calcFilePath), os.path.basename(calcFilePath))
('C:\\Windows\\System32', 'calc.exe')
```

- But os.path.split() is a nice shortcut if you need both values.
- os.path.split() does *not* take a file path and return a list of strings of each folder. For that, use the split() string method and split on the string in os.sep.

For example,

```
>>> calcFilePath.split(os.path.sep)
['C:', 'Windows', 'System32', 'calc.exe']
```

- On OS X and Linux systems, there will be a blank string at the start of the returned list:

```
>>> '/usr/bin'.split(os.path.sep)
['', 'usr', 'bin']
```

2.2 Finding File Sizes and Folder Contents

- folders. The os.path module provides functions for finding the size of a file in bytes and the files and folders inside a given folder
- Calling os.path.getsize(*path*) will return the size in bytes of the file in the *path* argument.
- Calling os.listdir(*path*) will return a list of filename strings for each file in the *path* argument. (Note that this function is in the os module, not os.path.)

Program:

```
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
776192
>>> os.listdir('C:\\Windows\\System32')
['0409', '12520437.cpx', '12520850.cpx', '5U877.ax', 'aaclient.dll',
--snip--
'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

- If you want to find the total size of all the files in this directory, I can use os.path.getsize() and os.listdir() together.

Program:

```
>>> totalSize = 0
>>> for filename in os.listdir('C:\\Windows\\System32'):
totalSize = totalSize + os.path.getsize(os.path.join('C:\\Windows\\System32',
filename))
>>> print(totalSize)
1117846456
```

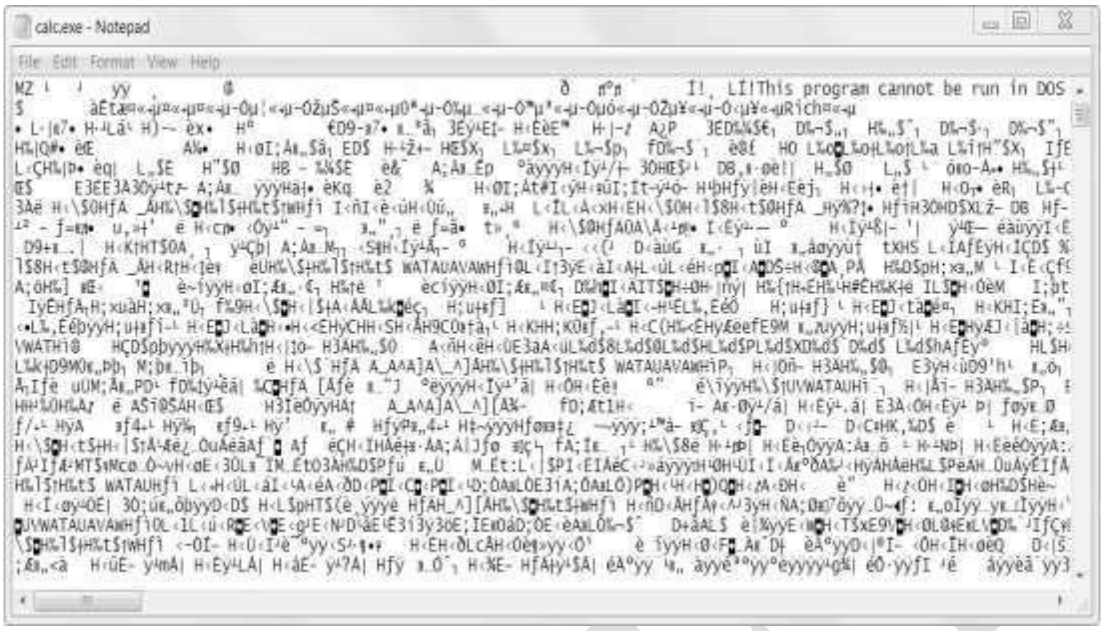
2.3 Checking Path Validity

- The os.path module provides functions to check whether a given path exists and whether it is a file or folder.
- Calling os.path.exists(*path*) will return True if the file or folder referred to in the argument exists and will return False if it does not exist.
- Calling os.path.isfile(*path*) will return True if the path argument exists and is a file and will return False otherwise.
- Calling os.path.isdir(*path*) will return True if the path argument exists and is a folder and will return False otherwise.

```
>>> os.path.exists('C:\\Windows')
True
>>> os.path.exists('C:\\some_made_up_folder')
False
>>> os.path.isdir('C:\\Windows\\System32')
True
>>> os.path.isfile('C:\\Windows\\System32')
False
>>> os.path.isdir('C:\\Windows\\System32\\calc.exe')
False
>>> os.path.isfile('C:\\Windows\\System32\\calc.exe')
True
```

3. The File Reading/Writing Process

- *Plaintext files* contain only basic text characters and do not include font, size, or color information.
- Text files with the *.txt* extension or Python script files with the *.py* extension are examples of plaintext files.
- These can be opened with Windows’s Notepad or OS X’s TextEdit application.
- *Binary files* are all other file types, such as word processing documents, PDFs, images, spreadsheets, and executable programs.
- If you open a binary file in Notepad or TextEdit, it will look like scrambled



```
MZ                                                     !This program cannot be run in DOS
```

Figure: The Windows calc.exe program opened in Notepad

- Since every different type of binary file must be handled in its own way, this book will not go into reading and writing raw binary files directly.
- There are three steps to reading or writing files in Python.
  1. Call the open() function to return a File object.
  2. Call the read() or write() method on the File object.
  3. Close the file by calling the close() method on the File object.

3.1 Opening Files with the open() Function

- To open a file with the open() function, you pass it a string path indicating the file you want to open; it can be either an absolute or relative path.
- The open() function returns a File object.
- Try it by creating a text file named *hello.txt* using Notepad or TextEdit. Type Hello world! as the content of this text file and save it in your user home folder.

```
>>> helloFile = open('C:\\Users\\your_home_folder\\hello.txt')
```

- If you're using OS X, enter the following into the interactive shell instead:

```
>>> helloFile = open('/Users/your_home_folder/hello.txt')
```

- When a file is opened in read mode, Python lets you only read data from the file; you can't write or modify it in any way.

- Read mode is the default mode for files you open in Python.
- if you don't want to rely on Python's defaults, you can explicitly specify the mode by passing the string value 'r' as a second argument to open().
- open('/Users/asweigart/ hello.txt', 'r') and open('/Users/asweigart/hello.txt')

3.2 Reading the Contents of Files

- If you want to read the entire contents of a file as a string value, use the File object's read() method

```
>>> helloContent = helloFile.read()
>>> helloContent
'Hello world!'
```

- Alternatively, you can use the readlines() method to get a *list* of string values from the file, one string for each line of text.
- For example, create a file named *sonnet29.txt* in the same directory as *hello.txt* and write the following text in it:
- Make sure to separate the four lines with line breaks

```
When, in disgrace with fortune and men's eyes,
I all alone beweep my outcast state,
And trouble deaf heaven with my bootless cries,
And look upon myself and curse my fate,
>>> sonnetFile = open('sonnet29.txt')
>>> sonnetFile.readlines()
[When, in disgrace with fortune and men's eyes,\n', ' I all alone beweep my
outcast state,\n', And trouble deaf heaven with my bootless cries,\n', And
look upon myself and curse my fate,']
```

3.3 Writing to Files

- Python allows you to write content to a file in a way similar to how the print() function "writes" strings to the screen.
- You can't write to a file you've opened in read mode, though. Instead, you need to open it in "write plaintext" mode or "append plaintext" mode, or write mode and append mode for short.
- Write mode will overwrite the existing file and start from scratch, just like when you overwrite a variable's value with a new value
- Pass 'w' as the second argument to open() to open the file in write mode Append mode, on the other hand, will append text to the end of the existing file.
- Pass 'a' as the second argument to open() to open the file in append mode.
- If the filename passed to open() does not exist, both write and append mode will create a new, blank file.
- Example:

```
>>> baconFile = open('bacon.txt', 'w')
>>> baconFile.write('Hello world!\n')
13
>>> baconFile.close()
>>> baconFile = open('bacon.txt', 'a')
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> baconFile = open('bacon.txt')
>>> content = baconFile.read()
>>> baconFile.close()
>>> print(content)
Hello world!
Bacon is not a vegetable.
```

4. <u>Saving Variables with the shelve Module</u>

- You can save variables in your Python programs to binary shelf files using the shelve module.
- This way, your program can restore data to variables from the hard drive.
- The shelve module will let you add Save and Open features to your program.
- For example, if you ran a program and entered some configuration settings, you could save those settings to a shelf file and then have the program load them the next time it is run.

```
>>> import shelve
>>> shelfFile = shelve.open('mydata')
>>> cats = ['Zophie', 'Pooka', 'Simon']
>>> shelfFile['cats'] = cats
>>> shelfFile.close()
```

- To read and write data using the shelve module, you first import shelve. Call shelve.open() and pass it a filename, and then store the returned shelf value in a variable.
- create a list cats and write shelfFile['cats'] = cats to store the list in shelfFile as a value associated with the key 'cats' (like in a dictionary). Then we call close() on shelfFile.
- programs can use the shelve module to later reopen and retrieve the data from these shelf files
- Shelf values don't have to be opened in read or write mode—they can do both once opened
- **Program:**

```
>>> shelfFile = shelve.open('mydata')
>>> type(shelfFile)
<class 'shelve.DbfilenameShelf'>
>>> shelfFile['cats']
['Zophie', 'Pooka', 'Simon']
>>> shelfFile.close()
```

- Just like dictionaries, shelf values have keys() and values() methods that will return list-like values of the keys and values in the shelf
- Since these methods return list-like values instead of true lists, you should pass them to the list() function to get them in list form

```
>>> shelfFile = shelve.open('mydata')
>>> list(shelfFile.keys())
['cats']
>>> list(shelfFile.values())
[['Zophie', 'Pooka', 'Simon']]
>>> shelfFile.close()
```

5. Saving Variables with the pprint.pformat() Function

- pprint.pprint() function will "pretty print" the contents of a list or dictionary to the screen,
- while the pprint.pformat() function will return this same text as a string instead of printing it.
- file will be your very own module that you can import whenever you want to use the variable stored in it.
- **For example,**

```
>>> import pprint
>>> cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> pprint.pformat(cats)
"[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]"
>>> fileObj = open('myCats.py', 'w')
>>> fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
83
>>> fileObj.close()
```

- Python programs can even generate other Python programs. You can then import these files into scripts.

```
>>> import myCats
>>> myCats.cats
[{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> myCats.cats[0]
{'name': 'Zophie', 'desc': 'chubby'}
>>> myCats.cats[0]['name']
'Zophie'
```

6. <u>Project: Generating Random Quiz Files</u>

- The program does:
  - Creates 35 different quizzes.
  - Creates 50 multiple-choice questions for each quiz, in random order.
  - Provides the correct answer and three random wrong answers for each question, in random order.
  - Writes the quizzes to 35 text files.
  - Writes the answer keys to 35 text files.

- This means the code will need to do the following:
  - Store the states and their capitals in a dictionary.
  - Call open(), write(), and close() for the quiz and answer key text files.
  - Use random.shuffle() to randomize the order of the questions and multiple-choice options.

Step 1: Store the Quiz Data in a Dictionary

- The first step is to create a skeleton script and fill it with your quiz data. Create a file named randomQuizGenerator.py,

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

 import random
# The quiz data. Keys are states and values are their capitals.
 capitals = {'Alabama': 'Montgomery', 'Alaska': 'Juneau', 'Arizona': 'Phoenix',
'Arkansas': 'Little Rock', 'California': 'Sacramento', 'Colorado': 'Denver',
'Connecticut': 'Hartford', 'Delaware': 'Dover', 'Florida': 'Tallahassee',
'Georgia': 'Atlanta', 'Hawaii': 'Honolulu', 'Idaho': 'Boise', 'Illinois':
'Springfield', 'Indiana': 'Indianapolis', 'Iowa': 'Des Moines', 'Kansas':
'Topeka', 'Kentucky': 'Frankfort', 'Louisiana': 'Baton Rouge', 'Maine':
'Augusta', 'Maryland': 'Annapolis', 'Massachusetts': 'Boston', 'Michigan':
'Lansing', 'Minnesota': 'Saint Paul', 'Mississippi': 'Jackson', 'Missouri':
'Jefferson City', 'Montana': 'Helena', 'Nebraska': 'Lincoln', 'Nevada':
'Carson City', 'New Hampshire': 'Concord', 'New Jersey': 'Trenton', 'New
Mexico': 'Santa Fe', 'New York': 'Albany', 'North Carolina': 'Raleigh',
'North Dakota': 'Bismarck', 'Ohio': 'Columbus', 'Oklahoma': 'Oklahoma City',
'Oregon': 'Salem', 'Pennsylvania': 'Harrisburg', 'Rhode Island': 'Providence',
'South Carolina': 'Columbia', 'South Dakota': 'Pierre', 'Tennessee':
'Nashville', 'Texas': 'Austin', 'Utah': 'Salt Lake City', 'Vermont':
'Montpelier', 'Virginia': 'Richmond', 'Washington': 'Olympia', 'West
Virginia': 'Charleston', 'Wisconsin': 'Madison', 'Wyoming': 'Cheyenne'}
# Generate 35 quiz files.
 for quizNum in range(35):
# TODO: Create the quiz and answer key files.
# TODO: Write out the header for the quiz.
# TODO: Shuffle the order of the states.
# TODO: Loop through all 50 states, making a question for each.
```

- Since this program will be randomly ordering the questions and answers, you’ll need to import the random module u to make use of its functions.
- The capitals variable v contains a dictionary with US states as keys and their capitals as values. And since you want to create 35 quizzes, the code that actually generates the quiz and answer key files (marked with TODO comments for now) will go inside a for loop that loops 35 times

Step 2: Create the Quiz File and Shuffle the Question Order

- The code in the loop will be repeated 35 times—once for each quiz— so you have to worry about only one quiz at a time within the loop
- First you’ll create the actual quiz file.
- It needs to have a unique filename and should also have some kind of standard header in it, with places for the student to fill in a name, date, and class period.
- Add the following lines of code to *randomQuizGenerator.py*:

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--snip—

# Generate 35 quiz files.
for quizNum in range(35):
# Create the quiz and answer key files.
quizFile = open('capitalsquiz%s.txt' % (quizNum + 1), 'w')
 answerKeyFile = open('capitalsquiz_answers%s.txt' % (quizNum + 1), 'w')
# Write out the header for the quiz.
w quizFile.write('Name:\n\nDate:\n\nPeriod:\n\n')
quizFile.write((' ' * 20) + 'State Capitals Quiz (Form %s)' % (quizNum + 1))
quizFile.write('\n\n')
# Shuffle the order of the states.
states = list(capitals.keys())
random.shuffle(states)

# TODO: Loop through all 50 states, making a question for each.
```

Step 3: Create the Answer Options

- Now you need to generate the answer options for each question, which will be multiple choice from A to D.
- You'll need to create another for loop—this one to generate the content for each of the 50 questions on the quiz

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--snip—

        # Loop through all 50 states, making a question for each.
        for questionNum in range(50):

        # Get right and wrong answers.
        correctAnswer = capitals[states[questionNum]]
        wrongAnswers = list(capitals.values())
        del wrongAnswers[wrongAnswers.index(correctAnswer)]
        wrongAnswers = random.sample(wrongAnswers, 3)
        answerOptions = wrongAnswers + [correctAnswer]
        random.shuffle(answerOptions)

        # TODO: Write the question and answer options to the quiz file.
        # TODO: Write the answer key to a file.
```

- The correct answer is easy to get—it's stored as a value in the capitals dictionary
- This loop will loop through the states in the shuffled states list, from states[0] to states[49], find each state in capitals, and store that state's corresponding capital in correctAnswer.

- The list of possible wrong answers is trickier. You can get it by duplicating *all* the values in the capitals dictionary
- deleting the correct answer w, and selecting three random values from this list
- The random.sample() function makes it easy to do this selection. Its first argument is the list you want to select from; the second argument is the number of values you want to select. The full list of answer options is the combination of these three wrong answers with the correct answers
- Finally, the answers need to be randomized z so that the correct response isn't always choice D.

Step 4: Write Content to the Quiz and Answer Key Files

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--snip—
# Loop through all 50 states, making a question for each.
for questionNum in range(50):
--snip—

# Write the question and the answer options to the quiz file.
quizFile.write('%s. What is the capital of %s?\n' % (questionNum + 1,
        states[questionNum]))
for i in range(4):
        quizFile.write(' %s. %s\n' % ('ABCD'[i], answerOptions[i]))
quizFile.write('\n')

# Write the answer key to a file.
answerKeyFile.write('%s. %s\n' % (questionNum + 1, 'ABCD'[
        answerOptions.index(correctAnswer)]))
quizFile.close()
answerKeyFile.close()
```

- A for loop that goes through integers 0 to 3 will write the answer options in the answerOptions list . The expression 'ABCD'[i] at treats the string 'ABCD' as an array and will evaluate to 'A','B', 'C', and then 'D' on each respective iteration through the loop.

```
Name:
Date:
Period:
                    State Capitals Quiz (Form 1)
1. What is the capital of West Virginia?
A. Hartford
B. Santa Fe
C. Harrisburg
D. Charleston

2. What is the capital of Colorado?
A. Raleigh
B. Harrisburg
C. Denver
D. Lincoln
--snip—
```

- The corresponding *capitalsquiz_answers1.txt* text file

```
1. D
2. C
3. A
4. C
--snip--
```

7. Project: Multiclipboard

- Say you have the boring task of filling out many forms in a web page or software with several text fields.
- The clipboard saves you from typing the same text over and over again. But only one thing can be on the clipboard at a time.
- The program will save each piece of clipboard text under a keyword.
- For example, when you run py mcb.pyw save spam, the current contents of the clipboard will be saved with the keyword *spam*.

  Here's what the program does:

  • The command line argument for the keyword is checked.

  • If the argument is save, then the clipboard contents are saved to the keyword.

  • If the argument is list, then all the keywords are copied to the clipboard.

  • Otherwise, the text for the keyword is copied to the keyboard. This means the code will need to do the following:

  • Read the command line arguments from sys.argv.

  • Read and write to the clipboard.

  • Save and load to a shelf file.

Step 1: Comments and Shelf Setup

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
# Usage: py.exe mcb.pyw save <keyword> - Saves clipboard to keyword.
        # py.exe mcb.pyw <keyword> - Loads keyword to clipboard.
        # py.exe mcb.pyw list - Loads all keywords to clipboard.
 import shelve, pyperclip, sys
 mcbShelf = shelve.open('mcb')

# TODO: Save clipboard content.
# TODO: List keywords and load content.
mcbShelf.close()
```

Step 2: Save Clipboard Content with a Keyword

- The program does different things depending on whether the user wants to save text to a keyword, load text into the clipboard, or list all the existing keywords.

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
--snip—

# Save clipboard content.
 if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
 mcbShelf[sys.argv[2]] = pyperclip.paste()
elif len(sys.argv) == 2:
 # TODO: List keywords and load content.

mcbShelf.close()
```

- If the first command line argument (which will always be at index 1 of the sys.argv list) is 'save'
- The second command line argument is the keyword for the current content of the clipboard.
- The keyword will be used as the key for mcbShelf, and the value will be the text currently on the clipboard

Step 3: List Keywords and Load a Keyword's Content

- The user wants to load clipboard text in from a keyword, or they want a list of all available keywords

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
--snip—

# Save clipboard content.
if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
        mcbShelf[sys.argv[2]] = pyperclip.paste()
elif len(sys.argv) == 2:
        # List keywords and load content.
        if sys.argv[1].lower() == 'list':
                pyperclip.copy(str(list(mcbShelf.keys())))
        elif sys.argv[1] in mcbShelf:
                pyperclip.copy(mcbShelf[sys.argv[1]])
mcbShelf.close()
```

- If there is only one command line argument, first let’s check whether it’s 'list'
- If so, a string representation of the list of shelf keys will be copied to the clipboard
- The user can paste this list into an open text editor to read it.
- Otherwise, you can assume the command line argument is a keyword. If this keyword exists in the mcbShelf shelf as a key, you can load the value onto the clipboard

CHAPTER 1-ORGANISING FILES

1. The shutil Module

- The shutil (or shell utilities) module has functions to let you copy, move, rename, and delete files in your Python programs

1.1 Copying Files and Folders

- The shutil module provides functions for copying files, as well as entire folders. Calling shutil.copy(source, destination) will copy the file at the path source to the folder at the path destination.
- If destination is a filename, it will be used as the new name of the copied file. This function returns a string of the path of the copied file.

Program:

```
>>> import shutil, os
>>> os.chdir('C:\\')
u>>> shutil.copy('C:\\spam.txt', 'C:\\delicious')
'C:\\delicious\\spam.txt'
 >>> shutil.copy('eggs.txt', 'C:\\delicious\\eggs2.txt')
'C:\\delicious\\eggs2.txt'
```

- The first shutil.copy() call copies the file at *C:\spam.txt* to the folder *C:\delicious*. The return value is the path of the newly copied file. Note that since a folder was specified as the destination
- The original *spam.txt* filename is used for the new, copied file's filename. The second shutil.copy()call also copies the file at *C:\eggs.txt* to the folder *C:\delicious* but gives the copied file the name *eggs2.txt.*

Program:

```
>>> import shutil, os
>>> os.chdir('C:\\')
>>> shutil.copytree('C:\\bacon', 'C:\\bacon_backup')
'C:\\bacon_backup'
```

- The shutil.copytree() call creates a new folder named *bacon_backup* with the same content as the original *bacon* folder

1.2 Moving and Renaming Files and Folders

- Calling shutil.move(source, destination) will move the file or folder at the path source to the path destination and will return a string of the absolute path of the new location.
- If destination points to a folder, the source file gets moved into destination and keeps its current filename

```
>>> import shutil
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs\\bacon.txt'
```

- Assuming a folder named eggs already exists in the C:\ directory, this shutil.move() calls says, “Move C:\bacon.txt into the folder C:\eggs.”
- The *destination* path can also specify a filename

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs\\new_bacon.txt')
'C:\\eggs\\new_bacon.txt'
```

- Both of the previous examples worked under the assumption that there was a folder *eggs* in the *C:\* directory. But if there is no *eggs* folder, then move() will rename *bacon.txt* to a file named *eggs*.

```
>>> shutil.mo
'C:\\eggs'
```

- The folders that make up the destination must already exist, or else Python will throw an exception

```
>>> shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
Traceback (most recent call last):
        File "C:\Python34\lib\shutil.py", line 521, in move
        os.rename(src, real_dst)
FileNotFoundError: [WinError 3] The system cannot find the path specified:
        'spam.txt' -> 'c:\\does_not_exist\\eggs\\ham'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
        File "<pyshell#29>", line 1, in <module>
                shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
        File "C:\Python34\lib\shutil.py", line 533, in move
                copy2(src, real_dst)
        File "C:\Python34\lib\shutil.py", line 244, in copy2
                copyfile(src, dst, follow_symlinks=follow_symlinks)
        File "C:\Python34\lib\shutil.py", line 108, in copyfile
                with open(dst, 'wb') as fdst:
        FileNotFoundError: [Errno 2] No such file or directory:
'c:\\does_not_exist\\eggs\\ham'
```

1.3 Permanently Deleting Files and Folders

- You can delete a single file or a single empty folder with functions in the os module, whereas to delete a folder and all of its contents, you use the shutil module.
- Calling os.unlink(*path*) will delete the file at *path*.
- Calling os.rmdir(*path*) will delete the folder at *path*. This folder must be empty of any files or folders

- Calling shutil.rmtree(*path*) will remove the folder at *path*, and all files and folders it contains will also be deleted.

```
import os
for filename in os.listdir():
if filename.endswith('.rxt'):
os.unlink(filename)
```

- If you had any important files ending with *.rxt*, they would have been accidentally, permanently deleted

```
impo
for filename in os
    if filename.endswith('.rxt
    #os.unlink(filename)
    print(filename)
```

- Now the os.unlink() call is commented, so Python ignores it. Instead, you will print the filename of the file that would have been deleted.

1.4 Safe Deletes with the send2trash Module

- Since Python's built-in shutil.rmtree() function irreversibly deletes files and folders, it can be dangerous to use
- A much better way to delete files and folders is with the third-party send2trash module
- You can install this module by running pip install send2trash from a Terminal window

- Using send2trash is much safer than Python’s regular delete functions, because it will send folders and files to your computer’s trash or recycle bin instead of permanently deleting them.

```
>>> import send2trash
>>> baconFile = open('bacon.txt', 'a') # creates the file
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> send2trash.send2trash('bacon.txt')
```

2. **Walking a Directory Tree**

- you want to rename every file in some folder and also every file in every subfolder of that folder.
- That is, you want to walk through the directory tree, touching each file as you go.
- Writing a program to do this could get tricky; fortunately, Python provides a function to handle this process for you.

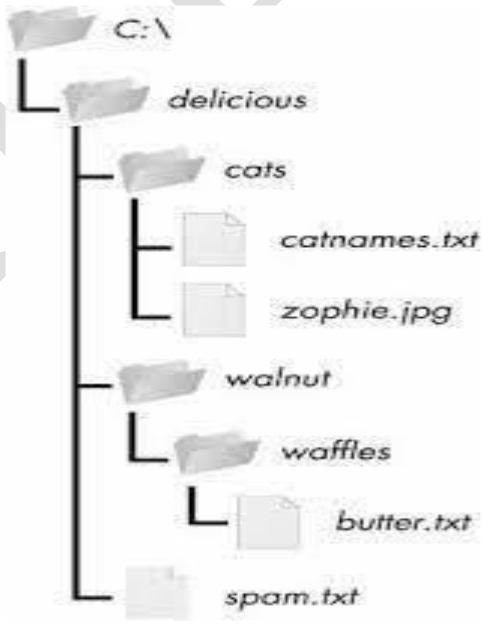


Figure: An example folder that contains three folders and four files

```
import os
for folderName, subfolders, filenames in os.walk('C:\\delicious'):
        print('The current folder is ' + folderName)
        for subfolder in subfolders:
                print('SUBFOLDER OF ' + folderName + ': ' + subfolder)
        for filename in filenames:
        print('FILE INSIDE ' + folderName + ': '+ filename)
                        print('')
```

- The os.walk() function is passed a single string value: the path of a folder. You can use os.walk() in a for loop statement to walk a directory tree, much like how you can use the range() function to walk over a range of numbers.
- Unlike range(), the os.walk() function will return three values on each iteration through the loop:
    1. A string of the current folder’s name
    2. A list of strings of the folders in the current folder
    3. A list of strings of the files in the current folder

```
The current folder is C:\delicious
SUBFOLDER OF C:\delicious: cats
SUBFOLDER OF C:\delicious: walnut
FILE INSIDE C:\delicious: spam.txt

The current folder is C:\delicious\cats
FILE INSIDE C:\delicious\cats: catnames.txt
FILE INSIDE C:\delicious\cats: zophie.jpg

The current folder is C:\delicious\walnut
SUBFOLDER OF C:\delicious\walnut: waffles

The current folder is C:\delicious\walnut\waffles
FILE INSIDE C:\delicious\walnut\waffles: butter.txt.
```

3. Compressing Files with the zipfile Module

- Compressing a file reduces its size, which is useful when transferring it over the Internet.
- since a ZIP file can also contain multiple files and subfolders, it's a handy way to package several files into one.
- This single file, called an *archive file*, can then be, say, attached to an email.

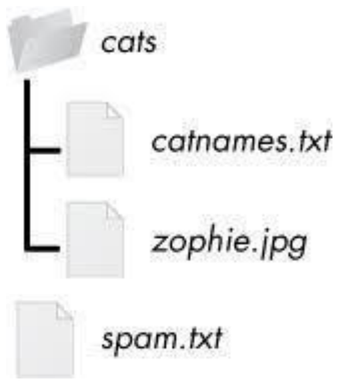


Figure : The contents of example.zip

3.1 Reading ZIP Files

- To read the contents of a ZIP file, first you must create a ZipFile object (note the capital letters *Z* and *F*).
- ZipFile objects are conceptually similar to the File objects you saw returned by the open() function
- They are values through which the program interacts with the file. To create a ZipFile object, call the zipfile.ZipFile() function

For example

```
>>> import zipfile, os
>>> os.chdir('C:\\') # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
>>> exampleZip.namelist()
['spam.txt', 'cats/', 'cats/catnames.txt', 'cats/zophie.jpg']
>>> spamInfo = exampleZip.getinfo('spam.txt')
>>> spamInfo.file_size
13908
>>> spamInfo.compress_size
3828
 >>> 'Compressed file is %sx smaller!' % (round(spamInfo.file_size / spamInfo
.compress_size, 2))
'Compressed file is 3.63x smaller!'
>>> exampleZip.close()
```

- A ZipFile object has a namelist() method that returns a list of strings for all the files and folders contained in the ZIP file.
- These strings can be passed to the getinfo() ZipFile method to return a ZipInfo object about that particular file. ZipInfo objects have their own attributes, such as file_size and compress_size in bytes, which hold integers of the original file size and compressed file size, respectively.

3.2 Extracting from ZIP Files

- The extractall() method for ZipFile objects extracts all the files and folders from a ZIP file into the current working directory.

```
>>> import zipfile, os
>>> os.chdir('C:\\') # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
 >>> exampleZip.extractall()
>>>  exampleZip.close()
```

- The extract() method for ZipFile objects will extract a single file from the ZIP file. Continue the interactive shell example

```
>>> e
'C:\\spam.txt'
>>> exampleZip.extract('spam.txt',
'C:\\some\\new\\folders\\spam.txt'
>>>  exampleZip.close()
```

3.3 Creating and Adding to ZIP Files

- To create your own compressed ZIP files, you must open the ZipFile object in *write mode* by passing 'w' as the second argument.
- When you pass a path to the write() method of a ZipFile object, Python will compress the file at that path and add it into the ZIP file.
- The write() method's first argument is a string of the filename to add.
- The second argument is the compression type parameter, which tells the computer what algorithm it should use to compress the files;

```
>>> import zipfile
>>> newZip = zipfile.ZipFile('new.zip', 'w')
>>> newZip.write('spam.txt', compress_type=zipfile.ZIP_DEFLATED)
>>> newZip.close()
```

4. Project 1: Renaming Files with American-Style Dates to European-Style Dates

The program does:

- It searches all the filenames in the current working directory for American-style dates.
- When one is found, it renames the file with the month and day swapped to make it European-style.
- This means the code will need to do the following:
- Create a regex that can identify the text pattern of American-style dates.
- Call os.listdir() to find all the files in the working directory.
- Loop over each filename, using the regex to check whether it has a date.
- If it has a date, rename the file with shutil.move().

Step 1: Create a Regex for American-Style Dates

- The first part of the program will need to import the necessary modules and create a regex that can identify MM-DD-YYYY dates.

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date
format
# to European DD-MM-YYYY.
 import shutil, os, re

# Create a regex that matches files with the American date format.
 datePattern = re.compile(r"""^(.*?)    # all text before the date
                    ((0|1)?\d)-     # one or two digits for the month
((0|1|2|3)?\d)-                          # one or two digits for the day
((19|20)\d\d)                            # four digits for the year
(.*?)$                                    # all text after the date
""", re.VERBOSE)

# TODO: Loop over the files in the working directory.
# TODO: Skip files without a date.
# TODO: Get the different parts of the filename.
# TODO: Form the European-style filename.
# TODO: Get the full, absolute file paths.
# TODO: Rename the files.
```

- After importing the re module at the top, call re.compile() to create a Regex object
- Passing re.VERBOSE for the second argument w will allow whitespace and comments in the regex string to make it more readable.

Step 2: Identify the Date Parts from the Filenames

- Next, the program will have to loop over the list of filename strings returned from os.listdir() and match them against the regex.

- The matched text will be stored in several variables

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date
format
# to European DD-MM-YYYY.

--snip—

# Loop over the files in the working directory.
        for amerFilename in os.listdir('.'):
        mo = datePattern.search(amerFilename)
        # Skip files without a date.
         if mo == None:
        continue

 # Get the different parts of the filename.
beforePart = mo.group(1)
monthPart = mo.group(2)
dayPart = mo.group(4)
yearPart = mo.group(6)
afterPart = mo.group(8)
        --snip--
```

- To keep the group numbers straight, try reading the regex from the beginning and count up each time you encounter an opening parenthesis.
- Without thinking about the code, just write an outline of the regular expression.

```
datePattern = re.compile(r"""^(1)       # all text before the date
(2 (3) )-                               # one or two digits for the month
(4 (5) )-                                # one or two digits for the day
(6 (7) )                                # four digits for the year
(8) $                                   # all text after the date
""", re.VERBOSE)
```

Step 3: Form the New Filename and Rename the Files

- As the final step, concatenate the strings in the variables made in the previous step with the European-style date:

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date
format
# to European DD-MM-YYYY.

--snip--
        # Form the European-style filename.
        euroFilename = beforePart + dayPart + '-' + monthPart + '-' + yearPart +
                    afterPart
# Get the full, absolute file paths.
absWorkingDir = os.path.abspath('.')
amerFilename = os.path.join(absWorkingDir, amerFilename)
euroFilename = os.path.join(absWorkingDir, euroFilename)

# Rename the files.
print('Renaming "%s" to "%s"...' % (amerFilename, euroFilename))
 #shutil.move(amerFilename, euroFilename)          # uncomment after testing
```

Project 2 : Backing Up a Folder into a ZI P File

Step 1: Figure Out the ZIP File's Name

- The code for this program will be placed into a function named backupToZip(). This will make it easy to copy and paste the function into other Python programs that need this functionality.

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

import zipfile, os
        def backupToZip(folder):
        # Backup the entire contents of "folder" into a ZIP file.
        folder = os.path.abspath(folder) # make sure folder is absolute
        # Figure out the filename this code should use based on
        # what files already exist.
         number = 1
        while True:
        zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
        if not os.path.exists(zipFilename):
                break
        number = number + 1
 # TODO: Create the ZIP file.
# TODO: Walk the entire folder tree and compress the files in each folder.
print('Done.')
backupToZip('C:\\delicious')
```

- Add the shebang (#!) line, describe what the program does, and import the zipfile and os modules

- Define a backupToZip() function that takes just one parameter, folder. This parameter is a string path to the folder whose contents should be backed up.
- The function will determine what filename to use for the ZIP file it will create; then the function will create the file, walk the folder folder, and each of the subfolders and files to the ZIP file.

Step 2: Create the New ZIP File

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

--snip--
 while True:
        zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
        if not os.path.exists(zipFilename):
                break
        number = number + 1
 # Create the ZIP file.
print('Creating %s...' % (zipFilename))
backupZip = zipfile.ZipFile(zipFilename, 'w')

# TODO: Walk the entire folder tree and compress the files in each folder.
print('Done.')

backupToZip('C:\\delicious')
```

Step 3: Walk the Directory Tree and Add to the ZIP File

- Now you need to use the os.walk() function to do the work of listing every file in the folder and its subfolders

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

--snip—

# Walk the entire folder tree and compress the files in each folder.
 for foldername, subfolders, filenames in os.walk(folder):
         print('Adding files in %s...' % (foldername))
# Add the current folder to the ZIP file.
backupZip.write(foldername)
# Add all the files in this folder to the ZIP file.
          for filename in filenames:
                   newBase / os.path.basename(folder) + '_'
                   if filename.startswith(newBase) and filename.endswith('.zip')
                   continue                    # don't backup the backup ZIP files
         backupZip.write(os.path.join(foldername, filename))
backupZip.close()
print('Done.')

backupToZip('C:\\delicious')
```

```
Creating delicious_1.zip...
Adding files in C:\delicious...
Adding files in C:\delicious\cats...
Adding files in C:\delicious\waffles...
Adding files in C:\delicious\walnut...
Adding files in C:\delicious\walnut\waffles...
Done.
```

CHAPTER - 4 DEBUGGING

1. Raising Exceptions

- Python raises an exception whenever it tries to execute invalid code.
- Raising an exception is a way of saying, "Stop running the code in this function and move the program execution to the except statement."
- Exceptions are raised with a raise statement. In code, a raise statement consists of the following:
- The raise keyword
- A call to the Exception() function
- A string with a helpful error message passed to the Exception() function
- For example

```
>>> raise Exception('This is the error message.')
Traceback (most recent call last):
File "<pyshell#191>", line 1, in <module>
raise Exception('This is the error message.')
Exception: This is the error message
```

-

- If there are no try and except statements covering the raise statement that raised the exception, the program simply crashes and displays the exception’s error message..

```
def boxPrint(symbol, width, height):
          if len(symbol) != 1:
            raise Exception('Symbol must be a single character string.')
          if width <= 2:
            raise Exception('Width must be greater than 2.')
          if height <= 2:
          raise Exception('Height must be greater than 2.')
print(symbol * width)
for i in range(height - 2):
          print(symbol + (' ' * (width - 2)) + symbol)
print(symbol * width)

for sym, w, h in (('*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
try:
    boxPrint(sym, w, h)
 except Exception as err:
     print('An exception happened: ' + str(err))
```

- This program uses the except Exception as err form of the except statement If an Exception object is returned from boxPrint()
- This except statement will store it in a variable named err. The Exception object can then be converted to a string by passing it to str() to produce a userfriendly error message

Output

```
****
* *
* *
****
OOOOOOOOOOOOOOOOOOOO
O                  O
O                  O
O                  O
OOOOOOOOOOOOOOOOOOOO
An exception happened: Width must be greater than 2.
An exception happened: Symbol must be a single character string.
```

2. Getting the Traceback as a String

- When Python encounters an error, it produces a treasure trove of error information called the traceback.
- The traceback includes the error message, the line number of the line that caused the error, and the sequence of the function calls that led to the error.

```
def sp
    bacon()
def bacon():
    raise Exception('This is the error message.')
spam()
```

Output:

```
Traceback (most recent call last):
   File "errorExample.py", line 7, in <module>
  spam()
File "errorExample.py", line 2, in spam
  bacon()
File "errorExample.py", line 5, in bacon
raise Exception('This is the error message.')
Exception: This is the error message.
```

- The traceback is displayed by Python whenever a raised exception goes unhandled
- But you can also obtain it as a string by calling traceback.format_exc(). This function is useful if you want the information from an exception's traceback but also want an except statement to gracefully handle the exception.
- For example,

```
>>> import traceback
>>> try:
raise Exception('This is the error message.')
except:
errorFile = open('errorInfo.txt', 'w')
errorFile.write(traceback.format_exc())
errorFile.close()
print('The traceback info was written to errorInfo.txt.')
116
The traceback info was written to errorInfo.txt.
```

- The 116 is the return value from the write() method, since 116 characters were written to the file.

```
Tracebac
File "<pyshell#28>", line 2, in <mo
Exception: This is the error message.
```

3. <u>Assertions</u>

- An *assertion* is a sanity check to make sure your code isn't doing something obviously wrong.
- These sanity checks are performed by assert statements. If the sanity check fails, then an AssertionError exception is raised.
- assert statement consists of the following:
- The assert keyword
    - A condition (that is, an expression that evaluates to True or False)
    - A comma
    - • A string to display when the condition is False
- For example,

```
>>> podBayDoorStatus = 'open'
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
>>> podBayDoorStatus = 'I\'m sorry, Dave. I\'m afraid I can't do that.''
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
Traceback (most recent call last):
File "<pyshell#10>", line 1, in <module>
assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
AssertionError: The pod bay doors need to be "open".
```

- set podBayDoorStatus to 'open', so from now on, we fully expect the value of this variable to be 'open'
- In a program that uses this variable, we might have written a lot of code under the assumption that the value is 'open'—code that depends on its being 'open' in order to work as we expect. So we add an assertion to make sure we're right to assume podBayDoorStatus is 'open'
- Here, we include the message 'The pod bay doors need to be "open".' so it'll be easy to see what's wrong if the assertion fails.

Using an Assertion in a Traffic Light Simulation

- The data structure representing the stoplights at an intersection is a dictionary with keys 'ns' and 'ew', for the stoplights facing north-south and east-west, respectively.
- The values at these keys will be one of the strings 'green', 'yellow', or 'red'. The code would look something like this:

```
market_2nd = {'ns': 'green', 'ew': 'red'}
mission_16th = {'ns': 'red', 'ew': 'green'}
```

- To start the project, you want to write a switchLights() function, which will take an intersection dictionary as an argument and switch the lights.

- At first, you might think that switchLights() should simply switch each light to the next color in the sequence: Any 'green' values should change to 'yellow', 'yellow' values should change to 'red', and 'red' values should change to 'green'.

Program:

```
def switchLights(stoplight):
        for key in stoplight.keys():
          if stoplight[key] == 'green':
                stoplight[key] = 'yellow'
        elif stoplight[key] == 'yellow':
                stoplight[key] = 'red'
        elif stoplight[key] == 'red':
                stoplight[key] = 'green'
    switchLights(market_2nd)
```

- while writing switchLights() you had added an assertion to check that at least one of the lights is always red,
- include the following at the bottom of the function:

```
assert 'red' in stoplight.values(), 'Neither light is red! ' + str(stoplight)
```

- With this assertion in place, your program would crash with this error message:

```
Traceback (most recent call last):
File "carSim.py", line 14, in <module>
switchLights(market_2nd)
File "carSim.py", line 13, in switchLights
assert 'red' in stoplight.values(), 'Neither light is red! ' + str(stoplight)
u AssertionError: Neither light is red! {'ns': 'yellow', 'ew': 'green'}
```

- The important line here is the AssertionError
- Neither direction of traffic has a red light, meaning that traffic could be going both ways. By failing fast early in the program’s execution

4. Logging

- Logging is a great way to understand what's happening in your program and in what order its happening.
- Python's logging module makes it easy to create a record of custom messages that you write.
- These log messages will describe when the program execution has reached the logging function call and list any variables you have specified at that point in time.
- On the other hand, a missing log message indicates a part of the code was skipped and never executed.

Using the logging Module

- To enable the logging module to display log messages on your screen as your program runs,

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
%(levelname)s
- %(message)s')
```

- when Python logs an event, it creates a LogRecord object that holds information about that event
- The logging module's basicConfig() function lets you specify what details about the LogRecord object you want to see and how you want those details displayed

Program:

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s
- %(message)s')
logging.debug('Start of program')
def factorial(n):
        logging.debug('Start of factorial(%s%%)' % (n))
        total = 1
        for i in range(n + 1):
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
logging.debug('End of factorial(%s%%)' % (n))
        return total
print(factorial(5))
logging.debug('End of program')
```

- debug() function will call basicConfig(), and a line of information will be printed.
- This information will be in the format we specified in basicConfig() and will include the messages we passed to debug().

Output:

```
2015-05-23 16:20:12,664 - DEBUG - Start of program
2015-05-23 16:20:12,664 - DEBUG - Start of factorial(5)
2015-05-23 16:20:12,665 - DEBUG - i is 0, total is 0
2015-05-23 16:20:12,668 - DEBUG - i is 1, total is 0
2015-05-23 16:20:12,670 - DEBUG - i is 2, total is 0
2015-05-23 16:20:12,673 - DEBUG - i is 3, total is 0
2015-05-23 16:20:12,675 - DEBUG - i is 4, total is 0
2015-05-23 16:20:12,678 - DEBUG - i is 5, total is 0
2015-05-23 16:20:12,680 - DEBUG - End of factorial(5)
0
2015-05-23 16:20:12,684 - DEBUG - End of program
```

- The factorial() function is returning 0 as the factorial of 5, which isn’t right.
- The for loop should be multiplying the value in total by the numbers from 1 to 5. But the log messages displayed by logging.debug() show that the i variable is starting at 0 instead of 1.
- Since zero times anything is zero, the rest of the iterations also have the wrong value for total
- Logging messages provide a trail of breadcrumbs that can help you figure out when things started to go wrong.
- Change the for i in range(n + 1): line to for i in range(**1,** n + 1):, and run the program again

Output

```
2015-05-23 17:13:40,650 - DEBUG - Start of program
2015-05-23 17:13:40,651 - DEBUG - Start of factorial(5)
2015-05-23 17:13:40,651 - DEBUG - i is 1, total is 1
2015-05-23 17:13:40,654 - DEBUG - i is 2, total is 2
2015-05-23 17:13:40,656 - DEBUG - i is 3, total is 6
2015-05-23 17:13:40,659 - DEBUG - i is 4, total is 24
2015-05-23 17:13:40,661 - DEBUG - i is 5, total is 120
2015-05-23 17:13:40,661 - DEBUG - End of factorial(5)
120
2015-05-23 17:13:40,666 - DEBUG - End of program
```

Logging Levels

- Logging levels provide a way to categorize your log messages by importance. There are five logging levels
- Messages can be logged at each level using a different logging function.

Level	Logging Function	Description
DEBUG	logging.debug()	The lowest level. Used for small details. Usually you care about these messages only when diagnosing problems.
INFO	logging.info()	Used to record information on general events in your program or confirm that things are working at their point in the program.
WARNING l	logging.warning()	Used to indicate a potential problem that doesn’t prevent the program from working but might do so in the future.
ERROR	logging.error()	Used to record an error that caused the program to fail to do something.
CRITICAL	logging.critical()	The highest level. Used to indicate a fatal error that has caused or is about to cause the program to stop running entirely.

Table 10-1: Logging Levels in Python

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
%(levelname)s - %(message)s')
>>> logging.debug('Some debugging details.')
2015-05-18 19:04:26,901 - DEBUG - Some debugging details.
>>> logging.info('The logging module is working.')
2015-05-18 19:04:35,569 - INFO - The logging module is working.
>>> logging.warning('An error message is about to be logged.')
2015-05-18 19:04:56,843 - WARNING - An error message is about to be
logged.
>>> logging.error('An error has occurred.')
2015-05-18 19:05:07,737 - ERROR - An error has occurred.
>>> logging.critical('The program is unable to recover!')
2015-05-18 19:05:45,794 - CRITICAL - The program is unable to recover!
```

Disabling Logging

- The logging.disable() function disables these so that you don't have to go into your program and remove all the logging calls by hand.
- pass logging.disable() a logging level, and it will suppress all log messages at that level or lower

```
>>> import logging
>>> logging.basicConfig(level=logging.INFO, format=' %(asctime)s -%(levelname)s -
%(message)s')
>>> logging.critical('Critical error! Critical error!')
2015-05-22 11:10:48,054 - CRITICAL - Critical error! Critical error!
>>> logging.disable(logging.CRITICAL)
>>> logging.critical('Critical error! Critical error!')
>>> logging.error('Error! Error!')
```

- Since logging.disable() will disable all messages after it, you will probably want to add it near the import logging line of code in your program

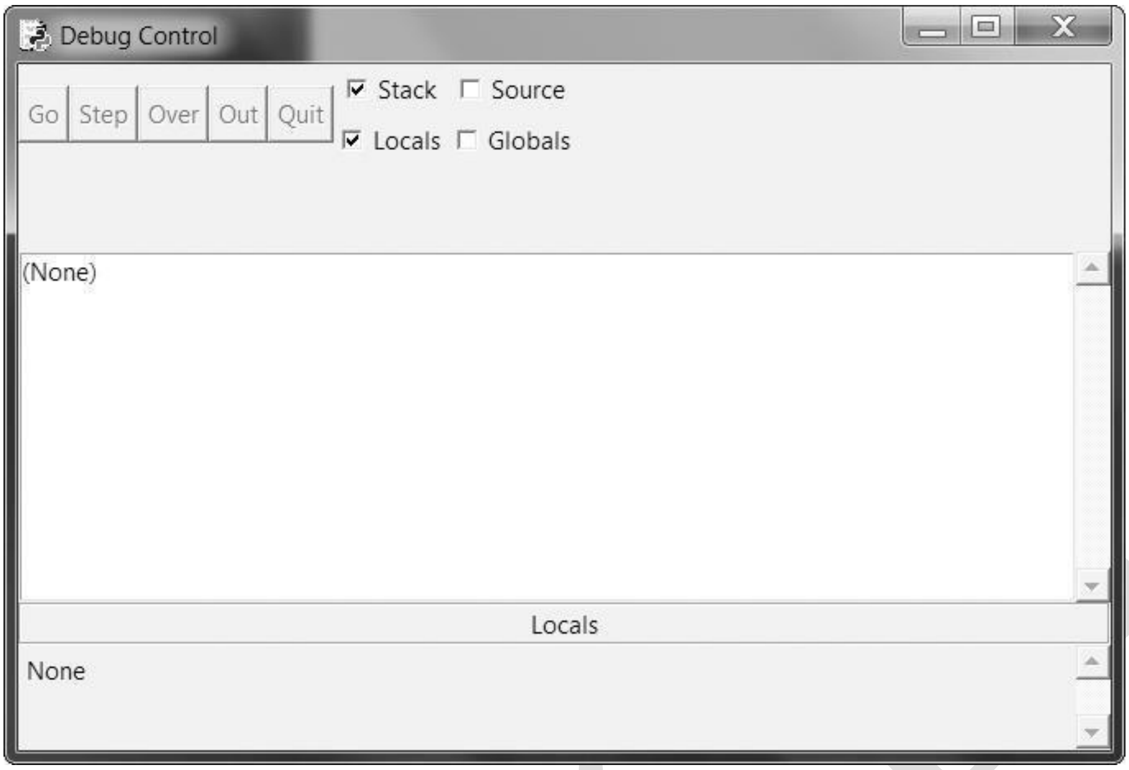
Logging to a File

- Instead of displaying the log messages to the screen, you can write them to a text file. The logging.basicConfig() function takes a filename keyword argument,

```
import logging
logging.basicConfig(filename='myProgramLog.txt',level=logging.DEBUG,
format=' %(asctime)s - %(levelname)s - %(message)s')
```

5. IDLE ’s Debugger

- The *debugger* is a feature of IDLE that allows you to execute your program one line at a time.
- The debugger will run a single line of code and then wait for you to tell it to continue
- To enable IDLE’s debugger, click Debug4Debugger in the interactive shell window.
- When the Debug Control window appears, select all four of the Stack, Locals, Source, and Globals checkboxes so that the window shows the full set of debug information
- While the Debug Control window is displayed, any time you run a program from the file editor
- debugger will pause execution before the first instruction and display the following:
- The line of code that is about to be executed
- A list of all local variables and their values
- A list of all global variables and their values



(None)

Locals

None

Figure: The Debug Control window

- You'll notice that in the list of global variables there are several variables you haven't defined, such as __builtins__, __doc__, __file__, and so on. These are variables that Python automatically sets whenever it runs a program.
- The program will stay paused until you press one of the five buttons in the Debug Control window: Go, Step, Over, Out, or Quit.

Go

- Clicking the Go button will cause the program to execute normally until it terminates or reaches a *breakpoint*
- If you are done debugging and want the program to continue normally, click the **Go** button.

Step

- Clicking the Step button will cause the debugger to execute the next line of code and then pause again
- The Debug Control window's list of global and local variables will be updated if their values change.
- If the next line of code is a function call, the debugger will "step into" that function and jump to the first line of code of that function.

Over

- Clicking the Over button will execute the next line of code, similar to the Step button.
- The Over button will "step over" the code in the function. The function's code will be executed at full speed, and the debugger will pause as soon as the function call returns.
- For example, if the next line of code is a print() call, you don't really care about code inside the built-in print() function; you just want the string you pass it printed to the screen.

Quit

- If you want to stop debugging entirely and not bother to continue executing the rest of the program, click the Quit button
- The Quit button will immediately terminate the program. If you want to run your program normally again, select Debug4Debugger again to disable the debugger.

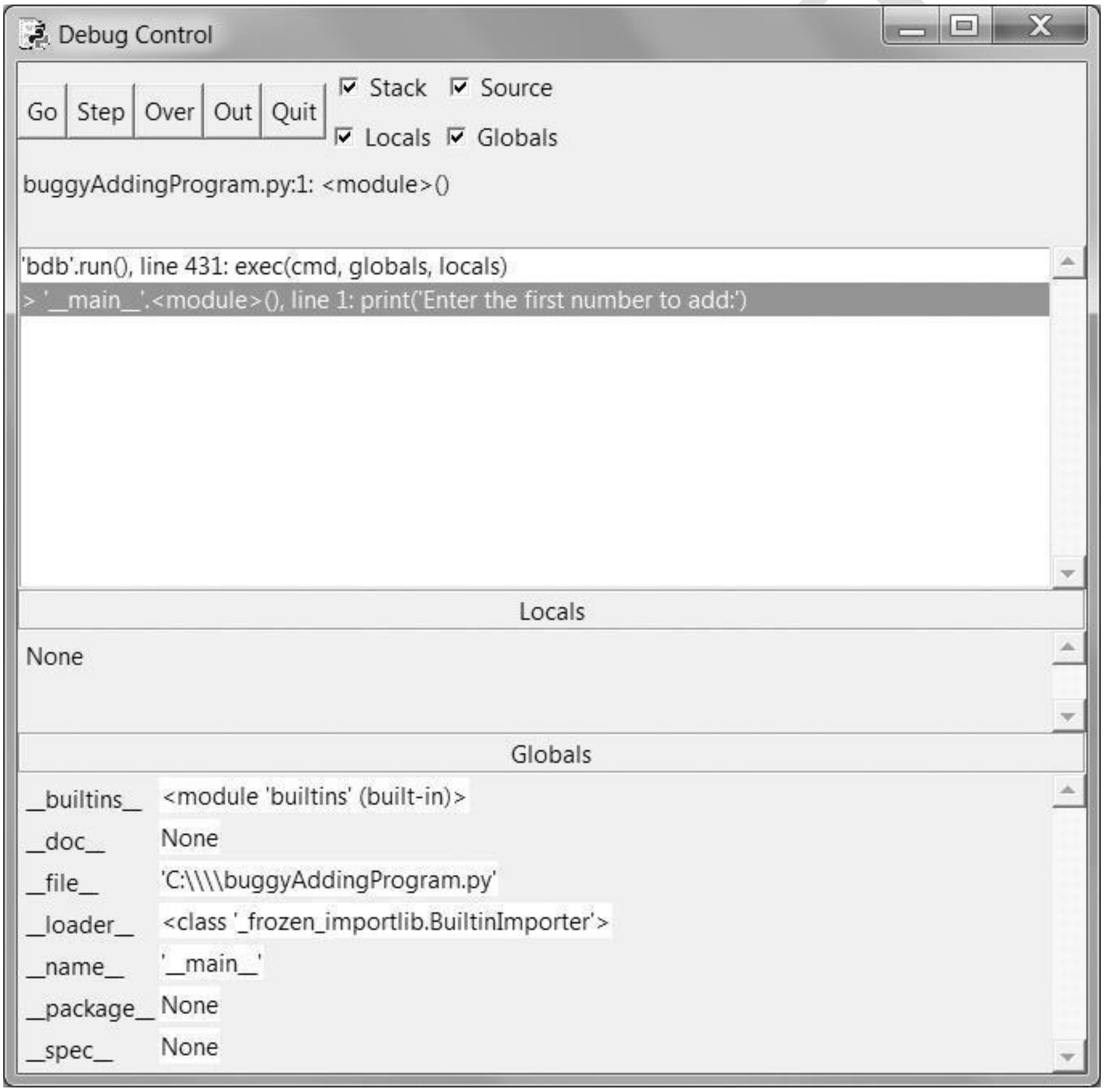
Debugging a Number Adding Program

```
print('Enter the first number to add:')
first = input()
print('Enter the second number to add:')
second = input()
print('Enter the third number to add:')
third = input()
print('The sum is ' + first + second + third)
```

Save it as *buggyAddingProgram.py* and run it first without the debugger enabled.

```
Enter the first number to add:
5
Enter the second number to add:
3
Enter the third number to add:
42
The sum is 5342
```

- The program hasn't crashed, but the sum is obviously wrong. Let's enable the Debug Control window and run it again, this time under the debugger
- When you press F5 or select **Run**4**Run Module** (with **Debug**4**Debugger** enabled and all four checkboxes on the Debug Control window checked), the program starts in a paused state on line 1.
- The debugger will always pause on the line of code it is about to execute.



```
buggyAddingProgram.py:1: <module>()

'bdb'.run(), line 431: exec(cmd, globals, locals)
> '__main__'.<module>(), line 1: print('Enter the first number to add:')
```

Locals

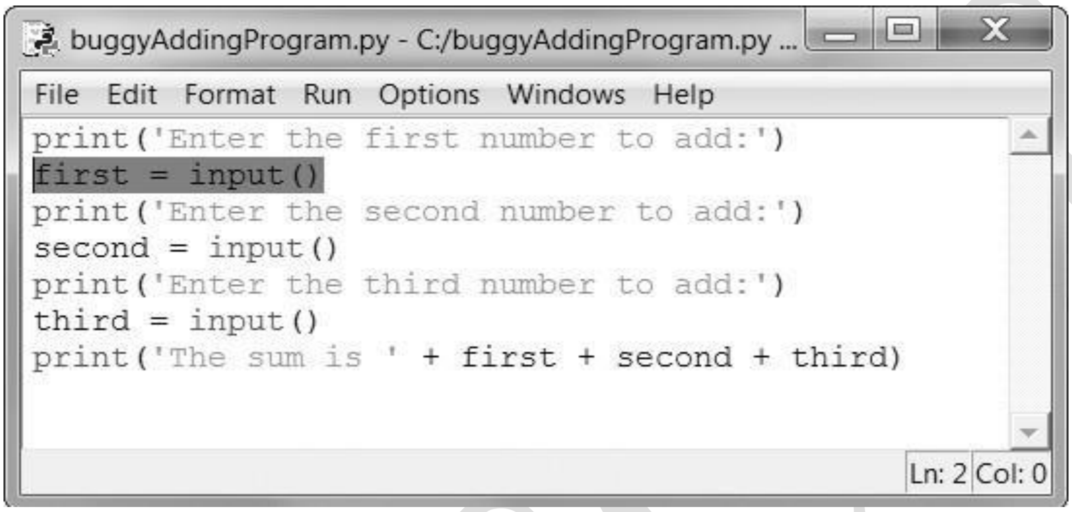
```
None
```

Globals

```
__builtins__  <module 'builtins' (built-in)>
__doc__       None
__file__      'C:\\\\buggyAddingProgram.py'
__loader__    <class '_frozen_importlib.BuiltinImporter'>
__name__      '__main__'
__package__   None
__spec__      None
```

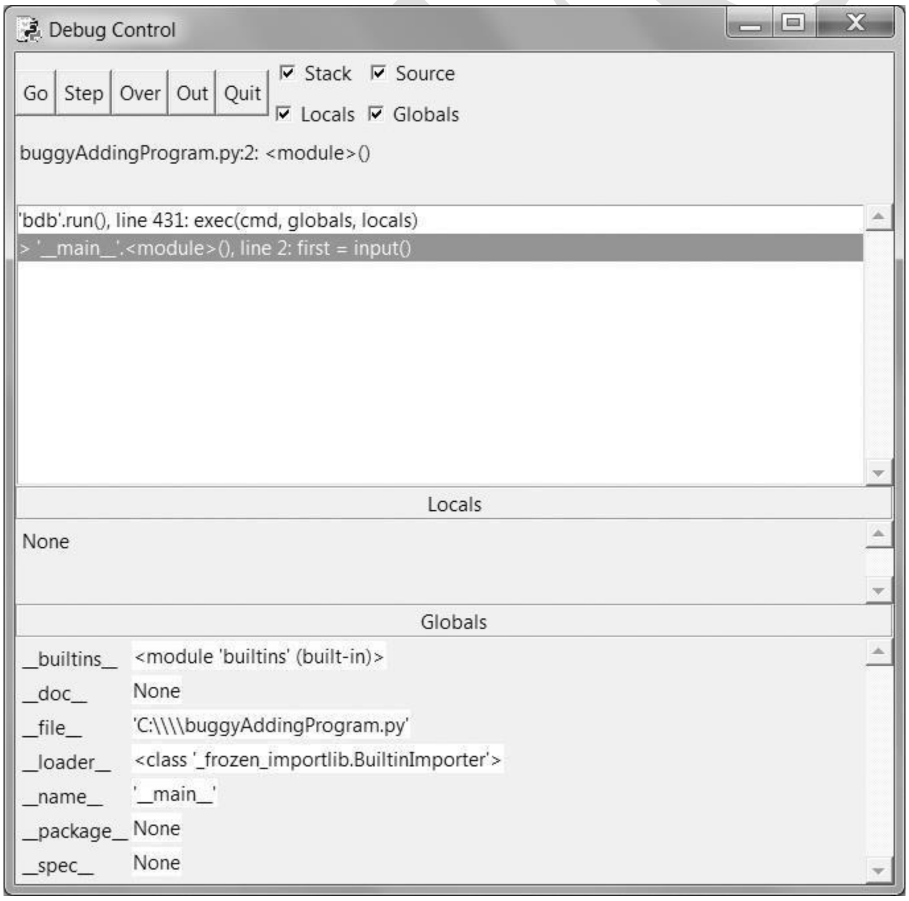
Figure The Debug Control window when the program first starts under the debugger

- Click the **Over** button once to execute the first print() call. You should use Over instead of Step here, since you don't want to step into the code for the print() function.
- The Debug Control window will update to line 2, and line 2 in the file editor window will be highlighted.


buggyAddingProgram.py - C:/buggyAddingProgram.py ...
File Edit Format Run Options Windows Help
Ln: 2 Col: 0

```
print('Enter the first number to add:')
first = input()
print('Enter the second number to add:')
second = input()
print('Enter the third number to add:')
third = input()
print('The sum is ' + first + second + third)
```

Figure: The Debug Control window after clicking Over



```
buggyAddingProgram.py:2: <module>()

'bdb'.run(), line 431: exec(cmd, globals, locals)
> '__main__'.<module>(), line 2: first = input()
```

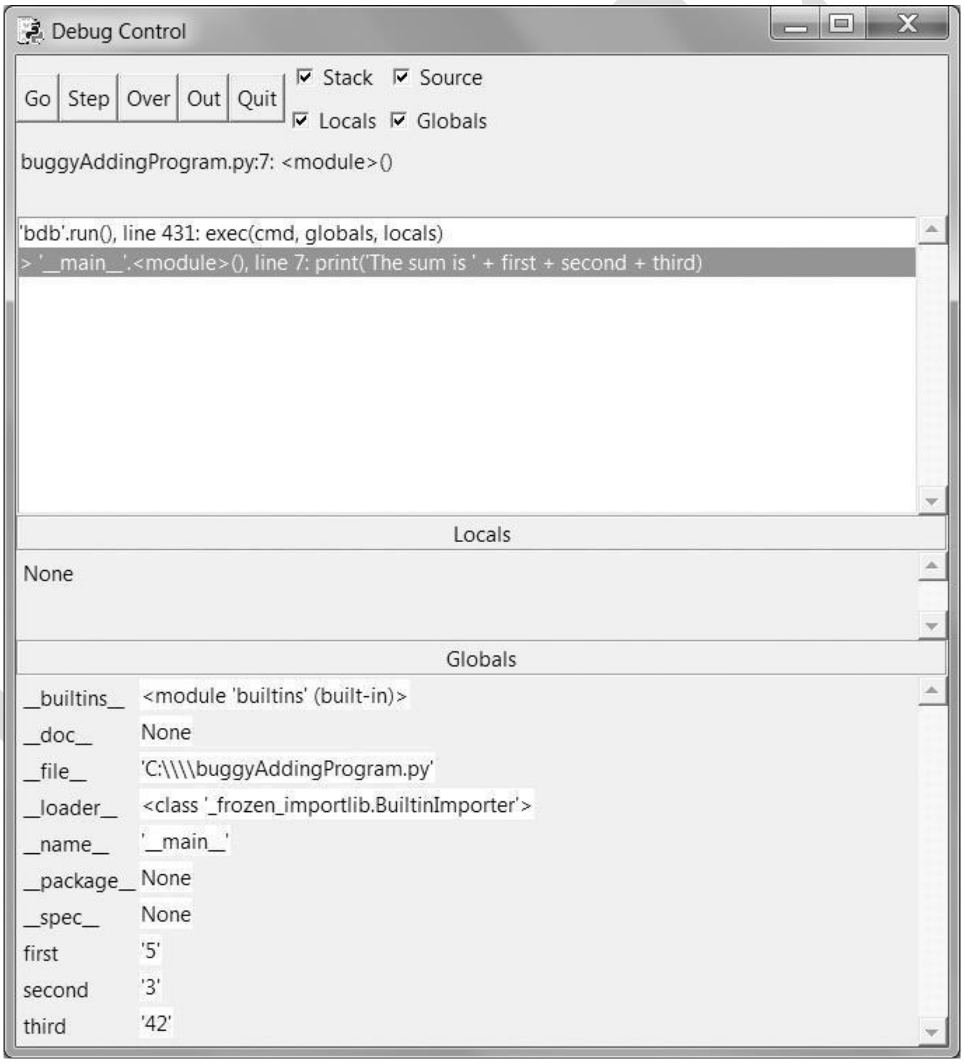
Locals:

```
None
```

Globals:

```
__builtins__  <module 'builtins' (built-in)>
__doc__       None
__file__      'C:\\\\buggyAddingProgram.py'
__loader__    <class '_frozen_importlib.BuiltinImporter'>
__name__      '__main__'
__package__   None
__spec__      None
```

- Click Over again to execute the input() function call, and the buttons in the Debug Control window will disable themselves while IDLE waits for you to type something for the input() call into the interactive shell window.
- Enter **5** and press Return. The Debug Control window buttons will be reenabled.
- Keep clicking Over, entering 3 and 42 as the next two numbers, until the debugger is on line 7, the final print() call in the program
- Globals section that the first, second, and third variables are set to string values '5', '3', and '42' instead of integer values 5, 3, and 42.
- When the last line is executed, these strings are concatenated instead of added together, causing the bug.



buggyAddingProgram.py:7: <module>()

```
'bdb'.run(), line 431: exec(cmd, globals, locals)
> '__main__'.<module>(), line 7: print('The sum is ' + first + second + third)
```

Locals

None

Globals

```
__builtins__  <module 'builtins' (built-in)>
__doc__       None
__file__      'C:\\\\buggyAddingProgram.py'
__loader__    <class '_frozen_importlib.BuiltinImporter'>
__name__      '__main__'
__package__   None
__spec__      None
first         '5'
second        '3'
third         '42'
```

Figure The Debug Control window on the last line. The variables are set to strings, causing the bug.

Breakpoints

- A *breakpoint* can be set on a specific line of code and forces the debugger to mpause whenever the program execution reaches that line.
- Open a new file editor window and enter the following program, which simulates flipping a coin 1,000 times.

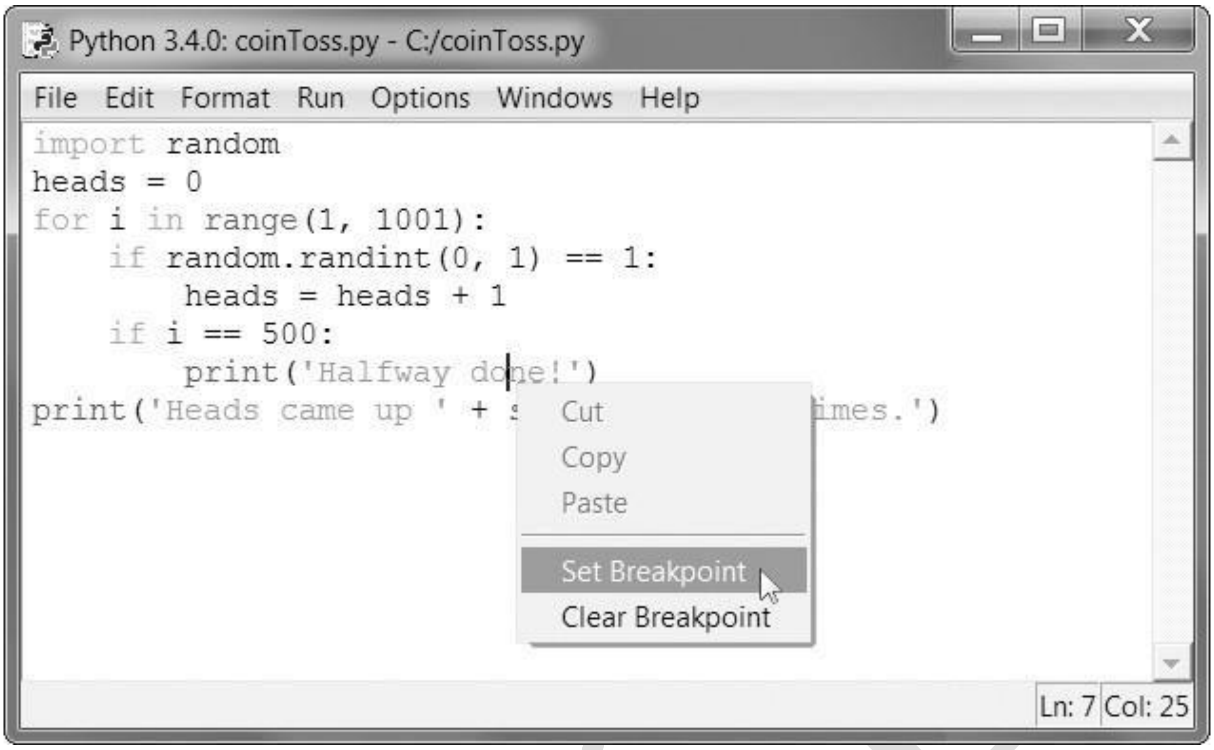
```
import random
heads = 0
for i in range(1, 1001):
 if random.randint(0, 1) == 1:
heads = heads + 1
if i == 500:
print('Halfway done!')
print('Heads came up ' + str(heads) + ' times.')
```

- The random.randint(0, 1) call u will return 0 half of the time and 1 the other half of the time.
- This can be used to simulate a 50/50 coin flip where 1 represents heads.

Output:

```
Halfway done!
 Heads came up 490 times.
```

- If you ran this program under the debugger, you would have to click the Over button thousands of times before the program terminated.
- If you were interested in the value of heads at the halfway point of the program's execution, when 500 of 1000 coin flips have been completed, you could instead just set a breakpoint on the line print('Halfway done!')
- To set a breakpoint, right-click the line in the file editor and select Set Breakpoint,


Python 3.4.0: coinToss.py - C:/coinToss.py
File Edit Format Run Options Windows Help
Cut
Copy
Paste
Set Breakpoint
Clear Breakpoint
Ln: 7 Col: 25

```
import random
heads = 0
for i in range(1, 1001):
    if random.randint(0, 1) == 1:
        heads = heads + 1
    if i == 500:
        print('Halfway done!')
print('Heads came up ' + s                    imes.')
```

Figure : Setting a breakpoint

CHAPTER 01

CLASSES AND OBJECTS

1. Programmer-defined types

- We have used many of Python’s built-in types; now we are going to define a new type. As an example, we will create a type called Point that represents a point in two-dimensional space.
- In mathematical notation, points are often written in parentheses with a comma separating the coordinates.
- For example, (0, 0) represents the origin, and (x, y) represents the point x units to the right and y units up from the origin.
- There are several ways we might represent points in Python:
  1. We could store the coordinates separately in two variables, x and y.
  2. We could store the coordinates as elements in a list or tuple.
  3. We could create a new type to represent points as objects.
- Creating a new type is more complicated than the other options, but it has advantages that will be apparent soon.
- A programmer-defined type is also called a class. A class definition looks like this:

```
class Point:
"""Represents a point in 2-D space."""
```

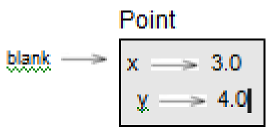


Figure 15.1: Object diagram.

- The header indicates that the new class is called Point. The body is a docstring that ex-plains what the class is for. You can define variables and methods inside a class definition, but we will get back to that later.
- Defining a class named Point creates a class object.

```
>>> Point
            <class '__main__.Point'>
```

- Because Point is defined at the top level, its “full name” is __main__.Point.
- The class object is like a factory for creating objects. To create a Point, you call Point as if it were a function.

```
blank = Point()
blank
<__main__.Point object at 0xb7e9d3ac>
```

- The return value is a reference to a Point object, which we assign to blank.
- Creating a new object is called instantiation, and the object is an instance of the class.
- When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix 0x means that the following number is in hexadecimal).

2. Attributes

- You can assign values to an instance using dot notation:

```
blank.x = 3.0
blank.y = 4.0
```

- This syntax is similar to the syntax for selecting a variable from a module, such as math.pi or string.whitespace .
- In this case, though, we are assigning values to named elements of an object. These elements are called attributes.
- A state diagram that shows an object and its attributes is called an object diagram; see Figure 15.1.
- The variable blank refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number.
- You can read the value of an attribute using the same syntax:

```
blank.y
4.0
```

```
x = blank.x
x
3.0
```

- The expression blank.x means, “Go to the object blank refers to and get the value of x.” In the example, we assign that value to a variable named x. There is no conflict between the variable x and the attribute x.
- You can use dot notation as part of any expression. For example:

```
'(%g, %g)' % (blank.x, blank.y)
 '(3.0, 4.0)'
```

```
distance = math.sqrt(blank.x**2 + blank.y**2)
distance
5.0
```

- You can pass an instance as an argument in the usual way. For example:

```
def print_point(p):
print('(%g, %g)' % (p.x, p.y))
```

- print_point takes a point as an argument and displays it in mathematical notation. To invoke it, you can pass blank as an argument:

```
print_point(blank) (3.0, 4.0)
```

- Inside the function, p is an alias for blank, so if the function modifies p, blank changes.

3. Rectangles

- Sometimes it is obvious what the attributes of an object should be, but other times you have to make decisions.

- For example, imagine you are designing a class to represent rectangles. What attributes would you use to specify the location and size of a rectangle?
- You can ignore angle; to keep things simple, assume that the rectangle is either vertical or horizontal.
- There are at least two possibilities:
    1. You could specify one corner of the rectangle (or the center), the width, and the height.
    2. You could specify two opposing corners.
- At this point it is hard to say whether either is better than the other, so we'll implement the first one, just as an example.
- Here is the class definition:

```
class Rectangle:
"""Represents a  rectangle.
attributes: width, height, corner."""
```

- The docstring lists the attributes: width and height are numbers; corner is a Point object that specifies the lower-left corner.
- To represent a rectangle, you have to instantiate a Rectangle object and assign values to the attributes:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

- The expression box.corner.x means, "Go to the object box refers to and select the attribute named corner; then go to that object and select the attribute named x."
- Figure 15.2 shows the state of this object. An object that is an attribute of another object is **embedded**.

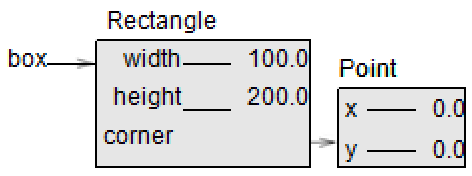


Figure 15.2: Object diagram.

4. Instances as return values

- Functions can return instances. For example, find_center takes a Rectangle as an argument and returns a Point that contains the coordinates of the center of the Rectangle:

```
def find_center(rect):
p = Point()
p.x = rect.corner.x + rect.width/2
p.y = rect.corner.y + rect.height/2
return p
```

- Here is an example that passes box as an argument and assigns the resulting Point to center:

```
center = find_center(box)
print_point(center)
(50, 100)
```

5. Objects are mutable

- You can change the state of an object by making an assignment to one of its attributes. For example, to change the size of a rectangle without changing its position, you can modify the values of width and height:

```
box.width = box.width + 50
box.height = box.height + 100
```

- You can also write functions that modify objects.
- For example, grow_rectangle takes a Rectangle object and two numbers, dwidth and dheight, and adds the numbers to the width and height of the rectangle:

```
def grow_rectangle(rect, dwidth, dheight):
rect.width += dwidth
rect.height += dheight
```

- Here is an example that demonstrates the effect:

```
box.width, box.height (150.0, 300.0)
grow_rectangle(box, 50, 100)
box.width, box.height
(200.0, 400.0)
```

- Inside the function, rect is an alias for box, so when the function modifies rect, box changes.

6. Copying

- Aliasing can make a program difficult to read because changes in one place might have unexpected effects in another place.
- It is hard to keep track of all the variables that might refer to a given object.
- Copying an object is often an alternative to aliasing. The copy module contains a function called copy that can duplicate any object:

```
p1 = Point()
p1.x = 3.0
p1.y = 4.0
```

```
import copy
p2 = copy.copy(p1)
```

- p1 and p2 contain the same data, but they are not the same Point.

```
print_point(p1)
(3, 4)
print_point(p2)
(3, 4)
p1 is p2
False
p1 == p2
False
```

- The is operator indicates that p1 and p2 are not the same object, which is what we expected. But you might have expected == to yield True because these points contain the same data.
- In that case, you will be disappointed to learn that for instances, the default behavior of the == operator is the same as the is operator; it checks object identity, not object equivalence.
- That’s because for programmer-defined types, Python doesn’t know what should be considered equivalent. At least, not yet.

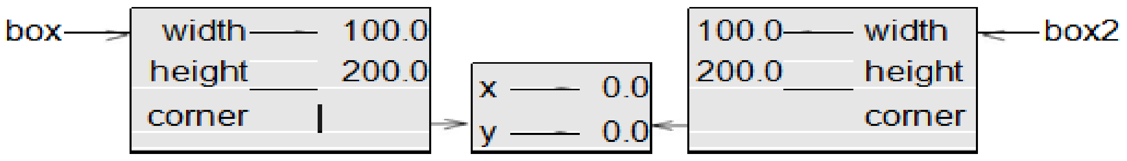


Figure 15.3: Object diagram.

- If you use copy.copy to duplicate a Rectangle, you will find that it copies the Rectangle object but not the embedded Point.

```
box2 = copy.copy(box)
box2 is box
False
box2.corner is box.corner
True
```

- Figure 15.3 shows what the object diagram looks like. This operation is called a **shallow copy** because it copies the object and any references it contains, but not the embedded objects.
- For most applications, this is not what you want.
- In this example, invoking grow_rectangle on one of the Rectangles would not affect the other, but invoking move_rectangle on either would affect both! This behavior is confusing and error-prone.
- Fortunately, the copy module provides a method named deepcopy that copies not only the object but also the objects it refers to, and the objects they refer to, and so on. You will not be surprised to learn that this operation is called a **deep copy.**

```
box3 = copy.deepcopy(box)
box3 is box
False
box3.corner is box.corner
False
```

- box3 and box are completely separate objects.

CHAPTER 02
CLASSES AND FUNCTIONS

1. Time

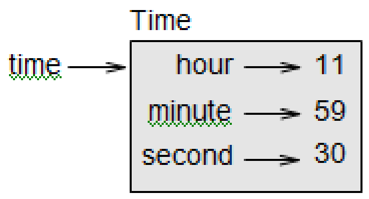
- ➢ As another example of a programmer-defined type, we'll define a class called Time that records the time of day. The class definition looks like this:

```
class Time:
    """Represents the time of day.
    attributes: hour, minute, second """
```

- ➢ We can create a new Time object and assign attributes for hours, minutes, and seconds:

```
time = Time( )
time.hour = 11
time.minute   =  59
time.second   =  30
```

- ➢ The state diagram for the Time object looks like Figure below.



2. Pure functions

- ➢ In the next few sections, we'll write two functions that add time values.
- ➢ They demonstrate two kinds of functions: pure functions and modifiers.
- ➢ They also demonstrate a development plan I'll call prototype and patch, which is a way of tackling a complex problem by starting with a simple prototype and incrementally dealing with the complications.
- ➢ Here is a simple prototype of add_time:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

➢ The function creates a new Time object, initializes its attributes, and returns a reference to the new object.

➢ This is called a pure function because it does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value.

- To test this function, let us create two Time objects: start contains the start time of a movie, like Monty Python and the Holy Grail, and duration contains the run time of the movie, which is one hour 35 minutes.
- add_time figures out when the movie will be done.

```
>>>  start  = Time()
>>>  start.hour  = 9
>>>   start.minute  = 45
>>>  start.second  =    0

>>>  duration   = Time()
>>>  duration.hour   = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

- The result, 10:80:00 might not be what you were hoping for.
- The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty.
- When that happens, we have to “carry” the extra seconds into the minute column or the extra minutes into the hour column.
- Here’s an improved version:

```
def add_time(t1, t2):
sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

if sum.second >= 60: sum.second -= 60
    sum.minute += 1

if sum.minute >= 60: sum.minute -= 60
    sum.hour += 1

    return sum
```

3. Modifiers

- Sometimes it is useful for a function to modify the objects it gets as parameters.
- In that case, the changes are visible to the caller. Functions that work this way are called modifiers.
- increment, which adds a given number of seconds to a Time object, can be written naturally as a modifier. Here is a rough draft:

```
def increment(time, seconds):
time.second += seconds

if time.second >= 60: time.second -= 60
time.minute += 1

if time.minute >= 60: time.minute -= 60
time.hour += 1
```

- The first line performs the basic operation; the remainder deals with the special cases we saw before.
- Is this function correct? What happens if seconds is much greater than sixty?
- In that case, it is not enough to carry once; we have to keep doing it until time.second is less than sixty.
- One solution is to replace the if statements with while statements. That would make the function correct, but not very efficient.
- Anything that can be done with modifiers can also be done with pure functions.

4. Prototyping versus planning

- The development plan, i.e. demonstrating is called “prototype and patch”. For each function, we wrote a prototype that performed the basic calculation and then tested it, patching errors along the way.
- This approach can be effective, especially if you don’t yet have a deep understanding of the problem.
- But incremental corrections can generate code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to know if you have found all the errors.
- Here is a function that converts Times to integers:

```
def time_to_int(time):
minutes = time.hour * 60 +time.minute
seconds = minutes * 60 + time.second return seconds
```

- And here is a function that converts an integer to a Time (recall that divmod divides the first argument by the second and returns the quotient and remainder as a tuple).

```
def int_to_time(seconds):
 time = Time()
minutes, time.second = divmod(seconds, 60)
time.hour, time.minute = divmod(minutes, 60)
return time
```

- Once we are convinced they are correct, you can use them to rewrite:

```
def add_time(t1, t2):
seconds = time_to_int(t1) + time_to_int(t2)
return int_to_time(seconds)
```

- This version is shorter than the original, and easier to verify.

CHAPTER 03
CLASSES AND METHODS

1. Object-Oriented Features

- Python is an object-oriented programming language, which means that it provides features that support object-oriented programming, which has these defining characteristics:

- Programs include class and method definitions.
- Most of the computation is expressed in terms of operations on objects.
- Objects often represent things in the real world, and methods often correspond to the ways things in the real world interact.

- A method is a function that is associated with a particular class.
- Methods are semantically the same as functions, but there are two syntactic differences:

- Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.
- The syntax for invoking a method is different from the syntax for calling a function.

2. Printing Objects

- We already defined a class named and also wrote a function named print_time:

```
class Time:
"""Represents the time of day."""

def print_time(time):
print('%.2d:%.2d:%.2d'     %  (time.hour,  time.minute,  time.second))
```

- To call this function, we have to pass a Time object as an argument:

```
>>>  start  = Time()
>>>  start.hour  = 9
>>>   start.minute  = 45
>>>   start.second  = 00
>>> print_time(start)
09:45:00
```

- To make print_time a method, all we have to do is move the function definition inside the class definition. Notice the change in indentation.

```
class Time:
    def print_time(time):
        print('%.2d:%.2d:%.2d'     %  (time.hour,  time.minute,  time.second))
```

- Now there are two ways to call print_time. The first (and less common) way is to use function syntax:

```
>>>Time.print_time(start)
09:45:00
```

- In this use of dot notation, Time is the name of the class, and print_time is the name of the method. start is passed as a parameter.
- The second (and more concise) way is to use method syntax:

```
>>> start.print_time()
09:45:00
```

- In this use of dot notation, print_time is the name of the method (again), and start is the object the method is invoked on, which is called the subject.
- Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is what the method is about.
- Inside the method, the subject is assigned to the first parameter, so in this case start is assigned to time.
- By convention, the first parameter of a method is called self, so it would be more common to write print_time like this:

```
class Time:
    def print_time(self):
        print('%.2d:%.2d:%.2d'     %  (self.hour, self.minute,  self.second))
```

- The reason for this convention is an implicit metaphor:

- The syntax for a function call, print_time(start), suggests that the function is the active agent. It says something like, “Hey print_time! Here’s an object for you to print.”
- In object-oriented programming, the objects are the active agents. A method invocation like start.print_time() says “Hey start! Please print yourself.”

3. <u>Another Example</u>

➢ Here’s a version of increment rewritten as a method:

```
# inside class Time:

def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

➢ This version assumes that time_to_int is written as a method. Also, note that it is a pure function, not a modifier.

➢ Here’s how you would invoke increment:

```
>>> start.print_time()

09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

➢ The subject, start, gets assigned to the first parameter, self. The argument, 1337, gets assigned to the second parameter, seconds.

➢ This mechanism can be confusing, especially if you make an error. For example, if you invoke increment with two arguments, you get:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

➢ The error message is initially confusing, because there are only two arguments in parentheses. But the subject is also considered an argument, so all together that’s three.

➢ By the way, a positional argument is an argument that doesn’t have a parameter name; that is, it is not a keyword argument. In this function call:

```
sketch(parrot, cage, dead=True)
```

- parrot and cage are positional, and dead is a keyword argument.

4. A More Complicated Example

- Rewriting is_after is slightly more complicated because it takes two Time objects as parameters.
- In this case it is conventional to name the first parameter self and the second parameter other:

```
# inside class Time:

def is_after(self, other):
return self.time_to_int() > other.time_to_int()
```

- To use this method, you have to invoke it on one object and pass the other as an argument:

```
>>> end.is_after(start)
True
```

5. The init Method

- The init method (short for "initialization") is a special method that gets invoked when an object is instantiated.
- Its full name is __init__(two underscore characters, followed by init, and then two more underscores).
- An init method for the Time class might look like this:

```
# inside class Time:
def _init_(self, hour=0, minute=0, second=0):
    self.hour = hour
    self.minute = minute
    self.second = second
```

- It is common for the parameters of__init__to have the same names as the attributes.
- The statement

```
self.hour = hour
```

- stores the value of the parameter hour as an attribute of self.
- The parameters are optional, so if you call Time with no arguments, you get the default values:

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

- If we provide one argument, it overrides hour:

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

- If we provide two arguments, they override hour and minute.

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

- And if we provide three arguments, they override all three default values

6. The __str__ Method

- __str__is a special method, like __init_, that is supposed to return a string representa- tion of an object.
- For example, here is a str method for Time objects:

```
# inside class Time:
def__str_(self):
        return  '%.2d:%.2d:%.2d' %  (self.hour,  self.minute,  self.second)
```

- When you print an object, Python invokes the str method:

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

7. Operator Overloading

- By defining other special methods, you can specify the behavior of operators on programmer-defined types.
- For example, if we define a method named __add__for the Time class, you can use the + operator on Time objects.
- Here is what the definition might look like:

```
def _add_(self,other):
     seconds=self.time_to_int()+other.time_to_int()
     return int_to_time(seconds)
```

- And here is how we could use it:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
```

- When you apply the + operator to Time objects, Python invokes __add__.
- When you print the result, Python invokes __str__. So there is a lot happening behind the scenes!
- Changing the behavior of an operator so that it works with programmer-defined types is called operator overloading.
- For every operator in Python there is a corresponding special method, like __add__.

8. Type-Based Dispatch

- The following is the version of _add_ that checks the type of other and invokes either add_time or increment:

```
def__add__(self,other):
      if isintance(other, Time):
            return self.add_time(other)
     else:
           return self.increment(other)

def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
         return int_to_time(seconds)

def increment(self, seconds):
            seconds += self.time_to_int()
            return int_to_time(seconds)
```

- The built-in function isinstance takes a value and a class object, and returns True if the value is an instance of the class.
- If other is a Time object, __add___invokes add_time. Otherwise it assumes that the parameter is a number and invokes increment.
- This operation is called a type-based dispatch because it dispatches the computation to different methods based on the type of the arguments.
- Here are examples that use the + operator with different types:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
```

```
11:20:00
>>> print(start + 1337)
10:07:17
```

- Unfortunately, this implementation of addition is not commutative. If the integer is the first operand, you get

```
>>> print(1337 + start)
TypeError:  unsupported  operand  type(s)  for  +:  'int'     and 'instance'
```

- The problem is, instead of asking the Time object to add an integer, Python is asking an integer to add a Time object, and it doesn't know how.
- But there is a clever solution for this problem: the special method __radd_, which stands for "right-side add".
- This method is invoked when a Time object appears on the right side of the + operator. Here's the definition:

```
# inside class Time:
def__radd__(self, other):
        return self._add_(other)
```

- And here's how it's used:

```
>>> print(1337 + start)
10:07:17
```

9. Polymorphism

- Type-based dispatch is useful when it is necessary, but (fortunately) it is not always necessary. Often you can avoid it by writing functions that work correctly for arguments with different types.
- Many of the functions we wrote for strings also work for other sequence types. For example, we used histogram to count the number of times each letter appears in a word.

```
def
histogram(s):
    d =  dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c]+1
    return d
```

- This function also works for lists, tuples, and even dictionaries, as long as the elements of s are hashable, so they can be used as keys in d:

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

- Functions that work with several types are called polymorphic. Polymorphism can facilitate code reuse.
- For example, the built-in function sum, which adds the elements of a sequence, works as long as the elements of the sequence support addition.

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 31)
>>> t3 = Time(7, 37)
>>> total = sum(t1, t2, t3)
>>> print(total)
23:01:00
```

- In general, if all of the operations inside a function work with a given type, the function works with that type.
- The best kind of polymorphism is the unintentional kind, where you discover that a func- tion you already wrote can be applied to a type you never planned for.