UNIT 4:FILES & LISTS

FILES:

4.1Persistence

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.

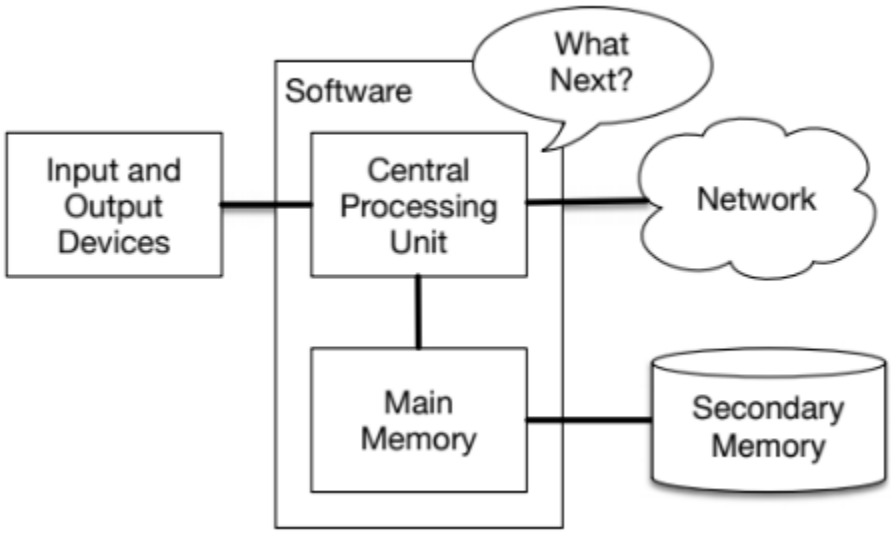

Figure 1: Secondary Memory

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.

4.2 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. You can download this file from

www.py4e.com/code3/mbox.txt

```
>>> 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.

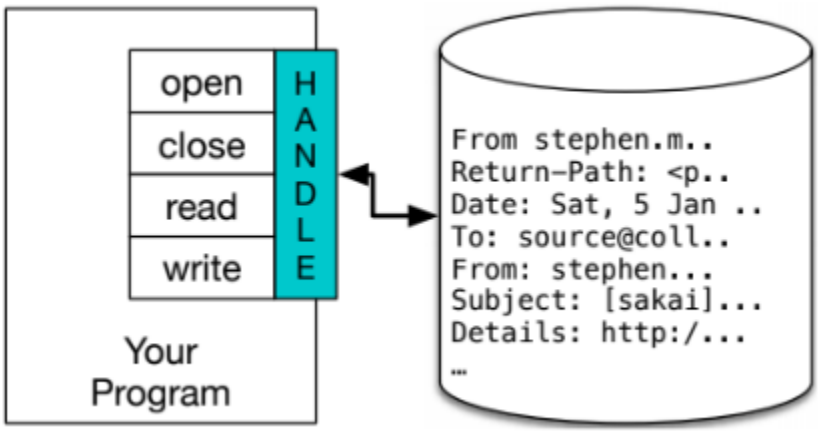

Figure 2: A File Handle

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'
```

4.3 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

The entire file of mail interactions is available from

www.py4e.com/code3/mbox.txt

and a shortened version of the file is available from

www.py4e.com/code3/mbox-short.txt

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. For more information about the mbox format, see https://en.wikipedia.org/wiki/Mbox.

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.

4.4 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)

# Code: http://www.py4e.com/code3/open.py
```

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. It is a good idea

to store the output of read as a variable because each call to read exhausts the

resource:

>>> fhand = open('mbox-short.txt')

>>> print(len(fhand.read()))

94626

>>> print(len(fhand.read()))

0

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.

4.5 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)

# Code: http://www.py4e.com/code3/search1.py
```

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)
# Code: http://www.py4e.com/code3/search2.py
```

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)

Code: http://www.py4e.com/code3/search3.py

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)

Code: http://www.py4e.com/code3/search4.py

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

...

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.

4.6 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 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)

# Code: http://www.py4e.com/code3/search6.py
```

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?”

4.7 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)

# Code: http://www.py4e.com/code3/search7.py
```

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:

python 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.

4.8 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 (andreading) 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.

4.9 Debugging

When you are reading and writing files, you might run into problems with whites- pace. These errors can be hard to debug because spaces, tabs, and newlines are

normally invisible:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2 3
4
```

The built-in function repr can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace

characters with backslash sequences:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

LISTS:

5.1 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]

['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] []

5.2 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-th 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 rela-tionship 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

5.3 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)

5.4. List Operations

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]]

5.4 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.

5.5 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 opera-

tions 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']
```

5.6 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 acciden- tally write t = t.sort(), you will be disappointed with the result.

5.7 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.

5.8 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)

# Code: http://www.py4e.com/code3/avenum.py
```

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)

# Code: http://www.py4e.com/code3/avelist.py
```

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.

5.9 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 “1”. 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.

5.10 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])

# Code: http://www.py4e.com/code3/search5.py
```

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.

5.11 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

b

‘banana’

‘banana’

a

b

‘banana’

Figure : Variables and Objects

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 oper-ator.

>>> 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.

5.12 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.

5.13 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']

5.14 Debugging

Careless use of lists (and other mutable objects) can lead to long hours of debugging.

Here are some common pitfalls and ways to avoid them:

1. Don’t forget that most list methods modify the argument and return None.

This is the opposite of the string methods, which return a new string andleave the original alone.If you are used to writing string code like this:

word = word.strip()

It is tempting to write list code like this:

t = t.sort() # WRONG!

Because sort returns None, the next operation you perform with t is likely to fail. Before using list methods and operators, you should read the documentation carefully and then test them in interactive mode. The methods and operators that lists share with other sequences (like strings) are documented at:

docs.python.org/library/stdtypes.html#common-sequence-operations

The methods and operators that only apply to mutable sequences are docu- mented at:

docs.python.org/library/stdtypes.html#mutable-sequence-types

2. Pick an idiom and stick with it.

Part of the problem with lists is that there are too many ways to do things.

For example, to remove an element from a list, you can use pop, remove, del,

or even a slice assignment.

To add an element, you can use the append method or the + operator. Butdon’t forget that these are right:

t.append(x)

t = t + [x]

And these are wrong:

t.append([x]) # WRONG!

t = t.append(x) # WRONG!

t + [x] # WRONG!

t = t + x # WRONG!

3. Make copies to avoid aliasing.

If you want to use a method like sort that modifies the argument, but you

need to keep the original list as well, you can make a copy.

5.14. DEBUGGING

orig = t[:]

t.sort()

In this example you could also use the built-in function sorted, which returns a new, sorted list and leaves the original alone. But in that case you should avoid using sorted as a variable name!

4. Lists, split, and files

When we read and parse files, there are many opportunities to encounterinput that can crash our program so it is a good idea to revisit the guardianpattern when it comes writing programs that read through a file and look for a "needle in the haystack". Let's revisit our program that is looking for the day of the week on the from lines of our file:

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

Since we are breaking this line into words, we could dispense with the use

of startswith and simply look at the first word of the line to determine if

we are interested in the line at all. We can use continue to skip lines that

don't have "From" as the first word as follows:

fhand = open('mbox-short.txt')

```
for line in fhand:

words = line.split()

if words[0] != 'From' : continue

print(words[2])
```

This looks much simpler and we don't even need to do the rstrip to remove the newline at the end of the file. But is it better? python search8.py

```
Sat

Traceback (most recent call last):

File "search8.py", line 5, in <module>

if words[0] != 'From' : continue

IndexError: list index out of range
```

It kind of works and we see the day from the first line (Sat), but then the program fails with a traceback error. What went wrong? What messed-up data caused our elegant, clever, and very Pythonic program to fail? You could stare at it for a long time and puzzle through it or ask someone for help, but the quicker and smarter approach is to add a print statement. The best place to add the print statement is right before the line where the program failed and print out the data that seems to be causing the failure.

Now this approach may generate a lot of lines of output, but at least you will immediately have some clue as to the problem at hand. So we add a print of the variable words right before line five. We even add a prefix "Debug:" to the line so we can keep our regular output separate from our debug output.

```
for line in fhand:

words = line.split()

print('Debug:', words)

if words[0] != 'From' : continue
```

print(words[2])

When we run the program, a lot of output scrolls off the screen but at the end, we see our debug output and the traceback so we know what happened just before the traceback.

Debug: ['X-DSPAM-Confidence:', '0.8475']

Debug: ['X-DSPAM-Probability:', '0.0000']

Debug: []

Traceback (most recent call last):

File "search9.py", line 6, in <module>

if words[0] != 'From' : continue

IndexError: list index out of range

Each debug line is printing the list of words which we get when we split the line into words. When the program fails, the list of words is empty [].

If we open the file in a text editor and look at the file, at that point it looks as follows:

X-DSPAM-Result: Innocent

X-DSPAM-Processed: Sat Jan 5 09:14:16 2008

X-DSPAM-Confidence: 0.8475

X-DSPAM-Probability: 0.0000

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772

The error occurs when our program encounters a blank line! Of course there are “zero words” on a blank line. Why didn’t we think of that when we were writing the code? When the code looks for the first word (word[0]) to checkto see if it matches “From”, we get an “index out of range” error.

This of course is the perfect place to add some guardian code to avoid check- ing the first word if the first word is not there. There are many ways to protect this code; we will choose to check the number of words we havebefore we look at the first word:

```
fhand = open('mbox-short.txt')

count = 0

for line in fhand:

words = line.split()

# print('Debug:', words)

if len(words) == 0 : continue

if words[0] != 'From' : continue

print(words[2])
```

First we commented out the debug print statement instead of removing it, in case our modification fails and we need to debug again. Then we added a guardian statement that checks to see if we have zero words, and if so, we use continue to skip to the next line in the file.