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.

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

The String Data Type

- Getting a string as input

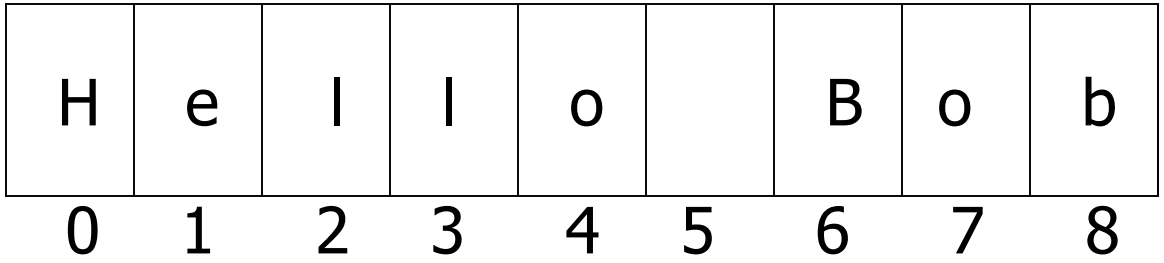
```
>>> firstName = input("Please enter your name: ")
Please enter your name: John
>>> print("Hello", firstName)
Hello John
```

- 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

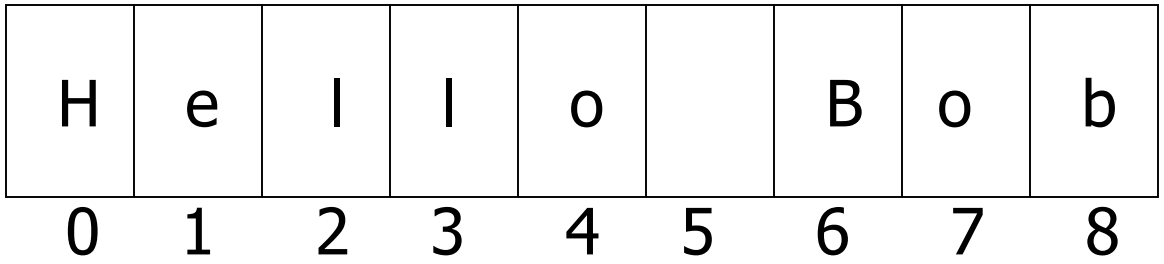
- We can access the individual characters in a string through *indexing*.
- The positions in a string are numbered from the left, starting with 0.
- The general form is <string>[<expr>], where the value of expr determines which character is selected from the string.

The String Data Type




```
>>> 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



- 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'
```

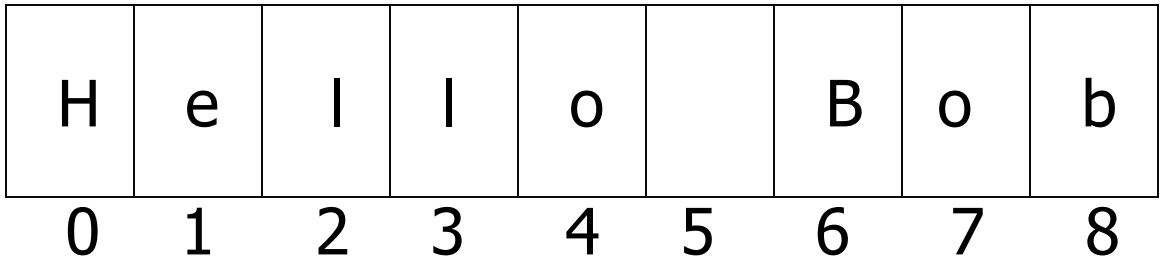
The String Data Type

- 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




```
>>> 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.

```
>>> “Hello" + “world"
‘HelloWorld'
>>> “Hello" + "And" + “world"
' Hello And world '
>>> 3 * "spam"
'spamspamspam'
>>> "spam" * 5
'spamspamspamspamspam'
>>> (3 * "spam") + ("eggs" * 5)
'spamspamspameggseggseggseggseggs'
```

The String Data Type

```
>>> len("spam")
4
>>> for ch in “Hello!":
        print (ch)

Hello!
```

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.

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 Algorithm 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 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: AP1 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: AP1 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

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()

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.