15-110 Principles of Computing – F21

Lecture 15:

Strings 1

Guest Teacher:
Hend Gedawy

Teacher:
Gianni A. Di Caro

Outline

Outline

String data type: sequence of characters

❖ **String data type** is used to represent sequences of characters, written in python enclosed in quotes

Single quotes:

```
'Hi'
'Hello!'
'z'
'abc'
'_wow_'
```

Double quotes:

```
"Number 5"
"abc"
"Hello!"
"   "
```

✓ Choose the right quotes if the string contains *quotes as part of the sequence*

```
"I'm Joe"

'This "trick" is cool!'
```

Triple quotes:

```
'''This is a very long line of text that it might be convenient
to write over multiple lines to make it well readable.
This is often the case with strings that are used to "describe" a
function or a piece of code'''
```

String data type: sequence of characters

✓ Virtually *any* character can be included in a string sequence!

```
'This is a   biZarre   sTRING! *&%^$ _-+@#//>><<}{}[]'
```

- Case matter!

```
su = 'Hello!'
sl = 'hello!'
```

su is different from sl

- Spaces are characters as others

```
s =  'Hello!'
sp = 'Hello! '
```

s is different from sp

➢ String data type is indicated with `str`

```
s =  'Hello!'
is_string = type(s) == str
print(is_string)
```

→ True

Outline

String data type vs. tuples: sequences, non-scalar, immutable

❖ String data types are alike tuples, and share with them a similar syntax, but have a different representation!

✓ Sequences → however restricted to characters → We can perform characters-specific operations!

```
s = 'This is a string of 33 characters'
```

As as tuple: s = ('T', 'h', 'i', ...) which is not really nice/flexible to represent text!

`'Hello Joe'`

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

As in tuples/list, each element in the string is paired to a position index

String data type vs. tuples: sequences, non-scalar, immutable

❖ String data types are alike tuples and share with them a similar syntax but have a different representation!

- ✓ Non-scalar → sequences where individual items or sub-sequences can be accessed with the operators for *indexing* and *slicing*: `[]`, `[:]`, `[::]`

```
s = 'This is a string of 33 characters'

print('s has', len(s), 'characters')

print('First character (at position index 0):', s[0])

print('6th character (at position index 5):', s[5])

print('Last character of the sequence:', s[-1])

print('Sub-sequence at position indices 7 to 12 (included):', s[7:13])

print('Sub-sequence of characters at even position indices:', s[0::2])
```

String data type vs. tuples: sequences, non-scalar, immutable

- String data types are alike tuples and share with them a similar syntax but have a different representation!
  - ✓ Immutable → Cannot be changed!

```
s = 'Principles of computing'
x = s[3]    # This is Ok!
s[3] = '?'
```

```
          TypeError: 'str' object does not support item assignment
```

Outline

- Defining String Data Type
- String Data VS. Tuples
- String Operators
- Built-In String Methods
- Practical Problems

String operators: `in` (part of, *membership*)

- **Part of,** `in` operator, *overloaded:* `s in p` returns `True` if s is contained within `p`, and `False` otherwise → **Membership**

```
s = 'Joe'
in_hello = s in 'Hello Joe'
in_food = s in 'Yummy meal'
print(in_hello, in_food, type(in_hello))
```

```
True False <class 'bool'>
```

- **Not part of,** `not in` operator, *overloaded:* `s not in p` returns `True` if s is *not* contained within `p`, and `False` otherwise

```
s = 'Joe'
in_hello = s not in 'Hello Joe'
in_food = s not in 'Yummy meal'
print(in_hello, in_food, type(in_hello))
```

```
False True <class 'bool'>
```

String operators: + (concatenation)

- **String concatenation**, + operator, *overloaded*: `s = s1 + s2` returns a new string `s` consisting of `s1` and `s2` ***joined together***

```
greet_joe = 'Hello Joe'
comma = ','
greet_mary = ' hello Mary'
greet = greet_joe + comma +  greet_mary
print(greet)
```

```
Hello Joe, hello Mary
```

➢ Can I do `greet + 1`? NO!

String operators: * (duplication)

- **String duplication**, * operator, *overloaded:* `sn = n * s` creates a ***new*** string consisting of **multiple copies** (`n`) of the string `s` (same as in lists!)
  - `s` is a string and `n` is an integer

```
s = 'Hello'
n = 4
print(s * n)
```

```
HelloHelloHelloHello
```

```
s = 'Hello'
n = 4
s4 = n * s
print(s4)
```

Commutative!

➢ What if `n` is a negative integer?

```
s = 'Hello'
n = -4
print(s * n)
```

➢ Can I do `s*s`? NO!

Augmented forms: **+=**, *=

- ✓ **Augmented (*shorthand*) form of** + operator: `s += x`
  - `s` must be already defined
  - Equivalent to `s = s + x`

```
s = 'abc'
s1 = 'defg'
s += s1
print(s)
```

```
abcdefg
```

- ✓ **Augmented form of** * operator: `s *= n`
  - `s` must be already defined
  - Equivalent to `s = s * n`

```
s = 'abc'
s *= 3
print(s)
```

```
abcabcabc
```

➢ Works also for numeric types!

Outline

- Defining String Data Type
- String Data VS. Tuples
- String Operators
- Built-In String Methods
    - Join, Split
    - endswith(), startswith(), Count()
    - Find, Replace
    - String Formatting
    - Character Classification
    - Case Conversion
- Practical Problems

Outline

- ❖ Defining String Data Type
- ❖ String Data VS. Tuples
- ❖ String Operators
- ❖ Built-In String Methods
  - Join, Split
  - endswith(), startswith(), Count()
  - Find, Replace
  - String Formatting
  - Character Classification
  - Case Conversion
- ❖ Practical Problems

`join(seq)` string method: a string out of a sequence

- `s.join(seq)` *string* method:
  - given an *iterable* object `seq` (e.g., `tuple, list, dict, set`) containing only string elements,
  - `s.join(seq)` returns a string in which the elements of `seq` have been joined by `s` as separator

```
l = ['1','2','3','4']
sep = "-"
s = sep.join(l)
print(s)
```

```
→   1-2-3-4
```

```
l = ['1','2','3','4']
sep = ''
s = sep.join(l)
print(s)
```

```
→   1234
```

```
l = ['1','2','3','4']
s = ''.join(l)
print(s)
```

```
→   1234
```

`join(seq)` string method: a string out of a sequence

```
l = ('1','2','3','4’)
sep = ", "
s = sep.join(l)
print(s)
```

```
→  1, 2, 3, 4
```

```
l = ('This','is','a','story')
sep = " "
s = sep.join(l)
print(s)
```

```
→  This is a story
```

A string `s` is treated as a <u>sequence of characters</u>

```
s = '123'
sep = 'abc'
print(sep.join(s))
```

```
→  1abc2abc3
```

```
s = 'abc'
sep = '123'
print(sep.join(s))
```

```
→  a123b123c
```

`split(seq)` string method: a list out of a string

- `s.split(sep, max)` *string* method:
  - given a string `s`,
  - the method <u>splits</u> the string in a list of strings, based on the separator string `sep`.

➢ The method <u>returns a list of strings.</u>

❖ The arguments `sep` and `max` are <u>optional</u>. If `sep` is not given, <u>white space is used as default separator</u> (*all* white spaces are removed in this case!). `max` indicates the maximum number of substrings in the list (plus one).

```
s = "I am John Smith"
ls = s.split()
print(ls)
```

```
['I', 'am', 'John', 'Smith']
```

```
s = "I  am     John  Smith"
ls = s.split()
print(ls)
```

```
['I', 'am', 'John', 'Smith']
```

```
s = "I am: John Smith"
ls = s.split(':')
print(ls)
```

```
['I am', ' John Smith']
```

`split(seq)` string method: a list out of a string

```
s = "I am   John    Smith"
ls = s.split(' ')
print(ls)
```

Note: if a white space `sep` is passed, the behavior is different from the default (in the default case, python removes iteratively all occurrences of white spaces!)

```
['I', 'am', '', '', 'John', '', '', '', 'Smith']
```

Outline

Built-in String Methods: endswith(), startswith()

- `s.endswith(<suffix>)`: returns True if s ends with the specified <suffix>, and False otherwise
  - `s = "crazy bar"`
  - `s.endswith("bar")` → True
- `s.endswith(<suffix>, <start>, <end>)`: as above, but now the comparison is restricted to the substring indicated by <start> and <end>
  - `s = "crazy bar"`
  - `s.endswith("bar", 0, 5)` → False
- `s.startswith(<prefix>)`: analogous to `endswith()`, but checking if the string begins with a given substring

Built-in String Methods: Count

- `s.count(<sub>)`**:** returns the number of non-overlapping occurrences of substring <sub> in s
  - `s = "moo ooh pooh"`
  - `s.count("oo")` → 3
  - `"moo ooh pooh".count("oo")` → 3

- `s.count(<sub>, <start>, <end>)`**:** returns the number of non-overlapping occurrences of substring <sub> in the s slice defined by <start> and <end>
  - `s = "moo ooh pooh"`
  - `s.count("oo", 3, len(s))` → 2

Outline

- ❖ Defining String Data Type
- ❖ String Data VS. Tuples
- ❖ String Operators
- ❖ Built-In String Methods
  - Join, Split
  - endswith(), startswith(), Count()
  - Find, Replace
  - String Formatting
  - Character Classification
  - Case Conversion
- ❖ Practical Problems

Built-in String Methods: Find

- `s.find(<sub>)`**:** returns the **lowest index** in s where the substring <sub> is found, **-1** is returned if the substring is not found
  - s = "crazy bar bar"
  - s.find ("bar") → 6
  - s.find("star") → -1
- `s.find(<sub>, <start>, <end>)`**:** as above, but now the search is **restricted** to the substring indicated by <start> and <end>
  - s = "crazy bar bar"
  - s.find("bar", 7, 13) → 10

Built-in String Methods: Find

- `s.rfind(<sub>)`**:** searches s starting from the end, such that it returns the **highest index** in s where the substring <sub> is found, **-1** is returned if the substring is not found
  - s = "crazy bar bar"
  - s.rfind ("bar") → 10
  - s.find("bar") → 6
- `s.rfind(<sub>, <start>, <end>)`**:** as above, but now the search is **restricted** to the substring indicated by <start> and <end>
  - s = "crazy bar bar"
  - s.rfind("bar", 0, 10) → 6

Built-in String Methods: Replace

- `s.replace(<old>, <new>)`: returns a *copy* of s with all occurrences of substring <old> replaced by new. If there are no occurrence of <old>, the copy is identical to the original (but it's still a different object)
  - s = "one step, two steps, three steps"
  - s.replace ("step", "stop")   → "one stop, two stops, three stops"

- `s.replace(<old>, <new>, <max>)`: as above, but now the number of replacements is limited to the <max> value
  - s = "one step, two steps, three steps"
  - s.replace("step", "stop", 2)   → "one stop, two stops, three steps"

Outline

Built-in String Methods: String formatting

- `s.center(<width>[, <fill>])` It creates a string of size <width> and aligns (s) in the center filling the right and left with the <fill> character specified. White Space is the default character.

```
s="Hello"
s.center(9)
```

```
'  Hello  '
```

```
s="Hello"
s.center(9, '*')
```

```
'**Hello**'
```

- `s.ljust(<width>[, <fill>])` same as before but aligns (s) on the left and fills character <fill> on the left of string (s)

```
s="Hello"
s.ljust(9)
```

```
'Hello    '
```

```
s="Hello"
s.ljust(9, '*')
```

```
'Hello****'
```

- `s.rjust(<width>[, <fill>])` same as before but aligns (s) on the right and fills character <fill> on the left of string (s)

```
s="Hello"
s.rjust(9)
```

```
'    Hello'
```

```
s="Hello"
s.rjust(9, '*')
```

```
'****Hello'
```

- `s.zfill(<width>)` same as before but fills character (0) on the left of string (s)

```
s="Hello"
s.zfill(9)
```

```
'0000Hello'
```

Built-in String Methods: String formatting

- `s.expandtabs(tabsize=<>)`: It replaces every tab character '\t' with a number of white species defined by tab size

```
s='\tHello'
s.expandtabs(tabsize=2)
```

```
'  Hello'
```

```
s= s='\t\t\tHello’
s.expandtabs(tabsize=2)
```

```
'      Hello'
```

- `s.lstrip([<chars>])`: It removes **all leading whitespaces** of a string and can also be used to remove a **particular character** from leading

```
s='    Hello'
s.lstrip()
```

```
'Hello'
```

```
s= '****Hello'
s.lstrip('*')
```

```
'Hello'
```

- `s.rstrip([<chars>])`: It removes **all trailing whitespaces** of a string and can also be used to remove a **particular character** from trailing

```
s= 'Hello    '
s.rstrip()
```

```
'Hello'
```

```
s='Hello****'
s.rstrip('*')
```

```
'Hello'
```

- `s.strip([<chars>])`: It removes **all leading and trailing whitespaces** of a string and can also be used to remove a **particular character** from both leading and trailing

```
s= '  Hello  '
s.strip()
```

```
'Hello'
```

```
s='**Hello**'
s.strip('*')
```

```
'Hello'
```

Outline

Built-in String Methods: Character classification

- `s.isalpha()`: `True` if **all** characters in `s` are **alphabetic** letters, `False` otherwise
- `s.isalnum()`: `True` if **all** characters in `s` are either **alphabetic** letters **or numeric** digits, `False` otherwise
- `s.isdigit()`: `True` if **all** characters in `s` are **numeric** digits, `False` otherwise
- `s.isidentifier()`: `True` if the string `s` could be used as **identifier (variable, function or class name),** `False` otherwise
- `s.islower()`: `True` if **all** characters in `s` are **lower case**, `False` otherwise
- `s.isupper()`: `True` if **all** characters in `s` are **upper case**, `False` otherwise
- `s.isspace()`: `True` if **all** characters in `s` are **white spaces**, `False` otherwise
- `s.istitle()`: `True` if the **first character** in `s` is **upper** case and **all** the **others** are **lower** case, `False` otherwise
- `s.isprintable()`: `True` if **all** characters in `s` are **printable** (digits, letters, punctuation characters, space), `False` otherwise (e.g. of strings that are not printable '\tHello', '\nHello')

Outline

Built-in String Methods: Case Conversion

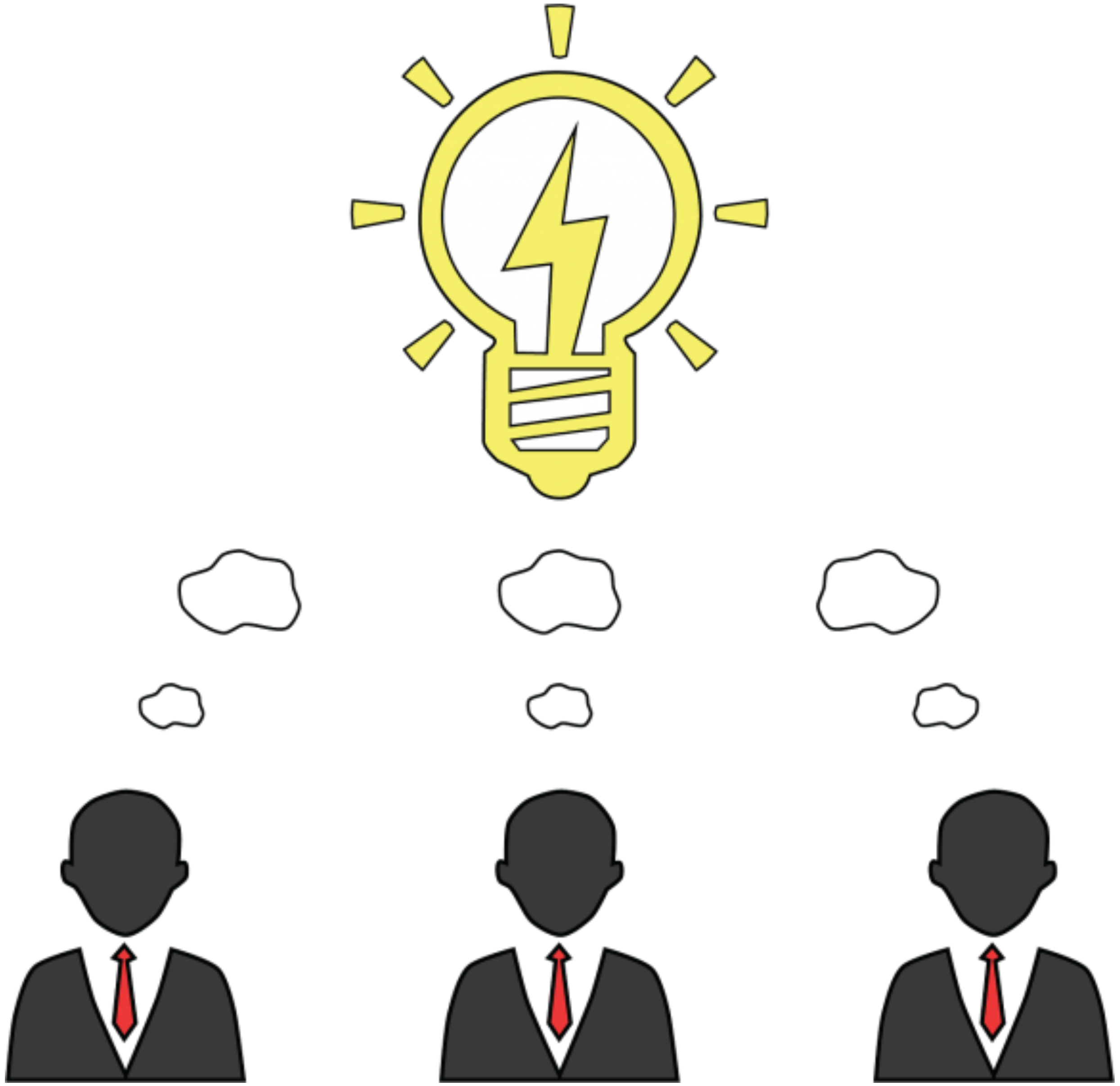
- `s.lower()`**:** returns **a *copy* of s** with **all** alphabetic **characters** converted to **lowercase**
- `s.upper()`**:** returns **a copy of s** with **all** alphabetic **characters** converted to **uppercase**
- `s.swapcase()`**:** returns **a copy of s** with **uppercase** alphabetic **characters** converted **to lowercase and vice versa**. Non-alphabetic characters are unchanged.
- `s.capitalize()`**:** returns **a copy of s** with the **first character** converted to **uppercase** and **all other** characters converted to **lowercase**
- `s.title()`**:** returns **a copy of s** in which the **first letter of each word** (separated by spaces) is converted to **uppercase** and **remaining** letters are **lowercase**

SUMMARY

Operators and Methods Summary

- **String Operators**
  - in, +, *, += , *=
- **Built-In String Methods**
  - L.join()
  - s.split()
  - s.startswith(), s.endswith()
  - s.count()
  - s.find(), s.rfind()
  - s.replace()
  - String Formatting
    - s.center(), s.expandtabs(), s.ljust(), s.rjust(), s.zfill(), s.lstrip(), s.rstrip(), s.strip()
  - Character Classification
    - s.isalpha(), s.isalnum(), s.isdigit(), s.isidentifier(), s.islower(), s.isupper(), s.isprintable(), s.isspace(), s.istitle()
  - Case Conversion
    - s.lower(), s.upper(), s.capitalize(), s.title()

Test your knowledge



Outline

- Defining String Data Type
- String Data VS. Tuples
- String Operators
- Built-In String Methods
- Practical Problems

Practical Problems

1

Problem 1: Find Words With a Special Character And Strip It

Function: problem1(text, character)

Input: String, character

Output: print the words that starts with the given special character after removing the character

Example:

- **Input:** ‘Universities in education city are: #1 CMU, #2 WCMC, #3 VCU, #4 ABP, #5 NW, #6 TAMU, and #7 GT’

- **Output:**
  - 1
  - 2
  - 3
  - 4
  - 5
  - 6
  - 7

Problem 1: Find Words With a Special Character And Strip It

Function: **problem1(text, character)**

Input: String, character

Output: print the words that starts with the given special character after removing the character

Solution:

```
def problem1(text, character):

    splitted= text.split()

    for s in splitted:
        if s.startswith(character):
            print(s.lstrip(character))
```

Practical Problems

2

Problem 2: Find Words of a Certain Type

Function: problem2(sentence)

Input: Text with some capitalized words and numbers

Output: print the capitalized words and numbers,
print the capitalized words converted to titles in one string separated by commas

Example:

- **Input:** ‘students in computer science program need to take courses JAVA 125 GTI 215 ML 312 ALGO 451’
- **Output:**
  - JAVA
  - 125
  - GTI
  - 215
  - ML
  - 312
  - 451
  - Titles Are: Java, Gti, Ml, Algo

Problem 2: Find Words of a Certain Type

Function: problem2(sentence)

Input: String with some capitalized words and numbers

Output: print the capitalized words and numbers,
print the capitalized words converted to titles in one string separated by commas

Solution:

```
def problem2(sentence):
    splitted= sentence.split()
    titlesList=[]

    for s in splitted:
        if s.isupper():
            print(s)
            titlesList.append(s.title())
        if s.isdigit():
            print(s)

    titlesString=','.join(titlesList)
    print('Titles Are:', titlesString)
```

Practical Problems

3

Problem 3: Find Unique Words And Count Their Appearance

Function: **problem3(repetitiveString)**

Input: String with words separated with commas

Output: String with unique words (remove repetitions), print of each unique word and how many times it appeared in the text

Example:

- **Input:** ‘Apple,Mango,Strawberry,Mango,Mango,Apple’
- **Output:**

```
Apple: 2
Mango: 3
Strawberry: 1
Apple Mango Strawberry
```

Problem 3: Find Unique Words And Count Their Appearance

Input: String with words separated with commas

Output: String with unique words (remove repetitions), print of each unique word and how many times it appeared in the text

Solution:

```
def problem3 (repetitiveString):
    uniqueStringsCombined=""

    repetitiveSplitted= repetitiveString.split(' ,')

    for s in repetitiveSplitted:
        if not s in uniqueStringsCombined:
            print(s,':',repetitiveString.count(s))
            uniqueStringsCombined+= (s+' ')

    print('Unique Text: ', uniqueStringsCombined)
```

```
def problem3WithJoin (repetitiveString):
    uniqueStrings=[]

    repetitiveSplitted= repetitiveString.split(' ,')

    for s in repetitiveSplitted:
        if not s in uniqueStrings:
            print(s,':',repetitiveString.count(s))
            uniqueStrings.append(s)

    uniqueStringsCombined= ' '.join(uniqueStrings)
    print('Unique Text: ', uniqueStringsCombined)
```