Python Programming Fundamentals

Grut Computing Systems, Inc.
333 Melrose Drive, #10B
Richardson, TX 75080
(940) 894-6623

Python Programming Fundamentals was developed and written by Grut Computing Systems, Inc.

Grut Computing Systems, Inc.
333 Melrose Drive, # 10B
Richardson, TX 75080
(940) 894-6623

Important Notice:

The purpose of this student manual is to serve as a supplement to the instructor-based technical presentation in the classroom. The work is not intended to be used as a self-contained reference manual on its own.

Revised November, 2017

COURSE ABSTRACT

The Python Programming Fundamentals class teaches Python programming in an interactive and scripting environment.

AUDIENCE

This course is designed for UNIX users needing development skills in Python.

DURATION

1 - 2 Days

LEARNING OBJECTIVES

Upon completion of this course, the participant will be able to:

- Understand the scalar and aggregate data types used with Python
- Understand the programming constructs used with Python
- Understand the development and invocation of Python functions
- Understand Python modules, attributes, and methods.

PREREQUISITES

Programming skills with a procedural or object language. Fundamental UNIX skills with knowledge of a UNIX text editor.

Table Of Contents

Python Programming

Python Programming

Comments

In the event you want to include comments in your python scripts, type a #. All characters from that point to the end of the line are treated as a comment.

A first look at a Python Program

```
#!/usr/bin/python

name = raw_input('Enter file:')
handle = open(name, 'r')
counts = dict()
for line in handle:
        words = line.split()
        for word in words:
                counts[word] = counts.get(word, 0) + 1
bigcount = None
bigword = None
for word, count in list(counts.items()):
        if bigcount is None or count > bigcount:
                bigword = word
                bigcount = count
print(bigword, bigcount)
```

A second look at a Python Program

```
#!/usr/bin/python
import sys
import os.path
for my_file in sys.argv[1:]:
    if not os.path.exists(my_file):
        print(my_file + " does not exist")
    elif os.path.isfile(my_file):
        print(my_file + " is a file")
    else:
        print(my_file + " is a directory")
```

Lab 23

Type in the above two programs. Save the first as wordcount1.py. Save the second as filecheck.py. Give the programs execute privileges. Run them and ensure they compile properly.

wordcount1.py
filecheck.py /etc abc /etc/passwd

Variables, expressions, and statements

Values and types

A *value* is one of the basic things a program works with, like a letter or a number.

Values belong to different *types*: 2 is an integer, and “Hello, World!” is a *string*, so called because it contains a “string” of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

The print statement also works for any type of data. We can use the python command to start the interactive interpreter.

```
python
>>> print(4)
4
```

If you are not sure what type a value has, the interpreter can tell you.

```
>>> type('Hello, World!')
<class 'str'>

>>> type(17)
<class 'int'>
```

Not surprisingly, strings belong to the type str and integers belong to the type int. Less obviously, numbers with a decimal point belong to a type called float, because these numbers are represented in a format called *floating point*.

```
>>> type(3.2)
<class 'float'>
```

What about values like “17” and “3.2”? They look like numbers, but they are in quotation marks like strings.

```
>>> type('17')
<class 'str'>

>>> type('3.2')
<class 'str'>
```

They’re strings.
When you type a large integer, you might be tempted to use commas between groups of three digits, as in 1,000,000. This is not a legal integer in Python, but it is legal:

```
>>> print(1,000,000)
1 0 0
```

Well, that’s not what we expected at all! Python interprets 1,000,000 as a comma separated sequence of integers, which it prints with spaces between.

This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn’t do the “right” thing.

Variables

One of the most powerful features of a programming language is the ability to manipulate *variables*. A variable is a name that refers to a value.
An *assignment statement* creates new variables and gives them values:

```
>>> message = 'And now for something completely different'

>>> n = 17

>>> pi = 3.14159265358979 31
```

This example makes three assignments. The first assigns a string to a new variable named message; the second assigns the integer 17 to n; the third assigns the (approximate) value of _ to pi.

To display the value of a variable, you can use a print statement:

```
>>> print(n)
17

>>> print(pi)
3.141592653589793
```

The type of a variable is the type of the value it refers to.

```
>>> type(message)
<class 'str'>

>>> type(n)
<class 'int'>

>>> type(pi)
<class 'float'>
```

Variable names and keywords

Programmers generally choose names for their variables that are meaningful and document what the variable is used for.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they cannot start with a number. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter (you'll see why later).

The underscore character (_) can appear in a name. It is often used in names with multiple words, such as my_name or airspeed_of_unladen_swallow.

Variable names can start with an underscore character, but we generally avoid doing this unless we are writing library code for others to use.

If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax

>>> more@ = 1000000
SyntaxError: invalid syntax

>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

76trombones is illegal because it begins with a number. more@ is illegal because it contains an illegal character, @. But what's wrong with class?

It turns out that class is one of Python's *keywords*. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

Python reserves 33 keywords:

and	del	from	None	True
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

Statements

A *statement* is a unit of code that the Python interpreter can execute. We have seen two kinds of statements: print being an expression statement and assignment.

When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one.

A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

For example, the script

```
print(1)
x = 2
print(x)
```

produces the output

```
1
2
```

The assignment statement produces no output.

Operators and operands

Operators are special symbols that represent computations like addition and multiplication.

The values the operator is applied to are called *operands*.

The operators +, -, *, /, and ** perform addition, subtraction, multiplication, division, and exponentiation, as in the following examples:

20 + 32 hour - 1 hour * 60 + minute minute /60 5 ** 2 (5 + 9) * (15 - 7)

There has been a change in the division operator between Python 2.x and Python 3.x. In Python 3.x, the result of this division is a floating point result:

```
>>> minute = 59

>>> minute/60
0.9833333333333333
```

The division operator in Python 2.0 would divide two integers and truncate the result to an integer:

```
>>> minute = 59
>>> minute / 60
0
```

To yield a floating point result in Python 2.0

```
>>> minute = 59
>>> float(minute) / 60
0.98333333333333328
```

To obtain the integer division answer in Python 3.0 use floored (// integer) division.

```
>>> minute = 59
>>> minute // 60
0
```

In Python 3.0 integer division functions much more as you would expect if you entered the expression on a calculator.

Expressions

An *expression* is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable x has been assigned a value):

```
17
x
x + 17
```

If you type an expression in interactive mode, the interpreter *evaluates* it and displays the result:

```
>>> 1 + 1
2
```

But in a script, an expression all by itself doesn't do anything!

Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the *rules of precedence*. For mathematical operators, Python follows mathematical convention. The acronym *PEMDAS* is a useful way to remember the rules:

• *P*arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 * (3-1) is 4, and (1+1)**(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute * 100) / 60, even if it doesn't change the result.

• *E*xponentiation has the next highest precedence, so 2**1+1 is 3, not 4, and 3*1**3 is 3, not 27.

• *M*ultiplication and *D*ivision have the same precedence, which is higher than *A*ddition and *S*ubtraction, which also have the same precedence. So 2*3-1 is 5, not 4, and 6+4/2 is 8.0, not 5.

• Operators with the same precedence are evaluated from left to right. So the expression 5-3-1 is 1, not 3, because the 5-3 happens first and then 1 is subtracted from 2.

When in doubt, always put parentheses in your expressions to make sure the computations are performed in the order you intend.

Modulus operator

The *modulus operator* works on integers and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators:

```
>>> quotient = 7 // 3
>>> print(quotient)
2

>>> remainder = 7 % 3
>>> print(remainder)
1
```

So 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another: if x % y is zero, then x is divisible by y.

You can also extract the right-most digit or digits from a number. For example, x % 10 yields the right-most digit of x (in base 10). Similarly, x % 100 yields the last two digits.

String operations

The + operator works with strings, but it is not addition in the mathematical sense. Instead it performs *concatenation*, which means joining the strings by linking them end to end. For example:

```
>>> first = 10
>>> second = 15
>>> print(first+second)
25

>>> first = '100'
>>> second = '150'
>>> print(first + second)
100150
```

The output of this program is 100150.

Asking the user for input

Sometimes we would like to take the value for a variable from the user via their keyboard (stdin). Python provides built-in functions (input and raw_input) that get input from the keyboard. When these functions are called, the program stops and waits for the user to type something. When the user presses Return or Enter, the program resumes and these functions return what the user typed in.

In Python 2.0, for strings, use function raw_input. It's not a bad idea to use raw_input for all user input and convert as needed.

```
>>> input = raw_input()
Some silly stuff

>>> print(input)
Some silly stuff
```

Before getting input from the user, it is a good idea to print a prompt telling the user what to input. You can pass a string to input to be displayed to the user before pausing for input:

```
>>> name = raw_input('What is your name?\n')
What is your name?
Chuck

>>> print(name)
Chuck
```

The sequence \n at the end of the prompt represents a *newline*, which is a special character that causes a line break. That's why the user's input appears below the prompt.

If you expect the user to type an integer, you can use the input (instead of raw_input) or try to convert the return value to int using the int() function:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)

What...is the airspeed velocity of an unladen swallow?
17

>>> int(speed)
17

>>> int(speed) + 5
22
```

But if the user types something other than a string of digits, you get an error:

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?

>>> int(speed)
ValueError: invalid literal for int() with base 10:
```

We will see how to handle this kind of error later.

Take a quick look at the following Python sample code which loops through some data. We will cover loops soon, but for now try to just puzzle through what this means:

```
for word in words:
        print(word)
```

What is happening here? Which of the tokens (for, word, in, etc.) are reserved words and which are just variable names? Does Python understand at a fundamental level the notion of words? Beginning programmers have trouble separating what parts of the code *must* be the same as this example and what parts of the code are simply choices made by the programmer.

The following code is equivalent to the above code:

```
for slice in pizza:
        print(slice)
```

It is easier for the beginning programmer to look at this code and know which parts are reserved words defined by Python and which parts are simply variable names chosen by the programmer. It is pretty clear that Python has no fundamental understanding of pizza and slices and the fact that a pizza consists of a set of one or more slices.

But if our program is truly about reading data and looking for words in the data, pizza and slice are very un-mnemonic variable names. Choosing them as variable names distracts from the meaning of the program.

After a pretty short period of time, you will know the most common reserved words and you will start to see the reserved words jumping out at you.

Many text editors are aware of Python syntax and will color reserved words differently to give you clues to keep your variables and reserved words separate. After a while you will begin to read Python and quickly determine what is a variable and what is a reserved word.

Conditional execution

Boolean expressions

A *boolean expression* is an expression that is either true or false. The following examples use the operator ==, which compares two operands and produces True if they are equal and False otherwise:

```
>>> 5 == 5
True

>>> 5 == 6
False
```

True and False are special values that belong to the class bool; they are not strings:

```
>>> type(True)
<class 'bool'>

>>> type(False)
<class 'bool'>
```

The == operator is one of the *comparison operators*; the others are:

```
x != y # x is not equal to y
x > y # x is greater than y
x < y # x is less than y
x >= y # x is greater than or equal to y
x <= y # x is less than or equal to y
x is y # x is the same as y
x is not y # x is not the same as y
```

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols for the same operations. A common error is to use a single equal sign (=) instead of a double equal sign (==). Remember that = is an assignment operator and == is a comparison operator. There is no such thing as =< or =>.

Logical operators

There are three *logical operators*: and, or, and not. The semantics (meaning) of these operators is similar to their meaning in English. For example, x > 0 and x < 10 is true only if x is greater than 0 *and* less than 10.

n%2 == 0 or n%3 == 0 is true if *either* of the conditions is true, that is, if the number is divisible by 2 *or* 3.

Finally, the not operator negates a boolean expression, so not (x > y) is true if x > y is false; that is, if x is less than or equal to y.

Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as "true."

```
>>> 17 and True
True
```

This flexibility can be useful, but there are some subtleties to it that might be confusing. You might want to avoid it until you are sure you know what you are doing.

Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. *Conditional statements* give us this ability. The simplest form is the if statement:

```
if x > 0 :
        print('x is positive')
```

The boolean expression after the if statement is called the *condition*. We end the if statement with a colon character (:) and the line(s) after the if statement are indented.

If the logical condition is true, then the indented statement gets executed. If the logical condition is false, the indented statement is skipped.

if statements have the same structure as function definitions or for loops. The statement consists of a header line that ends with the colon character (:) followed by an indented block. Statements like this are called *compound statements* because they stretch across more than one line.

There is no limit on the number of statements that can appear in the body, but there must be at least one. Occasionally, it is useful to have a body with no statements (usually as a place holder for code you haven't written yet). In that case, you can use the pass statement, which does nothing.

```
if x < 0 :
        pass # need to handle negative values!
```

If you enter an if statement in the Python interpreter, the prompt will change from three chevrons to three dots to indicate you are in the middle of a block of statements, as shown below:

```
>>> x = 3
>>> if x < 10:
... print('Small')
...
Small
>>>
```

Alternative execution

A second form of the if statement is *alternative execution*, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

```
if x%2 == 0 :
        print('x is even')
else :
        print('x is odd')
```

If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed.

Since the condition must either be true or false, exactly one of the alternatives will be executed. The alternatives are called *branches*, because they are branches in the flow of execution.

Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a *chained conditional*:

```
if x < y:
        print('x is less than y')
elif x > y:
        print('x is greater than y')
else:
        print('x and y are equal')
```

elif is an abbreviation of “else if.” Again, exactly one branch will be executed.

There is no limit on the number of elif statements. If there is an else clause, it has to be at the end, but there doesn’t have to be one.

```
if choice == 'a':
        print('Bad guess')
elif choice == 'b':
        print('Good guess')
elif choice == 'c':
        print('Close, but not correct')
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

Nested conditionals

One conditional can also be nested within another. We could have written the three-branch example like this:

```
if x == y:
        print('x and y are equal')
else:
        if x < y:
                print('x is less than y')
        else:
                print('x is greater than y')
```

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another if statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

Although the indentation of the statements makes the structure apparent, *nested conditionals* become difficult to read very quickly. In general, it is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
if 0 < x:
        if x < 10:
                print('x is a positive single-digit number.')
```

The print statement is executed only if we make it past both conditionals, so we can get the same effect with the and operator:

```
if 0 < x and x < 10:
        print('x is a positive single-digit number.')
```

Do Labs 1 thru 4

Catching exceptions using try and except

Earlier we saw a code segment where we used the input and int functions to read and parse an integer number entered by the user. We also saw how treacherous doing this could be:

```
>>> prompt = "What...is the airspeed velocity of an unladen swallow?\n"

>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?

>>> int(speed)
ValueError: invalid literal for int() with base 10:
>>>
```

When we are executing these statements in the Python interpreter, we get a new prompt from the interpreter, think “oops”, and move on to our next statement.

However if you place this code in a Python script and this error occurs, your script immediately stops in its tracks with a traceback. It does not execute the following statement.

Here is a sample program to convert a Fahrenheit temperature to a Celsius temperature:

```
inp = input('Enter Fahrenheit Temperature: ')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)
```

If we execute this code and give it invalid input, it simply fails with an unfriendly error message:

```
python fahren.py

Enter Fahrenheit Temperature:72
22.22222222222222

python fahren.py
Enter Fahrenheit Temperature:fred
Traceback (most recent call last):
File "fahren.py", line 2, in <module>
fahr = float(inp)
ValueError: could not convert string to float: 'fred'
```

There is a conditional execution structure built into Python to handle these types of expected and unexpected errors called “try / except”. The idea of try and except is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs. These extra statements (the except block) are ignored if there is no error.

You can think of the try and except feature in Python as an “insurance policy” on a sequence of statements.

We can rewrite our temperature converter as follows:

```
inp = input('Enter Fahrenheit Temperature:')
try:
        fahr = float(inp)
        cel = (fahr - 32.0) * 5.0 / 9.0
        print(cel)
except:
        print('Please enter a number')
```

Python starts by executing the sequence of statements in the try block. If all goes well, it skips the except block and proceeds. If an exception occurs in the try block, Python jumps out of the try block and executes the sequence of statements in the except block.

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.22222222222222

python fahren2.py
Enter Fahrenheit Temperature:fred
Please enter a number
```

Handling an exception with a try statement is called *catching* an exception. In this example, the except clause prints an error message. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

Do Labs 5 thru 7

Short-circuit evaluation of logical expressions

When Python is processing a logical expression such as x >= 2 and (x/y) > 2, it evaluates the expression from left to right. Because of the definition of and, if x is less than 2, the expression x >= 2 is False and so the whole expression is False regardless of whether (x/y) > 2 evaluates to True or False.

When Python detects that there is nothing to be gained by evaluating the rest of a logical expression, it stops its evaluation and does not do the computations in the rest of the logical expression. When the evaluation of a logical expression stops because the overall value is already known, it is called *short-circuiting* the evaluation.

While this may seem like a fine point, the short-circuit behavior leads to a clever technique called the *guardian pattern*. Consider the following code sequence in the Python interpreter:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
```

```
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False

>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

>>>
```

The third calculation failed because Python was evaluating (x/y) and y was zero, which causes a runtime error. But the second example did *not* fail because the first part of the expression x >= 2 evaluated to False so the (x/y) was not ever executed due to the *short-circuit* rule and there was no error.

We can construct the logical expression to strategically place a *guard* evaluation just before the evaluation that might cause an error as follows:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False

>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False

>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

In the first logical expression, x >= 2 is False so the evaluation stops at the and. In the second logical expression, x >= 2 is True but y != 0 is False so we never reach (x/y).

In the third logical expression, the y != 0 is *after* the (x/y) calculation so the expression fails with an error.

In the second expression, we say that y != 0 acts as a *guard* to insure that we only execute (x/y) if y is non-zero.

Functions

Function calls

In the context of programming, a *function* is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can "call" the function by name. We have already seen one example of a *function call*:

```
>>> type(32)
<class 'int'>
```

The name of the function is type. The expression in parentheses is called the *argument* of the function. The argument is a value or variable that we are passing into the function as input to the function. The result, for the type function, is the type of the argument.

It is common to say that a function "takes" an argument and "returns" a result. The result is called the *return value*.

Built-in functions

Python provides a number of important built-in functions that we can use without needing to provide the function definition. The creators of Python wrote a set of functions to solve common problems and included them in Python for us to use.

The max and min functions give us the largest and smallest values in a list, respectively:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
```

The max function tells us the "largest character" in the string (which turns out to be the letter "w") and the min function shows us the smallest character (which turns out to be a space).

Another very common built-in function is the len function which tells us how many items are in its argument. If the argument to len is a string, it returns the number of characters in the string.

```
>>> len('Hello world')
11
```

These functions are not limited to looking at strings. They can operate on any set of values, as we will see later. You should treat the names of built-in functions as reserved words (i.e., avoid using “max” as a variable name).

Type conversion functions

Python also provides built-in functions that convert values from one type to another. The int function takes any value and converts it to an integer, if it can, or complains otherwise:

```
>>> int('32')
32

>>> int('Hello')
ValueError: invalid literal for int() with base 10: 'Hello'
```

int can convert floating-point values to integers, but it doesn’t round off; it chops off the fraction part:

```
>>> int(3.99999)
3

>>> int(-2.3)
-2
```

float converts integers and strings to floating-point numbers:

```
>>> float(32)
32.0

>>> float('3.14159')
3.14159
```

Finally, str converts its argument to a string:

```
>>> str(32)
'32'

>>> str(3.14159)
'3.14159'
```

Random numbers

Given the same inputs, most computer programs generate the same outputs every time, so they are said to be *deterministic*. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

Making a program truly nondeterministic turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to use *algorithms* that generate *pseudorandom* numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

The random module provides functions that generate pseudo random numbers (which I will simply call “random” from here on).

The function random returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call random, you get the next number in a long series.

To see a sample, run this loop:

```
import random
for i in range(10):
        x = random.random()
        print(x)
```

This program produces the following list of 10 random numbers between 0.0 and up to but not including 1.0.

```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
```

The random function is only one of many functions that handle random numbers. The function randint takes the parameters low and high, and returns an integer between low and high (including both).

```
>>> random.randint(5, 10)
5

>>> random.randint(5, 10)
9
```

To choose an element from a sequence at random, you can use choice:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2

>>> random.choice(t)
3
```

The random module also provides functions to generate random values from continuous distributions including Gaussian, exponential, gamma, and a few more.

Do Lab 8

Math functions

Python has a math module that provides most of the familiar mathematical functions. Before we can use the module, we have to import it:

```
>>> import math
```

This statement creates a *module object* named math. If you print the module object, you get some information about it:

```
>>> print(math)
<module 'math' (built-in)>
```

The module object contains the functions and variables defined in the module. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called *dot notation*.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)
>>> radians = 0.7
>>> height = math.sin(radians)
```

The first example computes the logarithm base 10 of the signal-to-noise ratio. The math module also provides a function called log that computes logarithms base e.

The second example finds the sine of radians. The name of the variable is a hint that sin and the other trigonometric functions (cos, tan, etc.) take arguments in radians. To convert from degrees to radians, divide by 360 and multiply by 2pi:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865476
```

The expression math.pi gets the variable pi from the math module. The value of this variable is an approximation of pi, accurate to about 15 digits.

If you know your trigonometry, you can check the previous result by comparing it to the square root of two divided by two:

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

Adding new functions

So far, we have only been using the functions that come with Python, but it is also possible to add new functions. A *function definition* specifies the name of a new function and the sequence of statements that execute when the function is called.

Once we define a function, we can reuse the function over and over throughout our program.

Here is an example:

```
def print_lyrics():
        print("I'm a lumberjack, and I'm okay.")
        print('I sleep all night and I work all day.')
```

def is a keyword that indicates that this is a function definition. The name of the function is print_lyrics. The rules for function names are the same as for variable names: letters, numbers and some punctuation marks are legal, but the first character can't be a number. You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.

The empty parentheses after the name indicate that this function doesn't take any arguments. Later we will build functions that take arguments as their inputs.

The first line of the function definition is called the *header*; the rest is called the *body*. The header has to end with a colon and the body has to be indented. By convention, the indentation is always four spaces. The body can contain any number of statements.

The strings in the print statements are enclosed in quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.

If you type a function definition in interactive mode, the interpreter prints ellipses (. . .) to let you know that the definition isn't complete:

```
>>> def print_lyrics():
... print("I'm a lumberjack, and I'm okay.")
... print('I sleep all night and I work all day.')
...
```

To end the function, you have to enter an empty line (this is not necessary in a script).

Defining a function creates a variable with the same name.

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> print(type(print_lyrics))
<class 'function'>
```

The value of print_lyrics is a *function object*, which has type "function".
The syntax for calling the new function is the same as for built-in functions:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called repeat_lyrics:

```
def repeat_lyrics():
        print_lyrics()
        print_lyrics()
```

And then call repeat_lyrics:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

But that's not really how the song goes.

Definitions and uses

Pulling together the code fragments from the previous section, the whole program looks like this:

```
def print_lyrics():
        print("I'm a lumberjack, and I'm okay.")
        print('I sleep all night and I work all day.')
def repeat_lyrics():
        print_lyrics()
        print_lyrics()

repeat_lyrics()
```

This program contains two function definitions: print_lyrics and repeat_lyrics. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not get executed until the function is called, and the function definition generates no output.

As you might expect, you have to create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.

Flow of execution

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the *flow of execution*.

Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

Function *definitions* do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

What's the moral of this sordid tale? When you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

Parameters and arguments

Some of the built-in functions we have seen require arguments. For example, when you call math.sin you pass a number as an argument. Some functions take more than one argument: math.pow takes two, the base and the exponent.

Inside the function, the arguments are assigned to variables called *parameters*. Here is an example of a user-defined function that takes an argument:

```
def print_twice(bruce):
        print(bruce)
        print(bruce)
```

This function assigns the argument to a parameter named bruce. When the function is called, it prints the value of the parameter (whatever it is) twice.

This function works with any value that can be printed.

```
>>> print_twice('Spam')
Spam
Spam

>>> print_twice(17)
17
17
```

```
>>> import math
>>> print_twice(math.pi)
3.141592653589793
3.141592653589793
```

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for print_twice:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam

>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

The argument is evaluated before the function is called, so in the examples the expressions “Spam ’ *4 and math.cos(math.pi) are only evaluated once.

You can also use a variable as an argument:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

The name of the variable we pass as an argument (michael) has nothing to do with the name of the parameter (bruce). It doesn’t matter what the value was called back home (in the caller); here in print_twice, we call everybody bruce.

Do Lab 9

Fruitful functions and void functions

Some of the functions we are using, such as the math functions, yield results; for lack of a better name, I call them *fruitful functions*. Other functions, like print_twice, perform an action but don't return a value. They are called *void functions*.

When you call a fruitful function, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

When you call a function in interactive mode, Python displays the result:

```
>>> math.sqrt(5)
2.23606797749979
```

But in a script, if you call a fruitful function and do not store the result of the function in a variable, the return value vanishes into the mist!

```
math.sqrt(5)
```

This script computes the square root of 5, but since it doesn't store the result in a variable or display the result, it is not very useful.

Void functions might display something on the screen or have some other effect, but they don't have a return value. If you try to assign the result to a variable, you get a special value called None.

```
>>> result = print_twice('Bing')
Bing
Bing

>>> print(result)
None
```

The value None is not the same as the string "None". It is a special value that has its own type:

```
>>> print(type(None))
<class 'NoneType'>
```

To return a result from a function, we use the return statement in our function. For example, we could make a very simple function called addtwo that adds two numbers together and returns a result.

```
def addtwo(a, b):
        added = a + b
        return added
x = addtwo(3, 5)
print(x)
```

When this script executes, the print statement will print out “8” because the addtwo function was called with 3 and 5 as arguments. Within the function, the parameters a and b were 3 and 5 respectively. The function computed the sum of the two numbers and placed it in the local function variable named added. Then it used the return statement to send the computed value back to the calling code as the function result, which was assigned to the variable x and printed out.

Why functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

• Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read, understand, and debug.

• Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.

• Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.

• Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Throughout the rest of this class, often we will use a function definition to explain a concept. Part of the skill of creating and using functions is to have a function properly capture an idea such as “find the smallest value in a list of values”. Later we will show you code that finds the smallest in a list of values and we will present it to you as a function named min which takes a list of values as its argument and returns the smallest value in the list.

Iteration

Updating variables

A common pattern in assignment statements is an assignment statement that updates a variable, where the new value of the variable depends on the old.

```
x = x + 1
```

This means “get the current value of x, add 1, and then update x with the new value.”

If you try to update a variable that doesn’t exist, you get an error, because Python evaluates the right side before it assigns a value to x:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

Before you can update a variable, you have to *initialize* it, usually with a simple assignment:

```
>>> x = 0
>>> x = x + 1
```

Updating a variable by adding 1 is called an *increment*; subtracting 1 is called a *decrement*.

The while statement

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. Because iteration is so common, Python provides several language features to make it easier.

One form of iteration in Python is the while statement. Here is a simple program that counts down from five and then says “Blastoff!”.

```
n = 5
while n > 0:
        print(n)
        n = n - 1
print('Blastoff!')
```

You can almost read the while statement as if it were English. It means, “While n is greater than 0, display the value of n and then reduce the value of n by 1. When you get to 0, exit the while statement and display the word Blastoff!”

More formally, here is the flow of execution for a while statement:

1. Evaluate the condition, yielding True or False.

2. If the condition is false, exit the while statement and continue execution at the next statement.

3. If the condition is true, execute the body and then go back to step 1.

This type of flow is called a *loop* because the third step loops back around to the top. We call each time we execute the body of the loop an *iteration*. For the above loop, we would say, “It had five iterations”, which means that the body of the loop was executed five times.

The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. We call the variable that changes each time the loop executes and controls when the loop finishes the *iteration variable*. If there is no iteration variable, the loop will repeat forever, resulting in an *infinite loop*.

Do Lab 10

Infinite loops

An endless source of amusement for programmers is the observation that the directions on shampoo, “Lather, rinse, repeat,” are an infinite loop because there is no *iteration variable* telling you how many times to execute the loop.

In the case of countdown, we can prove that the loop terminates because we know that the value of n is finite, and we can see that the value of n gets smaller each time through the loop, so eventually we have to get to 0. Other times a loop is obviously infinite because it has no iteration variable at all.

“Infinite loops” and break

Sometimes you don’t know it’s time to end a loop until you get half way through the body. In that case you can write an infinite loop on purpose and then use the break statement to jump out of the loop.

This loop is obviously an *infinite loop* because the logical expression on the while statement is simply the logical constant True:

```
while True:
        print(n, end=' ')
        n = n - 1
print('Done!')
```

If you make the mistake and run this code, you will learn quickly how to stop a runaway Python process on your system or find where the power-off button is on your computer. This program will run forever or until your battery runs out because the logical expression at the top of the loop is always true by virtue of the fact that the expression is the constant value True.

While this is a dysfunctional infinite loop, we can still use this pattern to build useful loops as long as we carefully add code to the body of the loop to explicitly exit the loop using break when we have reached the exit condition.

For example, suppose you want to take input from the user until they type done. You could write:

```
while True:
        line = input('> ')
        if line == 'done':
                break
        print(line)
print('Done!')
```

The loop condition is True, which is always true, so the loop runs repeatedly until it hits the break statement.

Each time through, it prompts the user with an angle bracket. If the user types done, the break statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop. Here's a sample run:

```
> hello there
hello there

> finished
finished

> done
Done!
```

This way of writing while loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens.").

Finishing iterations with continue

Sometimes you are in an iteration of a loop and want to finish the current iteration and immediately jump to the next iteration. In that case you can use the continue statement to skip to the next iteration without finishing the body of the loop for the current iteration.

Here is an example of a loop that copies its input until the user types "done", but treats lines that start with the hash character as lines not to be printed (kind of like Python comments).

```
while True:
        line = input('> ')
        if line[0] == '#':
                continue
        if line == 'done':
                break
        print(line)
print('Done!')
```

Here is a sample run of this new program with continue added.

```
> hello there
hello there

> # don't print this
> print this!
print this!

> done
Done!
```

All the lines are printed except the one that starts with the hash sign because when the continue is executed, it ends the current iteration and jumps back to the while statement to start the next iteration, thus skipping the print statement.

Definite loops using for

Sometimes we want to loop through a *set* of things such as a list of words, the lines in a file, or a list of numbers. When we have a list of things to loop through, we can construct a *definite* loop using a for statement. We call the while statement an *indefinite* loop because it simply loops until some condition becomes False, whereas the for loop is looping through a known set of items so it runs through as many iterations as there are items in the set.

The syntax of a for loop is similar to the while loop in that there is a for statement and a loop body:

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
        print('Happy New Year:', friend)
print('Done!')
```

In Python terms, the variable friends is a list of three strings and the for loop goes through the list and executes the body once for each of the three strings in the list resulting in this output:

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
```

Translating this for loop to English is not as direct as the while, but if you think of friends as a *set*, it goes like this: “Run the statements in the body of the for loop once for each friend *in* the set named friends.”

Looking at the for loop, *for* and *in* are reserved Python keywords, and friend and friends are variables.

```
for friend in friends:
        print('Happy New Year:', friend)
```

In particular, friend is the *iteration variable* for the for loop. The variable friend changes for each iteration of the loop and controls when the for loop completes. The *iteration variable* steps successively through the three strings stored in the friends variable.

Loop patterns

Often we use a for or while loop to go through a list of items or the contents of a file and we are looking for something such as the largest or smallest value of the data we scan through.
These loops are generally constructed by:

• Initializing one or more variables before the loop starts

• Performing some computation on each item in the loop body, possibly changing the variables in the body of the loop

• Looking at the resulting variables when the loop completes

We will use a list of numbers to demonstrate the concepts and construction of these loop patterns.

Counting and summing loops

For example, to count the number of items in a list, we would write the following for loop:

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
        count = count + 1
print('Count: ', count)
```

We set the variable count to zero before the loop starts, then we write a for loop to run through the list of numbers. Our *iteration* variable is named itervar and while we do not use itervar in the loop, it does control the loop and cause the loop body to be executed once for each of the values in the list.

In the body of the loop, we add 1 to the current value of count for each of the values in the list. While the loop is executing, the value of count is the number of values we have seen “so far”.

Once the loop completes, the value of count is the total number of items. The total number “falls in our lap” at the end of the loop. We construct the loop so that we have what we want when the loop finishes.

Another similar loop that computes the total of a set of numbers is as follows:

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
        total = total + itervar
print('Total: ', total)
```

In this loop we *do* use the *iteration variable*. Instead of simply adding one to the count as in the previous loop, we add the actual number (3, 41, 12, etc.) to the running total during each loop iteration. If you think about the variable total, it contains the “running total of the values so far”. So before the loop starts total is zero because we have not yet seen any values, during the loop total is the running total, and at the end of the loop total is the overall total of all the values in the list.

As the loop executes, total accumulates the sum of the elements; a variable used this way is sometimes called an *accumulator*.

Neither the counting loop nor the summing loop are particularly useful in practice because there are built-in functions len() and sum() that compute the number of items in a list and the total of the items in the list respectively.

Do Lab 11

Maximum and minimum loops

[maximumloop] To find the largest value in a list or sequence, we construct the following loop:

```
largest = None
print('Before:', largest)
for itervar in [3, 41, 12, 9, 74, 15]:
        if largest is None or itervar > largest :
                largest = itervar
        print('Loop:', itervar, largest)
print('Largest:', largest)
```

When the program executes, the output is as follows:

```
Before: None
Loop: 3 3
Loop: 41 41
Loop: 12 41
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

The variable largest is best thought of as the “largest value we have seen so far”. Before the loop, we set largest to the constant None. None is a special constant value which we can store in a variable to mark the variable as “empty”.

Before the loop starts, the largest value we have seen so far is None since we have not yet seen any values. While the loop is executing, if largest is None then we take the first value we see as the largest so far. You can see in the first iteration when the value of itervar is 3, since largest is None, we immediately set largest to be 3.

After the first iteration, largest is no longer None, so the second part of the compound logical expression that checks itervar > largest triggers only when we see a value that is larger than the “largest so far”. When we see a new “even larger” value we take that new value for largest. You can see in the program output that largest progresses from 3 to 41 to 74.

At the end of the loop, we have scanned all of the values and the variable largest now does contain the largest value in the list.

To compute the smallest number, the code is very similar with one small change:

```
smallest = None
print('Before:', smallest)
for itervar in [3, 41, 12, 9, 74, 15]:
        if smallest is None or itervar < smallest:
                smallest = itervar
        print('Loop:', itervar, smallest)
print('Smallest:', smallest)
```

Again, smallest is the “smallest so far” before, during, and after the loop executes. When the loop has completed, smallest contains the minimum value in the list.

Again as in counting and summing, the built-in functions max() and min() make writing these exact loops unnecessary.

The following is a simple version of the Python built-in min() function:

```
def min(values):
        smallest = None
        for value in values:
                if smallest is None or value < smallest:
                        smallest = value
        return smallest
```

In the function version of the smallest code, we removed all of the print statements so as to be equivalent to the min function which is already built in to Python.

Do Lab 12 thru 14

Strings

A string is a sequence

A string is a *sequence* of characters. You can access the characters one at a time with the bracket operator:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

The second statement extracts the character at index position 1 from the fruit variable and assigns it to the letter variable.

The expression in brackets is called an *index*. The index indicates which character in the sequence you want (hence the name).

But you might not get what you expect:

```
>>> print(letter)
a
```

For most people, the first letter of “banana” is b, not a. But in Python, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
>>> letter = fruit[0]
>>> print(letter)
b
```

So b is the 0th letter (“zero-eth”) of “banana”, a is the 1th letter (“one-eth”), and n is the 2th (“two-eth”) letter.

You can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise you get:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

Getting the length of a string using len

len is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

The reason for the IndexError is that there is no letter in ’banana’ with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from length:

```
>>> last = fruit[length-1]
>>> print(last)
a
```

Alternatively, you can use negative indices, which count backward from the end of the string. The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.

Traversal through a string with a loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a *traversal*. One way to write a traversal is with a while loop:

```
index = 0
while index < len(fruit):
        letter = fruit[index]
        print(letter)
        index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is index < len(fruit), so when index is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index len(fruit)-1, which is the last character in the string.

Another way to write a traversal is with a for loop:

```
for char in fruit:
        print(char)
```

Each time through the loop, the next character in the string is assigned to the variable char. The loop continues until no characters are left.

String slices

A segment of a string is called a *slice*. Selecting a slice is similar to selecting a character:

```
>>> s = 'Monty Python'
>>> print(s[0:5])
Monty

>>> print(s[6:12])
Python
```

The operator returns the part of the string from the “n-eth” character to the “m-eth” character, including the first but excluding the last.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'

>>> fruit[3:]
'ana'
```

If the first index is greater than or equal to the second the result is an *empty string*, represented by two quotation marks:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

Strings are immutable

It is tempting to use the operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

The “object” in this case is the string and the “item” is the character you tried to assign. For now, an *object* is the same thing as a value, but we will refine that definition later. An *item* is one of the values in a sequence.

The reason for the error is that strings are *immutable*, which means you can’t change an existing string. The best you can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print(new_greeting)
Jello, world!
```

This example concatenates a new first letter onto a slice of greeting. It has no effect on the original string.

Looping and counting

The following program counts the number of times the letter a appears in a string:

```
word = 'banana'
count = 0
for letter in word:
        if letter == 'a':
                count = count + 1
print(count)
```

This program demonstrates another pattern of computation called a *counter*. The variable count is initialized to 0 and then incremented each time an a is found. When the loop exits, count contains the result: the total number of a's.

The in operator

The word in is a boolean operator that takes two strings and returns True if the first appears as a substring in the second:

```
>>> 'a' in 'banana'
True

>>> 'seed' in 'banana'
False
```

Do Lab 15

String comparison

The comparison operators work on strings. To see if two strings are equal:

```
if word == 'banana':
        print('All right, bananas.')
```

Other comparison operations are useful for putting words in alphabetical order:

```
if word < 'banana':
        print('Your word,' + word + ', comes before banana.')
elif word > 'banana':
        print('Your word,' + word + ', comes after banana.')
else:
        print('All right, bananas.')
```

Python does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters, so:

Your word, Pineapple, comes before banana.

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

string methods

Strings are an example of Python *objects*. An object contains both data (the actual string itself) and *methods*, which are effectively functions that are built into the object and are available to any *instance* of the object.

Python has a function called dir which lists the methods available for an object. The type function shows the type of an object and the dir function shows the available methods.

```
>>> stuff = 'Hello world'
>>> type(stuff)
<class 'str'>
>>> dir(stuff)
['capitalize', 'casefold', 'center', 'count', 'encode',
'endswith' , 'expandtabs', 'find', 'format', 'format_map',
'index' , 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier' , 'islower', 'isnumeric', 'isprintable',
'isspace' , 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip' , 'maketrans', 'partition', 'replace', 'rfind',
'rindex' , 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split' , 'splitlines', 'startswith', 'strip', 'swapcase',
'title' , 'translate', 'upper', 'zfill']

>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
        S.capitalize() -> str

        Return a capitalized version of S, i.e. make the first character
        have upper case and the rest lower case.
>>>
```

Calling a *method* is similar to calling a function (it takes arguments and returns a value) but the syntax is different. We call a method by appending the method name to the variable name using the period as a delimiter.

For example, the method upper takes a string and returns a new string with all uppercase letters:

Instead of the function syntax upper(word), it uses the method syntax word.upper().

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print(new_word)
BANANA
```

This form of dot notation specifies the name of the method, upper, and the name of the string to apply the method to, word. The empty parentheses indicate that this method takes no argument.

A method call is called an *invocation*; in this case, we would say that we are invoking upper on the word.

For example, there is a string method named find that searches for the position of one string within another:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print(index)
1
```

In this example, we invoke find on word and pass the letter we are looking for as a parameter.

The find method can find substrings as well as characters:

```
>>> word.find('na')
2
```

It can take as a second argument the index where it should start:

```
>>> word.find('na', 3)
4
```

One common task is to remove white space (spaces, tabs, or newlines) from the beginning and end of a string using the strip method:

```
>>> line = ' Here we go '
>>> line.strip()
'Here we go'
```

Some methods such as *startswith* return boolean values.

```
>>> line = 'Have a nice day'
>>> line.startswith('Have')
True

>>> line.startswith('h')
False
```

You will note that startswith requires case to match, so sometimes we take a line and map it all to lowercase before we do any checking using the lower method.

```
>>> line = 'Have a nice day'
>>> line.startswith('h')
False

>>> line.lower()
'have a nice day'

>>> line.lower().startswith('h')
True
```

In the last example, the method lower is called and then we use startswith to see if the resulting lowercase string starts with the letter “h”. As long as we are careful with the order, we can make multiple method calls in a single expression.

Parsing strings

Often, we want to look into a string and find a substring. For example if we were presented a series of lines formatted as follows:

From stephen.marquard@ *uct.ac.za* Sat Jan 5 09:14:16 2008

and we wanted to pull out only the second half of the address (i.e., uct.ac.za) from each line, we can do this by using the find method and string slicing.

First, we will find the position of the at-sign in the string. Then we will find the position of the first space *after* the at-sign. And then we will use string slicing to extract the portion of the string which we are looking for.

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print(atpos)
21

>>> sppos = data.find(' ',atpos)
>>> print(sppos)
31

>>> host = data[atpos+1:sppos]
>>> print(host)
uct.ac.za
```

We use a version of the find method which allows us to specify a position in the string where we want find to start looking. When we slice, we extract the characters from “one beyond the at-sign through up to *but not including* the space character”.

Format operator

The *format operator*, % allows us to construct strings, replacing parts of the strings with the data stored in variables. When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator.

The first operand is the *format string*, which contains one or more *format sequences* that specify how the second operand is formatted. The result is a string.

For example, the format sequence “%d” means that the second operand should be formatted as an integer (d stands for “decimal”):

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string “42”, which is not to be confused with the integer value 42.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

The following example uses “%d” to format an integer, “%g” to format a floatingpoint number (don’t ask why), and “%s” to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple must match the number of format sequences in the string. The types of the elements also must match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string

>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

In the first example, there aren’t enough elements; in the second, the element is the wrong type.
The format operator is powerful, but it can be difficult to use

Do Lab 16

Files

Persistence

So far, we have learned how to write programs and communicate our intentions to the *Central Processing Unit* using conditional execution, functions, and iterations. We have learned how to create and use data structures in the *Main Memory*. The CPU and memory are where our software works and runs. It is where all of the “thinking” happens.

But if you recall from our hardware architecture discussions, once the power is turned off, anything stored in either the CPU or main memory is erased. So up to now, our programs have just been transient fun exercises to learn Python.

In this section, we start to work with *Secondary Memory* (or files). Secondary memory is not erased when the power is turned off. Or in the case of a USB flash drive, the data we write from our programs can be removed from the system and transported to another system.

We will primarily focus on reading and writing text files such as those we create ina text editor. Later we will see how to work with database files which are binary files, specifically designed to be read and written through database software.

Opening files

When we want to read or write a file (say on your hard drive), we first must *open* the file. Opening the file communicates with your operating system, which knows where the data for each file is stored. When you open a file, you are asking the operating system to find the file by name and make sure the file exists. In this example, we open the file mbox.txt, which should be stored in the same folder that you are in when you start Python.

```
>>> fhand = open('mbox.txt')
>>> print(fhand)
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252'>
```

If the open is successful, the operating system returns us a *file handle*. The file handle is not the actual data contained in the file, but instead it is a “handle” that we can use to read the data. You are given a handle if the requested file exists and you have the proper permissions to read the file.

If the file does not exist, open will fail with a traceback and you will not get a handle to access the contents of the file:

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'stuff.txt'
```

Later we will use try and except to deal more gracefully with the situation where we attempt to open a file that does not exist.

Text files and lines

A text file can be thought of as a sequence of lines, much like a Python string can be thought of as a sequence of characters. For example, this is a sample of a text file which records mail activity from various individuals in an open source project development team:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
...
```

These files are in a standard format for a file containing multiple mail messages. The lines which start with “From” separate the messages and the lines which start with “From:” are part of the messages.

To break the file into lines, there is a special character that represents the “end of the line” called the *newline* character.

In Python, we represent the *newline* character as a backslash-n in string constants.

Even though this looks like two characters, it is actually a single character. When we look at the variable by entering “stuff” in the interpreter, it shows us the \n in the string, but when we use print to show the string, we see the string broken into two lines by the newline character.

```
>>> stuff = 'Hello\nWorld!'
>>> stuff
'Hello\nWorld!'

>>> print(stuff)
Hello
World!

>>> stuff = 'X\nY'
>>> print(stuff)
X
Y

>>> len(stuff)
3
```

You can also see that the length of the string X\nY is *three* characters because the newline character is a single character.

So when we look at the lines in a file, we need to *imagine* that there is a special invisible character called the newline at the end of each line that marks the end of the line.

So the newline character separates the characters in the file into lines.

Reading files

While the *file handle* does not contain the data for the file, it is quite easy to construct a for loop to read through and count each of the lines in a file:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
        count = count + 1
print('Line Count:', count)
```

We can use the file handle as the sequence in our for loop. Our for loop simply counts the number of lines in the file and prints them out. The rough translation of the for loop into English is, “for each line in the file represented by the file handle, add one to the count variable.”

The reason that the open function does not read the entire file is that the file might be quite large with many gigabytes of data. The open statement takes the same amount of time regardless of the size of the file. The for loop actually causes the data to be read from the file.

When the file is read using a for loop in this manner, Python takes care of splitting the data in the file into separate lines using the newline character. Python reads each line through the newline and includes the newline as the last character in the line variable for each iteration of the for loop.

Because the for loop reads the data one line at a time, it can efficiently read and count the lines in very large files without running out of main memory to store the data. The above program can count the lines in any size file using very little memory since each line is read, counted, and then discarded.

If you know the file is relatively small compared to the size of your main memory, you can read the whole file into one string using the read method on the file handle.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print(len(inp))
94626

>>> print(inp[:20])

From stephen.marquar
```

In this example, the entire contents (all 94,626 characters) of the file mbox-short.txt are read directly into the variable inp. We use string slicing to print out the first 20 characters of the string data stored in inp.

When the file is read in this manner, all the characters including all of the lines and newline characters are one big string in the variable *inp*. Remember that this form of the open function should only be used if the file data will fit comfortably in the main memory of your computer.

If the file is too large to fit in main memory, you should write your program to read the file in chunks using a for or while loop.

Searching through a file

When you are searching through data in a file, it is a very common pattern to read through a file, ignoring most of the lines and only processing lines which meet a particular condition. We can combine the pattern for reading a file with string methods to build simple search mechanisms.

For example, if we wanted to read a file and only print out lines which started with the prefix “From:”, we could use the string method *startswith* to select only those lines with the desired prefix:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
        if line.startswith('From:'):
                print(line)
```

When this program runs, we get the following output:

```
From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu
...
```

The output looks great since the only lines we are seeing are those which start with “From:”, but why are we seeing the extra blank lines? This is due to that invisible *newline* character. Each of the lines ends with a newline, so the print statement prints the string in the variable *line* which includes a newline and then print adds *another* newline, resulting in the double spacing effect we see.

We could use line slicing to print all but the last character, but a simpler approach is to use the *rstrip* method which strips whitespace from the right side of a string as follows:

```
fhand = open('mbox-short.txt')
for line in fhand:
        line = line.rstrip()
        if line.startswith('From:'):
                print(line)
```

When this program runs, we get the following output:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
...
```

As your file processing programs get more complicated, you may want to structure your search loops using continue. The basic idea of the search loop is that you are looking for “interesting” lines and effectively skipping “uninteresting” lines. And then when we find an interesting line, we do something with that line.

We can structure the loop to follow the pattern of skipping uninteresting lines as follows:

```
fhand = open('mbox-short.txt')
for line in fhand:
        line = line.rstrip()
# Skip 'uninteresting lines'
        if not line.startswith('From:'):
                continue
# Process our 'interesting' line
        print(line)
```

The output of the program is the same. In English, the uninteresting lines are those which do not start with “From:”, which we skip using continue. For the “interesting” lines (i.e., those that start with “From:”) we perform the processing on those lines.

We can use the find string method to simulate a text editor search that finds lines where the search string is anywhere in the line. Since find looks for an occurrence of a string within another string and either returns the position of the string or -1 if the string was not found, we can write the following loop to show lines which contain the string “@uct.ac.za” (i.e., they come from the University of Cape Town in South Africa):

```
fhand = open('mbox-short.txt')
for line in fhand:
        line = line.rstrip()
        if line.find('@uct.ac.za') == -1: continue
        print(line)
```

Which produces the following output:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan 4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
...
```

Letting the user choose the file name

We really do not want to have to edit our Python code every time we want to process a different file. It would be more usable to ask the user to enter the file name string each time the program runs so they can use our program on different files without changing the Python code.

This is quite simple to do by reading the file name from the user using raw_input as follows:

```
fname = input('Enter the file name: ')
fhand = open(fname)
count = 0
for line in fhand:
        if line.startswith('Subject:'):
                count = count + 1
print('There were', count, 'subject lines in', fname)
```

We read the file name from the user and place it in a variable named fname and open that file. Now we can run the program repeatedly on different files.

```
python search6.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search6.py
Enter the file name: mbox-short.txt
There were 27 subject lines in mbox-short.txt
```

Before peeking at the next section, take a look at the above program and ask yourself, “What could go possibly wrong here?” or “What might our friendly user do that would cause our nice little program to ungracefully exit with a traceback, making us look not-so-cool in the eyes of our users?”

Using try, except, and open

I told you not to peek. This is your last chance.
What if our user types something that is not a file name?

```
python search6.py
Enter the file name: missing.txt
Traceback (most recent call last):
File "search6.py", line 2, in <module>
fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'missing.txt'
```

```
python search6.py
Enter the file name: na na boo boo
Traceback (most recent call last):
File "search6.py", line 2, in <module>
fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'na na boo boo'
```

Do not laugh. Users will eventually do every possible thing they can do to break your programs, either on purpose or with malicious intent. As a matter of fact, an important part of any software development team is a person or group called *Quality Assurance* (or QA for short) whose very job it is to do the craziest things possible in an attempt to break the software that the programmer has created.

The QA team is responsible for finding the flaws in programs before we have delivered the program to the end users who may be purchasing the software or paying our salary to write the software. So the QA team is the programmer's best friend.

So now that we see the flaw in the program, we can elegantly fix it using the try/except structure. We need to assume that the open call might fail and add recovery code when the open fails as follows:

```
fname = input('Enter the file name: ')
try:
        fhand = open(fname)
except:
        print('File cannot be opened:', fname)
        exit()
count = 0
for line in fhand:
        if line.startswith('Subject:'):
                count = count + 1
print('There were', count, 'subject lines in', fname)
```

The exit function terminates the program. It is a function that we call that never returns. Now when our user (or QA team) types in silliness or bad file names, we "catch" them and recover gracefully:

```
7python search7.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search7.py
Enter the file name: na na boo boo
File cannot be opened: na na boo boo
```

Protecting the open call is a good example of the proper use of try and except in a Python program. We use the term "Pythonic" when we are doing something the "Python way". We might say that the above example is the Pythonic way to open a file.

Once you become more skilled in Python, you can engage in repartee with other Python programmers to decide which of two equivalent solutions to a problem is "more Pythonic". The goal to be "more Pythonic" captures the notion that programming is part engineering and part art. We are not always interested in just making something work, we also want our solution to be elegant and to be appreciated as elegant by our peers.

Writing files

To write a file, you have to open it with mode "w" as a second parameter:

```
>>> fout = open('output.txt', 'w')
>>> print(fout)
<_io.TextIOWrapper name='output.txt' mode='w' encoding='cp1252'>
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn't exist, a new one is created.

The write method of the file handle object puts data into the file, returning the number of characters written. The default write mode is text for writing (and reading) strings.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

Again, the file object keeps track of where it is, so if you call write again, it adds the new data to the end.

We must make sure to manage the ends of lines as we write to the file by explicitly inserting the newline character when we want to end a line. The print statement automatically appends a newline, but the write method does not add the newline automatically.

```
>>> line2 = 'the emblem of our land.\n'
>>> fout.write(line2)
24
```

When you are done writing, you have to close the file to make sure that the last bit of data is physically written to the disk so it will not be lost if the power goes off.

```
>>> fout.close()
```

We could close the files which we open for read as well, but we can be a little sloppy if we are only opening a few files since Python makes sure that all open files are closed when the program ends. When we are writing files, we want to explicitly close the files so as to leave nothing to chance.

Lists

A list is a sequence

Like a string, a *list* is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in list are called *elements* or sometimes *items*.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):

[10, 20, 30, 40] or [‘crunchy frog’, ‘ram bladder’, ‘lark vomit’]

The first example is a list of four integers. The second is a list of three strings. The elements of a list don’t have to be the same type. The following list contains a string, a float, an integer, and (lo!) another list:

```
['spam', 2.0, 5, [10, 20]]
```

A list within another list is *nested*. A list that contains no elements is called an empty list; you can create one with empty brackets, [].

As you might expect, you can assign list values to variables:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

Lists are mutable

The syntax for accessing the elements of a list is the same as for accessing the characters of a string: the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
>>> print(cheeses[0])
Cheddar
```

Unlike strings, lists are mutable because you can change the order of items in a list or reassign an item in a list. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print(numbers)
[17, 5]
```

The one-eth element of numbers, which used to be 123, is now 5.

You can think of a list as a relationship between indices and elements. This relationship is called a *mapping*; each index “maps to” one of the elements.

List indices work the same way as string indices:

• Any integer expression can be used as an index.

• If you try to read or write an element that does not exist, you get an IndexError.

• If an index has a negative value, it counts backward from the end of the list.

The in operator also works on lists.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True

>>> 'Brie' in cheeses
False
```

Traversing a list

The most common way to traverse the elements of a list is with a for loop. The syntax is the same as for strings:

```
for cheese in cheeses:
        print(cheese)
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions range and len:

```
for i in range(len(numbers)):
        numbers[i] = numbers[i] * 2
```

This loop traverses the list and updates each element. len returns the number of elements in the list. range returns a list of indices from 0 to *n* – 1, where *n* is the length of the list. Each time through the loop, i gets the index of the next element. The assignment statement in the body uses i to read the old value of the element and to assign the new value.

A for loop over an empty list never executes the body:

```
for x in empty:
        print('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

List operations

The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

Similarly, the * operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats four times. The second example repeats the list three times.

List slices

The slice operator also works on lists:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']

>>> t[:4]
['a', 'b', 'c', 'd']

>>> t[3:]
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle, or mutilate lists.

A slice operator on the left side of an assignment can update multiple elements:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

List methods

Python provides methods that operate on lists. For example, append adds a new element to the end of a list:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
```

extend takes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

This example leaves t2 unmodified.

sort arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

Most list methods are void; they modify the list and return None. If you accidentally write t = t.sort(), you will be disappointed with the result.

Deleting elements

There are several ways to delete elements from a list. If you know the index of the element you want, you can use pop:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']

>>> print(x)
b
```

pop modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

If you don't need the removed value, you can use the del operator:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print(t)
['a', 'c']
```

If you know the element you want to remove (but not the index), you can use remove:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
['a', 'c']
```

The return value from remove is None.

To remove more than one element, you can use del with a slice index:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print(t)
['a', 'f']
```

As usual, the slice selects all the elements up to, but not including, the second index.

Lists and functions

There are a number of built-in functions that can be used on lists that allow you to quickly look through a list without writing your own loops:

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6

>>> print(max(nums))
74

>>> print(min(nums))
3

>>> print(sum(nums))
154

>>> print(sum(nums)/len(nums))
25
```

The sum() function only works when the list elements are numbers. The other functions (max(), len(), etc.) work with lists of strings and other types that can be comparable.

We could rewrite an earlier program that computed the average of a list of numbers entered by the user using a list.

First, the program to compute an average without a list:

```
total = 0
count = 0
while (True):
        inp = input('Enter a number: ')
        if inp == 'done': break
        value = float(inp)
        total = total + value
        count = count + 1
average = total / count
print('Average:', average)
```

In this program, we have count and total variables to keep the number and running total of the user's numbers as we repeatedly prompt the user for a number.

We could simply remember each number as the user entered it and use built-in functions to compute the sum and count at the end.

```
numlist = list()
while (True):
        inp = input('Enter a number: ')
        if inp == 'done': break
        value = float(inp)
        numlist.append(value)
average = sum(numlist) / len(numlist)
print('Average:', average)
```

We make an empty list before the loop starts, and then each time we have a number, we append it to the list. At the end of the program, we simply compute the sum of the numbers in the list and divide it by the count of the numbers in the list to come up with the average.

Lists and strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use list:

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

Because list is the name of a built-in function, you should avoid using it as a variable name. I also avoid the letter l because it looks too much like the number. So that's why I use t.

The list function breaks a string into individual letters. If you want to break a string into words, you can use the split method:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print(t)
['pining', 'for', 'the', 'fjords']

>>> print(t[2])
the
```

Once you have used split to break the string into a list of words, you can use the index operator (square bracket) to look at a particular word in the list.

You can call split with an optional argument called a *delimiter* that specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

join is the inverse of split. It takes a list of strings and concatenates the elements. join is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

In this case the delimiter is a space character, so join puts a space between words. To concatenate strings without spaces, you can use the empty string, “”, as a delimiter.

Parsing lines

Usually when we are reading a file we want to do something to the lines other than just printing the whole line. Often we want to find the “interesting lines” and then *parse* the line to find some interesting *part* of the line. What if we wanted to print out the day of the week from those lines that start with “From”?

From stephen.marquard@uct.ac.za*Sat* Jan 5 09:14:16 2008

The split method is very effective when faced with this kind of problem. We can write a small program that looks for lines where the line starts with “From”, split those lines, and then print out the third word in the line:

```
fhand = open('mbox-short.txt')
for line in fhand:
        line = line.rstrip()
        if not line.startswith('From '): continue
        words = line.split()
        print(words[2])
```

Here we also use the contracted form of the if statement where we put the continue on the same line as the if. This contracted form of the if functions the same as if the continue were on the next line and indented.

The program produces the following output:

```
Sat
Fri
Fri
Fri
...
```

Later, we will learn increasingly sophisticated techniques for picking the lines to work on and how we pull those lines apart to find the exact bit of information we are looking for.

Objects and values

If we execute these assignment statements:

```
a = 'banana'
b = 'banana'
```

we know that a and b both refer to a string, but we don't know whether they refer to the *same* string. There are two possible states:
a is pointing to its own 'banana'
b is pointing to its own 'banana'

or

both a and b are pointing to the same banana

In one case, a and b refer to two different objects that have the same value. In the second case, they refer to the same object.

To check whether two variables refer to the same object, you can use the is operator.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

In this example, Python only created one string object, and both a and b refer to it.

But when you create two lists, you get two objects:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

In this case we would say that the two lists are *equivalent*, because they have the same elements, but not *identical*, because they are not the same object. If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

Until now, we have been using “object” and “value” interchangeably, but it is more precise to say that an object has a value. If you execute a = [1,2,3], a refers to a list object whose value is a particular sequence of elements. If another list has the same elements, we would say it has the same value.

Aliasing

If a refers to an object and you assign b = a, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

The association of a variable with an object is called a *reference*. In this example, there are two references to the same object.

An object with more than one reference has more than one name, so we say that the object is *aliased*.

If the aliased object is mutable, changes made with one alias affect the other:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.

For immutable objects like strings, aliasing is not as much of a problem. In this example:

```
a = 'banana'
b = 'banana'
```

it almost never makes a difference whether a and b refer to the same string or not.

List arguments

When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change.

For example, delete_head removes the first element from a list:

```
def delete_head(t):
        del t[0]
```

Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print(letters)
['b', 'c']
```

The parameter t and the variable letters are aliases for the same object.

It is important to distinguish between operations that modify lists and operations that create new lists. For example, the append method modifies a list, but the + operator creates a new list:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)
[1, 2, 3]

>>> print(t2)
None

>>> t3 = t1 + [3]
>>> print(t3)
[1, 2, 3]

>>> t2 is t3
False
```

This difference is important when you write functions that are supposed to modify lists. For example, this function *does not* delete the head of a list:

```
def bad_delete_head(t):
    t = t[1:] # WRONG!
```

The slice operator creates a new list and the assignment makes t refer to it, but none of that has any effect on the list that was passed as an argument.

An alternative is to write a function that creates and returns a new list. For example, tail returns all but the first element of a list:

```
def tail(t):
    return t[1:]
```

This function leaves the original list unmodified. Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print(rest)
['b', 'c']
```

Dictionaries

A *dictionary* is like a list, but more general. In a list, the index positions have to be integers; in a dictionary, the indices can be (almost) any type.

You can think of a dictionary as a mapping between a set of indices (which are called *keys*) and a set of values. Each key maps to a value. The association of a key and a value is called a *key-value pair* or sometimes an *item*.

As an example, we'll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

The function dict creates a new dictionary with no items. Because dict is the name of a built-in function, you should avoid using it as a variable name.

```
>>> eng2sp = dict()
>>> print(eng2sp)
{}
```

The curly brackets, {}, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

```
>>> eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key 'one' to the value "uno". If we print the dictionary again, we see a key-value pair with a colon between the key and value:

```
>>> print(eng2sp)
{'one': 'uno'}
```

This output format is also an input format. For example, you can create a new dictionary with three items. But if you print eng2sp, you might be surprised:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
>>> print(eng2sp)
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

The order of the key-value pairs is not the same. In fact, if you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

```
>>> print(eng2sp['two'])
'dos'
```

The key 'two' always maps to the value "dos" so the order of the items doesn't matter.

If the key isn't in the dictionary, you get an exception:

```
>>> print(eng2sp['four'])
KeyError: 'four'
```

The len function works on dictionaries; it returns the number of key-value pairs:

```
>>> len(eng2sp)
3
```

The in operator works on dictionaries; it tells you whether something appears as a *key* in the dictionary (appearing as a value is not good enough).

```
>>> 'one' in eng2sp
True

>>> 'uno' in eng2sp
False
```

To see whether something appears as a value in a dictionary, you can use the method values, which returns the values as a list, and then use the in operator:

```
>>> vals = list(eng2sp.values())
>>> 'uno' in vals
True
```

The in operator uses different algorithms for lists and dictionaries. For lists, it uses a linear search algorithm. As the list gets longer, the search time gets longer in direct proportion to the length of the list. For dictionaries, Python uses an algorithm called a *hash table* that has a remarkable property: the in operator takes about the same amount of time no matter how many items there are in a dictionary

Dictionary as a set of counters

Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.

2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function ord), use the number as an index into the list, and increment the appropriate counter.

3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way.

An *implementation* is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don’t have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

Here is what the code might look like:

```
word = 'brontosaurus'
d = dict()
for c in word:
        if c not in d:
                d[c] = 1
        else:
        d[c] = d[c] + 1
print(d)
```

We are effectively computing a *histogram*, which is a statistical term for a set of counters (or frequencies).

The for loop traverses the string. Each time through the loop, if the character c is not in the dictionary, we create a new item with key c and the initial value 1 (since we have seen this letter once). If c is already in the dictionary we increment d[c].

Here's the output of the program:

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

The histogram indicates that the letters 'a' and "b" appear once; "o" appears twice, and so on.

Dictionaries have a method called get that takes a key and a default value. If the key appears in the dictionary, get returns the corresponding value; otherwise it returns the default value. For example:

```
>>> counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
>>> print(counts.get('jan', 0))
100

>>> print(counts.get('tim', 0))
0
```

We can use get to write our histogram loop more concisely. Because the get method automatically handles the case where a key is not in a dictionary, we can reduce four lines down to one and eliminate the if statement.

```
word = 'brontosaurus'
d = dict()
for c in word:
        d[c] = d.get(c,0) + 1
print(d)
```

The use of the get method to simplify this counting loop ends up being a very commonly used “idiom” in Python and we will use it many times in the rest of the book. So you should take a moment and compare the loop using the if statement and in operator with the loop using the get method. They do exactly the same thing, but one is more succinct.

Dictionaries and files

One of the common uses of a dictionary is to count the occurrence of words in a file with some written text. Let’s start with a very simple file of words taken from the text of *Romeo and Juliet*.

For the first set of examples, we will use a shortened and simplified version of the text with no punctuation. Later we will work with the text of the scene with punctuation included.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

We will write a Python program to read through the lines of the file, break each line into a list of words, and then loop through each of the words in the line and count each word using a dictionary.

You will see that we have two for loops. The outer loop is reading the lines of the file and the inner loop is iterating through each of the words on that particular line. This is an example of a pattern called *nested loops* because one of the loops is the *outer* loop and the other loop is the *inner* loop.

Because the inner loop executes all of its iterations each time the outer loop makes a single iteration, we think of the inner loop as iterating “more quickly” and the outer loop as iterating more slowly.

The combination of the two nested loops ensures that we will count every word on every line of the input file.

```
fname = input('Enter the file name: ')
try:
        fhand = open(fname)
except:
        print('File cannot be opened:', fname)
        exit()
counts = dict()
for line in fhand:
        words = line.split()
        for word in words:
                if word not in counts:
                        counts[word] = 1
                else:
                        counts[word] += 1
print(counts)
```

When we run the program, we see a raw dump of all of the counts in unsorted hash order.

```
python count1.py
Enter the file name: romeo.txt
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1,
'is': 3, 'through': 1, 'pale': 1, 'yonder': 1,
'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1,
'window': 1, 'sick': 1, 'east': 1, 'breaks': 1,
'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1,
'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

It is a bit inconvenient to look through the dictionary to find the most common words and their counts, so we need to add some more Python code to get us the output that will be more helpful.

Looping and dictionaries

If you use a dictionary as the sequence in a for statement, it traverses the keys of the dictionary. This loop prints each key and the corresponding value:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
        print(key, counts[key])
```

Here's what the output looks like:

```
jan 100
chuck 1
annie 42
```

Again, the keys are in no particular order.

We can use this pattern to implement the various loop idioms that we have described earlier. For example if we wanted to find all the entries in a dictionary with a value above ten, we could write the following code:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
        if counts[key] > 10 :
                print(key, counts[key])
```

The for loop iterates through the *keys* of the dictionary, so we must use the index operator to retrieve the corresponding *value* for each key. Here's what the output looks like:

```
jan 100
annie 42
```

We see only the entries with a value above 10.

If you want to print the keys in alphabetical order, you first make a list of the keys in the dictionary using the keys method available in dictionary objects, and then sort that list and loop through the sorted list, looking up each key and printing out key-value pairs in sorted order as follows:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
lst = list(counts.keys())
print(lst)
lst.sort()
for key in lst:
        print(key, counts[key])
```

Here's what the output looks like:

```
['jan', 'chuck', 'annie']
annie 42
chuck 1
jan 100
```

First you see the list of keys in unsorted order that we get from the keys method. Then we see the key-value pairs in order from the for loop.

Advanced text parsing

In the above example using the file romeo.txt, we made the file as simple as possible by removing all punctuation by hand. The actual text has lots of punctuation, as shown below.

But, soft! what light through yonder window breaks?
It is the east, and Juliet is the sun.
Arise, fair sun, and kill the envious moon,
Who is already sick and pale with grief,

Since the Python split function looks for spaces and treats words as tokens separated by spaces, we would treat the words “soft!” and “soft” as *different* words and create a separate dictionary entry for each word.

Also since the file has capitalization, we would treat “who” and “Who” as different words with different counts.

We can solve both these problems by using the string methods lower, punctuation, and translate. The translate is the most subtle of the methods. Here is the documentation for translate:

line.translate(str.maketrans(fromstr, tostr, deletestr))

Replace the characters in fromstr with the character in the same position in tostr and delete all characters that are in deletestr. The fromstr and tostr can be empty strings and the deletestr parameter can be omitted.

We will not specify the table but we will use the deletechars parameter to delete all of the punctuation. We will even let Python tell us the list of characters that it considers “punctuation”:

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

The parameters used by translate were different in Python 2.0.

We make the following modifications to our program:

```
import string
fname = input('Enter the file name: ')
try:
        fhand = open(fname)
except:
        print('File cannot be opened:', fname)
        exit()
counts = dict()
for line in fhand:
        line = line.rstrip()
        line = line.translate(line.maketrans('', '', string.punctuation))
        line = line.lower()
        words = line.split()
        for word in words:
                if word not in counts:
                        counts[word] = 1
                else:
                        counts[word] += 1
print(counts)
```

Part of learning the “Art of Python” or “Thinking Pythonically” is realizing that Python often has built-in capabilities for many common data analysis problems. Over time, you will see enough example code and read enough of the documentation to know where to look to see if someone has already written something that makes your job much easier.

The following is an abbreviated version of the output:

```
Enter the file name: romeo-full.txt
{'swearst': 1, 'all': 6, 'afeard': 1, 'leave': 2, 'these': 2,
'kinsmen': 2, 'what': 11, 'thinkst': 1, 'love': 24, 'cloak': 1,
a': 24, 'orchard': 2, 'light': 5, 'lovers': 2, 'romeo': 40,
'maiden': 1, 'whiteupturned': 1, 'juliet': 32, 'gentleman': 1,
'it': 22, 'leans': 1, 'canst': 1, 'having': 1, ...}
```

Looking through this output is still unwieldy and we can use Python to give us exactly what we are looking for, but to do so, we need to learn about Python *tuples*.

We will pick up this example once we learn about tuples.

Tuples

Tuples are immutable

A tuple is a sequence of values much like a list. The values stored in a tuple can be any type, and they are indexed by integers. The important difference is that tuples are *immutable*. Tuples are also *comparable* and *hashable* so we can sort lists of them and use tuples as key values in Python dictionaries.

Syntactically, a tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses to help us quickly identify tuples when we look at Python code:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, you have to include the final comma:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

Without the comma Python treats (’a’) as an expression with a string in parentheses that evaluates to a string:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

Another way to construct a tuple is the built-in function tuple. With no argument, it creates an empty tuple:

```
>>> t = tuple()
>>> print(t)
()
```

If the argument is a sequence (string, list, or tuple), the result of the call to tuple is a tuple with the elements of the sequence:

```
>>> t = tuple('lupins')
>>> print(t)
('l', 'u', 'p', 'i', 'n', 's')
```

Because tuple is the name of a constructor, you should avoid using it as a variable name.

Most list operators also work on tuples. The bracket operator indexes an element:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

And the slice operator selects a range of elements.

```
>>> print(t[1:3])
('b', 'c')
```

But if you try to modify one of the elements of the tuple, you get an error:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

You can't modify the elements of a tuple, but you can replace one tuple with another:

```
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd', 'e')
```

Comparing tuples

The comparison operators work with tuples and other sequences. Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next element, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

```
>>> (0, 1, 2) < (0, 3, 4)
True

>>> (0, 1, 2000000) < (0, 3, 4)
True
```

The sort function works the same way. It sorts primarily by first element, but in the case of a tie, it sorts by second element, and so on.

This feature lends itself to a pattern called *DSU* for

Decorate a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence,

Sort the list of tuples using the Python built-in sort, and

Undecorate by extracting the sorted elements of the sequence.

For example, suppose you have a list of words and you want to sort them from longest to shortest:

```
txt = 'but soft what light in yonder window breaks'
words = txt.split()
t = list()
for word in words:
        t.append((len(word), word))
t.sort(reverse=True)
res = list()
for length, word in t:
        res.append(word)
print(res)
```

The first loop builds a list of tuples, where each tuple is a word preceded by its length.

sort compares the first element, length, first, and only considers the second element to break ties. The keyword argument reverse=True tells sort to go in decreasing order.

The second loop traverses the list of tuples and builds a list of words in descending order of length. The four-character words are sorted in *reverse* alphabetical order, so “what” appears before “soft” in the following list.

The output of the program is as follows:

```
['yonder', 'window', 'breaks', 'light', 'what',
'soft', 'but', 'in']
```

Of course the line loses much of its poetic impact when turned into a Python list and sorted in descending word length order.

Tuple assignment

One of the unique syntactic features of the Python language is the ability to have a tuple on the left side of an assignment statement. This allows you to assign more than one variable at a time when the left side is a sequence.

In this example we have a two-element list (which is a sequence) and assign the first and second elements of the sequence to the variables x and y in a single statement.

```
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
'have'

>>> y
'fun'
```

It is not magic, Python *roughly* translates the tuple assignment syntax to be the following:

```
>>> m = [ 'have', 'fun' ]
>>> x = m[0]
>>> y = m[1]
>>> x
'have'

>>> y
'fun'
```

Stylistically when we use a tuple on the left side of the assignment statement, we omit the parentheses, but the following is an equally valid syntax:

```
>>> m = [ 'have', 'fun' ]
>>> (x, y) = m
>>> x
'have'

>>> y
'fun'
```

A particularly clever application of tuple assignment allows us to *swap* the values of two variables in a single statement:

```
>>> a, b = b, a
```

Python does not translate the syntax literally. For example, if you try this with a dictionary, it will not work as might expect.

Both sides of this statement are tuples, but the left side is a tuple of variables; the right side is a tuple of expressions. Each value on the right side is assigned to its respective variable on the left side. All the expressions on the right side are evaluated before any of the assignments.

The number of variables on the left and the number of values on the right must bethe same:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

More generally, the right side can be any kind of sequence (string, list, or tuple).

For example, to split an email address into a user name and a domain, you could write:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

The return value from split is a list with two elements; the first element is assigned to uname, the second to domain.

```
>>> print(uname)
Monty

>>> print(domain)
python.org
```

Dictionaries and tuples

Dictionaries have a method called items that returns a list of tuples, where each tuple is a key-value pair:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> print(t)
[('b', 1), ('a', 10), ('c', 22)]
```

As you should expect from a dictionary, the items are in no particular order.

However, since the list of tuples is a list, and tuples are comparable, we can now sort the list of tuples. Converting a dictionary to a list of tuples is a way for us to output the contents of a dictionary sorted by key:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> t
[('b', 1), ('a', 10), ('c', 22)]

>>> t.sort()
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

The new list is sorted in ascending alphabetical order by the key value.

Multiple assignment with dictionaries

Combining items, tuple assignment, and for, you can see a nice code pattern for traversing the keys and values of a dictionary in a single loop:

```
for key, val in list(d.items()):
        print(val, key)
```

This loop has two *iteration variables* because items returns a list of tuples and key, val is a tuple assignment that successively iterates through each of the key-value pairs in the dictionary.

For each iteration through the loop, both key and value are advanced to the next key-value pair in the dictionary (still in hash order).

The output of this loop is:

```
10 a
22 c
1 b
```

Again, it is in hash key order (i.e., no particular order).

If we combine these two techniques, we can print out the contents of a dictionary sorted by the *value* stored in each key-value pair.

To do this, we first make a list of tuples where each tuple is (value, key). The items method would give us a list of (key, value) tuples, but this time we want to sort by value, not key. Once we have constructed the list with the value-key tuples, it is a simple matter to sort the list in reverse order and print out the new, sorted list.

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
>>> for key, val in d.items() :
... l.append( (val, key) )
...
>>> l
[(10, 'a'), (22, 'c'), (1, 'b')]

>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
```

By carefully constructing the list of tuples to have the value as the first element of each tuple, we can sort the list of tuples and get our dictionary contents sorted by value.

The most common words

Coming back to our running example of the text from *Romeo and Juliet* Act 2, Scene 2, we can augment our program to use this technique to print the ten most common words in the text as follows:

```
import string
fhand = open('romeo-full.txt')
counts = dict()
for line in fhand:
        line = line.translate(str.maketrans('', '', string.punctuation))
        line = line.lower()
        words = line.split()
        for word in words:
                if word not in counts:
                        counts[word] = 1
                else:
                        counts[word] += 1
# Sort the dictionary by value
lst = list()
for key, val in list(counts.items()):
        lst.append((val, key))
lst.sort(reverse=True)
for key, val in lst[:10]:
        print(key, val)
```

The first part of the program which reads the file and computes the dictionary that maps each word to the count of words in the document is unchanged. But instead of simply printing out counts and ending the program, we construct a list of (val, key) tuples and then sort the list in reverse order.

Since the value is first, it will be used for the comparisons. If there is more than one tuple with the same value, it will look at the second element (the key), so tuples where the value is the same will be further sorted by the alphabetical order of the key.

At the end we write a nice for loop which does a multiple assignment iteration and prints out the ten most common words by iterating through a slice of the list (lst[:10]).

So now the output finally looks like what we want for our word frequency analysis.

```
61 i
42 and
40 romeo
34 to
34 the
32 thou
32 juliet
30 that
29 my
24 thee
```

The fact that this complex data parsing and analysis can be done with an easy-tounderstand 19-line Python program is one reason why Python is a good choice as a language for exploring information.

Using tuples as keys in dictionaries

Because tuples are *hashable* and lists are not, if we want to create a *composite* key to use in a dictionary we must use a tuple as the key.

We would encounter a composite key if we wanted to create a telephone directory that maps from last-name, first-name pairs to telephone numbers. Assuming that we have defined the variables last, first, and number, we could write a dictionary assignment statement as follows:

```
directory[last,first] = number
```

The expression in brackets is a tuple. We could use tuple assignment in a for loop to traverse this dictionary.

```
for last, first in directory:
        print(first, last, directory[last,first])
```

This loop traverses the keys in directory, which are tuples. It assigns the elements of each tuple to last and first, then prints the name and corresponding telephone number.

Sequences: strings, lists, and tuples - Oh My!

I have focused on lists of tuples, but almost all of the examples in this section also work with lists of lists, tuples of tuples, and tuples of lists. To avoid enumerating the possible combinations, it is sometimes easier to talk about sequences of sequences.

In many contexts, the different kinds of sequences (strings, lists, and tuples) can be used interchangeably. So how and why do you choose one over the others?

To start with the obvious, strings are more limited than other sequences because the elements have to be characters. They are also immutable. If you need the ability to change the characters in a string (as opposed to creating a new string), you might want to use a list of characters instead.

Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

1. In some contexts, like a return statement, it is syntactically simpler to create a tuple than a list. In other contexts, you might prefer a list.

2. If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.

3. If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.

Because tuples are immutable, they don't provide methods like sort and reverse, which modify existing lists. However Python provides the built-in functions sorted and reversed, which take any sequence as a parameter and return a new sequence with the same elements in a different order.

Regular expressions

So far we have been reading through files, looking for patterns and extracting various bits of lines that we find interesting. We have been using string methods like split and find and using lists and string slicing to extract portions of the lines.

This task of searching and extracting is so common that Python has a very powerful library called *regular expressions* that handles many of these tasks quite elegantly. The reason we have not introduced regular expressions earlier in the book is because while they are very powerful, they are a little complicated and their syntax takes some getting used to.

Regular expressions are almost their own little programming language for searching and parsing strings. As a matter of fact, entire books have been written on the topic of regular expressions. In this section, we will only cover the basics of regular expressions.

The regular expression library re must be imported into your program before you can use it. The simplest use of the regular expression library is the search() function. The following program demonstrates a trivial use of the search function.

```
# Search for lines that contain 'From'
import re
hand = open('mbox-short.txt')
for line in hand:
        line = line.rstrip()
        if re.search('From:', line):
                print(line)
```

We open the file, loop through each line, and use the regular expression search() to only print out lines that contain the string “From:”. This program does not use the real power of regular expressions, since we could have just as easily used line.find() to accomplish the same result.

The power of the regular expressions comes when we add special characters to the search string that allow us to more precisely control which lines match the string. Adding these special characters to our regular expression allow us to do sophisticated matching and extraction while writing very little code.

For example, the caret character is used in regular expressions to match “the beginning” of a line. We could change our program to only match lines where “From:” was at the beginning of the line as follows:

```
# Search for lines that start with 'From'
import re
hand = open('mbox-short.txt')
for line in hand:
        line = line.rstrip()
        if re.search('^From:', line):
                print(line)
```

Now we will only match lines that *start with* the string “From:”. This is still a very simple example that we could have done equivalently with the startswith() method from the string library. But it serves to introduce the notion that regular expressions contain special action characters that give us more control as to what will match the regular expression.

Character matching in regular expressions

There are a number of other special characters that let us build even more powerful regular expressions. The most commonly used special character is the period or full stop, which matches any character.

In the following example, the regular expression “F..m:” would match any of the strings “From:”, “Fxxm:”, “F12m:”, or “F!@m:” since the period characters in the regular expression match any character.

```
# Search for lines that start with 'F', followed by
# 2 characters, followed by 'm:'
import re
hand = open('mbox-short.txt')
for line in hand:
        line = line.rstrip()
        if re.search('^F..m:', line):
                print(line)
```

This is particularly powerful when combined with the ability to indicate that a character can be repeated any number of times using the “*” or “+” characters in your regular expression. These special characters mean that instead of matching a single character in the search string, they match zero-or-more characters (in the case of the asterisk) or one-or-more of the characters (in the case of the plus sign).

We can further narrow down the lines that we match using a repeated *wild card* character in the following example:

```
# Search for lines that start with From and have an at sign
import re
hand = open('mbox-short.txt')
for line in hand:
        line = line.rstrip()
        if re.search('^From:.+@', line):
                print(line)
```

The search string “^From:.+@” will successfully match lines that start with “From:”, followed by one or more characters (“.+”), followed by an at-sign. So this will match the following line:

From: uct.ac.za

You can think of the “.+” wildcard as expanding to match all the characters between the colon character and the at-sign.

From:

It is good to think of the plus and asterisk characters as “pushy”. For example, the following string would match the last at-sign in the string as the “.+” pushes outwards, as shown below:

From: iupui.edu

It is possible to tell an asterisk or plus sign not to be so “greedy” by adding another character. See the detailed documentation for information on turning off the greedy behavior.

Extracting data using regular expressions

If we want to extract data from a string in Python we can use the findall() method to extract all of the substrings which match a regular expression. Let’s use the example of wanting to extract anything that looks like an email address from any line regardless of format. For example, we want to pull the email addresses from each of the following lines:

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
for <source@collab.sakaiproject.org>;
Received: (from apache@localhost)
Author: stephen.marquard@uct.ac.za

We don’t want to write code for each of the types of lines, splitting and slicing differently for each line. This following program uses findall() to find the lines with email addresses in them and extract one or more addresses from each of those lines.

```
import re
s = 'A message from csev@umich.edu to cwen@iupui.edu about meeting
@2PM'
lst = re.findall('\S+@\S+', s)
print(lst)
```

The findall() method searches the string in the second argument and returns a list of all of the strings that look like email addresses. We are using a two-character sequence that matches a non-whitespace character (\S).

The output of the program would be:

['csev@umich.edu', 'cwen@iupui.edu']

Translating the regular expression, we are looking for substrings that have at least one non-whitespace character, followed by an at-sign, followed by at least one more non-whitespace character. The "\S+" matches as many non-whitespace characters as possible.

The regular expression would match twice (csev@umich.edu and cwen@iupui.edu), but it would not match the string "@2PM" because there are no non-blank characters *before* the at-sign. We can use this regular expression in a program to read all the lines in a file and print out anything that looks like an email address as follows:

```
# Search for lines that have an at sign between characters
import re
hand = open('mbox-short.txt')
for line in hand:
        line = line.rstrip()
        x = re.findall('\S+@\S+', line)
        if len(x) > 0:
                print(x)
```

We read each line and then extract all the substrings that match our regular expression. Since findall() returns a list, we simply check if the number of elements in our returned list is more than zero to print only lines where we found at least one substring that looks like an email address.

If we run the program on mbox.txt we get the following output:

```
['wagnermr@iupui.edu']
['cwen@iupui.edu']
 ['<postmaster@collab.sakaiproject.org>']
['<200801032122.m03LMFo4005148@nakamura.uits.iupui.edu>']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['apache@localhost)']
['source@collab.sakaiproject.org;']
```

Some of our email addresses have incorrect characters like "<" or ";" at the beginning or end. Let's declare that we are only interested in the portion of the string that starts and ends with a letter or a number.

To do this, we use another feature of regular expressions. Square brackets are used to indicate a set of multiple acceptable characters we are willing to consider matching. In a sense, the "\S" is asking to match the set of "non-whitespace characters". Now we will be a little more explicit in terms of the characters we will match.

Here is our new regular expression:

[a-zA-Z0-9]\S*@\S*[a-zA-Z]

This is getting a little complicated and you can begin to see why regular expressions are their own little language unto themselves. Translating this regular expression, we are looking for substrings that start with a *single* lowercase letter, uppercase letter, or number "[a-zA-Z0-9]", followed by zero or more non-blank characters ("\S*"), followed by an at-sign, followed by zero or more non-blank characters ("\S*"), followed by an uppercase or lowercase letter. Note that we switched from "+" to "*" to indicate zero or more non-blank characters since "[a-zA-Z0-9]" is already one non-blank character. Remember that the "*" or "+" applies to the single character immediately to the left of the plus or asterisk.

If we use this expression in our program, our data is much cleaner:

```
# Search for lines that have an at sign between characters
# The characters must be a letter or number
import re
hand = open('mbox-short.txt')
for line in hand:
        line = line.rstrip()
        x = re.findall('[a-zA-Z0-9]\S+@\S+[a-zA-Z]', line)
        if len(x) > 0:
                print(x)
...
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['postmaster@collab.sakaiproject.org']
['200801032122.m03LMFo4005148@nakamura.uits.iupui.edu']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['apache@localhost']
```

Notice that on the "source@collab.sakaiproject.org" lines, our regular expression eliminated two letters at the end of the string (">;"). This is because when we append "[a-zA-Z]" to the end of our regular expression, we are demanding that whatever string the regular expression parser finds must end with a letter. So when it sees the ">" after "sakaiproject.org>;" it simply stops at the last "matching" letter it found (i.e., the "g" was the last good match).

Also note that the output of the program is a Python list that has a string as the single element in the list.

Combining searching and extracting

If we want to find numbers on lines that start with the string “X-” such as:

X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000

we don’t just want any floating-point numbers from any lines. We only want to extract numbers from lines that have the above syntax.

We can construct the following regular expression to select the lines:

^X-.*: [0-9.]+

Translating this, we are saying, we want lines that start with “X-”, followed by zero or more characters (“.*”), followed by a colon (“:”) and then a space. After the space we are looking for one or more characters that are either a digit (0-9) or a period “[0-9.]+”. Note that inside the square brackets, the period matches an actual period (i.e., it is not a wildcard between the square brackets).

This is a very tight expression that will pretty much match only the lines we are interested in as follows:

```
# Search for lines that start with 'X' followed by any non
# whitespace characters and ':'
# followed by a space and any number.
# The number can include a decimal.
import re
hand = open('mbox-short.txt')
for line in hand:
        line = line.rstrip()
        if re.search('^X\S*: [0-9.]+', line):
                print(line)
```

When we run the program, we see the data nicely filtered to show only the lines we are looking for.

X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000

But now we have to solve the problem of extracting the numbers. While it would be simple enough to use split, we can use another feature of regular expressions to both search and parse the line at the same time.

Parentheses are another special character in regular expressions. When you add parentheses to a regular expression, they are ignored when matching the string. But when you are using findall(), parentheses indicate that while you want the whole expression to match, you only are interested in extracting a portion of the substring that matches the regular expression.

So we make the following change to our program:

```
# Search for lines that start with 'X' followed by any
# non whitespace characters and ':' followed by a space
# and any number. The number can include a decimal.
# Then print the number if it is greater than zero.
import re
hand = open('mbox-short.txt')
for line in hand:
        line = line.rstrip()
        x = re.findall('^X\S*: ([0-9.]+)', line)
        if len(x) > 0:
                print(x)
```

Instead of calling search(), we add parentheses around the part of the regular expression that represents the floating-point number to indicate we only want findall() to give us back the floating-point number portion of the matching string.

The output from this program is as follows:

```
['0.8475']
['0.0000']
['0.6178']
['0.0000']
['0.6961']
['0.0000']
..
```

The numbers are still in a list and need to be converted from strings to floating point, but we have used the power of regular expressions to both search and extract the information we found interesting.

As another example of this technique, if you look at the file there are a number of lines of the form:

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772

If we wanted to extract all of the revision numbers (the integer number at the end of these lines) using the same technique as above, we could write the following program:

```
# Search for lines that start with 'Details: rev='
# followed by numbers and '.'
# Then print the number if it is greater than zero
import re
hand = open('mbox-short.txt')
for line in hand:
        line = line.rstrip()
        x = re.findall('^Details:.*rev=([0-9.]+)', line)
        if len(x) > 0:
                print(x)
```

Translating our regular expression, we are looking for lines that start with “Details:”, followed by any number of characters (“.*”), followed by “rev=”, and then by one or more digits. We want to find lines that match the entire expression but we only want to extract the integer number at the end of the line, so we surround “[0-9]+” with parentheses.

When we run the program, we get the following output:

```
['39772']
['39771']
['39770']
['39769']
...
```

Remember that the “[0-9]+” is “greedy” and it tries to make as large a string of digits as possible before extracting those digits. This “greedy” behavior is why we get all five digits for each number. The regular expression library expands in both directions until it encounters a non-digit, or the beginning or the end of a line.

Now we can use regular expressions to redo an exercise from earlier in the material where we were interested in the time of day of each mail message. We looked for lines of the form:

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

and wanted to extract the hour of the day for each line. Previously we did this with two calls to split. First the line was split into words and then we pulled out the fifth word and split it again on the colon character to pull out the two characters we were interested in.

While this worked, it actually results in pretty brittle code that is assuming the lines are nicely formatted. If you were to add enough error checking (or a big try/except block) to insure that your program never failed when presented with incorrectly formatted lines, the code would balloon to 10-15 lines of code that was pretty hard to read.

We can do this in a far simpler way with the following regular expression:

^From .* [0-9][0-9]:

The translation of this regular expression is that we are looking for lines that start with “From” (note the space), followed by any number of characters (“.*”), followed by a space, followed by two digits “[0-9][0-9]”, followed by a colon character. This is the definition of the kinds of lines we are looking for.

In order to pull out only the hour using findall(), we add parentheses around the two digits as follows:

^From .* ([0-9][0-9]):

This results in the following program:

```
# Search for lines that start with From and a character
# followed by a two digit number between 00 and 99 followed by ':'
# Then print the number if it is greater than zero
import re
hand = open('mbox-short.txt')
for line in hand:
        line = line.rstrip()
        x = re.findall('^From .* ([0-9][0-9]):', line)
        if len(x) > 0: print(x)
```

When the program runs, it produces the following output:

```
['09']
['18']
['16']
['15']
...
```

Escape character

Since we use special characters in regular expressions to match the beginning or end of a line or specify wild cards, we need a way to indicate that these characters are “normal” and we want to match the actual character such as a dollar sign or caret.

We can indicate that we want to simply match a character by prefixing that character with a backslash. For example, we can find money amounts with the following regular expression.

```
import re
x = 'We just received $10.00 for cookies.'
y = re.findall('\$[0-9.]+',x)
```

Since we prefix the dollar sign with a backslash, it actually matches the dollar sign in the input string instead of matching the “end of line”, and the rest of the regular expression matches one or more digits or the period character. *Note:* Inside square brackets, characters are not “special”. So when we say “[0-9.]”, it really means digits or a period. Outside of square brackets, a period is the “wildcard” character and matches any character. Inside square brackets, the period is a period.

Summary

While this only scratched the surface of regular expressions, we have learned a bit about the language of regular expressions. They are search strings with special characters in them that communicate your wishes to the regular expression system as to what defines “matching” and what is extracted from the matched strings.

Here are some of those special characters and character sequences:

ˆ Matches the beginning of the line.

$ Matches the end of the line.

. Matches any character (a wildcard).

\s Matches a whitespace character.

\S Matches a non-whitespace character (opposite of \s).

* Applies to the immediately preceding character and indicates to match zero or more of the preceding character(s).

*? Applies to the immediately preceding character and indicates to match zero or more of the preceding character(s) in “non-greedy mode”.

+ Applies to the immediately preceding character and indicates to match one or more of the preceding character(s).

+? Applies to the immediately preceding character and indicates to match one or more of the preceding character(s) in “non-greedy mode”.

[aeiou] Matches a single character as long as that character is in the specified set. In this example, it would match “a”, “e”, “i”, “o”, or “u”, but no other characters.

[a-z0-9] You can specify ranges of characters using the minus sign. This example is a single character that must be a lowercase letter or a digit.

[^A-Za-z] When the first character in the set notation is a caret, it inverts the logic. This example matches a single character that is anything *other than* an uppercase or lowercase letter.

( ) When parentheses are added to a regular expression, they are ignored for the purpose of matching, but allow you to extract a particular subset of the matched string rather than the whole string when using findall().

\b Matches the empty string, but only at the start or end of a word.

\B Matches the empty string, but not at the start or end of a word.

\d Matches any decimal digit; equivalent to the set [0-9].

\D Matches any non-digit character; equivalent to the set [^0-9].

Appendix A

Copyright Detail

I would have preferred to license the book under the less restrictive CC-BY-SA license. But unfortunately there are a few unscrupulous organizations who search for and find freely licensed books, and then publish and sell virtually unchanged copies of the books on a print on demand service such as LuLu or CreateSpace.

CreateSpace has (thankfully) added a policy that gives the wishes of the actual copyright holder preference over a non-copyright holder attempting to publish a freely licensed work. Unfortunately there are many print-on-demand services and very few have as well-considered a policy as CreateSpace.

Regretfully, I added the NC element to the license this book to give me recourse in case someone tries to clone this book and sell it commercially. Unfortunately, adding NC limits uses of this material that I would like to permit. So I have added this section of the document to describe specific situations where I am giving my permission in advance to use the material in this book in situations that some might consider commercial.

• If you are printing a limited number of copies of all or part of this book for use in a course (e.g., like a coursepack), then you are granted CC-BY license to these materials for that purpose.

• If you are a teacher at a university and you translate this book into a language other than English and teach using the translated book, then you can contact me and I will granted you a CC-BY-SA license to these materials with respect to the publication of your translation. In particular, you will be permitted to sell the resulting translated book commercially.

If you are intending to translate the book, you may want to contact me so we can make sure that you have all of the related course materials so you can translate them as well.

Of course, you are welcome to contact me and ask for permission if these clauses are not sufficient. In all cases, permission to reuse and remix this material will be granted as long as there is clear added value or benefit to students or teachers that will accrue as a result of the new work.

Charles Severance
www.dr-chuck.com
Ann Arbor, MI, USA
September 9, 2013

Appendix B (vi Cheat Sheet)

Modes

vi has two modes, insertion mode and command mode. The editor begins in command mode, where the cursor movement and text deletion and pasting occur. Insertion mode begins upon entering an insertion or change command. [ESC] returns the editor to command mode (where you can quit, for example by typing :q!). Most commands execute as soon as you type them except for "colon" commands which execute when you press the return key.

Quitting

:x	Exit, saving changes
:q	Exit as long as there have been no changes
ZZ	Exit and save changes if any have been made
:q!	Exit and ignore any changes

Inserting Text

i	Insert before cursor
I	Insert before line
a	Append after cursor
A	Append after line
o	Open a new line after current line
O	Open a new line before current line
r	Replace one character
R	Replace many characters

Motion

h	Move left
j	Move down
k	Move up
l	Move right
w	Move to next word
W	Move to next blank delimited word
b	Move to the beginning of the word
B	Move to the beginning of blank delimited word
e	Move to the end of the word
E	Move to the end of Blank delimited word
(	Move a sentence back
)	Move a sentence forward
{	Move a paragraph back
}	Move a paragraph forward
0	Move to the beginning of the line
$	Move to the end of the line
1G	Move to the first line of the file
G	Move to the last line of the file
nG	Move to nth line of the file
:n	Move to nth line of the file
fc	Move forward to c
Fc	Move back to c
H	Move to top of screen
M	Move to middle of screen
L	Move to bottom of screen
%	Move to associated ( ), { }, [ ]

Deleting Text

Almost all deletion commands are performed by typing d followed by a motion. For example, dw deletes a word. A few other deletes are:

x	Delete character to the right of cursor
X	Delete character to the left of cursor
D	Delete to the end of the line
dd	Delete current line
:d	Delete current line

Yanking Text

Like deletion, almost all yank commands are performed by typing y followed by a motion. For example, y$ yanks to the end of the line. Two other yank commands are:

yy	Yank the current line
:y	Yank the current line

Changing text

The change command is a deletion command that leaves the editor in insert mode. It is performed by typing c followed by a motion. For example cw changes a word. A few other change commands are:

C	Change to the end of the line
cc	Change the whole line

Putting text

p	Put after the position or after the line
P	Put before the position or before the line

Buffers

Named buffers may be specified before any deletion, change, yank or put command. The general prefix has the form "c where c is any lowercase character. for example, "adw deletes a word into buffer a. It may thereafter be put back into text with an appropriate "ap.

Markers

Named markers may be set on any line in a file. Any lower case letter may be a marker name. Markers may also be used as limits for ranges.

mc	Set marker c on this line
`c	Go to beginning of marker c line.
'c	Go to first non-blank character of marker c line.

Search for strings

/*string*	Search forward for *string*
?*string*	Search back for *string*
n	Search for next instance of *string*
N	Search for previous instance of *string*

Replace

The search and replace function is accomplished with the :s command. It is commonly used in combination with ranges or the :g command (below).

:s/*pattern*/*string*/*flags*	Replace *pattern* with *string* according to *flags*.
g	Flag - Replace all occurrences of pattern
c	Flag - Confirm replaces.
&	Repeat last :s command

Regular Expressions

. (dot)	Any single character except newline
*	zero or more occurrences of any character
[...]	Any single character specified in the set
[^...]	Any single character not specified in the set
^	Anchor - beginning of the line
$	Anchor - end of line
\<	Anchor - beginning of word
\>	Anchor - end of word
\(...\)	Grouping - usually used to group conditions
\n	Contents of nth grouping

[...] - Set Examples	
[A-Z]	The SET from Capital A to Capital Z
[a-z]	The SET from lowercase a to lowercase z
[0-9]	The SET from 0 to 9 (All numerals)
[./=+]	The SET containing . (dot), / (slash), =, and +
[-A-F]	The SET from Capital A to Capital F and the dash (dashes must be specified first)
[0-9 A-Z]	The SET containing all capital letters and digits and a space
[A-Z][a-zA-Z]	In the first position, the SET from Capital A to Capital Z In the second character position, the SET containing all letters

Regular Expression Examples	
/Hello/	Matches if the line contains the value Hello
/^TEST$/	Matches if the line contains TEST by itself
/^[a-zA-Z]/	Matches if the line starts with any letter
/^[a-z].*/	Matches if the first character of the line is a-z and there is at least one more of any character following it
/2134$/	Matches if line ends with 2134
/\(21|35\)/	Matches is the line contains 21 or 35 Note the use of ( ) with the pipe symbol to specify the 'or' condition
/[0-9]*/	Matches if there are zero or more numbers in the line
/^[^#]/	Matches if the first character is not a # in the line
Notes: 1. Regular expressions are case sensitive 2. Regular expressions are to be used where *pattern* is specified	

Counts

Nearly every command may be preceded by a number that specifies how many times it is to be performed. For example, 5dw will delete 5 words and 3fe will move the cursor forward to the 3rd occurrence of the letter e. Even insertions may be repeated conveniently with this method, say to insert the same line 100 times.

Ranges

Ranges may precede most "colon" commands and cause them to be executed on a line or lines. For example :3,7d would delete lines 3-7. Ranges are commonly combined with the :s command to perform a replacement on several lines, as with :.,$s/pattern/string/g to make a replacement from the current line to the end of the file.

:n,m	Range – Lines n-m
:.	Range - Current line
:$	Range - Last line
:'c	Range - Marker c
:%	Range - All lines in file
:g/*pattern*/	Range - All lines that contain *pattern*

Files

:w *file*	Write to *file*
:r *file*	Read *file* in after line
:n	Go to next file
:p	Go to previous file
:e *file*	Edit *file*
!!*program*	Replace line with output from *program*

Other

~	Toggle upper and lower case
J	Join lines
.	Repeat last text-changing command
u	Undo last change
U	Undo all changes to line

Appendix 2 (ASCII Table)

ASCII	Oct	Dec	Hex	ASCII	Oct	Dec	Hex
(null)	000	0	00	3 063	51	33	
(SOH)	001	1	01	4 064	52	34	
(STX)	002	2	02	5 065	53	35	
(ETX)	003	3	03	6 066	54	36	
(EOT)	004	4	04	7 067	55	37	
(ENQ)	005	5	05	8 070	56	38	
(ACK)	006	6	06	9 071	57	39	
(beep)	007	7	07	: 072	58	3A	
(BS)	010	8	08	; 073	59	3B	
(tab)	011	9	09	< 074	60	3C	
(line feed)	012	10	0A	= 075	61	3D	
(home)	013	11	0B	> 076	62	3E	
(form feed)	014	12	0C	? 077	63	3F	
(carriage ret)	015	13	0D	@ 100	64	40	
(SO)	016	14	0E	A 101	65	41	
(SI)	017	15	0F	B 102	66	42	
(DLE)	020	16	10	C 103	67	43	
(DC1)	021	17	11	D 104	68	44	
(DC2)	022	20	12	E 105	69	45	
(DC3)	023	19	13	F 106	70	46	
(DC4)	024	20	14	G 107	71	47	
(NAK)	025	21	15	H 110	72	48	
(SYN)	026	22	16	I 111	73	49	
(ETB)	027	23	17	J 112	74	4A	
(CAN)	030	24	18	K 113	75	4B	
(EM)	031	25	19	L 114	76	4C	
(SUB)	032	26	1A	M 115	77	4D	
(ESC)	033	27	1B	N 116	78	4E	
(cursor right)	034	28	1C	O 117	79	4F	
(cursor left)	035	29	1D	P 120	80	50	
(cursor up)	036	30	1E	Q 121	81	51	
(cursor down)	037	31	1F	R 122	82	52	
(space)	040	32	20	S 123	83	53	
! 041	33	21		T 124	84	54	
“ 042	34	22		U 125	85	55	
# 043	35	23		V 126	86	56	
$ 044	36	24		W 127	87	57	
% 045	37	25		X 130	88	58	
& 046	38	26		Y 131	89	59	
‘ 047	39	27		Z 132	90	5A	
(050	40	28		[133	91	5B	
) 051	41	29		\ 134	92	5C	
* 052	42	2A		] 135	93	5D	
+ 053	43	2B		^ 136	94	5E	
, 054	44	2C		_ 137	95	5F	
- 055	45	2D		` 140	96	60	
. 056	46	2E		a 141	97	61	
/ 057	47	2F		b 142	98	62	
0 060	48	30		c 143	99	63	
1 061	49	31		d 144	100	64	
2 062	50	32		e 145	101	65	

Appendix 2 (ASCII Table)

ASCII		Oct	Dec	Hex	ASCII		Oct	Dec	Hex
f	146	102	66		ô	223	147	93	
g	147	103	67		ö	224	148	94	
h	150	104	68		ò	225	149	95	
i	151	105	69		û	226	150	96	
j	152	106	6A		ù	227	151	97	
k	153	107	6B		ÿ	230	152	98	
l	154	108	6C		Ö	231	153	99	
m	155	109	6D		Ü	232	154	9A	
n	156	110	6E		¢	233	155	9B	
o	157	111	6F		£	234	156	9C	
p	160	112	70		¥	235	157	9D	
q	161	113	71		P	236	158	9E	
r	162	114	72		ƒ	237	159	9F	
s	163	115	73		á	240	160	A0	
t	164	116	74		í	241	161	A1	
u	165	117	75		ó	242	162	A2	
v	166	118	76		ú	243	163	A3	
w	167	119	77		ñ	244	164	A4	
x	170	120	78		Ñ	245	165	A5	
y	171	121	79		ª	246	166	A6	
z	172	122	7A		º	247	167	A7	
{	173	123	7B		¿	248	168	A8	
|	174	124	7C		©	249	169	A9	
}	175	125	7D		¬	250	170	AA	
~	176	126	7E		½	251	171	AB	
_	177	127	7F		¼	252	172	AC	
Ç	200	128	80		¡	253	173	AD	
ü	201	129	81		«	254	174	AE	
é	202	130	82		»	255	175	AF	
â	203	131	83						
ä	204	132	84						
à	205	133	85						
å	206	134	86						
ç	207	135	87						
ê	210	136	88						
ë	211	137	89						
è	212	138	8A						
ï	213	139	8B						
î	214	140	8C						
ì	215	141	8D						
Ä	216	142	8E						
Å	217	143	8F						
É	220	144	90						
æ	221	145	91						
Æ	222	146	92						

&

.

/

A

B

C

D

E

F

G

H

I

J

K

L

M

N

P

Q

R

S

T

U

V

W

X