INLS 560
Programming for Information Professionals

Strings

UNC
SCHOOL OF INFORMATION AND LIBRARY SCIENCE

Joan Boone
jpboone@email.unc.edu

Part 1

- Basic string operations

Part 2

- Modify, search, replace, and splitting strings

Strings are text

Most applications work with text in some format

- Google Docs, word processors
- Email
- Social media
- Search engines
- Databases
- Data and text mining analyze text by deriving patterns and trends

Some familiar Python examples

- `resting_HR = input('Enter your resting heart rate: ')`
- `print('You qualify for the loan.')`
- `steps_file = open('steps.txt', 'r')`

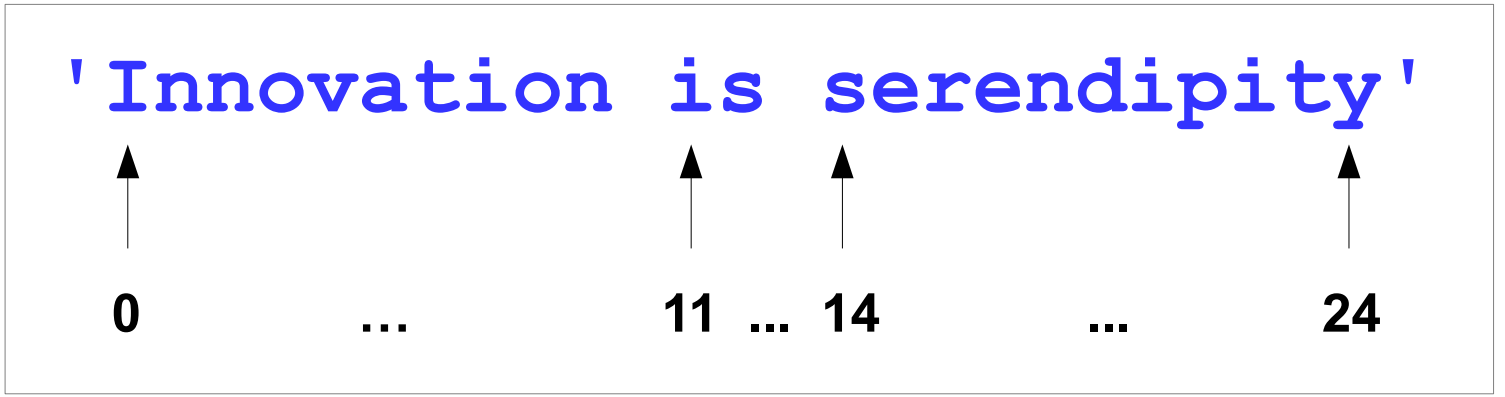
Basic String Operations: Iteration

Very similar to **`list`** and **`dictionary`** iteration: use a **`for`** loop

```
# Count the number of times a letter occurs in a string

def main():
    # Define a counter
    count = 0

    # Get a string from the user.
    input_string = input('Enter a sentence: ')

    # Count occurrences of letter E or e
    for letter in input_string:
        if letter == 'E' or letter == 'e':
            count = count + 1

    print('The letter E appears', count, 'times.')

main()
```

letter_counter.py

Basic String Operations: Indexing




```
text = 'Innovation is serendipity'
print(text[3], text[12], text[24])
```

→ o s y

`IndexError` Exception occurs if an index is out of range for a string.

Common error:
looping beyond end of a string

```
index = 0
while index < 30:
    print(text[index])
    index = index + 1
```

How to avoid:

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

string_indexing.py

Basic String Operations: Concatenation

Concatenation is a common operation where one string is concatenated, or appended, to the end of another string

```
first_name = 'Monty'
last_name = 'Python'
full_name = first_name + last_name
print(full_name)
MontyPython

full_name = first_name + ' ' + last_name
print(full_name)
Monty Python
```

Using concatenation in the input prompt

```
for month in range(1, 13):
    inches = float(input('Enter rainfall for month ' + str(month) + ': '))
    total = total + inches

Enter rainfall for month 1: 5
Enter rainfall for month 2: 10
...
```

Strings are Immutable
(so are integers and floats)

- In Python, strings cannot be modified once they are created. Some operations appear to modify a string, but they do not.

Figure 8-4 The string 'Carmen' assigned to `name`

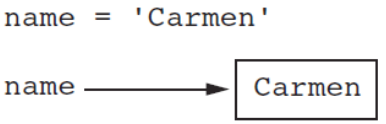
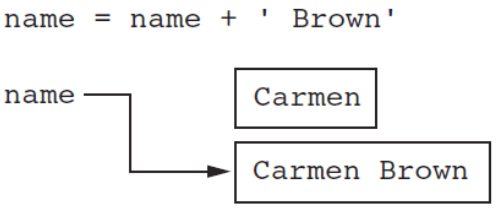


Figure 8-5 The string 'Carmen Brown' assigned to `name`



- Takeaway: you cannot use an expression in the form `string[index]` on the left side of an assignment operator, i.e., you cannot modify a character in a string using an index.

```
text = 'Innovation is serendipity'
text[14] = 'S'  ⟶  TypeError: 'str' object does not support item assignment

text = 'Innovation is Serendipity'  ⟵  Correct way to modify string
```

Source: Starting Out with Python by Tony Gaddis

Basic String Operations: Slicing

String slices select a subset of characters in a string.

A string slice is also called a ***substring***.

Very similar to `list` slicing

```
days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday', 'Saturday']

weekdays = days[1:6]
```

String slicing: `string[start : end]`

```
python_author = 'Guido van Rossum'
first_name = python_author[:5]
last_name = python_author[6:]
print(first_name, last_name)
```

```
Guido van Rossum
```

Testing Strings with `in` and `not in`

`in` and `not in` operators return `True` or `False`

```
opening_text = 'It was a dark and stormy night'


if 'stormy' in opening_text:
    print('The string “stormy” was found')
else:
    print('The string "stormy" was not found')
```

Other String Operations using Methods

- Testing for values of strings
- Performing various modifications
- Searching for sub-strings and replacing sequences of characters

Source: Starting Out with Python by Tony Gaddis

Methods for Testing Values of Strings

Each method returns `True` or `False`, and assumes the string contains at least one character

Method	Description
`isalnum()`	Returns true if string contains only alphabetic letters or digits
`isalpha()`	Returns true if string contains only alphabetic letters
`islower()`	Returns true if all of the alphabetic letters in the string are lowercase
`isupper()`	Returns true if all of the alphabetic letters in the string are uppercase
`isnumeric()`	Returns true if all characters are numeric (0-9)
`isspace()`	Returns true if the string contains only whitespace characters, e.g., newlines (\n) and tabs (\t)

Python documentation for String methods

Testing Values of Strings for Input Validation

To validate an input string, often there are several requirements that must be met for validation to be successful. Here's a general algorithm that uses String methods for validation.

- Use boolean variables to specify whether a validation requirement has been met (is it True or False?), e.g, is the string numeric, at least 8 characters long, etc.
- Initially, set all of these variables to False, i.e., assume the validation will fail. If a validation requirement is met, then set its variable to True
- Loop through each character of the string, and determine if the requirements are met.
- After evaluating the string, check to see if all of the boolean variables have been set to True
  - If all are true, then the input string is valid
  - If one or more are false, the input string is invalid

Example: Password Validation

Prompts for a password, and validates it according to these rules:

- at least 7 characters in length
- contains at least one uppercase letter
- contains at least one lowercase letter
- contains at least one digit

```
def valid_password(password):
    # Set the Boolean variables to false.
    correct_length = False
    has_uppercase = False
    has_lowercase = False
    has_digit = False

    # Validate length first
    if len(password) >= 7:
        correct_length = True

        # Test each character
        for character in password:
            if character.isupper():
                has_uppercase = True
            if character.islower():
                has_lowercase = True
            if character.isdigit():
                has_digit = True

    # Are requirements met?
    if correct_length and has_uppercase and
        has_lowercase and has_digit:
        is_valid = True
    else:
        is_valid = False

    # Return the is_valid variable.
    return is_valid
```

validate_password.py

EXERCISE: Password Validation

Add another validation rule: *the first character must be alphabetic.*

Prompts for a password, and validates it according to these rules:

- at least 7 characters in length
- contains at least one uppercase letter
- contains at least one lowercase letter
- contains at least one digit

```
def valid_password(password):
    # Set the Boolean variables to false.
    correct_length = False
    has_uppercase = False
    has_lowercase = False
    has_digit = False

    # Validate length first
    if len(password) >= 7:
        correct_length = True

        # Test each character
        for character in password:
            if character.isupper():
                has_uppercase = True
            if character.islower():
                has_lowercase = True
            if character.isdigit():
                has_digit = True

    # Are requirements met?
    if correct_length and has_uppercase and
        has_lowercase and has_digit:
        is_valid = True
    else:
        is_valid = False

    # Return the is_valid variable.
    return is_valid
```

validate_password.py

Part 1

- Basic string operations

Part 2

- Modify, search, replace, and splitting strings

Methods to Modify Strings

Method	Description
`lower()`	Returns a copy of string with all alphabetic letters converted to lowercase
`upper()`	Returns a copy of string with all alphabetic letters converted to uppercase
`lstrip()`	Returns a copy of string with all leading whitespace characters removed
`lstrip(char)`	Returns a copy of string with all instances of **`char`** that appear at the beginning of the string removed
`rstrip()`	Returns a copy of string with all trailing whitespace characters removed
`rstrip(char)`	Returns a copy of string with all instances of **`char`** that appear at the end of the string removed
`strip()`	Returns a copy of string with all leading and trailing whitespace characters removed
`strip(char)`	Returns a copy of string with all instances of **`char`** that appear at the beginning and the end of the string removed

Python documentation for String methods

Example: Case-insensitive Comparison

```
# This program makes a case-insensitive comparison
# of a user's response to a prompt

again = 'y'
while again.lower() == 'y':
    print('Hello')
    print('Do you want to see that again?')
    again = input('y = yes, anything else = no: ')
```

```
# This program makes a case-insensitive comparison
# of a user's response to a prompt

again = 'y'
while again.upper() == 'Y':
    print('Goodbye')
    print('Do you want to see that again?')
    again = input('y = yes, anything else = no: ')
```

Source: Starting Out with Python by Tony Gaddis

Methods to Search and Replace Strings

Method	Description
`find(substring)`	The ***substring*** argument is a string. The method returns the lowest index in the string where ***substring*** is found. If substring is not found, the method returns -1.
`replace(old, new)`	The ***old*** and ***new*** arguments are both strings. The method returns a copy of the string with all instances of ***old*** replaced by ***new***.
`startswith(substring)`	The ***substring*** argument is a string. The method returns true if the string starts with ***substring.***
`endswith(substring)`	The ***substring*** argument is a string. The method returns true if the string ends with ***substring.***

Python documentation for String methods

Splitting a String to create a List

- **`split`** method returns a list containing words in the string
- By default, the method uses spaces as separators

```
def main():
    # Create a string with multiple words.
    my_string = 'One two three four'

    # Split the string.
    word_list = my_string.split()

    print(word_list)

main()
```

```
['One', 'two', 'three', 'four']
```

- To specify a different separator, pass as an argument:

```
    date_string = '10/08/2019'
    date_list = date_string.split('/')
    print(date_list)
```

```
['10', '08', '2019']
```

Source: Starting Out with Python by Tony Gaddis

Example: Parsing email addresses

- Suppose you have a list or file of email addresses and you want to extract the domain part of each address
- One approach is to use string slicing:

```
email_addr = 'newhire@startup.com'

local_part = email_addr[0:7]

domain_part = email_addr[8:]

print(domain_part)
```

- Is there a better approach?

email_addresses.py

Example: Phone Number Translator
Exercise 5, Chapter 8

Many companies use phone numbers like 555-GET-FOOD so the number is easier to remember. On a standard phone, the alphabetic letters are mapped to numbers.

How to write a program that prompts user for a phone number in XXX-XXX-XXXX format and translates any alphabetic characters to numeric?


```
Enter the phone number in the format XXX-XXX-XXXX: 555-GET-FOOD
The phone number is 555-438-3663
```

phone_number_translator.py