Python Cheatsheet

Table of Content:

Python Basics

Math Operators

From **Higuest** to **Lowest** precedence:

Operators	Operation	Example
**	Exponent	2 ** 3 = 8

%	Modulus/Remaider	22 % 8 = 16
//	Integer division	22 // 8 = 2
/	Division	22 / 8 = 2.75
*	Multiplication	3 * 3 = 15
-	Subtraction	5 - 2 = 3
+	Addition	2 + 2 = 4

Examples of expressions in the interactive shell:

```
>>> 2 + 3 * 6
20

>>> (2 + 3) * 6
30

>>> 2 ** 8
256

>>> 23 // 7
3

>>> 23 % 7
2

>>> (5 - 1) * ((7 + 1) / (3 - 1))
16.0
```

Return to the Index

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'

Return to the Index

String Concatenation and Replication

String concatenation:

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

String Replication:

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

Return to the Index

Variables

You 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.

Example:

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

Return to the Index

Comments

Inline comment:

```
# This is a comment
```

Multiline Comment:

```
"""
This is a Multiline Comment
You can also use:
''' multiline comment '''
"""
```

Return to the Index

The print() Function

```
>>> print('Hello world!')
Hello world!
```

Return to the Index

The input() Function

Example Code:

```
>>> print('What is your name?')    # ask for their name
>>> myName = input()
>>> print('It is good to meet you, ' + myName)
```

Output:

```
What is your name?
Al
It is good to meet you, Al
```

Return to the Index

The len() Function

Evaluates to the integer value of the number of characters in a string:

```
>>> len('hello')
5
```

Return to the Index

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

Convert Between Data Types:

Integer to String or Float:

```
>>> str(29)
'29'

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

```
I am 29 years old.

>>> str(-3.14)
'-3.14'
```

Float to Integer:

```
>>> int(7.7)
7

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

Return to the Index

Flow Control

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 you give them:

Examples:

```
>>> 42 == 42
True

>>> 40 == 42
False

>>> 'hello' == 'hello'
True

>>> 'hello' == 'Hello'
False
```

```
>>> 'dog' != 'cat'
True

>>> True == True
True

>>> True != False
True

>>> 42 == 42.0
True

>>> 42 == '42'
False
```

Return to the Index

Boolean Operators

There are three Boolean operators: and, or, and not.

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

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

The *not* Operator's *Truth* Table:

Expression	Evaluates to
not True	False
not False	True

Return to the Index

Mixing Boolean and Comparison Operators

```
>>> (4 < 5) and (5 < 6)
True

>>> (4 < 5) and (9 < 6)
False

>>> (1 == 2) or (2 == 2)
True
```

You 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
```

Return to the Index

if Statements

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

Return to the Index

else Statements

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

Return to the Index

elif Statements

```
name = 'Bob'
age = 5
if name == 'Alice':
```

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

```
name = 'Bob'
age = 30
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.')
```

Return to the Index

while Loop Statements

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

Return to the Index

break Statements

If the execution reaches a break statement, it immediately exits the while loop's clause.

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

Return to the Index

continue Statements

When the program execution reaches a continue statement, the program execution immediately jumps back to the start of the loop.

```
while True:
```

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

Return to the Index

for Loops and the range() Function

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

Output:

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

The *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)
```

Output:

```
0
2
4
6
8
```

You 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)
```

Output:

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

Return to the Index

Importing Modules

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

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

```
from random import *.
```

Return to the Index

Ending a Program Early with sys.exit()

```
import sys

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

Return to the Index

Functions

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

hello('Alice')
hello('Bob')
```

Output:

```
Hello Alice
Hello Bob
```

Return to the Index

Return Values and return Statements

When creating a function using the def statement, you can specify what the return value should be with a return statement. A return statement consists of the following:

- The return keyword.
- The value or expression that the function should return.

```
import random
def getAnswer(answerNumber):
    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'
```

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

Return to the Index

The None Value

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

>>> None == spam
True
```

Return to the Index

Keyword Arguments and print()

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

Output:

```
HelloWorld
```

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

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

Return to the Index

Local and Global Scope

- Code in the global scope cannot use any local variables.
- However, a local scope can access global variables.

- Code in a function's local scope cannot use variables in any other local scope.
- You 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.

Return to the Index

The global Statement

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

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

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

Output:

```
spam
```

There are four rules to tell whether a variable is in a local scope or global scope:

1. If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
2. If there is a global statement for that variable in a function, it is a global variable.
3. Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
4. But if the variable is not used in an assignment statement, it is a global variable.

Return to the Index

Exception Handling

```
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
```

Return to the Index

Lists

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

>>> spam
['cat', 'bat', 'rat', 'elephant']
```

Return to the Index

Getting Individual Values in a List with Indexes

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

>>> spam[0]
'cat'

>>> spam[1]
'bat'

>>> spam[2]
'rat'

>>> spam[3]
'elephant'
```

Return to the Index

Negative Indexes

```
>>> 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.'
```

Return to the Index

Getting Sublists with Slices

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

>>> spam[0:4]
['cat', 'bat', 'rat', 'elephant']

>>> spam[1:3]
['bat', 'rat']

>>> spam[0:-1]
['cat', 'bat', 'rat']
```

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

>>> spam[:2]
['cat', 'bat']

>>> spam[1:]
['bat', 'rat', 'elephant']

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

Return to the Index

Getting a List's Length with len()

```
>>> spam = ['cat', 'dog', 'moose']

>>> len(spam)
3
```

Return to the Index

Changing Values in a List with Indexes

```
>>> 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]
```

Return to the Index

List Concatenation and List Replication

```
>>> [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']
```

Return to the Index

Removing Values from Lists with del Statements

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

>>> del spam[2]

>>> spam
['cat', 'bat', 'elephant']
```

```
>>> del spam[2]

>>> spam
['cat', 'bat']
```

Return to the Index

Using for Loops with Lists

```
>>> supplies = ['pens', 'staplers', 'flame-throwers', 'binders']

>>> for i in range(len(supplies)):
        print('Index ' + str(i) + ' in supplies is: ' + supplies[i])
```

Output:

```
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
```

Return to the Index

The in and not in Operators

```
>>> '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
```

Return to the Index

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. So instead of doing this:

```
>>> cat = ['fat', 'orange', 'loud']

>>> size = cat[0]

>>> color = cat[1]

>>> disposition = cat[2]
```

you could type this line of code:

```
>>> cat = ['fat', 'orange', 'loud']

>>> size, color, disposition = cat
```

The multiple assignment trick can also be used to swap the values in two variables:

```
>>> a, b = 'Alice', 'Bob'

>>> a, b = b, a

>>> print(a)
'Bob'

>>> print(b)
'Alice'
```

Return to the Index

Augmented Assignment Operators

Operator	Equivalent
spam += 1	spam = spam + 1
spam -= 1	spam = spam - 1
spam *= 1	spam = spam * 1
spam /= 1	spam = spam / 1
spam %= 1	spam = spam % 1

Examples:

```
>>> spam = 'Hello'
>>> spam += ' world!'
>>> spam
'Hello world!'

>>> bacon = ['Zophie']
>>> bacon *= 3
>>> bacon
['Zophie', 'Zophie', 'Zophie']
```

Return to the Index

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

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

>>> spam.index('Pooka')
1
```

Return to the Index

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

append():

```
>>> spam = ['cat', 'dog', 'bat']

>>> spam.append('moose')

>>> spam
['cat', 'dog', 'bat', 'moose']
```

insert():

```
>>> spam = ['cat', 'dog', 'bat']

>>> spam.insert(1, 'chicken')

>>> spam
['cat', 'chicken', 'dog', 'bat']
```

Return to the Index

Removing Values from Lists with remove()

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

>>> spam.remove('bat')

>>> spam
['cat', 'rat', 'elephant']
```

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

Return to the Index

Sorting the Values in a List 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']
```

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)

>>> spam
```

```
['a', 'A', 'z', 'Z']
```

Return to the Index

Tuple Data Type

```
>>> eggs = ('hello', 42, 0.5)

>>> eggs[0]
'hello'

>>> eggs[1:3]
(42, 0.5)

>>> len(eggs)
3
```

The main way that tuples are different from lists is that tuples, like strings, are immutable.

Return to the Index

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']
```

Return to the Index

Dictionaries and Structuring Data

Example Dictionary:

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

Return to the Index

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

values():

```
>>> spam = {'color': 'red', 'age': 42}

>>> for v in spam.values():
        print(v)
```

Output:

```
red
42
```

keys():

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

Output:

```
color
age
```

items():

```
>>> for i in spam.items():
        print(i)
```

Output:

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

Using the keys(), values(), and items() methods, a for loop can iterate over the keys, values, or key-value pairs in a dictionary, respectively

```
>>> spam = {'color': 'red', 'age': 42}
```

```
>>> for k, v in spam.items():
        print('Key: ' + k + ' Value: ' + str(v))
```

Output:

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

Return to the Index

Checking Whether a 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
```

Return to the Index

The get() Method

```
>>> 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.'
```

Return to the Index

The setdefault() Method

```
spam = {'name': 'Pooka', 'age': 5}

if 'color' not in spam:
    spam['color'] = 'black'
```

The above code is equal to:

```
>>> 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'}
```

Return to the Index

Pretty Printing

```
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}
```

Return to the Index

Manipulating Strings

Escape Characters

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

Example:

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

Output:

```
Hello there!
How are you?
I'm doing fine.
```

Return to the Index

Raw Strings

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

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

Output:

```
That is Carol\'s cat.
```

Return to the Index

Multiline Strings with Triple Quotes

```
print('''Dear Alice,

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

Sincerely,
Bob''')
```

Output:

```
Dear Alice,

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

Sincerely,
Bob
```

Return to the Index

Indexing and Slicing Strings

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

```
>>> spam = 'Hello world!'
```

```
>>> spam[0]
'H'

>>> spam[4]
'o'

>>> spam[-1]
'!'

>>> spam[0:5]
'Hello'

>>> spam[:5]
'Hello'

>>> spam[6:]
'world!'
```

Slicing:

```
>>> spam = 'Hello world!'

>>> fizz = spam[0:5]

>>> fizz
'Hello'
```

Return to the Index

The in and not in Operators with Strings

```
>>> 'Hello' in 'Hello World'
True

>>> 'Hello' in 'Hello'
True

>>> 'HELLO' in 'Hello World'
False

>>> '' in 'spam'
True

>>> 'cats' not in 'cats and dogs'
False
```

Return to the Index

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

upper() and lower():

```
>>> spam = 'Hello world!'

>>> spam = spam.upper()

>>> spam
'HELLO WORLD!'

>>> spam = spam.lower()

>>> spam
'hello world!'
```

isupper() and islower():

```
>>> spam = 'Hello world!'

>>> spam.islower()
False

>>> spam.isupper()
False

>>> 'HELLO'.isupper()
True

>>> 'abc12345'.islower()
True

>>> '12345'.islower()
False

>>> '12345'.isupper()
False
```

Return to the Index

The isX String Methods

- **isalpha()** returns True if the string consists only of letters and is not blank.
- **isalnum()** returns True if the string consists only of lettersand numbers and is not blank.
- **isdecimal()** returns True if the string consists only ofnumeric 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 wordsthat begin with an uppercase letter followed by onlylowercase letters.

Return to the Index

The startswith() and endswith() String Methods

```
>>> 'Hello world!'.startswith('Hello')
True

>>> 'Hello world!'.endswith('world!')
True

>>> 'abc123'.startswith('abcdef')
False

>>> 'abc123'.endswith('12')
False

>>> 'Hello world!'.startswith('Hello world!')
True

>>> 'Hello world!'.endswith('Hello world!')
True
```

Return to the Index

The join() and split() String Methods

join():

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

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

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

split():

```
>>> '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']
```

Return to the Index

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

rjust() and ljust():

```
>>> 'Hello'.rjust(10)
'     Hello'

>>> 'Hello'.rjust(20)
'               Hello'

>>> 'Hello World'.rjust(20)
'         Hello World'

>>> 'Hello'.ljust(10)
'Hello     '
```

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:

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

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

center():

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

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

Return to the Index

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

```
>>> spam = '    Hello World     '

>>> spam.strip()
'Hello World'

>>> spam.lstrip()
'Hello World '

>>> spam.rstrip()
'    Hello World'
```

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

'BaconSpamEggs'
```

Return to the Index

Copying and Pasting Strings with the pyperclip Module

```
>>> import pyperclip

>>> pyperclip.copy('Hello world!')

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

Return to the Index

Regular Expressions

1. Import the regex module with import re.
2. Create a Regex object with the re.compile() function. (Remember to use a raw string.)
3. Pass the string you want to search into the Regex object's search() method. This returns a Match object.
4. Call the Match object's group() method to return a string of the actual matched text.

All the regex functions in Python are in the re module:

```
>>> import re
```

Return to the Index

Matching Regex Objects

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')

>>> mo = phoneNumRegex.search('My number is 415-555-4242.')

>>> print('Phone number found: ' + mo.group())
Phone number found: 415-555-4242
```

Return to the Index

Grouping with Parentheses

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')

>>> mo = phoneNumRegex.search('My number is 415-555-4242.')

>>> mo.group(1)
'415'

>>> mo.group(2)
'555-4242'

>>> mo.group(0)
'415-555-4242'

>>> mo.group()
'415-555-4242'
```

To retrieve all the groups at once: groups() method—note the plural form for the name.

```
>>> mo.groups()
('415', '555-4242')

>>> areaCode, mainNumber = mo.groups()

>>> print(areaCode)
415

>>> print(mainNumber)
555-4242
```

Return to the Index

Matching Multiple Groups with the Pipe

The | character is called a pipe. You can use it anywhere you want to match one of many expressions. For example, the regular expression r'Batman|Tina Fey' will match either 'Batman' or 'Tina Fey'.

```
>>> heroRegex = re.compile (r'Batman|Tina Fey')

>>> mo1 = heroRegex.search('Batman and Tina Fey.')

>>> mo1.group()
'Batman'

>>> mo2 = heroRegex.search('Tina Fey and Batman.')

>>> mo2.group()
'Tina Fey'
```

You can also use the pipe to match one of several patterns as part of your regex:

```
>>> batRegex = re.compile(r'Bat(man|mobile|copter|bat)')

>>> mo = batRegex.search('Batmobile lost a wheel')

>>> mo.group()
'Batmobile'

>>> mo.group(1)
'mobile'
```

Return to the Index

Optional Matching with the Question Mark

The ? character flags the group that precedes it as an optional part of the pattern.

```
>>> batRegex = re.compile(r'Bat(wo)?man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
```

Return to the Index

Matching Zero or More with the Star

The * (called the star or asterisk) means “match zero or more”—the group that precedes the star can occur any number of times in the text.

```
>>> batRegex = re.compile(r'Bat(wo)*man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'

>>> mo3 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo3.group()
'Batwowowowoman'
```

Return to the Index

Matching One or More with the Plus

While * means “match zero or more,” the + (or plus) means “match one or more”. The group preceding a plus must appear at least once. It is not optional:

```
>>> batRegex = re.compile(r'Bat(wo)+man')
>>> mo1 = batRegex.search('The Adventures of Batwoman')
>>> mo1.group()
'Batwoman'

>>> mo2 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo2.group()
'Batwowowowoman'

>>> mo3 = batRegex.search('The Adventures of Batman')
>>> mo3 == None
True
```

Return to the Index

Matching Specific Repetitions with Curly Brackets

If you have a group that you want to repeat a specific number of times, follow the group in your regex with a number in curly brackets. For example, the regex (Ha){3} will match the string 'HaHaHa', but it will not match 'HaHa', since the latter has only two repeats of the (Ha) group.

Instead of one number, you can specify a range by writing a minimum, a comma, and a maximum in between the curly brackets. For example, the regex (Ha){3,5} will match 'HaHaHa', 'HaHaHaHa', and 'HaHaHaHaHa'.

```
>>> haRegex = re.compile(r'(Ha){3}')
>>> mo1 = haRegex.search('HaHaHa')
>>> mo1.group()
'HaHaHa'

>>> mo2 = haRegex.search('Ha')
>>> mo2 == None
True
```

Return to the Index

Greedy and Nongreedy Matching

Python's regular expressions are greedy by default, which means that in ambiguous situations they will match the longest string possible. The non-greedy version of the curly brackets, which matches the shortest string possible, has the closing curly bracket followed by a question mark.

```
>>> greedyHaRegex = re.compile(r'(Ha){3,5}')
>>> mo1 = greedyHaRegex.search('HaHaHaHaHa')
>>> mo1.group()
'HaHaHaHaHa'

>>> nongreedyHaRegex = re.compile(r'(Ha){3,5}?')
>>> mo2 = nongreedyHaRegex.search('HaHaHaHaHa')
>>> mo2.group()
'HaHaHa'
```

Return to the Index

The findall() Method

In addition to the search() method, Regex objects also have a findall() method. While search() will return a Match object of the first matched text in the searched string, the findall() method will return the strings of every match in the searched string.

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # has no groups

>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
['415-555-9999', '212-555-0000']
```

To summarize what the findall() method returns, remember the following:

- When called on a regex with no groups, such as \d-\d\d\d-\d\d\d\d, the method findall() returns a list of ng matches, such as ['415-555-9999', '212-555-0000'].

- When called on a regex that has groups, such as (\d\d\d)-d\d)-(\d\ d\d\d), the method findall() returns a list of es of strings (one string for each group), such as [('415', ', '9999'), ('212', '555', '0000')].

Return to the Index

Making Your Own Character Classes

There are times when you want to match a set of characters but the shorthand character classes (\d, \w, \s, and so on) are too broad. You can define your own character class using square brackets. For example, the character class [aeiouAEIOU] will match any vowel, both lowercase and uppercase.

```
>>> vowelRegex = re.compile(r'[aeiouAEIOU]')

>>> vowelRegex.findall('Robocop eats baby food. BABY FOOD.')
['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

You can also include ranges of letters or numbers by using a hyphen. For example, the character class [a-zA-Z0-9] will match all lowercase letters, uppercase letters, and numbers.

By placing a caret character (^) just after the character class's opening bracket, you can make a negative character class. A negative character class will match all the characters that are not in the character class. For example, enter the following into the interactive shell:

```
>>> consonantRegex = re.compile(r'[^aeiouAEIOU]')

>>> consonantRegex.findall('Robocop eats baby food. BABY FOOD.')
['R', 'b', 'c', 'p', ' ', 't', 's', ' ', 'b', 'b', 'y', ' ', 'f', 'd', '.', '
', 'B', 'B', 'Y', ' ', 'F', 'D', '.']
```

Return to the Index

The Caret and Dollar Sign Characters

- You can also use the caret symbol (^) at the start of a regex to indicate that a match must occur at the beginning of the searched text.
- Likewise, you can put a dollar sign ($) at the end of the regex to indicate the string must end with this regex pattern.
- And you can use the ^ and $ together to indicate that the entire string must match the regex—that is, it's not enough for a match to be made on some subset of the string.

The r'^Hello' regular expression string matches strings that begin with 'Hello':

```
>>> beginsWithHello = re.compile(r'^Hello')
```

```
>>> beginsWithHello.search('Hello world!')
<_sre.SRE_Match object; span=(0, 5), match='Hello'>

>>> beginsWithHello.search('He said hello.') == None
True
```

The r'\d$' regular expression string matches strings that end with a numeric character from 0 to 9:

```
>>> wholeStringIsNum = re.compile(r'^\d+$')

>>> wholeStringIsNum.search('1234567890')
<_sre.SRE_Match object; span=(0, 10), match='1234567890'>

>>> wholeStringIsNum.search('12345xyz67890') == None
True

>>> wholeStringIsNum.search('12 34567890') == None
True
```

Return to the Index

The Wildcard Character

The . (or dot) character in a regular expression is called a wildcard and will match any character except for a newline:

```
>>> atRegex = re.compile(r'.at')

>>> atRegex.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']
```

Return to the Index

Matching Everything with Dot-Star

```
>>> nameRegex = re.compile(r'First Name: (.*) Last Name: (.*)')

>>> mo = nameRegex.search('First Name: Al Last Name: Sweigart')

>>> mo.group(1)
'Al'

>>> mo.group(2)
'Sweigart'
```

The dot-star uses greedy mode: It will always try to match as much text as possible. To match any and all text in a nongreedy fashion, use the dot, star, and question mark (.*?). The question mark tells Python to match in a nongreedy way:

```
>>> nongreedyRegex = re.compile(r'<.*?>')
>>> mo = nongreedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man>'

>>> greedyRegex = re.compile(r'<.*>')
>>> mo = greedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man> for dinner.>'
```

Return to the Index

Matching Newlines with the Dot Character

The dot-star will match everything except a newline. By passing re.DOTALL as the second argument to re.compile(), you can make the dot character match all characters, including the newline character:

```
>>> noNewlineRegex = re.compile('.*')
>>> noNewlineRegex.search('Serve the public trust.\nProtect the
innocent.\nUphold the law.').group()
'Serve the public trust.'

>>> newlineRegex = re.compile('.*', re.DOTALL)
>>> newlineRegex.search('Serve the public trust.\nProtect the
innocent.\nUphold the law.').group()
'Serve the public trust.\nProtect the innocent.\nUphold the law.'
```

Return to the Index

Review of Regex Symbols

Symbol	Matches
?	zero or one of the preceding group.
*	zero or more of the preceding group.
+	one or more of the preceding group.
{n}	exactly n of the preceding group.
{n,}	n or more of the preceding group.
{,m}	0 to m of the preceding group.

{n,m}	at least n and at most m of the preceding p.
{n,m}? or *? or +?	performs a nongreedy match of the preceding p.
^spam	means the string must begin with spam.
spam$	means the string must end with spam.
.	any character, except newline characters.
\d, \w, and \s	a digit, word, or space character, ectively.
\D, \W, and \S	anything except a digit, word, or space acter, respectively.
[abc]	any character between the brackets (such as a, b,).
[^abc]	any character that isn't between the brackets.

Return to the Index

Case-Insensitive Matching

To make your regex case-insensitive, you can pass re.IGNORECASE or re.I as a second argument to re.compile():

```
>>> robocop = re.compile(r'robocop', re.I)

>>> robocop.search('Robocop is part man, part machine, all cop.').group()
'Robocop'

>>> robocop.search('ROBOCOP protects the innocent.').group()
'ROBOCOP'

>>> robocop.search('Al, why does your programming book talk about robocop so
much?').group()
'robocop'
```

Return to the Index

Substituting Strings with the sub() Method

The sub() method for Regex objects is passed two arguments:

1. The first argument is a string to replace any matches.
2. The second is the string for the regular expression.

The sub() method returns a string with the substitutions applied:

```
>>> namesRegex = re.compile(r'Agent \w+')

>>> namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent
Bob.')
```

```
'CENSORED gave the secret documents to CENSORED.'
```

Another example:

```
>>> agentNamesRegex = re.compile(r'Agent (\w)\w*')

>>> agentNamesRegex.sub(r'\1****', 'Agent Alice told Agent Carol that Agent
Eve knew Agent Bob was a double agent.')
A**** told C**** that E**** knew B**** was a double agent.'
```

Return to the Index

Managing Complex Regexes

To tell the re.compile() function to ignore whitespace and comments inside the regular expression string, "verbose mode" can be enabled by passing the variable re.VERBOSE as the second argument to re.compile().

Now instead of a hard-to-read regular expression like this:

```
phoneRegex = re.compile(r'((\d{3}|\(\d{3}\))?(\s|-|\.)?\d{3}(\s|-|\.)\d{4}(\s*
(ext|x|ext.)\s*\d{2,5})?)')
```

you can spread the regular expression over multiple lines with comments like this:

```
phoneRegex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?            # area code
    (\s|-|\.)?                    # separator
    \d{3}                         # first 3 digits
    (\s|-|\.)                     # separator
    \d{4}                         # last 4 digits
    (\s*(ext|x|ext.)\s*\d{2,5})?  # extension
    )''', re.VERBOSE)
```

Return to the Index

Reading and Writing Files

Backslash on Windows and Forward Slash on OS X and Linux

On Windows, paths are written using backslashes () as the separator between folder names. OS X and Linux, however, use the forward slash (/) as their path separator.

Fortunately, this is simple to do with the os.path.join() function. If you pass it the string values of individual file and folder names in your path, os.path.join() will return a string with a file path using the correct path

separators.

```
>>> import os

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

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

```
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']

>>> for filename in myFiles:
        print(os.path.join('C:\\Users\\asweigart', filename))
```

Output:

```
C:\Users\asweigart\accounts.txt
C:\Users\asweigart\details.csv
C:\Users\asweigart\invite.docx
```

Return to the Index

The Current Working Directory

```
>>> import os

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

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

Return to the Index

Absolute vs. Relative Paths

There are two ways to specify a file path.

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

There are also the dot (.) and dot-dot (..) folders. These are not real folders but special names that can be used

in a path. A single period (“dot”) for a folder name is shorthand for “this directory.” Two periods (“dot-dot”) means “the parent folder.”

Return to the Index

Creating New Folders with os.makedirs()

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

Return to the Index

Handling Absolute and Relative Paths

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

Return to the Index

Finding File Sizes and Folder Contents

- Calling os.path.getsize(path) will return the size in bytes of the file in the path argument.
- Calling os.listdir(path) will return a list of filename strings for each file in the path argument. (Note that this function is in the os module, not os.path.)

```
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
776192

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

To find the total size of all the files in this directory, use os.path.getsize() and os.listdir() together:

```
>>> totalSize = 0

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

```
>>> print(totalSize)
1117846456
```

Return to the Index

Checking Path Validity

- Calling os.path.exists(path) will return True if the file or er referred to in the argument exists and will return False t does not exist.
- Calling os.path.isfile(path) will return True if the path ment exists and is a file and will return False otherwise.
- Calling os.path.isdir(path) will return True if the path ment exists and is a folder and will return False otherwise.

Return to the Index

The File Reading/Writing Process

There are three steps to reading or writing files in Python.

1. Call the open() function to return a File object.
2. Call the read() or write() method on the File object.
3. Close the file by calling the close() method on the File object.

Return to the Index

Opening Files with the open() Function

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

Return to the Index

Reading the Contents of Files

```
>>> helloContent = helloFile.read()

>>> helloContent
'Hello world!'
```

Alternatively, you can use the *readlines()* method to get a list of string values from the file, one string for each line of text:

```
>>> sonnetFile = open('sonnet29.txt')

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

Return to the Index

Writing to Files

```
>>> baconFile = open('bacon.txt', 'w')

>>> baconFile.write('Hello world!\n')
13

>>> baconFile.close()

>>> baconFile = open('bacon.txt', 'a')

>>> baconFile.write('Bacon is not a vegetable.')
25

>>> baconFile.close()

>>> baconFile = open('bacon.txt')

>>> content = baconFile.read()

>>> baconFile.close()

>>> print(content)
Hello world!
Bacon is not a vegetable.
```

Return to the Index

Saving Variables with the shelve Module

To save variables:

```
>>> import shelve

>>> shelfFile = shelve.open('mydata')

>>> cats = ['Zophie', 'Pooka', 'Simon']
```

```
>>> shelfFile['cats'] = cats

>>> shelfFile.close()
```

To open and read variables:

```
>>> shelfFile = shelve.open('mydata')

>>> type(shelfFile)
<class 'shelve.DbfilenameShelf'>

>>> shelfFile['cats']
['Zophie', 'Pooka', 'Simon']

>>> shelfFile.close()
```

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

```
>>> shelfFile = shelve.open('mydata')

>>> list(shelfFile.keys())
['cats']

>>> list(shelfFile.values())
[['Zophie', 'Pooka', 'Simon']]

>>> shelfFile.close()
```

Return to the Index

Saving Variables with the pprint.pformat() Function

```
>>> import pprint

>>> cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc':
'fluffy'}]

>>> pprint.pformat(cats)
"[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]"

>>> fileObj = open('myCats.py', 'w')
```

```
>>> fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
83

>>> fileObj.close()
```

Return to the Index

Copying Files and Folders

The shutil module provides functions for copying files, as well as entire folders.

```
>>> import shutil, os

>>> os.chdir('C:\\')

>>> shutil.copy('C:\\spam.txt', 'C:\\delicious')
   'C:\\delicious\\spam.txt'

>>> shutil.copy('eggs.txt', 'C:\\delicious\\eggs2.txt')
   'C:\\delicious\\eggs2.txt'
```

While shutil.copy() will copy a single file, shutil.copytree() will copy an entire folder and every folder and file contained in it:

```
>>> import shutil, os

>>> os.chdir('C:\\')

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

Return to the Index

Moving and Renaming Files and Folders

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

The destination path can also specify a filename. In the following example, the source file is moved and renamed:

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

```
'C:\\eggs\\new_bacon.txt'
```

If there is no eggs folder, then move() will rename bacon.txt to a file named eggs.

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

Return to the Index

Permanently Deleting Files and Folders

- Calling os.unlink(path) will delete the file at path.
- Calling os.rmdir(path) will delete the folder at path. This folder must be empty of any files or folders.
- Calling shutil.rmtree(path) will remove the folder at path, and all files and folders it contains will also be deleted.

Return to the Index

Safe Deletes with the send2trash Module

You can install this module by running pip install send2trash from a Terminal window.

```
>>> import send2trash

>>> baconFile = open('bacon.txt', 'a') # creates the file

>>> baconFile.write('Bacon is not a vegetable.')
25

>>> baconFile.close()

>>> send2trash.send2trash('bacon.txt')
```

Return to the Index

Walking a Directory Tree

```
import os

for folderName, subfolders, filenames in os.walk('C:\\delicious'):
    print('The current folder is ' + folderName)

    for subfolder in subfolders:
```

```
        print('SUBFOLDER OF ' + folderName + ': ' + subfolder)
    for filename in filenames:
        print('FILE INSIDE ' + folderName + ': '+ filename)

    print('')
```

Output:

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

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

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

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

Return to the Index

Reading ZIP Files

```
>>> import zipfile, os

>>> os.chdir('C:\\')    # move to the folder with example.zip

>>> exampleZip = zipfile.ZipFile('example.zip')

>>> exampleZip.namelist()
['spam.txt', 'cats/', 'cats/catnames.txt', 'cats/zophie.jpg']

>>> spamInfo = exampleZip.getinfo('spam.txt')

>>> spamInfo.file_size
13908

>>> spamInfo.compress_size
3828

>>> 'Compressed file is %sx smaller!' % (round(spamInfo.file_size /
spamInfo.compress_size, 2))
'Compressed file is 3.63x smaller!'
```

```
>>> exampleZip.close()
```

Return to the Index

Extracting from ZIP Files

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

```
>>> import zipfile, os

>>> os.chdir('C:\\')    # move to the folder with example.zip

>>> exampleZip = zipfile.ZipFile('example.zip')

>>> exampleZip.extractall()

>>> exampleZip.close()
```

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

```
>>> exampleZip.extract('spam.txt')
'C:\\spam.txt'

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

>>> exampleZip.close()
```

Return to the Index

Creating and Adding to ZIP Files

```
>>> import zipfile

>>> newZip = zipfile.ZipFile('new.zip', 'w')

>>> newZip.write('spam.txt', compress_type=zipfile.ZIP_DEFLATED)

>>> newZip.close()
```

This code will create a new ZIP file named new.zip that has the compressed contents of spam.txt.

Return to the Index

Debugging

Raising Exceptions

Exceptions are raised with a raise statement. In code, a raise statement consists of the following:

- The raise keyword
- A call to the Exception() function
- A string with a helpful error message passed to the Exception() function

```
>>> raise Exception('This is the error message.')
Traceback (most recent call last):
  File "<pyshell#191>", line 1, in <module>
    raise Exception('This is the error message.')
Exception: This is the error message.
```

Often it's the code that calls the function, not the function itself, that knows how to handle an expection. So you will commonly see a raise statement inside a function and the try and except statements in the code calling the function.

```
def boxPrint(symbol, width, height):
    if len(symbol) != 1:
      raise Exception('Symbol must be a single character string.')
    if width <= 2:
      raise Exception('Width must be greater than 2.')
    if height <= 2:
      raise Exception('Height must be greater than 2.')
    print(symbol * width)
    for i in range(height - 2):
        print(symbol + (' ' * (width - 2)) + symbol)
    print(symbol * width)
for sym, w, h in (('*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
    try:
        boxPrint(sym, w, h)
  except Exception as err:
      print('An exception happened: ' + str(err))
```

Return to the Index

Getting the Traceback as a String

The traceback is displayed by Python whenever a raised exception goes unhandled. But can also obtain it as a string by calling traceback.format_exc(). This function is useful if you want the information from an exception's traceback but also want an except statement to gracefully handle the exception. You will need to import

Python's traceback module before calling this function.

```
>>> import traceback

>>> try:
        raise Exception('This is the error message.')
    except:
        errorFile = open('errorInfo.txt', 'w')
        errorFile.write(traceback.format_exc())
        errorFile.close()
        print('The traceback info was written to errorInfo.txt.')
```

Output:

```
116
The traceback info was written to errorInfo.txt.
```

The 116 is the return value from the write() method, since 116 characters were written to the file. The traceback text was written to errorInfo.txt.

```
Traceback (most recent call last):
  File "<pyshell#28>", line 2, in <module>
Exception: This is the error message.
```

Return to the Index

Assertions

An assertion is a sanity check to make sure your code isn't doing something obviously wrong. These sanity checks are performed by assert statements. If the sanity check fails, then an AssertionError exception is raised. In code, an assert statement consists of the following:

- The assert keyword
- A condition (that is, an expression that evaluates to True or False)
- A comma
- A string to display when the condition is False

```
>>> podBayDoorStatus = 'open'

>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'

>>> podBayDoorStatus = 'I\'m sorry, Dave. I\'m afraid I can\'t do that.'
```

```
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'

Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
AssertionError: The pod bay doors need to be "open".
```

In plain English, an assert statement says, "I assert that this condition holds true, and if not, there is a bug somewhere in the program." Unlike exceptions, your code should not handle assert statements with try and except; if an assert fails, your program should crash. By failing fast like this, you shorten the time between the original cause of the bug and when you first notice the bug. This will reduce the amount of code you will have to check before finding the code that's causing the bug.

Disabling Assertions

Assertions can be disabled by passing the -O option when running Python.

Return to the Index

Logging

To enable the logging module to display log messages on your screen as your program runs, copy the following to the top of your program (but under the #! python shebang line):

```
import logging

logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s-
%(message)s')
```

Say you wrote a function to calculate the factorial of a number. In mathematics, factorial 4 is 1 × 2 × 3 × 4, or 24. Factorial 7 is 1 × 2 × 3 × 4 × 5 × 6 × 7, or 5,040. Open a new file editor window and enter the following code. It has a bug in it, but you will also enter several log messages to help yourself figure out what is going wrong. Save the program as factorialLog.py.

```
import logging

logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s-
%(message)s')

logging.debug('Start of program')

def factorial(n):

    logging.debug('Start of factorial(%s)' % (n))
    total = 1

    for i in range(n + 1):
```

```
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))

    logging.debug('End of factorial(%s)' % (n))

    return total

print(factorial(5))
logging.debug('End of program')
```

Output:

```
2015-05-23 16:20:12,664 - DEBUG - Start of program
2015-05-23 16:20:12,664 - DEBUG - Start of factorial(5)
2015-05-23 16:20:12,665 - DEBUG - i is 0, total is 0
2015-05-23 16:20:12,668 - DEBUG - i is 1, total is 0
2015-05-23 16:20:12,670 - DEBUG - i is 2, total is 0
2015-05-23 16:20:12,673 - DEBUG - i is 3, total is 0
2015-05-23 16:20:12,675 - DEBUG - i is 4, total is 0
2015-05-23 16:20:12,678 - DEBUG - i is 5, total is 0
2015-05-23 16:20:12,680 - DEBUG - End of factorial(5)
0
2015-05-23 16:20:12,684 - DEBUG - End of program
```

Return to the Index

Logging Levels

Logging levels provide a way to categorize your log messages by importance. There are five logging levels, described in Table 10-1 from least to most important. Messages can be logged at each level using a different logging function.

Level	Logging Function	Description
DEBUG	logging.debug()	The lowest level. Used for small details. Usually you care about these messages only when diagnosing problems.
INFO	logging.info()	Used to record information on general events in your program or confirm that things are working at their point in the program.
WARNING	logging.warning()	Used to indicate a potential problem that doesn't prevent the program from working but might do so in the future.
ERROR	logging.error()	Used to record an error that caused the program to fail to do something.
CRITICAL	logging.critical()	The highest level. Used to indicate a fatal error that has caused or is about to cause the program to stop running entirely.

Return to the Index

Disabling Logging

After you've debugged your program, you probably don't want all these log messages cluttering the screen. The logging.disable() function disables these so that you don't have to go into your program and remove all the logging calls by hand.

```
>>> import logging

>>> logging.basicConfig(level=logging.INFO, format=' %(asctime)s -%
(levelname)s - %(message)s')

>>> logging.critical('Critical error! Critical error!')
2015-05-22 11:10:48,054 - CRITICAL - Critical error! Critical error!

>>> logging.disable(logging.CRITICAL)

>>> logging.critical('Critical error! Critical error!')

>>> logging.error('Error! Error!')
```

Return to the Index

Logging to a File

Instead of displaying the log messages to the screen, you can write them to a text file. The logging.basicConfig() function takes a filename keyword argument, like so:

```
import logging

logging.basicConfig(filename='myProgramLog.txt', level=logging.DEBUG,
format='%(asctime)s - %(levelname)s - %(message)s')
```

Return to the Index

Virtual Environment

The use of a Virtual Environment is to test python code in encapsulated environments and to also avoid filling the base Python installation with libraries we might use for only one project.

Return to the Index

Windows

1. Install virtualenv:

```
pip install virtualenv
```

2. Install virtualenvwrapper-win:

```
pip install virtualenvwrapper-win
```

Usage:

1. Make a Virtual Environemt:

```
mkvirtualenv HelloWold
```

Anything we install now will be specific to this project. And available to the projects we connect to this environment.

2. Set Project Directory:

To bind our virtualenv with our current working directory we simply enter:

```
setprojectdir .
```

3. Deactivate:

To move onto something else in the command line type 'deactivate' to deactivate your environment.

```
deactivate
```

Notice how the parenthesis disappear.

4. Workon:

Open up the command prompt and type 'workon HelloWold' to activate the environment and move into your root project folder:

```
workon HelloWold
```

Return to the Index

Lambda Functions

This function:

```
>>> def add(x, y):
        return x + y

>>> add(5, 3)
8
```

Is equivalente to the *lambda* function:

```
>>> add = lambda x, y: x + y
>>> add(5, 3)
8
```

It's not even need to bind it to a name like add before:

```
>>> (lambda x, y: x + y)(5, 3)
8
```

Like regular nested functions, lambdas also work as lexical closures:

```
>>> def make_adder(n):
        return lambda x: x + n

>>> plus_3 = make_adder(3)
>>> plus_5 = make_adder(5)

>>> plus_3(4)
7
>>> plus_5(4)
9
```

Return to the Index

Ternary Conditional Operator

Many programming languages have a ternary operator, which define a conditional expression. The most common usage is to make a terse simple conditional assignment statement. In other words, it offers one-line code to evaluate the first expression if the condition is true, otherwise it evaluates the second expression.

```
<expression1> if <condition> else <expression2>
```

Example:

```
>>> age = 15

>>> print('kid' if age < 18 else 'adult')
kid
```

Ternary operators can be changed:

```
>>> age = 15

>>> print('kid' if age < 13 else 'teenager' if age < 18 else 'adult')
teenager
```

The code above is equivalent to:

```
if age < 18:
    if age < 12:
        print('kid')
    else:
        print('teenager')
else:
    print('adult')
```

Return to the Index