15-110 Principles of Computing – F21

Lecture 15:
Strings 1

Teacher:
Gianni A. Di Caro

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
```

`is_string` → `True`

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*: `[ ]`, `[ : ]`, `[ : : ]`

```python
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
```

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`
  - `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
```

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

```
s = 'abc'
s1 *= 3
```

➢ Works also for numeric types!

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.capitalize()`: returns a copy of `s` with the first character converted to uppercase and all other characters converted to lowercase
- `s.swapcase()`: returns a copy of `s` with uppercase alphabetic characters converted to lowercase and vice versa. Non-alphabetic characters are unchanged.
- `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

Built-in String Methods: Count, Find, and Replace

- `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")

- `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

Built-in String Methods: Count, Find, and Replace

- `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, Find, and Replace

- `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: Count, Find, and Replace

- `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: Count, Find, and 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"

Built-in String Methods: String formatting

- `s.center(<width>[, <fill>])`
- `s.expandtabs(tabsize=8)`
- `s.ljust(<width>[, <fill>])`
- `s.rjust(<width>[, <fill>])`
- `s.zfill(<width>)`

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

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 lower case, `False` otherwise
- `s.isprintable()`: `True` if all characters in `s` are printable, `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