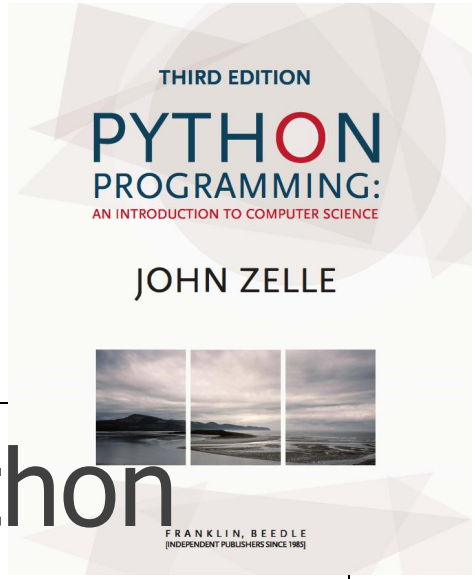


An Introduction to Python Programming

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 on strings through built-in functions and the string library.
- To understand the basic idea of **sequences** and indexing in strings and lists.
- 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.

Sequences:Strings,Lists,and Files

1. The String Data Type

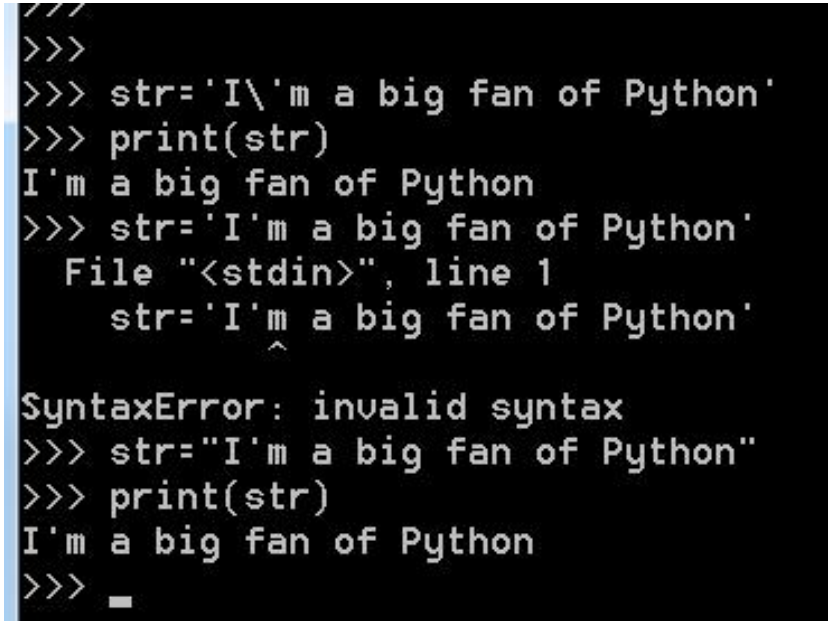
- The most common use of personal computers is word processing.
- Text is represented in programs by the ***string*** **data type**.
- A string is a sequence of characters enclosed within quotation marks (") or apostrophes (').

Sequences:Strings,Lists,and Files

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

In this case，There are no difference

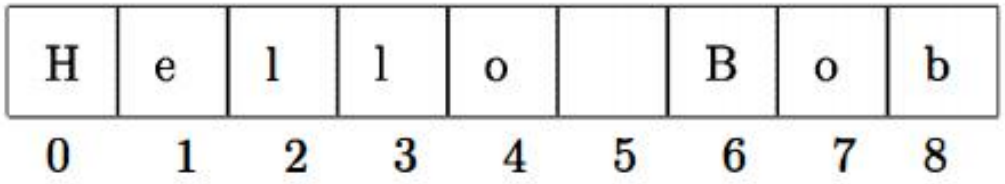
Sequences:Strings,Lists,and Files



more complicated

Sequences:Strings,Lists,and Files

- Besides input(),print() functions, 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.



Sequences:Strings,Lists,and Files

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

That's the difference between sting objects and the actual printed output.

Sequences:Strings,Lists,and Files

- By the way:

```
>>> greet[-1]
'b'
>>> greet[0:3]
'Hel'
>>> greet[5:9]
' Bob'
>>> greet[:5]
'Hello'
>>> greet[5:]
' Bob'
>>> greet[:]
'Hello Bob'
```

Sequences:Strings,Lists,and Files

- We can also access a contiguous sequence of characters, called a *substring*, through a process called ***slicing***.
- Indexing and slicing are useful operations for chopping strings **into smaller pieces**.
- Also,two handy operators are **concatenation (+) and repetition (*)** .

Sequences:Strings,Lists,and Files

```
>>> "spam" + "eggs"
'spameggs'
>>> "Spam" + "And" + "Eggs"
'SpamAndEggs'
>>> 3 * "spam"
'spamspamspam'
>>> "spam" * 5
'spamspamspamspamspam'
>>> (3 * "spam") + ("eggs" * 5)
'spamspamspameggseggseggseggseggs'
>>> len("spam")
4
>>> len("SpamAndEggs")
11
>>> for ch in "Spam!":
        print(ch, end=" ")
S p a m !
```

Sequences:Strings,Lists,and Files

- These basic string operations are summarized.

operator	meaning
+	concatenation
*	repetition
<string>[ ]	indexing
<string>[:]	slicing
len(<string>)	length
for <var> in <string>	iteration through characters

Sequences:Strings,Lists,and Files

2. Simple String Processing

- For example,usernames on a computer system
  - Putting the newline **character (\n)** at the end of the string in the first print statement caused the output to skip down an extra line.

```
def main():
    print("This program generates computer usernames.\n")

    # 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): ")
```

Sequences:Strings,Lists,and Files

```
This program generates computer usernames.

Please enter your first name (all lowercase): zaphod
Please enter your last name (all lowercase): beeblebrox
Your username is: zbeebleb
```

This will look better

Sequences:Strings,Lists,and Files

- 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]**

Sequences:Strings,Lists,and Files

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

Sequences:Strings,Lists,and Files

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

Sequences:Strings,Lists,and Files

3. Lists as Sequences

- The operations in upper tabe are not really just string operations.They are operations that apply to **sequences**.
- Python **lists** are also a kind of sequence.We can also

index, slice, and concatenate lists.

Sequences:Strings,Lists,and Files

```
>>> [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
```

Sequences:Strings,Lists,and Files

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

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

  - Using a list of strings, we can rewrite our month program and make it even simpler

Sequences:Strings,Lists,and Files

```
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 = int(input("Enter a month number (1-12): "))

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

Sequences:Strings,Lists,and Files

- The code that creates the list is **split over two lines for more readable**. Normally a Python statement is written on a single line,but in this case Python knows the list isn't finished until the closing bracket **"] "** is encountered.
- Lists, are indexed **starting with 0**, so in this list the value months [0] is the string "Jan" .
- It is also more **flexible**.To change the program so that it prints out the **entire name** of the month. All we need is **a new defnition of the lookup list.**

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

Sequences:Strings,Lists,and Files

- Lists are **mutable**, since Strings cannot be changed "in place."

```
>>> myList = [34, 26, 15, 10]
>>> myList[2]
15
>>> myList[2] = 0
>>> myList
[34, 26, 0, 10]
>>> myString = "Hello World"
>>> myString[2]
'l'
>>> myString[2] = 'z'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Sequences:Strings,Lists,and Files

4. String Representation and Message Encoding

- String Representation
  - 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,just like encoding.
  - In the early days of computers, each manufacturer **used their own encoding of numbers for characters**.
  - Today, Computers use the **ASCII system** (American Standard Code for Information Interchange).

Sequences:Strings,Lists,and Files

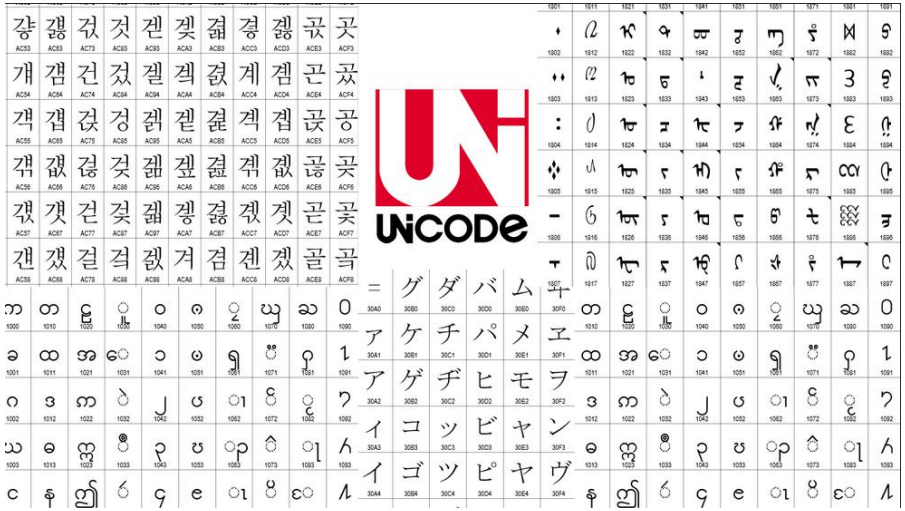
- **0 – 127** are used to represent the characters typically found on American keyboards.
  - 65 – 90 are “A” – “Z”
  - 97 – 122 are “a” – “z”
  - 48 – 57 are “0” – “9”
- The others are punctuation and *control codes* used to coordinate the sending and receiving of information.

Sequences:Strings,Lists,and Files

- One **major problem** with ASCII is that it's American-centric, it doesn't have many of the symbols necessary for other languages.
- Most modern systems are moving to **Unicode**, a much larger standard that aims to include the characters of nearly all written languages.
- Python strings support the Unicode Standard.
- Python provides **a couple of built-in functions** that allow us to switch back and forth,for example **ord & char**

Sequences:Strings,Lists,and Files

```
>>> ord("a")
97
>>> ord("A")
65
>>> chr(97)
'a'
>>> chr(90)
'Z'
```



- By design, Unicode uses the same codes as ASCII for the 127 characters . But Unicode includes many more exotic characters. For example,the Greek letter pi is character 960, and the symbol for the Euro is character 8364.

Sequences:Strings,Lists,and Files

- Another puzzle:How to **store characters** in computer memory?
- The number of possible Unicode characters is **100,000+ ,Much larger than 2^8 = 256.**
- To get around this problem, the Unicode Standard defines the most common encoding **UTF-8.**
- UTF-8 is a **variable-length encoding** scheme that uses a single byte to store characters in the ASCII subset, but may need up to four bytes to represent some of the more esoteric characters.

A string of length 10 characters will get a sequence of between 10 and 40 bytes.

A a rule of thumb for Latin alphabets , A character requires about one byte of storage on average.

Sequences:Strings,Lists,and Files

- Programming an Encoder
  - Using the Python **ord and chr functions**, we can write simple programs turning messages into sequences of numbers and back again----**encoding**.
  - We need to do something for **each character of the message.** A for loop will make.
  - To convert each character to a number, The simplest approach is to use **the Unicode number**.

Sequences:Strings,Lists,and Files

```
# text2numbers.py
#     A program to convert a textual message into a sequence of
#         numbers, utilizing 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() # blank line before prompt
```

Sequences:Strings,Lists,and Files

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

Please enter the message to encode: What a Sourpuss!

Here are the Unicode codes:
87 104 97 116 32 97 32 83 111 117 114 112 117 115 115 33
```

Sequences:Strings,Lists,and Files

5. String Methods

- Programming a Decoder

Let's make a similar program to turn the numbers back into a readable message.

- The decoding version will **collect the characters of the message** in a string and print out the entire message.
- We need to use **an accumulator variable**, which should be **initialized to be an empty string.**
- Each time of the loop, the input number is converted into an character and **append to the end of the message**.

Sequences:Strings,Lists,and Files

- But, **How exactly do we get the sequence of numbers to decode?** We don't even know how many numbers there will be.
- First, we read the entire sequence of numbers as a single string using **input**.
- Then we **split** the big string into a sequence of smaller strings.
- Finally,we **convert** each into a number.

Sequences:Strings,Lists,and Files

- Python **provides some functions** that do just what we need.
- By **virtue of being objects**, strings have some **built-in methods** .
- **Split method**----splits a string into a list of substings.

```
>>> myString = "Hello, string methods!"
>>> myString.split()
['Hello,', 'string', 'methods!']
>>> "32,24,25,57".split(",")
['32', '24', '25', '57']
```

This is very useful!

Sequences:Strings,Lists,and Files

- We could get the x and y values of a point by input, **turn it into a list using the split method**, and then index it.

```
>>> coords = input("Enter the point coordinates (x,y): ").split(",")
Enter the point coordinates (x,y): 3.4, 6.25
>>> coords
['3.4', '6.25']
>>> coords[0]
'3.4'
>>coords[1]
'6.25'
```

Sequences:Strings,Lists,and Files

- More String Methods

function	meaning
`s.capitalize()`	Copy of `s` with only the first character capitalized.
`s.center(width)`	Copy of `s` centered in a field of given `width`.
`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` into a string, using `s` as separator.
`s.ljust(width)`	Like `center`, but `s` is left-justified.
`s.lower()`	Copy of `s` in all lowercase characters.
`s.lstrip()`	Copy of `s` with leading white space removed.
`s.replace(oldsub,newsub)`	Replace all occurrences of `oldsub` in `s` with `newsub`.
`s.rfind(sub)`	Like `find`, but returns the rightmost position.
`s.rjust(width)`	Like `center`, but `s` is right-justified.
`s.rstrip()`	Copy of `s` with trailing white space removed.
`s.split()`	Split `s` into a list of substrings (see text).
`s.title()`	Copy of `s` with first character of each word capitalized.
`s.upper()`	Copy of `s` with all characters converted to upper case.

Sequences:Strings,Lists,and Files

```
>>> s="hello,I came here for an argument"
>>> s.capitalize()
'Hello,i came here for an argument'
>>> s.title()
'Hello,I Came Here For An Argument'
>>> s.replace("I","you")
'hello,you came here for an argument'
>>> s.center(30)
'hello,I came here for an argument'
>>> s.center(10)
'hello,I came here for an argument'
>>> s.center(50)
'        hello,I came here for an argument         '
>>> s.center(100)
'                                hello,I came here for an argument
                    .

>>> s.count('r')
3
>>> s.find('r')
15
>>> " ".join(["h","a","v","e","!"])
'h a v e !'
>>> " A".join(["h","a","v","e","!"])
'h Aa Av Ae A!'
```

Sequences:Strings,Lists,and Files

6. Lists Have Methods, Too

- The **append** method can be used to add an item at the end of a list.

```
squares = []
for x in range(1,101):
    squares.append(x*x)
```

- When the loop is done, squares will be the list: **[1 , 4, 9, . . . , 10000] .**

Sequences:Strings,Lists,and Files

- As the decoding program,in older version, **string concatenation** could be a slow operation.
- Programmers often used other techniques to **accumulate a long string.**

```
message = message + chr(codeNum)
```

- One way is to **use a list.**
- Lists are **mutable**, so changing the list "in place," **without having to copy the existing contents over to a new object.**
- We can use the **join operation** to concatenate the characters into a string.

Sequences:Strings,Lists,and Files

```
# Loop through each substring and build Unicode message
chars = []
for numStr in inString.split():
    codeNum = int(numStr)              # convert digits to a number
    chars.append(chr(codeNum))          # accumulate new character

message = "".join(chars)
print("\nThe decoded message is:", message)
```

Quite effcient!

Sequences:Strings,Lists,and Files

7. From Encoding to Encryption

- In unicode coding,there is **nothing really secret** about this code at all.
- Since each letter is always **encoded by the same symbol**.
- The process of encoding information for the purpose of keeping it secret or transmitting it privately is called **encryption.**

Sequences:Strings,Lists,and Files

Our simple encoding/decoding programs use a very weak form of encryption known as a **substitution cipher**.

Each character of the original message, called the **plaintext**, is replaced by a corresponding symbol (in our case a number) from a **cipher alphabet**. The resulting code is called the **ciphertet**.

Sequences:Strings,Lists,and Files

- Modem approaches to **encryption start** by translating a message into numbers, much like our encoding program.
- Then **sophisticated mathematical algorithms** are employed to transform these numbers into other numbers.
  - Usually, the transformation is based on combining the message with some other special value called **the key**.
  - To decrypt the message, the party on the receiving end needs to have an **appropriate key.**

Sequences:Strings,Lists,and Files

- Encryption approaches come in two flavors: **private key**(the same key is used for encryting and decrypting messages) and **public key**(the encrytion is publicly available, while the decryption key is kept private).

Sequences:Strings,Lists,and Files

8. Input/Output as String Manipulation

- Example Application: Date Conversion

The user will input a date such as "05/24/2020," and the program will display the date as "May 24, 2020."

```
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 look up the month name
Create a new date string in form Month Day, Year
Output the new date string
```

Sequences:Strings,Lists,and Files

- We can **implement the first two lines** of our algorithm directly in code using string operations.

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

- Then "unpacked" the list of three strings into the variables **monthStr, dayStr, and yearStr** using simultaneous assignment.
- The next, **convert** monthStr into an appropriate number and then use this value to **look up** the correct month name.

```
months = ["January", "February", "March", "April",
          "May", "June", "July", "August",
          "September", "October", "November", "December"]
monthStr = months[int(monthStr)-1]
```

Sequences:Strings,Lists,and Files

- The last step is to **piece together** the date in the new format:

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

- The complete program is on the P152.
- When run, the output looks like this:

```
Enter a date (mm/dd/yyyy): 05/24/2020
The converted date is: May 24, 2020
```

Sequences:Strings,Lists,and Files

- In Python, most data types can be **converted into strings** using the str function.

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

Sequences:Strings,Lists,and Files

- The complete set of **operations** for **converting values among various Python data types**.

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.

Sequences:Strings,Lists,and Files

- String Formatting
  - Here is the **change-counting** program

```
Change Counter

Please enter the count of each coin type.
How many quarters do you have? 6
How many dimes do you have? 0
How many nickels do you have? 0
How many pennies do you have? 0
The total value of your change is 1.5
```

Sequences:Strings,Lists,and Files

- **Change** the **very last line** of the program:

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

- The program **prints this message**:

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

- The **format method** is a built-in for Python strings.

```
<template-string>.format(<values>)
```

- The information inside the curly braces tells **which value** goes in the slot and **how** it should be **formatted**.

Sequences:Strings,Lists,and Files

- **The slot descriptions** will always have the form:

  {<index>: <format-specifier>}

- **The index** tells which of the parameters is inserted into the slot.
- Index **0** is used to say **the first parameter** .
- The **format** of this **specifer** is **<width>.<precision><type >**.

  The **width** specifes **how many "spaces"** the value should take up.Putting a **0** here essentially says "**use as much space as you need**."

  The precision is 2, which tells Python to **round the value to two decimal places**.

The type character **f** says the value should be displayed as a **fixed-point** number.

Sequences:Strings,Lists,and Files

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

- Control the **width and/or precision** of a numeric value.

```
>>> "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 width 10".format(7)
'This int,          7, was placed in a field of width 10'
```

Sequences:Strings,Lists,and Files

```
>>> "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, is fixed at 5 decimal places'

>>> "This float, {0:0.5}, has width 0 and precision 5".format(3.1415926)
'This float, 3.1416, has width 0 and precision 5'

>>> "Compare {0} and {0:0.20}".format(3.14)
'Compare 3.14 and 3.14000000000000001243'
```

Sequences:Strings,Lists,and Files

- For **normal (not fxed-point) floating-point** numbers, the precision specifes the number of **signifcant digits** to print.
- For **fixed-point** (indicated by the f at the end of the specifer) the precision gives the number of **decimal places**.
- Print **enough digits of a floating-point number**, you will almost always fnd a "**surprise**."
- If **not given an explicit precision**, Python will print the number out to a few decimal places.

Sequences:Strings,Lists,and Files

- We can **justifcation character** at the beginning of the format specifer.
- The necessary characters are **<, >, and ^** for **left, right, and center** justifcation.

```
>>> "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! '
```

Sequences:Strings,Lists,and Files

- Better Change Counter
  - It's uneasy about using float numbers to **represent money precisely**.
  - We can do that by keeping track of the money **in cents** and using an **int** to store it.
  - If **total** represents the value in cents, then we can get the number of **dollars** by integer division **total / / 100** and the cents from **total % 100**.

```
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))
```

Pad the feld
with zeroes instead of spaces.

Sequences:Strings,Lists,and Files

9. File Processing

- Multi-line Strings
  - In Python, text files can be very **flexible**, since it is easy to **convert back and forth** beteen strings and other types.
  - Typical files contains more than a single line of **text**. A **special character** or sequence of characters is used to mark the **end** of each line.
  - Python just uses the regular **newline character (\n)** to indicate line breaks.

Sequences:Strings,Lists,and Files

```
Hello
World

Goodbye 32
```

- When stored to a file, you get this **sequence of characters**:

```
Hello\nWorld\n\nGoodbye 32\n
```

- Also:

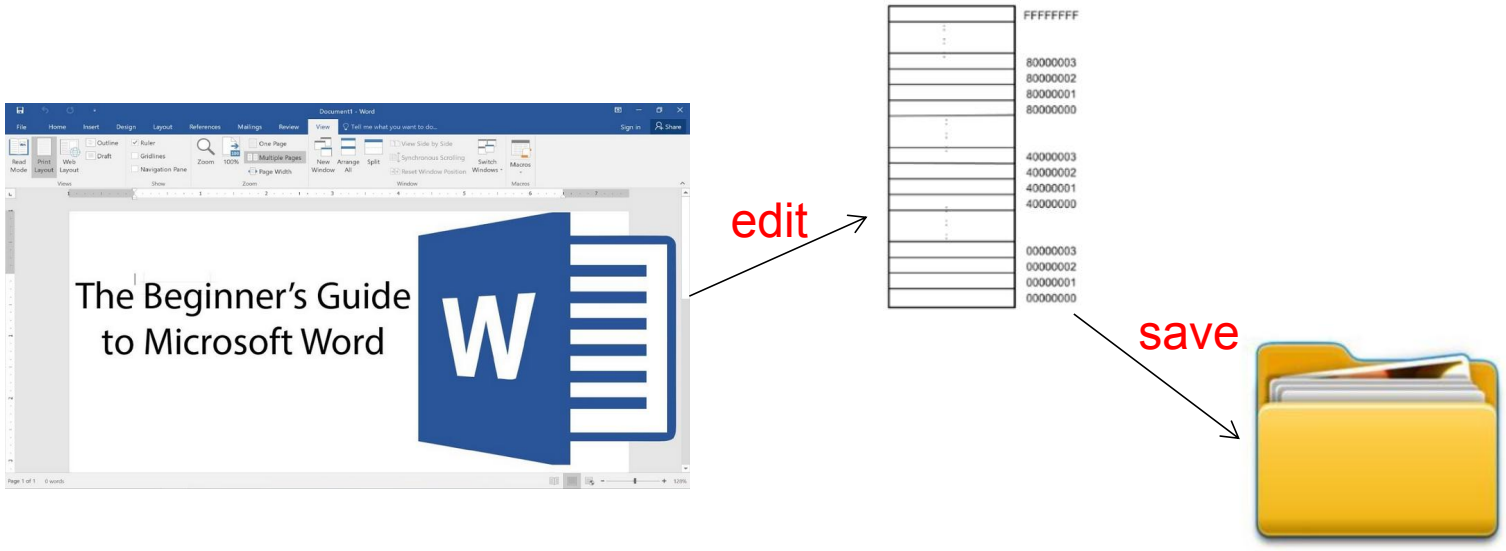
```
>>> print("Hello\nWorld\n\nGoodbye 32\n")
Hello
World

Goodbye 32
```

Sequences:Strings,Lists,and Files

- File Processing
  - 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. Lines of text are separated with a special character, **the *newline* character**.
  - You can think of *newline* as the character produced when you press the **<Enter> key**.
  - In Python, this character is represented as **'\n'**, just as tab is represented as **'\t'**.

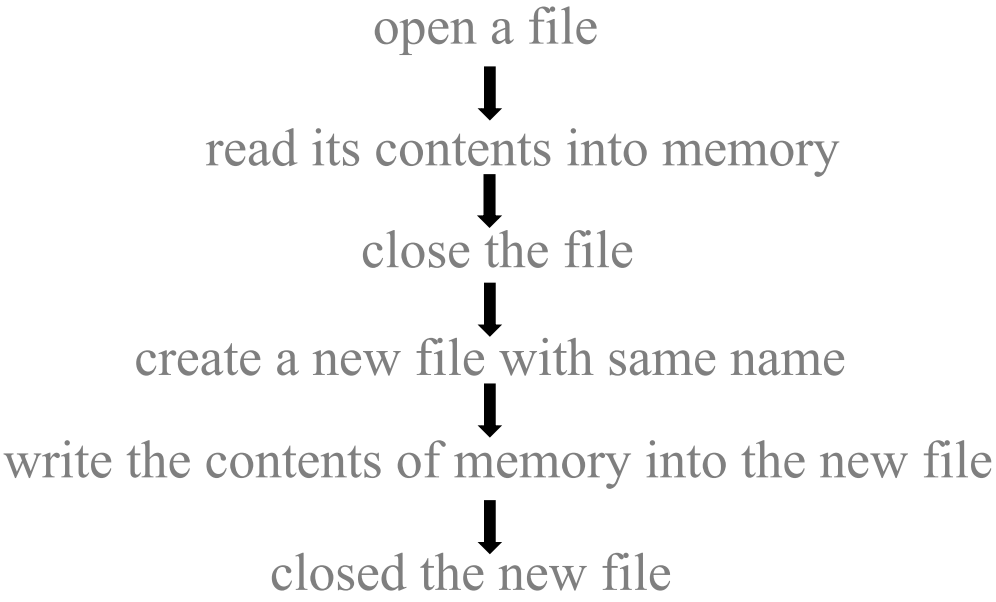
Sequences:Strings,Lists,and Files

- First, we need some way to **associate a file** on disk with an **object** in a program. (**opening a file**)
- Second, we need **a set of operations**,like read and write,etc.
- Finally when we are fnished with a file, it is **closed,**and makes sure any **bookkeeping**.



Sequences:Strings,Lists,and Files

- From the **program's perspective**:



Sequences:Strings,Lists,and Files

- Working with text files is easy in Python.

Creating a file object corresponding to a file on disk is done using the **open** function.

```
<variable> = open(<name>, <mode>)
infile = open("numbers.dat", "r")
```

Now we can use the **file object infile** to read the contents of numbers . dat from the disk , **on P161.**

```
<file>.read()
<file>.readline()
<file>.readlines()
```

Sequences:Strings,Lists,and Files

- Here's an example program.

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

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

main()
```

Sequences:Strings,Lists,and Files

- Attention: the string returned by **readline** will always **end with a newline character**, whereas input **discards** the newline character.

```
infile = open(someFile, "r")
for i in range(5):
line = infile.readline()
print(line[:-1])
```

- Aternatively you could print the whole line, but simply tell print not **to add its own newline character**.

```
print(line, end="")
```

Sequences:Strings,Lists,and Files

- **Loop through** the entire contents of a file just like this.

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

- **Drawback** of this approach is the fact that the file may be very large, and reading it into a list all at once may **take up too much RAM**.

Sequences:Strings,Lists,and Files

- Python treats the file itself as **a sequence of lines**. So looping through the lines of a file can be done like this:

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

- When writing to a file, make sure you **do not clobber** any files you will need later!
- An example of opening a file for output:

```
outfile = open("mydata.out", "w")
```

Sequences:Strings,Lists,and Files

- 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**.A new user is separated by one *or more spaces.*
  - *Let's go to P163.*

```
# 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")
```

```
# process each line of the input file
for line in infile:
    # get the first and last names from line
    first, last = line.split()
    # create the 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)

main()
```

Sequences:Strings,Lists,and Files

- It's not unusual for programs to have **multiple files open** for reading and writing at the same time.
- The **lower function** is used to convert the names into all lower case, in the event the names are mixed upper and lower case.
- We need to **concatenate '\n'** to our output to the file, otherwise the user names would be all run together on one line.

Programming Exercises

- A certain CS professor gives 5-point quizzes that are graded on the scale 5-A, 4-B, 3-C, 2-D, 1-F, 0-F. Write a program that accepts a quiz score as an input and prints out the corresponding grade.

Programming Exercises

- Write an improved version of the chaos . py program that allows a user to input two initial values and the number of iterations,5.11. Exercises and then prints a nicely formatted table showing how the values change over time. For example, if the starting values were . 25 and . 26 with 10 iterations, the table might look like this:

Programming Exercises

index	0.25	0.26
1	0.731250	0.750360
2	0.766441	0.730547
3	0.698135	0.767707
4	0.821896	0.695499
5	0.570894	0.825942
6	0.955399	0.560671
7	0.166187	0.960644
8	0.540418	0.147447
9	0.968629	0.490255
10	0.118509	0.974630