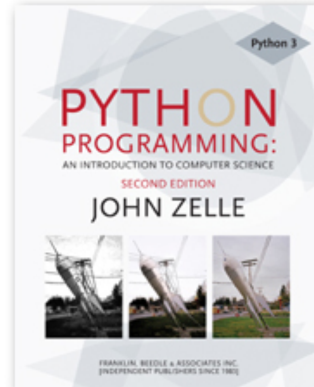
Python Programming: An Introduction to Computer Science



Chapter 5

Sequences: Strings, Lists, and Files

Objectives

- To understand the string data type and how strings are represented in the computer.
- To be familiar with various operations that can be performed on strings through built-in functions and the string library.

Objectives (cont.)

- To understand the basic idea of sequences and indexing as they apply to Python strings and lists.
- To be able to apply string formatting to produce attractive, informative program output.
- To understand basic file processing concepts and techniques for reading and writing text files in Python.

Objectives (cont.)

- To understand basic concepts of cryptography.
- To be able to understand and write programs that process textual information.

The String Data Type

- The most common use of personal computers is word processing.
- Text is represented in programs by the *string* data type.
- A string is a sequence of characters enclosed within quotation marks (") or apostrophes (').

The String Data Type

```
>>> str1="Hello"
>>> str2='spam'
>>> print(str1, str2)
Hello spam
>>> type(str1)
<class 'str'>
>>> type(str2)
<class 'str'>
```

The String Data Type

- Getting a string as input

```
>>> firstName = input("Please enter your name: ")
Please enter your name: John
>>> print("Hello", firstName)
Hello John
```

- Notice that the input is not `eval`uated. We want to store the typed characters, not to evaluate them as a Python expression.

The String Data Type

- We can access the individual characters in a string through *indexing*.
- The positions in a string are numbered from the left, starting with 0.
- The general form is <string>[<expr>], where the value of expr determines which character is selected from the string.

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet = "Hello Bob"
>>> greet[0]
'H'
>>> print(greet[0], greet[2], greet[4])
H l o
>>> x = 8
>>> print(greet[x - 2])
B
```

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

- In a string of *n* characters, the last character is at position *n-1* since we start counting with 0.
- We can index from the right side using negative indexes.

```
>>> greet[-1]
'b'
>>> greet[-3]
'B'
```

The String Data Type

- Indexing returns a string containing a single character from a larger string.
- We can also access a contiguous sequence of characters, called a *substring*, through a process called *slicing*.

The String Data Type

- Slicing:
  <string>[<start>:<end>]
- start and end should both be ints
- The slice contains the substring beginning at position start and runs up to **but doesn't include** the position end.

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet[0:3]
'Hel'
>>> greet[5:9]
' Bob'
>>> greet[:5]
'Hello'
>>> greet[5:]
' Bob'
>>> greet[:]
'Hello Bob'
```

The String Data Type

- If either expression is missing, then the start or the end of the string are used.
- Can we put two strings together into a longer string?
- *Concatenation* “glues” two strings together (+)
- *Repetition* builds up a string by multiple concatenations of a string with itself (*)

The String Data Type

- The function *len* will return the length of a string.

```
>>> "spam" + "eggs"
'spameggs'
>>> "Spam" + "And" + "Eggs"
'SpamAndEggs'
>>> 3 * "spam"
'spamspamspam'
>>> "spam" * 5
'spamspamspamspamspam'
>>> (3 * "spam") + ("eggs" * 5)
'spamspamspameggseggseggseggseggs'
```

The String Data Type

```
>>> len("spam")
4
>>> for ch in "Spam!":
        print (ch, end=" ")

S p a m !
```

The String Data Type

Operator	Meaning
+	Concatenation
*	Repetition
<string>[]	Indexing
<string>[:]	Slicing
len(<string>)	Length
for <var> in <string>	Iteration through characters

Simple String Processing

- Usernames on a computer system
  - First initial, first seven characters of last name

```
# get user's first and last names
first = input("Please enter your first name (all lowercase): ")
last = input("Please enter your last name (all lowercase): ")

# concatenate first initial with 7 chars of last name
uname = first[0] + last[:7]
```

Simple String Processing

```
>>>
Please enter your first name (all lowercase): john
Please enter your last name (all lowercase): doe
uname =  jdoe

>>>
Please enter your first name (all lowercase): donna
Please enter your last name (all lowercase): rostenkowski
uname =  drostenk
```

Simple String Processing

- Another use – converting an int that stands for the month into the three letter abbreviation for that month.
- Store all the names in one big string: “JanFebMarAprMayJunJulAugSepOctNovDec”
- Use the month number as an index for slicing this string: monthAbbrev = months[pos:pos+3]

Simple String Processing

Month	Number	Position
Jan	1	0
Feb	2	3
Mar	3	6
Apr	4	9

- To get the correct position, subtract one from the month number and multiply by three

Simple String Processing

```
# month.py
#  A program to print the abbreviation of a month, given its number


def main():

    # months is used as a lookup table
    months = "JanFebMarAprMayJunJulAugSepOctNovDec"

    n = eval(input("Enter a month number (1-12): "))

    # compute starting position of month n in months
    pos = (n-1) * 3

    # Grab the appropriate slice from months
    monthAbbrev = months[pos:pos+3]

    # print the result
    print ("The month abbreviation is", monthAbbrev + ".")

main()
```

Simple String Processing

```
>>> main()
Enter a month number (1-12): 1
The month abbreviation is Jan.
>>> main()
Enter a month number (1-12): 12
The month abbreviation is Dec.
```

- One weakness – this method only works where the potential outputs all have the same length.
- How could you handle spelling out the months?

Strings, Lists, and Sequences

- It turns out that strings are really a special kind of *sequence*, so these operations also apply to sequences!

```
>>> [1,2] + [3,4]
[1, 2, 3, 4]
>>> [1,2]*3
[1, 2, 1, 2, 1, 2]
>>> grades = ['A', 'B', 'C', 'D', 'F']
>>> grades[0]
'A'
>>> grades[2:4]
['C', 'D']
>>> len(grades)
5
```

Strings, Lists, and Sequences

- Strings are always sequences of characters, but *lists* can be sequences of arbitrary values.
- Lists can have numbers, strings, or both!

  myList = [1, "Spam ", 4, "U"]

Strings, Lists, and Sequences

- We can use the idea of a list to make our previous month program even simpler!
- We change the lookup table for months to a list:

```
months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
"Aug", "Sep", "Oct", "Nov", "Dec"]
```

Strings, Lists, and Sequences

- To get the months out of the sequence, do this:

  ```
  monthAbbrev = months[n-1]
  ```

  Rather than this:

  ```
  monthAbbrev = months[pos:pos+3]
  ```

Strings, Lists, and Sequences

```
# month2.py
#  A program to print the month name, given it's number.
#  This version uses a list as a lookup table.

def main():

    # months is a list used as a lookup table
    months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
              "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

    n = eval(input("Enter a month number (1-12): "))

    print ("The month abbreviation is", months[n-1] + ".")

main()
```

- Note that the months line overlaps a line. Python knows that the expression isn't complete until the closing] is encountered.

Strings, Lists, and Sequences

```
# month2.py
#  A program to print the month name, given it's number.
#  This version uses a list as a lookup table.

def main():

    # months is a list used as a lookup table
    months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
              "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

    n = eval(input("Enter a month number (1-12): "))

    print ("The month abbreviation is", months[n-1] + ".")

main()
```

- Since the list is indexed starting from 0, the *n-1* calculation is straight-forward enough to put in the print statement without needing a separate step.

Strings, Lists, and Sequences

- This version of the program is easy to extend to print out the whole month name rather than an abbreviation!

```
months = ["January", "February", "March", "April", "May", "June",
          "July", "August", "September", "October", "November", "December"]
```

Strings, Lists, and Sequences

- Lists are *mutable*, meaning they can be changed. Strings can **not** be changed.

```
>>> myList = [34, 26, 15, 10]
>>> myList[2]
15
>>> myList[2] = 0
>>> myList
[34, 26, 0, 10]
>>> myString = "Hello World"
>>> myString[2]
'l'
>>> myString[2] = "p"

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    myString[2] = "p"
TypeError: object doesn't support item assignment
```

Strings and Secret Codes

- Inside the computer, strings are represented as sequences of 1's and 0's, just like numbers.
- A string is stored as a sequence of binary numbers, one number per character.
- It doesn't matter what value is assigned as long as it's done consistently.

Strings and Secret Codes

- In the early days of computers, each manufacturer used their own encoding of numbers for characters.
- ASCII system (American Standard Code for Information Interchange) uses 127 bit codes
- Python supports Unicode (100,000+ characters)

Strings and Secret Codes

- The *ord* function returns the numeric (ordinal) code of a single character.
- The *chr* function converts a numeric code to the corresponding character.

```
>>> ord("A")
65
>>> ord("a")
97
>>> chr(97)
'a'
>>> chr(65)
'A'
```

Strings and Secret Codes

- Using ord and char we can convert a string into and out of numeric form.
- The encoding algorithm is simple:
  get the message to encode
  for each character in the message:
     print the letter number of the character
- A for loop iterates over a sequence of objects, so the for loop looks like:
  for ch in <string>

Strings and Secret Codes

```
# text2numbers.py
#     A program to convert a textual message into a sequence of
#         numbers, utlilizing the underlying Unicode encoding.

def main():
    print("This program converts a textual message into a sequence")
    print ("of numbers representing the Unicode encoding of the message.\n")

    # Get the message to encode
    message = input("Please enter the message to encode: ")

    print("\nHere are the Unicode codes:")

    # Loop through the message and print out the Unicode values
    for ch in message:
        print(ord(ch),  end=" ")

    print()

main()
```

Strings and Secret Codes

- We now have a program to convert messages into a type of "code", but it would be nice to have a program that could decode the message!
- The outline for a decoder:

```
get the sequence of numbers to decode
message = ""
for each number in the input:
   convert the number to the appropriate character
   add the character to the end of the message
print the message
```

Strings and Secret Codes

- The variable *message* is an accumulator variable, initially set to the *empty string*, the string with no characters (“”).
- Each time through the loop, a number from the input is converted to the appropriate character and appended to the end of the accumulator.

Strings and Secret Codes

- How do we get the sequence of numbers to decode?
- Read the input as a single string, then split it apart into substrings, each of which represents one number.

Strings and Secret Codes

- The new algorithm

```
get the sequence of numbers as a string, inString
message = “”
for each of the smaller strings:
   change the string of digits into the number it represents
   append the ASCII character for that number to message
print message
```

- Strings are objects and have useful methods associated with them

Strings and Secret Codes

- One of these methods is *split*. This will split a string into substrings based on spaces.

```
>>> "Hello string methods!".split()
['Hello', 'string', 'methods!']
```

Strings and Secret Codes

- Split can be used on characters other than space, by supplying the character as a parameter.

```
>>> "32,24,25,57".split(",")
['32', '24', '25', '57']
>>>
```

Strings and Secret Codes

- How can we convert a string containing digits into a number?
- Use our friend eval.

```
>>> numStr = "500"
>>> eval(numStr)
500
>>> x = eval(input("Enter a number "))
Enter a number 3.14
>>> print x
3.14
>>> type (x)
<type 'float'>
```

Strings and Secret Codes

```
# numbers2text.py
#     A program to convert a sequence of Unicode numbers into
#         a string of text.

def main():
    print ("This program converts a sequence of Unicode numbers into")
    print ("the string of text that it represents.\n")

    # Get the message to encode
    inString = input("Please enter the Unicode-encoded message: ")

    # Loop through each substring and build Unicde message
    message = ""
    for numStr in inString.split(i):
        # convert the (sub)string to a number
        codeNum = eval(numStr)
        # append character to message
        message = message + chr(codeNum)

    print("\nThe decoded message is:", message)

main()
```

Strings and Secret Codes

- The split function produces a sequence of strings. numString gets each successive substring.
- Each time through the loop, the next substring is converted to the appropriate Unicode character and appended to the end of message.

Strings and Secret Codes

```
--------------------------------------------------------------------------
This program converts a textual message into a sequence
of numbers representing the Unicode encoding of the message.

Please enter the message to encode: CS120 is fun!

Here are the Unicode codes:
67 83 49 50 48 32 105 115 32 102 117 110 33


--------------------------------------------------------------------------
This program converts a sequence of Unicode numbers into
the string of text that it represents.

Please enter the ASCII-encoded message: 67 83 49 50 48 32 105 115 32 102 117 110 33
The decoded message is: CS120 is fun!
```

Other String Methods

- There are a number of other string methods. Try them all!
  - s.capitalize() – Copy of s with only the first character capitalized
  - s.title() – Copy of s; first character of each word capitalized
  - s.center(width) – Center s in a field of given width

Other String Operations

- s.count(sub) – Count the number of occurrences of sub in s
- s.find(sub) – Find the first position where sub occurs in s
- s.join(list) – Concatenate list of strings into one large string using s as separator.
- s.ljust(width) – Like center, but s is left-justified

Other String Operations

- s.lower() – Copy of s in all lowercase letters
- s.lstrip() – Copy of s with leading whitespace removed
- s.replace(oldsub, newsub) – Replace occurrences of oldsub in s with newsub
- s.rfind(sub) – Like find, but returns the right-most position
- s.rjust(width) – Like center, but s is right-justified

Other String Operations

- s.rstrip() – Copy of s with trailing whitespace removed
- s.split() – Split s into a list of substrings
- s.upper() – Copy of s; all characters converted to uppercase

From Encoding to Encryption

- The process of encoding information for the purpose of keeping it secret or transmitting it privately is called *encryption*.
- *Cryptography* is the study of encryption methods.
- Encryption is used when transmitting credit card and other personal information to a web site.

From Encoding to Encryption

- Strings are represented as a sort of encoding problem, where each character in the string is represented as a number that's stored in the computer.
- The code that is the mapping between character and number is an industry standard, so it's not "secret".

From Encoding to Encryption

- The encoding/decoding programs we wrote use a *substitution cipher*, where each character of the original message, known as the *plaintext*, is replaced by a corresponding symbol in the *cipher alphabet*.
- The resulting code is known as the *ciphertext*.

From Encoding to Encryption

- This type of code is relatively easy to break.
- Each letter is always encoded with the same symbol, so using statistical analysis on the frequency of the letters and trial and error, the original message can be determined.

From Encoding to Encryption

- Modern encryption converts messages into numbers.
- Sophisticated mathematical formulas convert these numbers into new numbers – usually this transformation consists of combining the message with another value called the "*key*"

From Encoding to Encryption

- To decrypt the message, the receiving end needs an appropriate key so the encoding can be reversed.
- In a *private key* system the same key is used for encrypting and decrypting messages. Everyone you know would need a copy of this key to communicate with you, but it needs to be kept a secret.

From Encoding to Encryption

- In *public key* encryption, there are separate keys for encrypting and decrypting the message.
- In public key systems, the encryption key is made publicly available, while the decryption key is kept private.
- Anyone with the public key can send a message, but only the person who holds the private key (decryption key) can decrypt it.

Input/Output as String Manipulation

- Often we will need to do some string operations to prepare our string data for output ("pretty it up")
- Let's say we want to enter a date in the format "05/24/2003" and output "May 24, 2003." How could we do that?

Input/Output as String Manipulation

- Input the date in mm/dd/yyyy format (dateStr)
- Split dateStr into month, day, and year strings
- Convert the month string into a month number
- Use the month number to lookup the month name
- Create a new date string in the form “Month Day, Year”
- Output the new date string

Input/Output as String Manipulation

- The first two lines are easily implemented!

```
dateStr = input("Enter a date (mm/dd/yyyy): ")
monthStr, dayStr, yearStr = dateStr.split("/")
```

- The date is input as a string, and then "unpacked" into the three variables by splitting it at the slashes and using simultaneous assignment.

Input/Output as String Manipulation

- Next step: Convert monthStr into a number
- We can use the *int* function on monthStr to convert "05", for example, into the integer 5. (int("05") = 5)

Input/Output as String Manipulation

- Note: eval would work, but for the leading 0

```
>>> int("05")
5
>>> eval("05")
 Traceback (most recent call last):
File "<pyshell#9>", line 1, in <module>
eval("05")
File "<string>", line 1
05
^
SyntaxError: invalid token
```

- This is historical baggage. A leading 0 used to be used for base 8 (octal) literals in Python.

Input/Output as String Manipulation

```
months = ["January", "February", …, "December"]
monthStr = months[int(monthStr) – 1]
```

- Remember that since we start counting at 0, we need to subtract one from the month.
- Now let's concatenate the output string together!

Input/Output as String Manipulation

```
print ("The converted date is:", monthStr, dayStr+",", yearStr)
```

- Notice how the comma is appended to dayStr with concatenation!
- >>> main()
  Enter a date (mm/dd/yyyy): 01/23/2010
  The converted date is: January 23, 2010

Input/Output as String Manipulation

- Sometimes we want to convert a number into a string.
- We can use the *str* function.

```
>>> str(500)
'500'
>>> value = 3.14
>>> str(value)
'3.14'
>>> print("The value is", str(value) + ".")
The value is 3.14.
```

Input/Output as String Manipulation

- If value is a string, we can concatenate a period onto the end of it.
- If value is an int, what happens?

```
>>> value = 3.14
>>> print("The value is", value + ".")
The value is

Traceback (most recent call last):
  File "<pyshell#10>", line 1, in -toplevel-
    print "The value is", value + "."
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

Input/Output as String Manipulation

- We now have a complete set of type conversion operations:

Function	Meaning
float(<expr>)	Convert expr to a floating point value
int(<expr>)	Convert expr to an integer value
str(<expr>)	Return a string representation of expr
eval(<string>)	Evaluate string as an expression

String Formatting

- String formatting is an easy way to get beautiful output!

```
Change Counter

Please enter the count of each coin type.
Quarters: 6
Dimes: 0
Nickels: 0
Pennies: 0

The total value of your change is 1.5
```

- Shouldn’t that be more like $1.50??

String Formatting

- We can format our output by modifying the print statement as follows:

```
print("The total value of your change is ${0:0.2f}".format(total))
```

- Now we get something like:

```
The total value of your change is $1.50
```

- Key is the string format method.

String Formatting

- <template-string>.format(<values>)
- {} within the template-string mark "slots" into which the values are inserted.
- Each slot has description that includes *format specifier* telling Python how the value for the slot should appear.

String Formatting

```
print("The total value of your change is ${0:0.2f}".format(total)
```

- The template contains a single slot with the description: 0:0.2f
- Form of description: <index>:<format-specifier>
- Index tells which parameter to insert into the slot. In this case, total.

String Formatting

- The formatting specifier has the form: <width>.<precision><type>
- f means "fixed point" number
- <width> tells us how many spaces to use to display the value. 0 means to use as much space as necessary.
- <precision> is the number of decimal places.

String Formatting

```
>>> "Hello {0} {1}, you may have won ${2}" .format("Mr.", "Smith", 10000)
'Hello Mr. Smith, you may have won $10000'

>>> 'This int, {0:5}, was placed in a field of width 5'.format(7)
'This int,     7, was placed in a field of width 5'

>>> 'This int, {0:10}, was placed in a field of witdh 10'.format(10)
'This int,         10, was placed in a field of witdh 10'

>>> 'This float, {0:10.5}, has width 10 and precision 5.'.format(3.1415926)
'This float,     3.1416, has width 10 and precision 5.'

>>> 'This float, {0:10.5f},  is fixed at 5 decimal places.'.format(3.1415926)
'This float,   3.14159, has width 0 and precision 5.'
```

String Formatting

- If the width is wider than needed, numeric values are right-justified and strings are left-justified, by default.
- You can also specify a justification before the width.

```
>>> "left justification: {0:<5}.format("Hi!")
'left justification: Hi!  '
>>> "right justification: {0:>5}.format("Hi!")
'right justification:   Hi!'
>>> "centered: {0:^5}".format("Hi!")
'centered:  Hi! '
```

Better Change Counter

- With what we know now about floating point numbers, we might be uneasy about using them in a money situation.
- One way around this problem is to keep trace of money in cents using an int or long int, and convert it into dollars and cents when output.

Better Change Counter

- If total is a value in cents (an int),
  dollars = total//100
  cents = total%100
- Cents is printed using width 0>2 to right-justify it with leading 0s (if necessary) into a field of width 2.
- Thus 5 cents becomes '05'

Better Change Counter

```
# change2.py
#   A program to calculate the value of some change in dollars.
#   This version represents the total cash in cents.

def main():
    print ("Change Counter\n")

    print ("Please enter the count of each coin type.")
    quarters = eval(input("Quarters: "))
    dimes = eval(input("Dimes: "))
    nickels = eval(input("Nickels: "))
    pennies = eval(input("Pennies: "))
    total = quarters * 25 + dimes * 10 + nickels * 5 + pennies

    print ("The total value of your change is ${0}.{1:0>2}"
                .format(total//100, total%100))
```

Better Change Counter

```
>>> main()
Change Counter

Please enter the count of each coin type.
Quarters: 0
Dimes: 0
Nickels: 0
Pennies: 1

The total value of your change is $0.01
```

```
>>> main()
Change Counter

Please enter the count of each coin type.
Quarters: 12
Dimes: 1
Nickels: 0
Pennies: 4

The total value of your change is $3.14
```

Files: Multi-line Strings

- A *file* is a sequence of data that is stored in secondary memory (disk drive).
- Files can contain any data type, but the easiest to work with are text.
- A file usually contains more than one line of text.
- Python uses the standard newline character (\n) to mark line breaks.

Multi-Line Strings

- Hello
  World

  Goodbye 32
- When stored in a file:
  Hello\nWorld\n\nGoodbye 32\n

Multi-Line Strings

- This is exactly the same thing as embedding \n in print statements.
- Remember, these special characters only affect things when printed. They don't do anything during evaluation.

File Processing

- The process of *opening* a file involves associating a file on disk with an object in memory.
- We can manipulate the file by manipulating this object.
  - Read from the file
  - Write to the file

File Processing

- When done with the file, it needs to be *closed*. Closing the file causes any outstanding operations and other bookkeeping for the file to be completed.
- In some cases, not properly closing a file could result in data loss.

File Processing

- Reading a file into a word processor
  - File opened
  - Contents read into RAM
  - File closed
  - Changes to the file are made to the copy stored in memory, not on the disk.

File Processing

- Saving a word processing file
  - The original file on the disk is reopened in a mode that will allow writing (this actually erases the old contents)
  - File writing operations copy the version of the document in memory to the disk
  - The file is closed

File Processing

- Working with text files in Python
  - Associate a disk file with a file object using the open function
    <filevar> = open(<name>, <mode>)
  - Name is a string with the actual file name on the disk. The mode is either ‘r’ or ‘w’ depending on whether we are reading or writing the file.
  - Infile = open("numbers.dat", "r")

File Methods

- <file>.read() – returns the entire remaining contents of the file as a single (possibly large, multi-line) string
- <file>.readline() – returns the next line of the file. This is all text up to *and including* the next newline character
- <file>.readlines() – returns a list of the remaining lines in the file. Each list item is a single line including the newline characters.

File Processing

```
# printfile.py
#     Prints a file to the screen.

def main():
    fname = input("Enter filename: ")
    infile = open(fname,'r')
    data = infile.read()
    print(data)

main()
```

- First, prompt the user for a file name
- Open the file for reading
- The file is read as one string and stored in the variable data

File Processing

- readline can be used to read the next line from a file, including the trailing newline character
- ```
  infile = open(someFile, "r")
  for i in range(5):
      line = infile.readline()
      print line[:-1]
  ```
- This reads the first 5 lines of a file
- Slicing is used to strip out the newline characters at the ends of the lines
  ```

File Processing

- Another way to loop through the contents of a file is to read it in with readlines and then loop through the resulting list.
- ```
  infile = open(someFile, "r")
  for line in infile.readlines():
      # Line processing here
  infile.close()
  ```
  ```

File Processing

- Python treats the file itself as a sequence of lines!
- Infile = open(someFile, "r")
  for line in infile:
      # process the line here
  infile.close()

```
Infile = open(someFile, "r")
for line in infile:
    # process the line here
infile.close()
```

File Processing

- Opening a file for writing prepares the file to receive data
- If you open an existing file for writing, you wipe out the file's contents. If the named file does not exist, a new one is created.
- Outfile = open("mydata.out", "w")
- print(<expressions>, file=Outfile)

Example Program: Batch Usernames

- *Batch* mode processing is where program input and output are done through files (the program is not designed to be interactive)
- Let's create usernames for a computer system where the first and last names come from an input file.

Example Program: Batch Usernames

```
# userfile.py
#    Program to create a file of usernames in batch mode.

def main():
    print ("This program creates a file of usernames from a")
    print ("file of names.")

    # get the file names
    infileName = input("What file are the names in? ")
    outfileName = input("What file should the usernames go in? ")

    # open the files
    infile = open(infileName, 'r')
    outfile = open(outfileName, 'w')
```

Example Program: Batch Usernames

```
# process each line of the input file
for line in infile:
    # get the first and last names from line
    first, last = line.split()
    # create a username
    uname = (first[0]+last[:7]).lower()
    # write it to the output file
    print(uname, file=outfile)

# close both files
infile.close()
outfile.close()

print("Usernames have been written to", outfileName)
```

Example Program: Batch Usernames

- Things to note:
  - It's not unusual for programs to have multiple files open for reading and writing at the same time.
  - The lower method is used to convert the names into all lower case, in the event the names are mixed upper and lower case.