**Automate the Boring Stuff with Python: Practical Programming for Total Beginners**

**ALBERT SWEIGART**

You will learn the basics of programming so that you can automate simple tasks such as the following:

* Moving and renaming thousands of files and sorting them into folders
* Filling out online forms, no typing required
* Downloading files or copy text from a website whenever it updates
* Having your computer text you custom notifications
* Updating or formatting Excel spreadsheets
* Checking your email and sending out prewritten responses

**Conventions**

This book is not designed as a reference manual; it’s a guide for beginners. The coding style sometimes goes against best practices (for example, some programs use global variables), but that’s a trade-off to make the code simpler to learn. This book is made for people to write throwaway code, so there’s not much time spent on style and elegance. Sophisticated programming concepts—like object-oriented programming, list comprehensions, and generators—aren’t covered because of the complexity they add. Veteran programmers may point out ways the code in this book could be changed to improve efficiency, but this book is mostly concerned with getting programs to work with the least amount of effort.

**About This Book**

The first part of this book covers basic Python programming concepts, and the second part covers various tasks you can have your computer

automate. Each chapter in the second part has project programs for you to study. Here’s a brief rundown of what you’ll find in each chapter:

**Part I**

Chapter 1. Covers expressions, the most basic type of Python instruction, and how to use the Python interactive shell software to experiment with code.

Chapter 2. Explains how to make programs decide which instructions to execute so your code can intelligently respond to different conditions.

Chapter 3. Instructs you on how to define your own functions so that you can organize your code into more manageable chunks.

Chapter 4. Introduces the list data type and explains how to organize data.

Chapter 5. Introduces the dictionary data type and shows you more powerful ways to organize data.

Chapter 6. Covers working with text data (called strings in Python).

**Part II**

Chapter 7. Covers how Python can manipulate strings and search for text patterns with regular expressions.

Chapter 8. Explains how your programs can read the contents of text files and save information to files on your hard drive.

Chapter 9. Shows how Python can copy, move, rename, and delete large numbers of files much faster than a human user can. It also explains compressing and decompressing files.

Chapter 10. Shows how to use Python’s various bug-finding and bugfixing tools.

Chapter 11. Shows how to write programs that can automatically download web pages and parse them for information. This is called web scraping.

Chapter 12. Covers programmatically manipulating Excel spreadsheets so that you don’t have to read them. This is helpful when the number of documents you have to analyze is in the hundreds or thousands.

Chapter 13. Covers programmatically reading Word and PDF documents.

Chapter 14. Continues to explain how to programmatically manipulate documents with CSV and JSON files.

Chapter 15. Explains how time and dates are handled by Python programs and how to schedule your computer to perform tasks at certain times. This chapter also shows how your Python programs can launch non-Python programs.

Chapter 16. Explains how to write programs that can send emails and text messages on your behalf.

Chapter 17. Explains how to programmatically manipulate images such as JPEG or PNG files.

Chapter 18. Explains how to programmatically control the mouse and keyboard to automate clicks and keypresses.

**Downloading and Installing Python**

You can download Python for Windows, OS X, and Ubuntu for free from http://python.org/downloads/. If you download the latest version from the website’s download page, all of the programs in this book should work.

**Starting IDLE**

While the Python interpreter is the software that runs your Python programs, the interactive development environment (IDLE) software is where you’ll enter your programs, much like a word processor.

Let’s start IDLE now. On Windows 7 or newer, click the Start icon in the lower-left corner of your screen, enter IDLE in the search box, and select IDLE (Python GUI).

**When asking programming questions, remember to do the following**:

Explain what you are trying to do, not just what you did. This lets your helper know if you are on the wrong track.

Specify the point at which the error happens. Does it occur at the very start of the program or only after you do a certain action?

Copy and paste the entire error message and your code to http://pastebin.com/ or http://gist.github.com/.

These websites make it easy to share large amounts of code with people over the Web, without the risk of losing any text formatting. You can then put the URL of the posted code in your email or forum post. For example, here some pieces of code I’ve posted: http://pastebin.com/SzP2DbFx/ and https://gist.github.com/asweigart/6912168/.

* Explain what you’ve already tried to do to solve your problem. This tells people you’ve already put in some work to figure things out on your own.
* List the version of Python you’re using. (There are some key differences between version 2 Python interpreters and version 3 Python interpreters.) Also, say which operating system and version you’re running.
* If the error came up after you made a change to your code, explain exactly what you changed. Say whether you’re able to reproduce the error every time you run the program or whether it happens only after you perform certain actions.
* Explain what those actions are, if so. Always follow good online etiquette as well.

**Chapter 1. Python Basics**

Entering Expressions into the Interactive Shell You run the interactive shell by launching IDLE, which you installed with Python in the introduction.

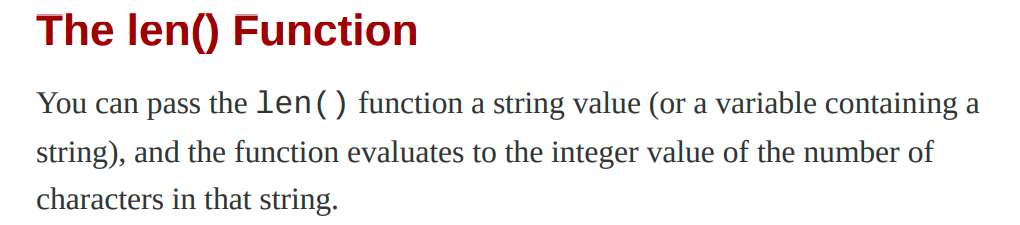
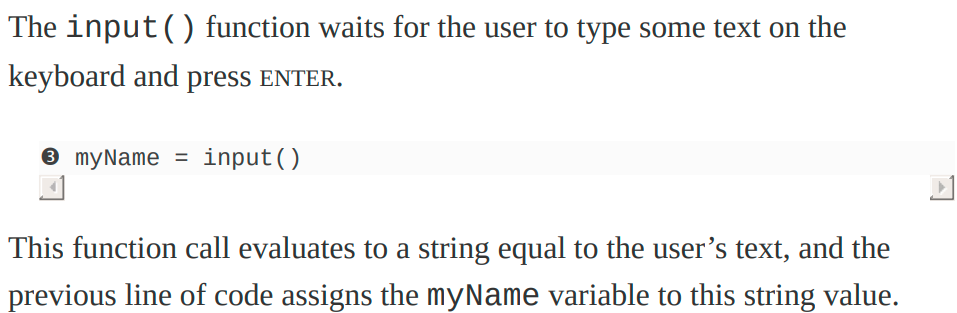
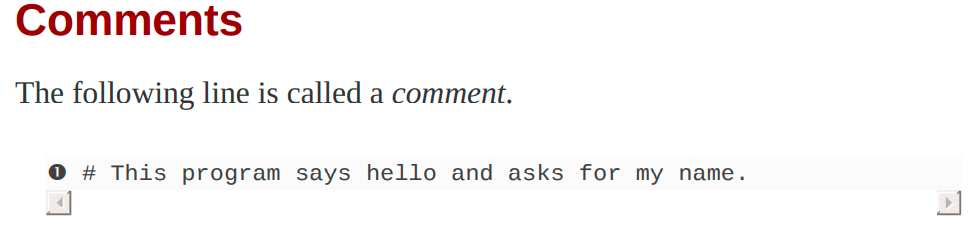
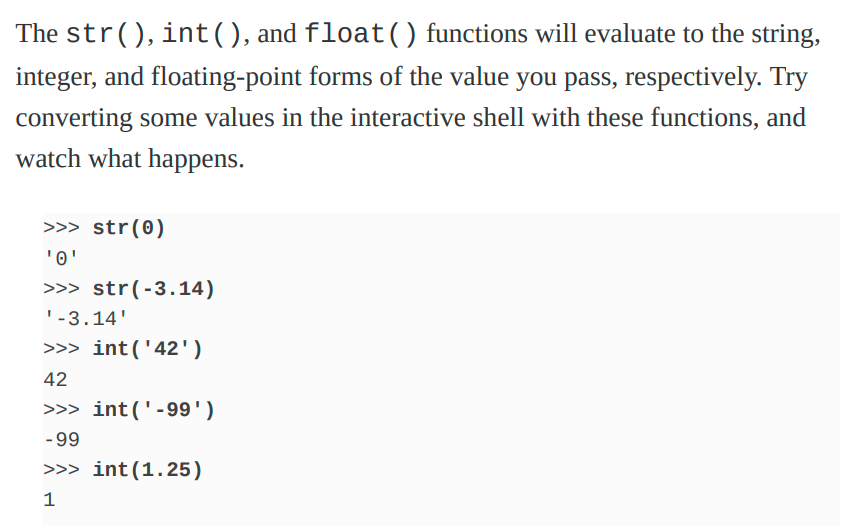
On Windows, open the Start menu, select All Programs ▸ Python 3.3, and then select IDLE (Python GUI).

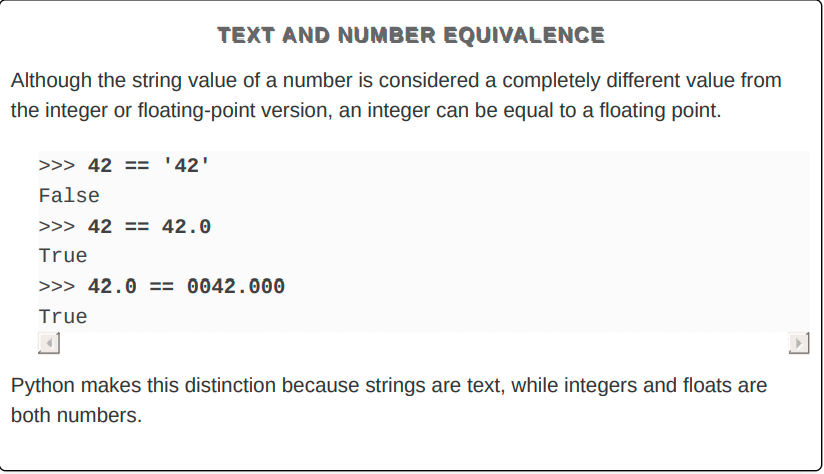
**Your First Program**

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 the file editor in IDLE, select File▸New Window.

Once you’ve entered your source code, save it so that you won’t have to retype it each time you start IDLE. From the menu at the top of the file editor window, select File▸Save As. In the Save As window, enter hello.py in the File Name field and then click Save.

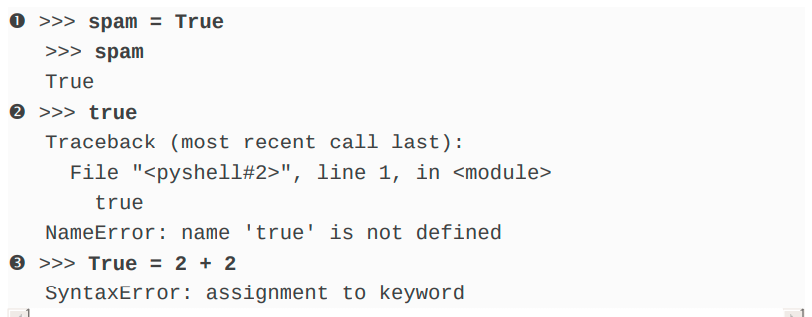
..

.

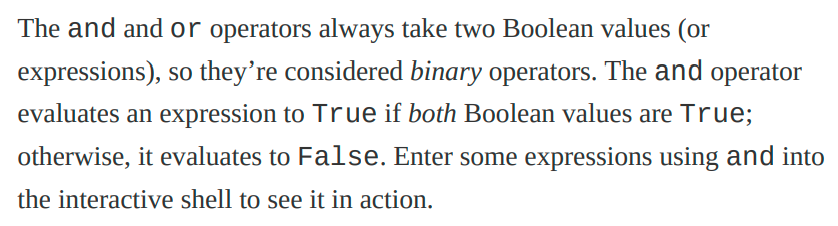
.

.

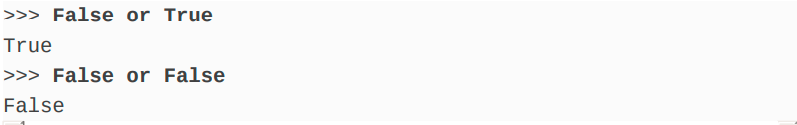
**Chapter 2. Flow Control**

**.**

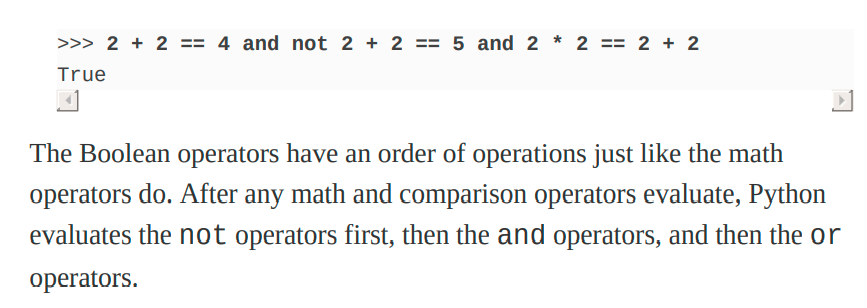
**Binary Boolean Operators**

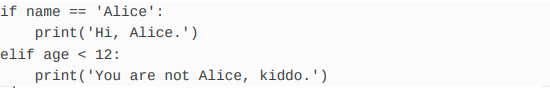
.

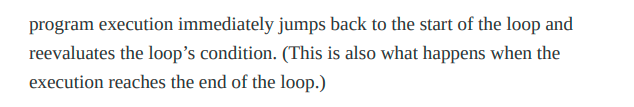
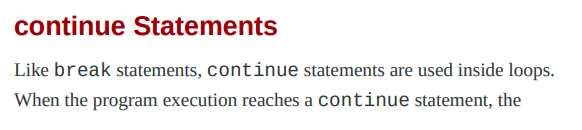
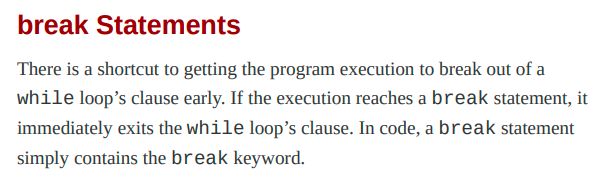
.

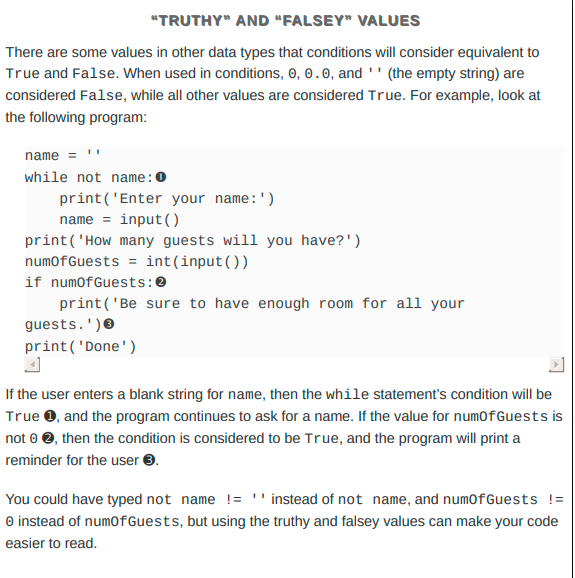
**The not Operator**

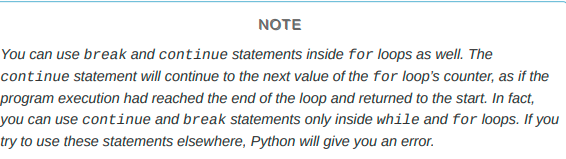
The not operator simply evaluates to the opposite Boolean value.

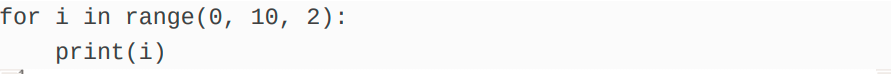
**Elif Statements** 

.

..mk

.

.

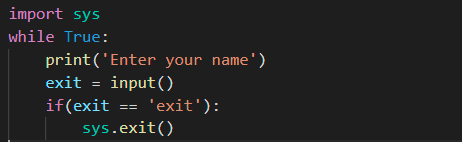
...

.

..

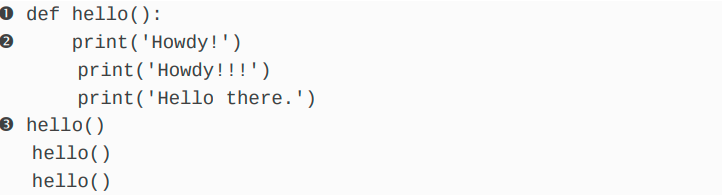
..

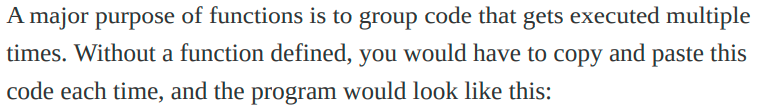
**Ending a Program Early with sys.exit() -**

.

..

**Chapter 3. Functions**

.

import random

def getAnswer(num):

if(num==1):

return "one"

elif(num==2):

return "two"

elif(num==3):

return "three"

r = random.randint(1,3)

print(getAnswer(r))

.

**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 None Type data type. (Other programming languages might call this value null, nil, or undefined.)

Just like the Boolean True and False values, None must be typed with a capital N.

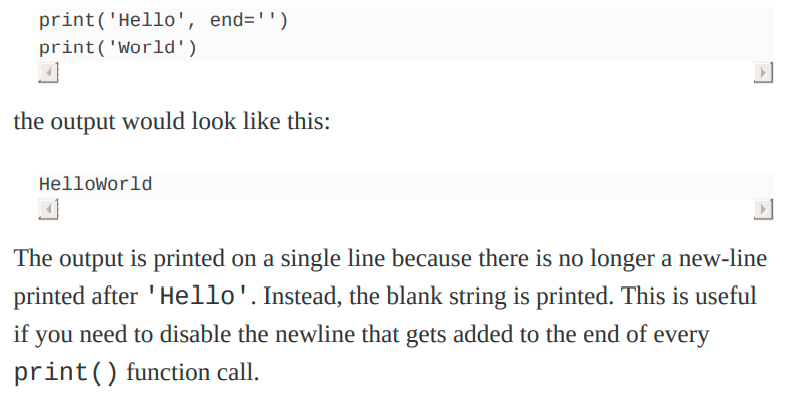
>>> spam = print('Hello!')

Hello!

>>> None == spam

True

Behind the scenes, Python adds return None to the end of any function definition with no return statement. This is similar to how a while or for loop implicitly ends with a continue statement. Also, if you use a return statement without a value (that is, just the return keyword by itself), then None is returned.

**Keyword Arguments and print() .**

**.**

Similarly, when you pass multiple string values to print(), the function will automatically separate them with a single space. Enter the following into the interactive shell:

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

**cats dogs mice.**

**.**

**def spam():**

**eggs = 'spam local'**

**print(eggs)**

**def bacon():**

**eggs = 'bacon local'**

**print(eggs)**

**spam()**

**print(eggs)**

**eggs = 'global'**

**bacon()**

**print(eggs)**

**# bacon local**

**# spam local**

**# bacon local**

**# global**

**.**

**The global Statement**

**def spam():**

**global eggs**

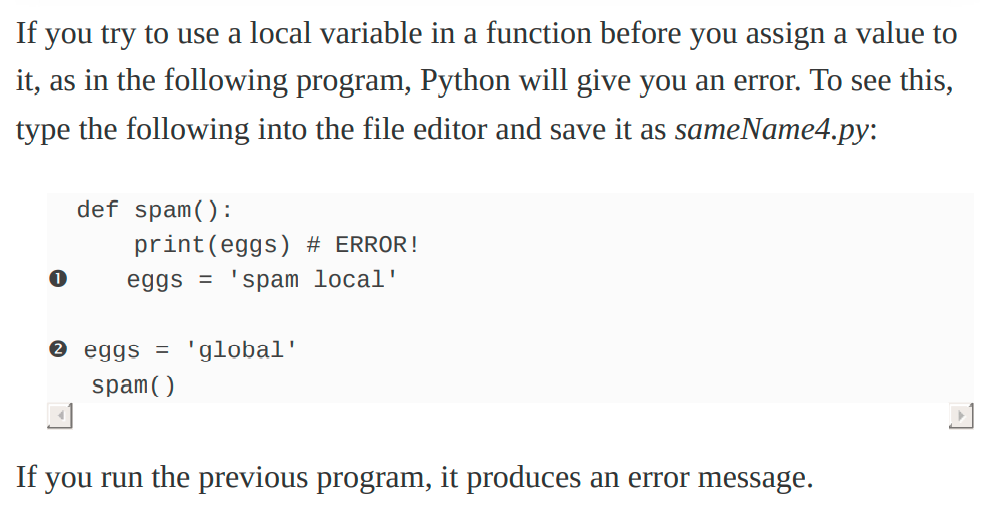
**eggs = 'spam'**

**eggs = 'global'**

**spam()**

**print(eggs) // #spam**

If you need to modify a global variable from within a function, use the global statement.



**Exception Handling**

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

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))

.

# 21.0

# 3.5

# Error: Invalid argument.

# None

# 42.0

.

NUMBER GUESSER GAME

import random

print('think of a number between 1 to 20')

print('you have 6 attempts')

ans = random.randint(1,20)

attempts =1

for attempts in range(1,7):

    user\_input = int(input())

    if(user\_input > ans):

        print("too high")

    elif(user\_input < ans):

        print("too low")

    else:

        break

    print('you have '+str(6- attempts)+' attempts')

if user\_input==ans:

    print('You guessed the number'+str(ans)+' correctly in '+  str(attempts) +' attempts')

else:

    print('You could not  guess the number '+str(ans))

.

CHECK INTEGER INPUT THROW ERROR –

try:

    user = int(input())

except ValueError :

    print('please enter an integer') // please enter an integer

**Chapter 4. Lists**

grocery = ['apple','mango','banana']

print(grocery) #['apple', 'mango', 'banana']

spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]

print(spam[0])

print(spam[0][1])

print(spam[1][4])

# ['cat', 'bat']

# bat

# 50

**.**

**Negative Indexes**

**Getting Sublists with Slices**

spam = ['cat', 'bat', 'rat', 'elephant']

print(spam[0:4])

#['cat', 'bat', 'rat', 'elephant']

print(spam[1:3])

#['bat', 'rat']

print(spam[0:-1])

#['cat', 'bat', 'rat']

**.**

spam = ['cat', 'bat', 'rat', 'elephant']

print(spam[:2])

#['cat', 'bat']

print(spam[1:])

#['bat', 'rat', 'elephant']

print(spam[:])

#['cat', 'bat', 'rat', 'elephant']

**.**

**Changing Values in a List with Indexes**

**List Concatenation and List Replication**

**Text

Description automatically generated.**

**.**

**.**

**.**

**Removing Values from Lists with del Statements**

**Graphical user interface

Description automatically generated with medium confidence.**

**Working with Lists**

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)

**..**

**Text

Description automatically generated.**

**The in and not in Operators**

**Text

Description automatically generated.**

**The Multiple Assignment Trick**

cat = ['fat', 'black', 'loud']

size, color, disposition = cat

print(cat) # ['fat', 'black', 'loud']

**.**

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

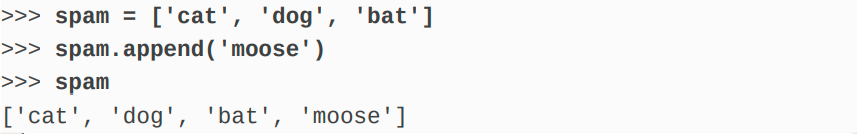
List values have an index() method that can be passed a value, and if that value exists in the list, the index of the value is returned.

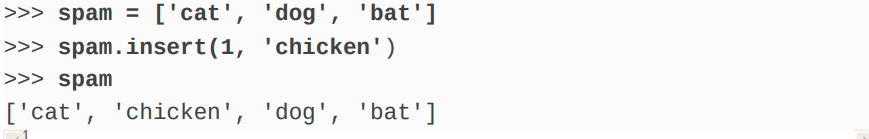
spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']

print(spam.index('Pooka')) # 1

.

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

**.**

**.**

**Removing Values from Lists with remove() , del()**

spam = ['cat', 'bat', 'rat', 'elephant']

spam.remove('bat')

print(spam) # ['cat', 'rat', 'elephant']

del(spam[0])

print(spam) # ['rat', 'elephant']

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

spam = [2, 5, 3.14, 1, -7]

spam.sort()

print(spam) #   [-7, 1, 2, 3.14, 5]

spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']

spam.sort()

print(spam) #   ['ants', 'badgers', 'cats', 'dogs', 'elephants']

spam.sort(reverse=True)

print(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 capture the return value by writing code like spam = spam.sort().

.

Third, sort() uses “ASCIIbetical order” rather than actual alphabetical order for sorting strings.

spam = ['Alice', 'ants', 'Bob', 'badgers', 'Carol', 'cats']

spam.sort()

print(spam) # ['Alice', 'Bob', 'Carol', 'ants', 'badgers', 'cats']

.

If you 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)

print(spam) # ['a', 'A', 'z', 'Z']

.

**Mutable and Immutable Data Types**

List is mutable. However, a string is immutable.

name = 'Zophie a cat'

newName = name[0:7] + 'the' + name[8:12]

print(name)

# 'Zophie a cat'

print(newName)

# 'Zophie the cat'

.

eggs = [1, 2, 3]

del eggs[2]

del eggs[1]

del eggs[0]

eggs.append(4)

eggs.append(5)

eggs.append(6)

print(eggs)

# [4, 5, 6]

**Tuples –**

First, tuples are typed with parentheses, (), instead of square brackets, [].

Tuples are immutable.

.

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

**.**

print(type(('hello',)))

print(type(('hello')))

# <class 'tuple'>

# <class 'str'>

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

tuple(['cat', 'dog', 5])

# ('cat', 'dog', 5)

list(('cat', 'dog', 5))

# ['cat', 'dog', 5]

list('hello')

# ['h', 'e', 'l', 'l', 'o']

.

**References**

When you assign a list to a variable, you 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.

spam = [0, 1, 2, 3, 4, 5]

cheese = spam

cheese[1] = 'Hello!'

spam

# [0, 'Hello!', 2, 3, 4, 5]

cheese

# [0, 'Hello!', 2, 3, 4, 5]

.

**Passing References**

When a function is called, the values of the arguments are copied to the parameter variables. For lists (and dictionaries), this means a copy of the reference is used for the parameter.

def eggs(someParameter):

    someParameter.append('Hello')

spam = [1, 2, 3]

eggs(spam)

print(spam)

#[1, 2, 3, 'Hello']

.

Keep this behavior in mind: Forgetting that Python handles list and dictionary variables this way can lead to confusing bugs.

.

**The copy Module’s copy() and deepcopy() Functions**

If the function modifies the list or dictionary that is passed, you 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.

import copy

spam = ['A', 'B', 'C', 'D']

cheese = copy.copy(spam)

cheese[1] = 42

print(spam)

# ['A', 'B', 'C', 'D']

print(cheese)

# ['A', 42, 'C', 'D']

.

If the list you need to copy contains lists, then use the copy.deepcopy() function instead of copy.copy(). The deepcopy() function will copy these inner lists as well.

.

**SUMMARY –**

Lists are mutable, meaning that their contents can change. Tuples and strings, although list-like in some respects, are immutable and cannot be changed. A variable that contains a tuple or string value can be overwritten with a new tuple or string value, but this is not the same thing as modifying the existing value in place—like, say, the append() or remove() methods do on lists.

.

.

**Chapter 5. Dictionaries and Structuring Data**

**.**

Indexes for dictionaries are called keys, and a key with its associated value is called a key-value pair.

In code, a dictionary is typed with braces, {}.

myCat = {'size': 'fat', 'color': 'gray', 'disposition':

'loud'}

print(myCat)

print(myCat['color'])

**Dictionaries vs. Lists**

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']

print(spam == bacon) #False

eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}

ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}

print(eggs == ham) #True

**.**

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

spam = {'color': 'red', 'age': 42}

for k in spam.keys():

     print(k)

for v in spam.values():

     print(v)

for i in spam.items():

    print(i)

# color

# age

# red

# 42

# ('color', 'red')

# ('age', 42)

**.**

for k in spam.keys():

     print(spam[k])

**.**

**Converting to list , tuples**

spam = {'color': 'red', 'age': 42}

print(list(spam.keys()))

print(tuple(spam.keys()))

for k , v in spam.items():

    print('Key : ' + k + " : " + str(v))

.

**Checking Whether a Key or Value Exists in a Dictionary**

spam = {'color': 'red', 'age': 42}

print('color' in spam.keys()) #True

print('color' in spam.values()) #True

.

**The get() Method**

It’s tedious to check whether a key exists in a dictionary before accessing that key’s value.

Fortunately, 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}

print('I am bringing ' + str(picnicItems.get('cups', -1)) + ' cups.')

.

**The setdefault() Method**

You’ll often have 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}

spam.setdefault('color', 'black')

print(spam) #{'name': 'Pooka', 'age': 5, 'color': 'black'}

spam.setdefault('color', 'white')

print(spam) #{'name': 'Pooka', 'age': 5, 'color': 'black'}

.

The **setdefault() method**

is a nice shortcut to ensure that a key exists. Here is a short program that counts the number of occurrences of each letter in a string.

.

The **setdefault()** method is a nice shortcut to ensure that a key exists.

Here is a short program that counts the number of occurrences of each letter in a string

**Pretty Printing** If you import the pprint module into your programs, you’ll have access to the pprint() and pformat() functions that will “pretty print” a dictionary’s values.

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)

.

**Using Data Structures to Model Real-World Things**

**.**

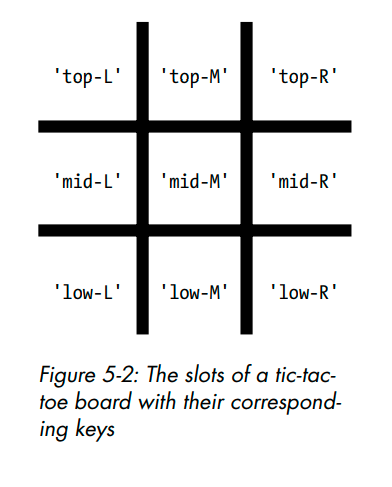
**A Tic-Tac-Toe Board**

theBoard =

{'top-L': ' ', 'top-M': ' ', 'top-R': ' ',

'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',

'low-L': ' ', 'low-M': ' ', 'low-R': ' '}

****.

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)

**.**

turn = 'X'

for i in range(9):

 printBoard(theBoard)

 print('Turn for ' + turn + '. Move on which space?')

 move = input()

 theBoard[move] = turn

 if turn == 'X':

    turn = 'O'

 else:

    turn = 'X'

**.**

**Nested Dictionaries and Lists**

As you model more complicated things, you may find you need dictionaries and lists that contain.

Lists are useful to contain an ordered series of values, and dictionaries are useful for associating keys with values.

For example, here’s a program that uses a dictionary that contains other dictionaries in order to see who is bringing what to a picnic.

The totalBrought() function can read this data structure and calculate the total number of an item being brought by all the guests.

.

allGuests = {'Alice': {'apples': 5, 'pretzels': 12},

            'Bob': {'ham sandwiches': 3, 'apples': 2},

            'Carol': {'cups': 3, 'apple pies': 1}}

def totalBrought(guests, item):

    numBrought = 0

    for k, v in guests.items():

        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')))

**.**

**.**

**.**

**Chapter 6. Manipulating Strings**

**.**

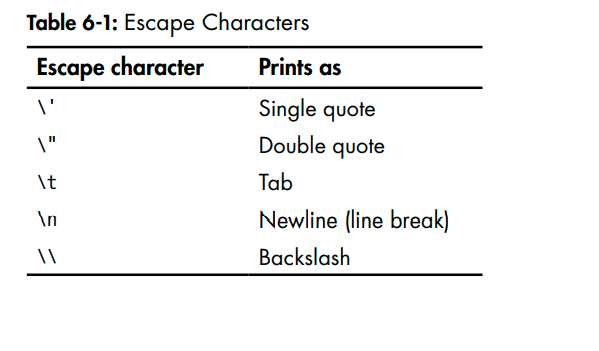
**String Literals**

Using single quotes & double quotes.

spam = "That is Alice's cat."

Escape Character

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

****

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

**Raw Strings**

You can place an r before the beginning quotation mark of a string to make it a raw string.

A raw string completely ignores all escape characters and prints any backslash that appears in the string.

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

.

**Multiline Strings with Triple Quotes**

Python’s indentation rules for blocks do not apply to lines inside a multiline string.

print('''Dear Alice,

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

Sincerely,

Bob''')

**.**

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

**.**

**Multiline Comments**

While the hash character (#) marks the beginning of a comment for the rest of the line, a multiline string is often used for comments that span multiple lines.

"""This is a test Python program.

Written by Al Sweigart al@inventwithpython.com

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

"""

**.**

**Indexing and Slicing Strings**

Note that slicing a string does not modify the original string. You can capture a slice from one variable in a separate variable.

spam = 'Hello world!'

fizz = spam[0:5]

print(fizz) # Hello

**The in and not in Operators with String**

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

print('Hello' in 'Hello World') # True

print('Hello' in 'Hello') # False

print('HELLO' in 'Hello World') # False

**Useful String Method**

The upper(), lower(), isupper(), islower().

print('Hello'.upper())

#'HELLO'

print('Hello'.upper().lower())

#'hello'

**.**

**The isX String Method**

* isalpha() returns True if the string consists only of letters and is not blank.
* isalnum() returns True if the string consists only of letters and numbers and is not blank.
* isdecimal() returns True if the string consists only of numeric characters and is not blank.
* isspace() returns True if the string consists only of spaces, tabs, and new lines and is not blank.
* istitle() returns True if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

**A LOGIN VALIDATOR**

while True:

    print('Enter your age:')

    age = input()

    if age.isdecimal():

        break

    print('Age can only be a number')

while True:

    print('Enter your password (only alphanumeric)')

    password = input()

    if password.isalnum():

        break;

    print('A password can only be alphanumeric.')

.

**The startswith() and endswith() String Methods**

**The join() and split() String Methods**

The join() method is useful when you have a list of strings that need to be joined together into a single string value.

The join() method is called on string , gets passed a list of strings, and returns a string.

', '.join(['cats', 'rats', 'bats'])

#'cats, rats, bats'

' '.join(['My', 'name', 'is', 'Simon'])

#'My name is Simon'

'ABC'.join(['My', 'name', 'is', 'Simon'])

#'MyABCnameABCisABCSimon'

**.**

**The split() method**

does the opposite: It’s called on a string value and returns a list of strings.

print('My name is Simon'.split())

#['My', 'name', 'is', 'Simon']

'MyABCnameABCisABCSimon'.split('ABC')

#['My', 'name', 'is', 'Simon']

'My name is Simon'.split('m')

#['My na', 'e is Si', 'on']

spam = '''Dear Alice,

How have you been? I am fine.

There is a container in the fridge

that is labeled "Milk Experiment".

Please do not drink it.

Sincerely,

Bob'''

spam.split('\n')

'''['Dear Alice,', 'How have you been? I am fine.', 'There is a container in the

fridge', 'that is labeled "Milk Experiment".', '', 'Please do not drink it.',

'Sincerely,', 'Bob']'''

**.**

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

The rjust() and ljust() string methods return a padded version of the string they are called on, with spaces inserted to justify the text.

The first argument to both methods is an integer length for the justified string.

print('Hello'.rjust(10))

#     Hello

print('Hello'.rjust(20))

#               Hello

print('Hello World'.ljust(30)+'!')

#Hello World                   !

'Hello'.rjust(10) says that we want to right-justify 'Hello' in a string of total length 10. 'Hello' is five characters, so five spaces will be added to its left, giving us a string of 10 characters with 'Hello' justified right.

.

An optional second argument to rjust() and ljust() will specify a fill

character other than a space character. Enter the following into the interactive shell:

print('Hello'.rjust(20, '\*'))

#'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Hello'

print('Hello'.ljust(20, '-'))

#'Hello---------------'

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

.

print('Hello'.center(20, '='))

#'=======Hello========'

**.**

These methods are especially useful when you need to print tabular data that has the correct spacing.

.

def printPicnic(itemsDict,leftWidth,rightWidth):

    print('PICNIC ITEMS'.center(leftWidth+rightWidth,'-'))

    for k, v in itemsDict.items():

        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))

picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4, 'cookies': 8000}

printPicnic(picnicItems, 12, 6)

'''---PICNIC ITEMS---

sandwiches..     4

apples......    12

cups........     4

cookies.....  8000'''

**.**

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

Sometimes you may want to strip off whitespace characters (space, tab, and newline) from the left side, right side, or both sides of a string. The strip() string method will return a new string without any whitespace

.

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

spam = 'SpamSpamBaconSpamEggsSpamSpam'

spam.strip('ampS')

'BaconSpamEggs'

Passing strip() the argument 'ampS' will tell it to strip occurences of a, m, p, and capital S from the ends of the string stored in spam.

The order of the characters in the string passed to strip() does not matter: strip('ampS') will do the same thing as strip('mapS') or strip('Spam').

**Copying and Pasting Strings with the pyperclip Module**

The pyperclip module has copy() and paste() functions that can send text to and receive text from your computer’s clipboard. Sending the output of your program to the clipboard will make it easy to paste it to an email, word processor, or some other software.

Pyperclip does not come with Python. To install it, follow the directions for installing third-party modules in Appendix A.{ pip install pyperclip}

import pyperclip

pyperclip.copy('Hello world!')

print(pyperclip.paste())

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

.