MODULE 1

Python Basics: Entering Expressions into the Interactive Shell, The Integer, Floating-Point, and String Data Types, String Concatenation and Replication, Storing Values in Variables, Your First Program, Dissecting Your Program.

Flow control: Boolean Values, Comparison Operators, Boolean Operators, Mixing Boolean and Comparison Operators, Elements of Flow Control, Program Execution, Flow Control Statements, Importing Modules, Ending a Program Early with sys.exit().

Functions: def Statements with Parameters, Return Values and return Statements, The None Value, Keyword Arguments and print(), Local and Global Scope, The global Statement, Exception Handling, A Short Program: Guess the Number.

Text Book1: Chapter 1, 2,3

CHAPTER 1: PYTHON BASICS

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.

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

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

- However, when + is used on two string values, it joins the strings as the string concatenation operator.
- 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'
```

1.4 Storing Values in Variables

- 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.

Assignment Statements

- 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'
```

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.
- A good variable name describes the data it contains.

1.5 Your First Program

- While the interactive shell is good for running Python instructions one at a time, to write entire Python programs, you'll type the instructions into the file editor.
- The file editor is similar to text editors such as Notepad or TextMate, but it has some specific features for typing in source code. To Open a new **file → New**
- 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:

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

print('Hello world!')
print('What is your name?')    # ask for their name
myName = input()
print('It is good to meet you, ' + myName)
print('The length of your name is:')
print(len(myName))

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))
```

len(myName) evaluates to an integer. It is then passed to print() to be displayed on the screen

- In the interactive shell:

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

- 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.

```
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

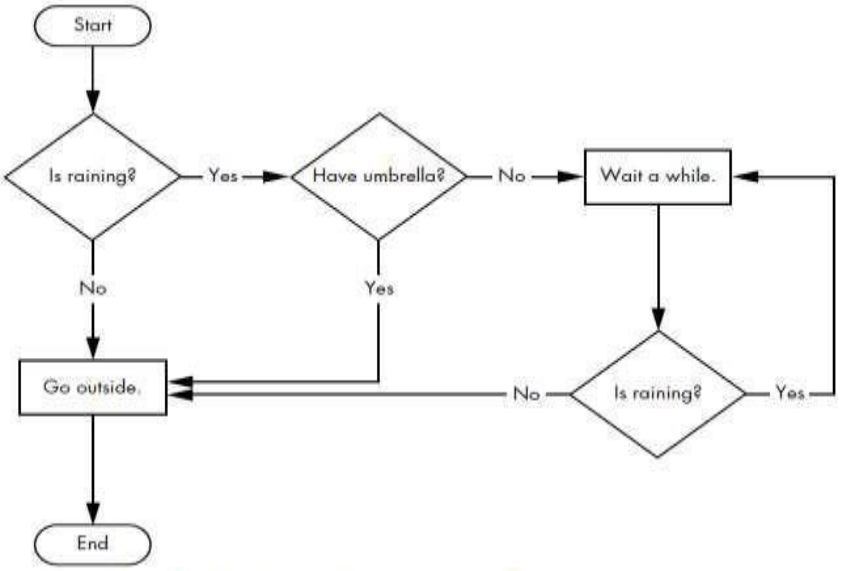
- Although the string value of a number is considered a completely different value from the integer or floating-point version, an integer can be equal to a floating point.

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

CHAPTER 2: FLOW CONTROL

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.



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 / Unary 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

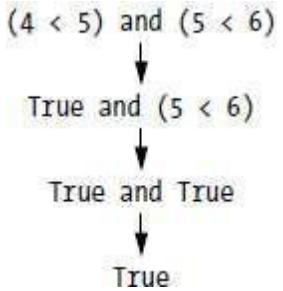
```
>>> not True
False
❶ >>> not not not not True
True
```

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:



- 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.

2.4 Elements of Flow Control

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

Conditions:

- 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.

Blocks of Code:

- 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:***

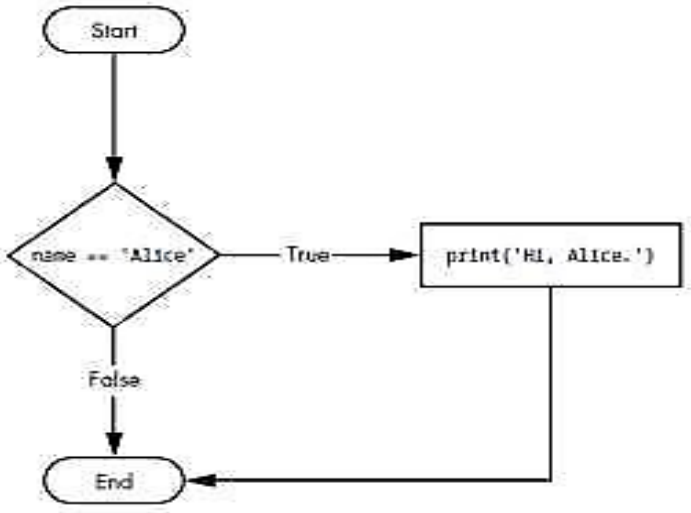
- The most common type of flow control statement is the if statement.
- An if statement‘s clause (that is, the block following the if statement) will execute if the statement‘s condition is True. The clause is skipped if the condition is False.
- In plain English, an if statement could be read as, ―If 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

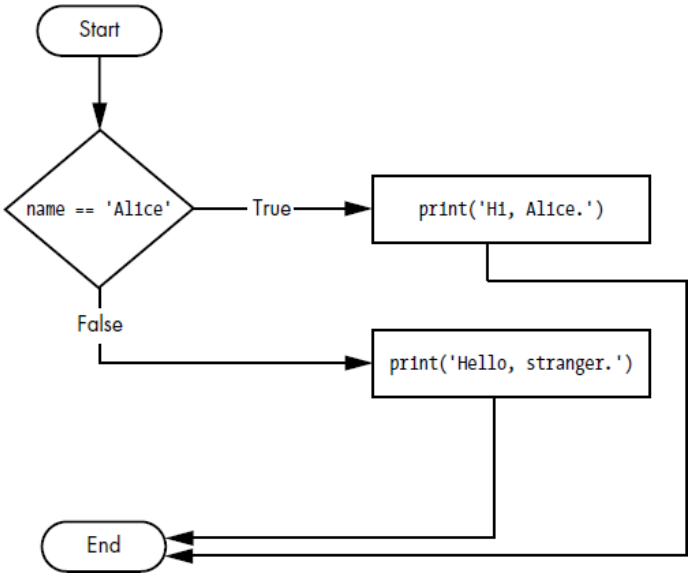
2. A colon

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)

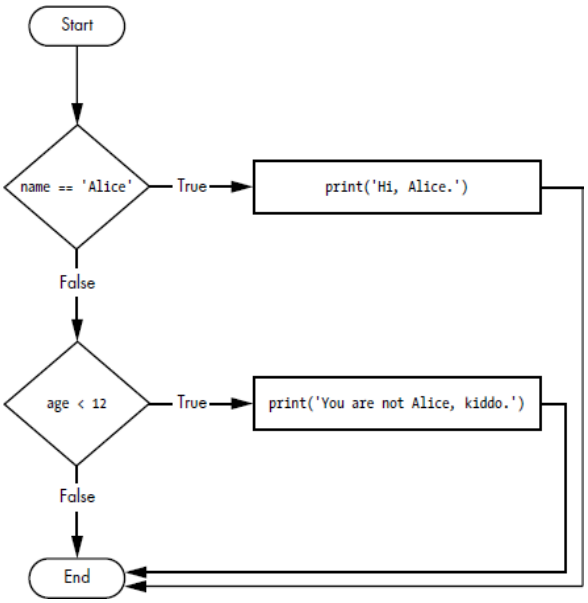
3. A colon

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.')
```

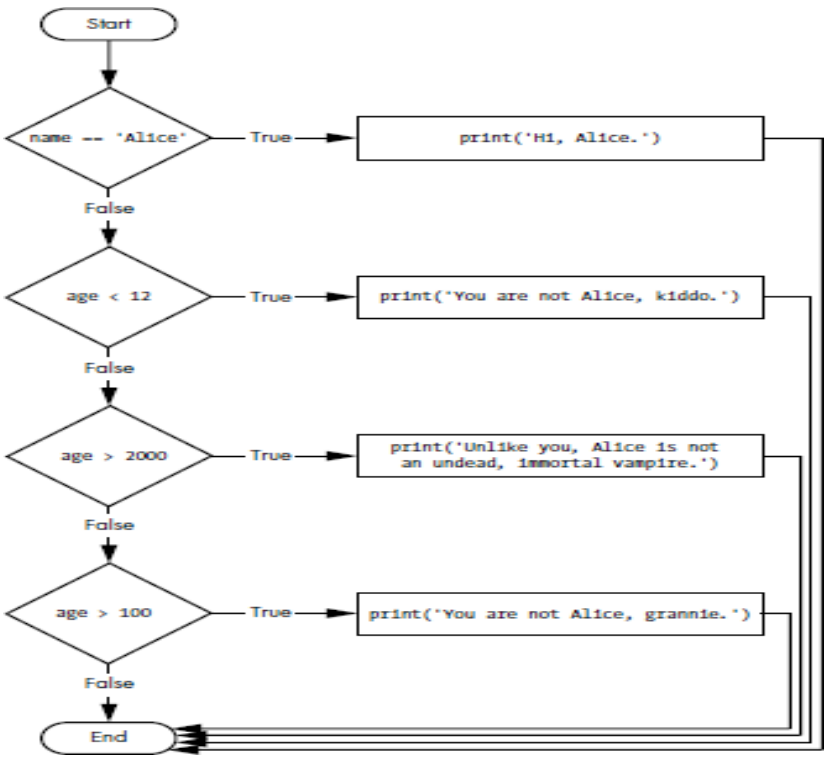
- Flowchart:



- **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:



- 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.

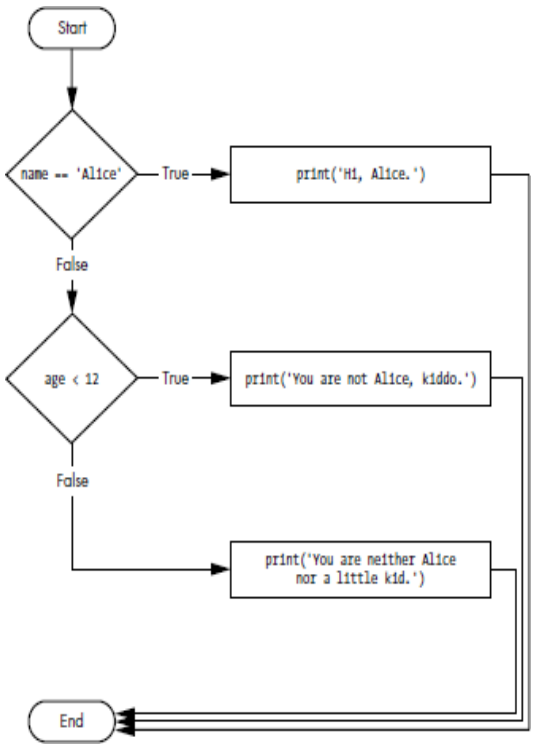
```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
elif age > 100:
    print('You are not Alice, grannie.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
```

- 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:

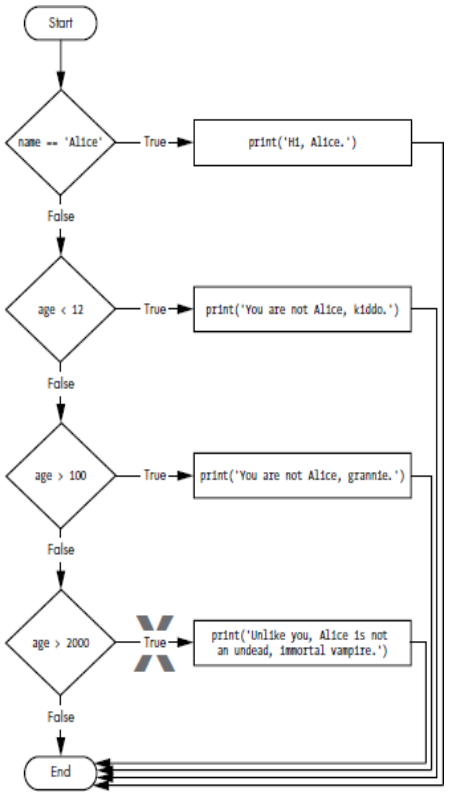
```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')
```

➢ Flowchart → (2)

Flowchart 1



Flowchart 2



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.

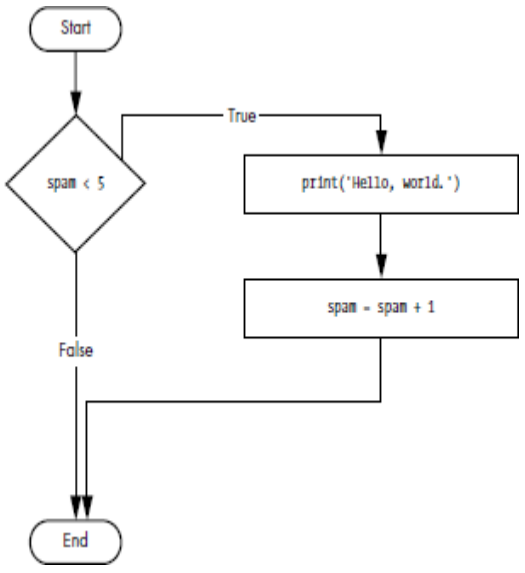
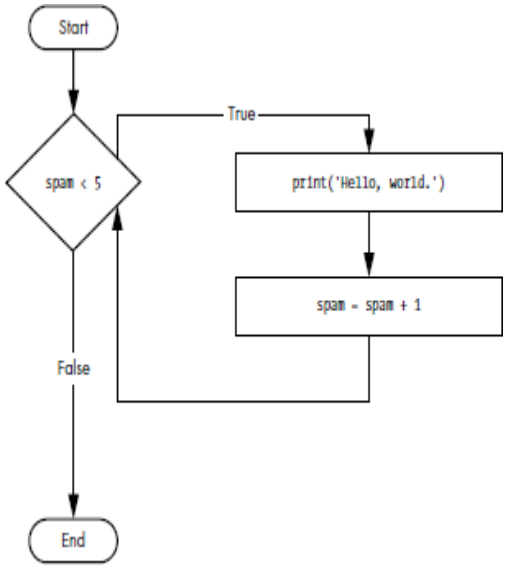
➢ But, at the end of a while clause, the program execution jumps back to the start of the while statement. The while clause is often called the while loop or just the loop.

- 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
	

- 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.

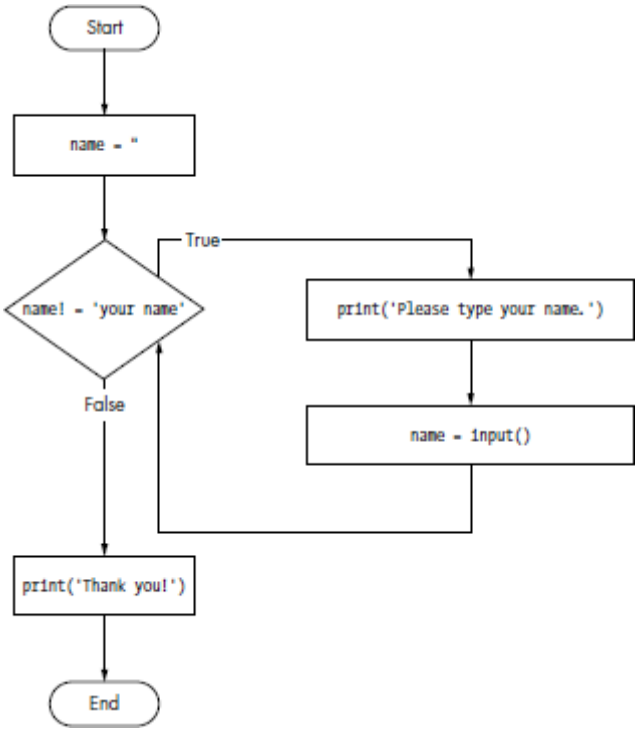
An annoying while loop:

- 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!

- 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:***

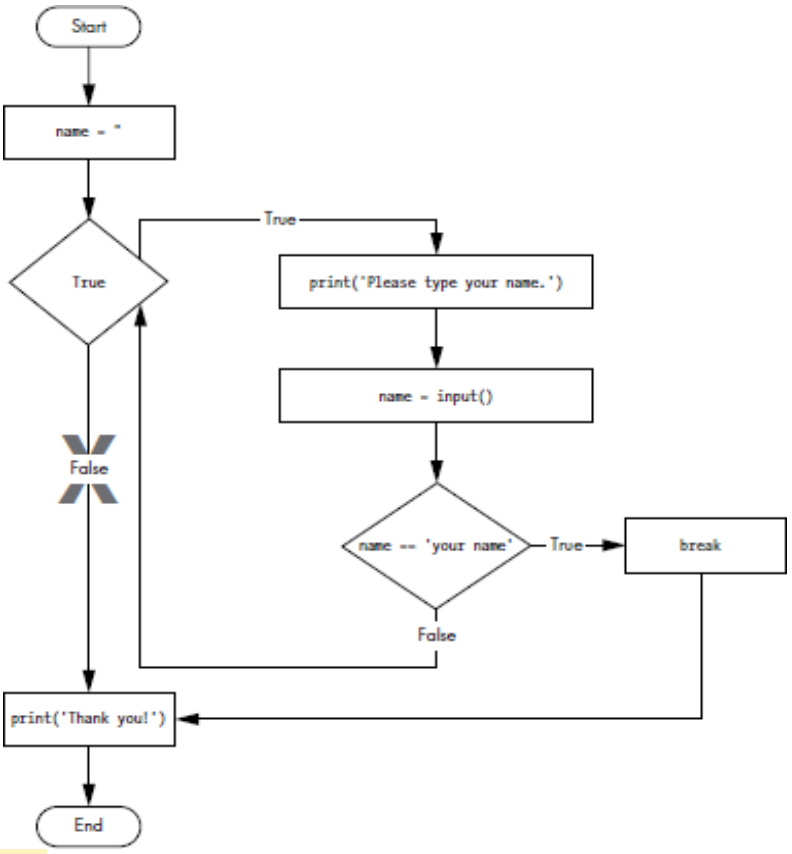
- There is a shortcut to getting the program execution to break out of a while loop's clause early.
- If the execution reaches a break statement, it immediately exits the while loop's clause
- In code, a break statement simply contains the break keyword.
- Example:

```
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:



Continue statement

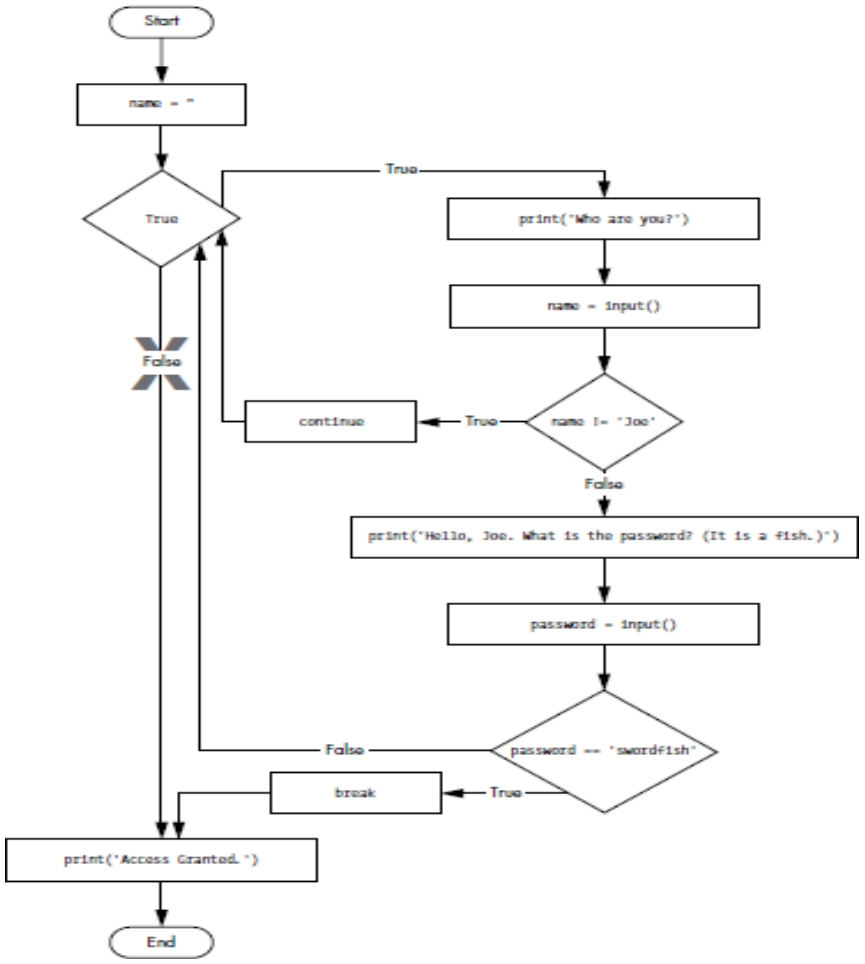
- Like break statements, continue statements are used inside loops.
- When the program execution reaches a continue statement, the program execution immediately jumps back to the start of the loop and reevaluates the loop's condition.

- 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.



- 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')
```

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)

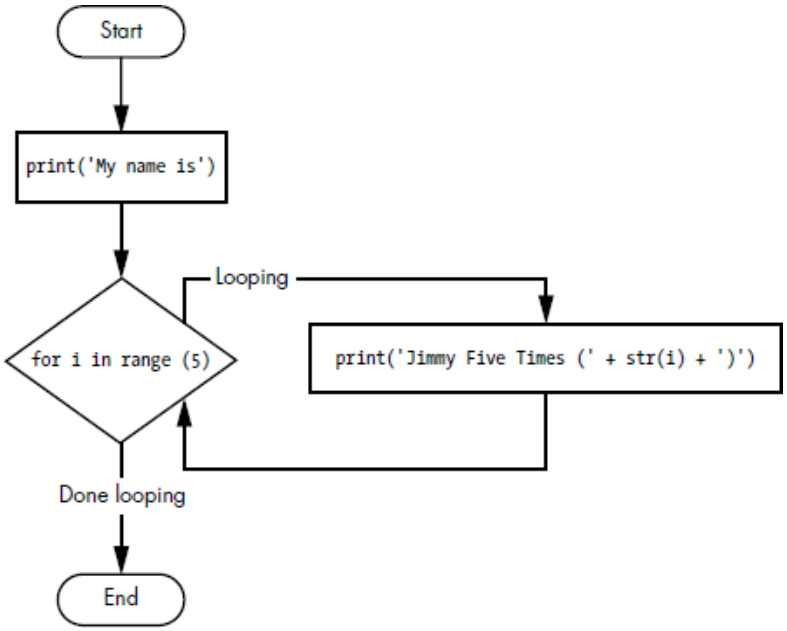
- Example and output:

Example	Output

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

6. The Starting, Stopping, and Stepping Arguments to range()

- 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.

from import Statements

- 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

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.
  2. However, a local scope can access global variables. Code in the global scope cannot use any local 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 Output
- The error happens because the eggs variable exists only in the local scope created when spam() is called.

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

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

- Once the program execution returns from spam, that local scope is destroyed, and there is no longer a variable named eggs.

Local Scopes Cannot Use Variables in Other Local Scopes

- 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

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

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

Output

```
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

```
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)
```

Output

```
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 ❸, eggs is the global variable, because there is no assignment statement or ham()

  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

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

❷ eggs = 'global'
spam()
```

Output

```
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. The List Data Type
2. Working with Lists
3. Augmented Assignment Operators
4. Methods
5. Example Program: Magic 8 Ball with a List
6. List-like Types: Strings and Tuples
7. References

1.1 The List Data Type

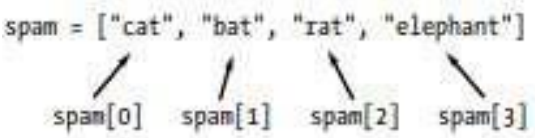
- A list is a value that contains multiple values in an ordered sequence.
- A list value looks like this: ['cat', 'bat', 'rat', 'elephant'].
- A list begins with an opening square bracket and ends with a closing square bracket, [].
- Values inside the list are also called items and are separated with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

Negative Indexes

- 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'
```

Removing Values from Lists with remove()

- 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]).

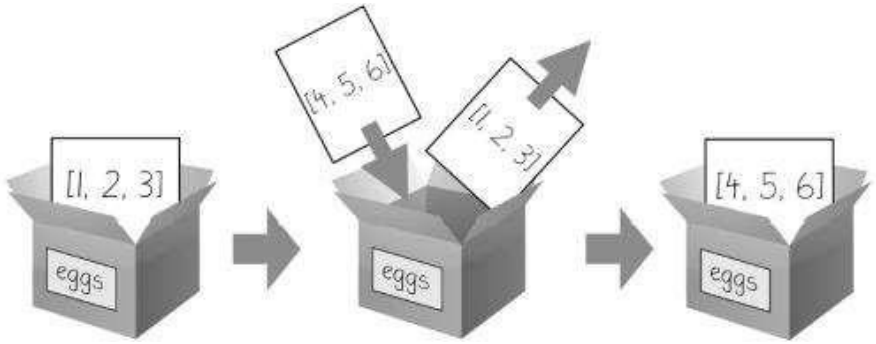


Figure: When eggs = [4, 5, 6] is executed, the contents of eggs are replaced with a new list value.

- 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]
```

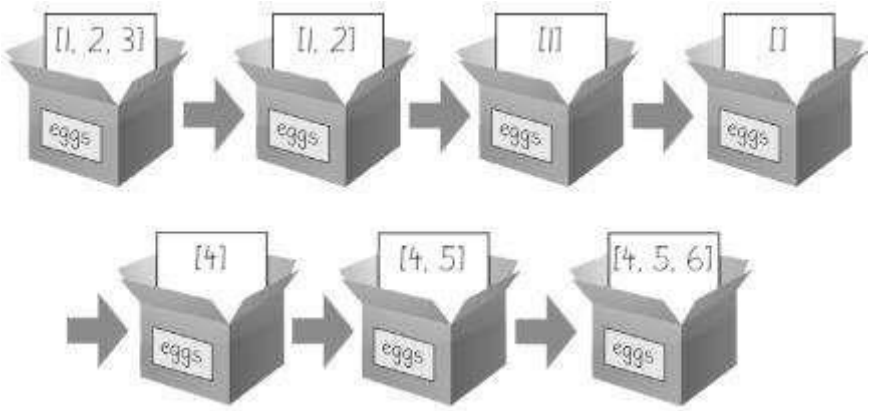


Figure: The del statement and the append() method modify the same list value in place.

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.

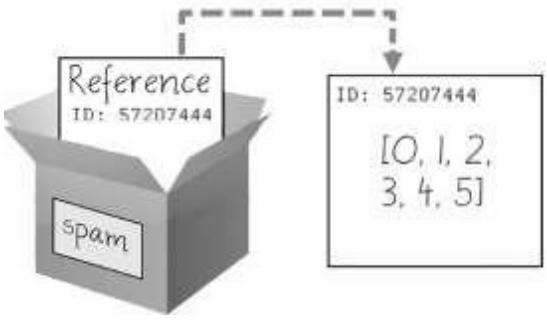


Figure: spam = [0, 1, 2, 3, 4, 5] stores a reference to a list, not the actual list.

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

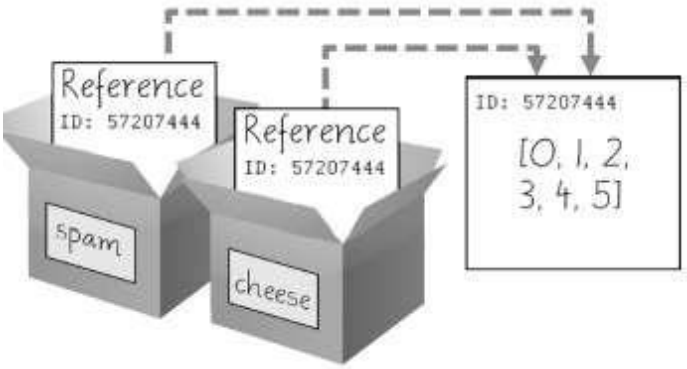


Figure: spam = cheese copies the reference, not the list

- 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.

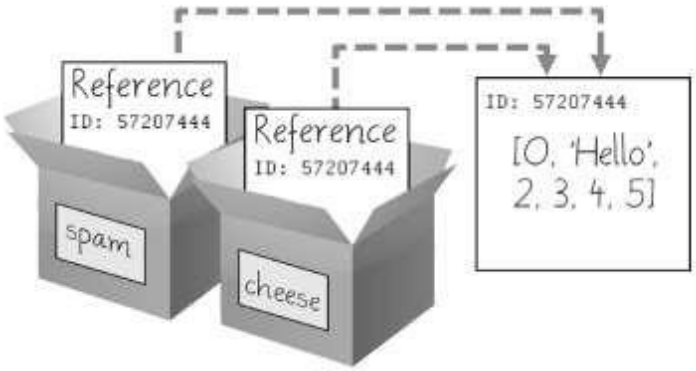


Figure: cheese[1] = 'Hello!' modifies the list that both variables refer to

- 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

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

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

Output

```
[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

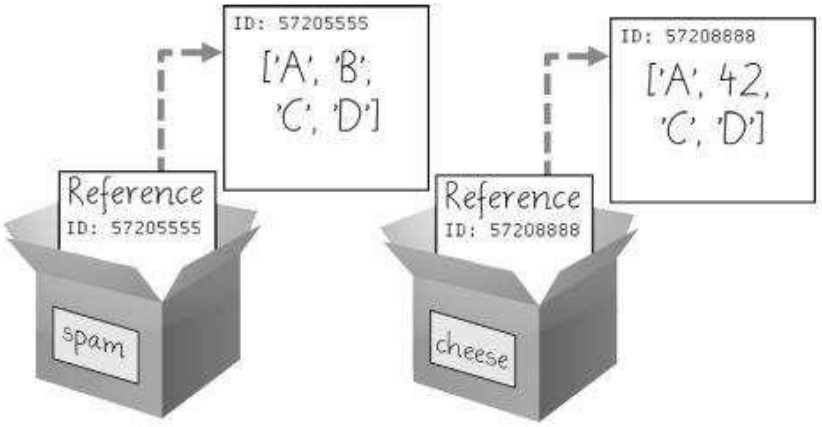


Figure: cheese = copy.copy(spam) creates a second list that can be modified independently of the first.

>>> **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'}
```

Dictionaries vs. Lists

- 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
```

The get() Method

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.'
```

The setdefault() Method

- 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.

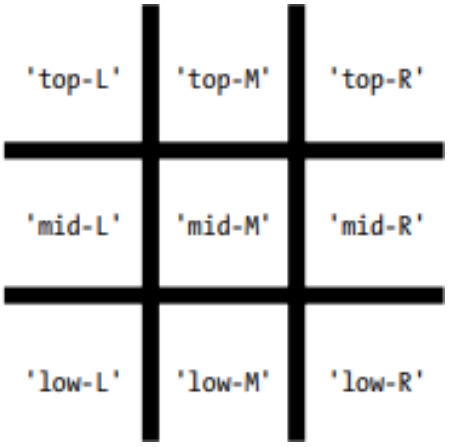


Figure: The slots of a tic-tactoe board with their corresponding keys

- 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.

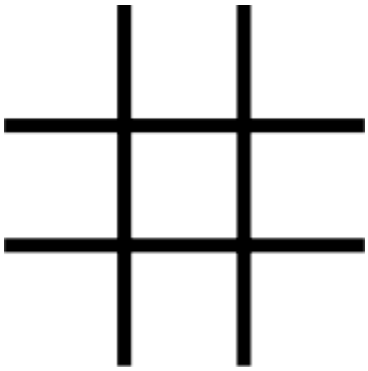


Figure: An empty tic-tac-toe board

- 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:

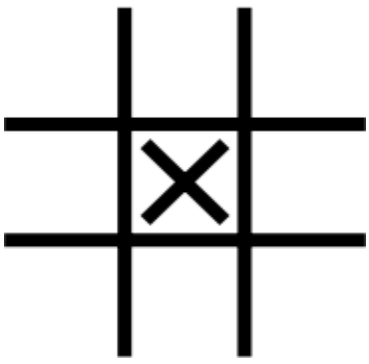


Figure: A first move

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

```
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'}
```

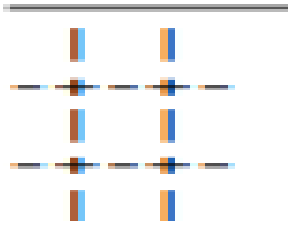
- The data structure in theBoard now represents tic-tac-toe board in the below Figure.

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.

```
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
```