Python for Data Analytics

Strings

Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

Systems Software &
Architecture Lab.

Seoul National University

Jan. 6 – 17, 2020

String Data Type

- A string is a sequence of characters

- For strings, + means "concatenation" (same as lists)

- When a string contains numbers, it is still a string

- We can convert numbers in a string into a number using `int()`

```
>>> str1 = "Hello"
>>> str2 = 'there'
>>> bob = str1 + str2
>>> print(bob)
Hellothere
>>> str3 = '123'
>>> str3 = str3 + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
TypeError: must be str, not int
>>> x = int(str3) + 1
>>> print(x)
124
```

Looking Inside Strings

- We can get at any single character is a string using an index specified in square brackets
- The index value must be an integer and starts at zero
- The index value can be an expression that is computed
- You will get a python error if you attempt to index beyond the end of a string

b	a	n	a	n	a
0	1	2	3	4	5

```
>>> fruit = 'banana'
>>> letter = fruit[1]
>>> print(letter)
a
>>> x = 3
>>> w = fruit[x-1]
>>> print(w)
n
>>> print(fruit[6])
```

Strings Have Length

- `len(`*str*`)`
  - The built-in function `len()` gives you the length of a string
- `len()` also works for
  - Lists
  - Tuples
  - Dictionaries
  - Sets
  - …

b	a	n	a	n	a
0	1	2	3	4	5

```
>>> fruit = 'banana'
>>> print(len(fruit))
6
>>> empty = ''
>>> print(len(empty))
0
>>> nl = '\n'
>>> print(len(nl))
1
```

Looping Through Strings

- Using while statement

```python
fruit = 'banana'
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

- Using for statement
  - More elegant (or "Pythonic")

```python
fruit = 'banana'
for letter in fruit:
    print(letter)
```

Counting Character(s)

- Loop through each letter in a string and counts the number of times the loop encounters the 'a' character

```
fruit = 'banana'
count = 0
for letter in fruit:
    if letter == 'a':
        count = count + 1
print(count)
```

- *str*.count(*s*)
  - Return the number of non-overlapping occurrences of substring *s*

```
fruit = 'banana'
print(fruit.count('a'))
print(fruit.count('na'))
```

Slicing Strings

- *str*[*start*:*stop*:*step*]
- Same as list slicing

M	o	n	t	y		P	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> s = 'Monty Python'
>>> print(s[1:2])
o
>>> print(s[8:])
thon
>>> print(s[:])
Month Python
>>> print(s[::2])
MntPto
```

```
>>> print(s[-4:])
thon
>>> print(s[:-5])
Monty P
>>> print(s[:-6:-1])
nohty
>>> print(s[::-1])
nohtyP ytnoM
```

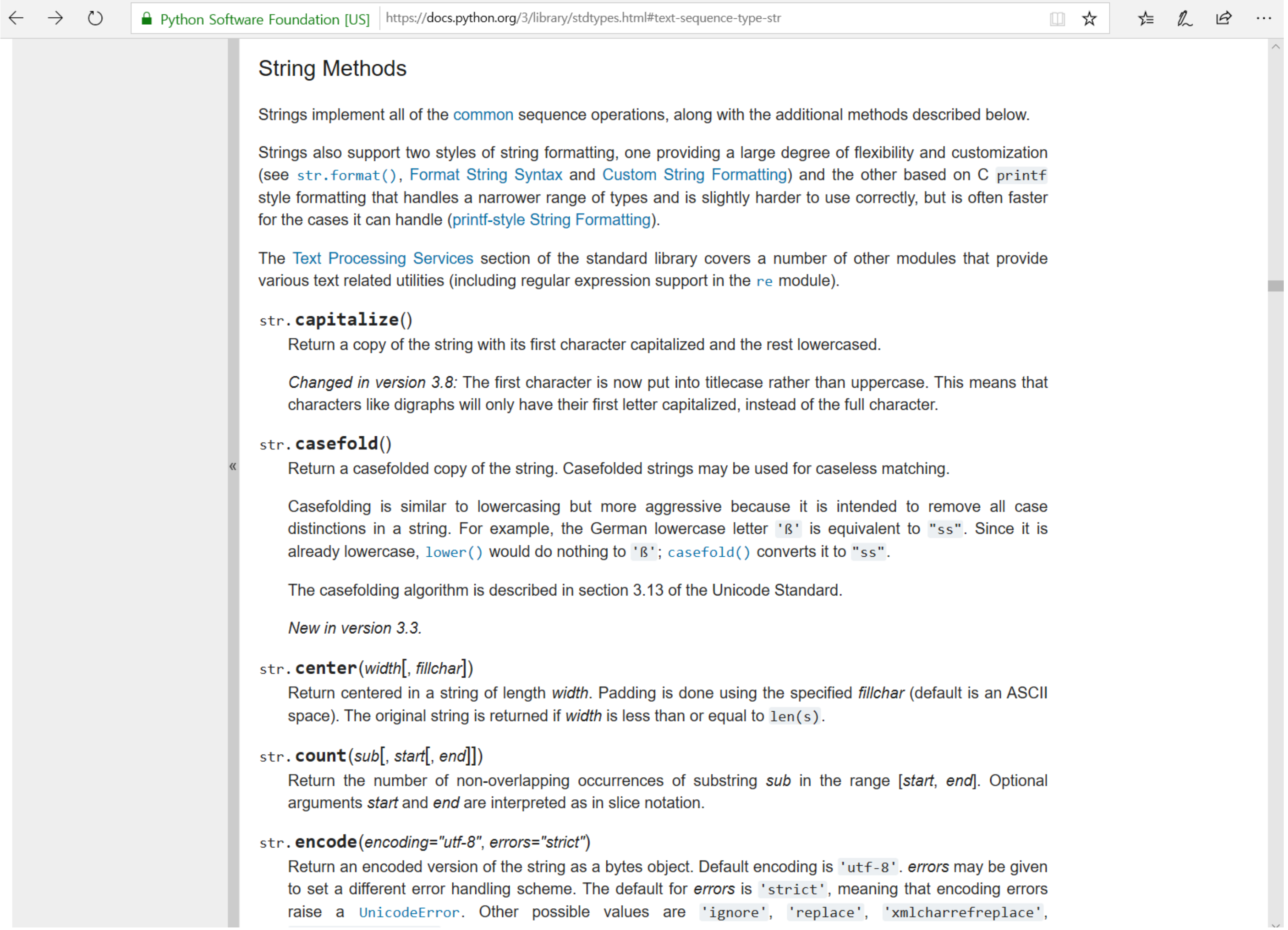
The in Operator

- Check to see if one string is in another string

- Return True or False

```
>>> fruit = 'banana'
>>> 'n' in fruit
True
>>> 'm' in fruit
False
>>> 'nan' in fruit
True
>>> if 'a' in fruit:
...     print('Found it!')
Found it!
```

String Methods


Python Software Foundation [US]
https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str

String Methods

Strings implement all of the common sequence operations, along with the additional methods described below.

Strings also support two styles of string formatting, one providing a large degree of flexibility and customization (see `str.format()`, Format String Syntax and Custom String Formatting) and the other based on C `printf` style formatting that handles a narrower range of types and is slightly harder to use correctly, but is often faster for the cases it can handle (printf-style String Formatting).

The Text Processing Services section of the standard library covers a number of other modules that provide various text related utilities (including regular expression support in the `re` module).

`str.`**`capitalize`**`()`

Return a copy of the string with its first character capitalized and the rest lowercased.

Changed in version 3.8: The first character is now put into titlecase rather than uppercase. This means that characters like digraphs will only have their first letter capitalized, instead of the full character.

`str.`**`casefold`**`()`

Return a casefolded copy of the string. Casefolded strings may be used for caseless matching.

Casefolding is similar to lowercasing but more aggressive because it is intended to remove all case distinctions in a string. For example, the German lowercase letter `'ß'` is equivalent to `"ss"`. Since it is already lowercase, `lower()` would do nothing to `'ß'`; `casefold()` converts it to `"ss"`.

The casefolding algorithm is described in section 3.13 of the Unicode Standard.

New in version 3.3.

`str.`**`center`**`(`*width*[, *fillchar*]`)`

Return centered in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

`str.`**`count`**`(`*sub*[, *start*[, *end*]]`)`

Return the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

`str.`**`encode`**`(`*encoding="utf-8", errors="strict"*`)`

Return an encoded version of the string as a bytes object. Default encoding is `'utf-8'`. *errors* may be given to set a different error handling scheme. The default for *errors* is `'strict'`, meaning that encoding errors raise a `UnicodeError`. Other possible values are `'ignore'`, `'replace'`, `'xmlcharrefreplace'`,

Test

- *str*.`startswith(`*prefix*[,*start*[,*end*]]`)`
  - `True` if string starts with the *prefix*
- *str*.`endswith(`*suffix* [,*start*[,*end*]]`)`
  - `True` if string ends with the suffix
- *str*.`isalpha()`
  - `True` if all characters are alphabetic
- *str*.`isdigit()`
  - `True` if all characters are digits
- *str*.`isprintable()`
  - `True` if all characters are printable
- *str*.`islower()`
  - `True` if all characters are lower case
- *str*.`isupper()`
  - `True` if all characters are uppercase
- *str*.`isspace()`
  - `True` if there are only whitespace characters

Find / Replace

- *str*.count(*sub*[,*start*[,*end*]])
- *str*.find(*sub*[,*start*[,*end*]]))
  - Return the lowest index where substring *sub* is found (-1 if *sub* is not found)
- *str*.index(*sub*[,*start*[,*end*]]))
  - Like find(), but raise ValueError if *sub* is not found
- *str*.replace(*old*, *new*[,*count*]))
  - Return a copy of the string with all occurrences of substring *old* replaced by *new*

```
>>> b = 'banana'
>>> print(b.count('a'))
3

>>> print(b.find('x'))
-1

>>> print(b.index('na'))
2

>>> print(b.replace('a','x'))
bxnxnx
```

Reformat (1)

- *str*.`lower()`
  - Return a copy of the string with all the characters converted to lowercase
- *str*.`upper()`
  - Return a copy of the string with all the characters converted to uppercase
- *str*.`capitalize()`
  - Return a copy of the string with its first character capitalized and the rest lowercased

```
>>> s = 'MoNtY PyThOn'

>>> print(s.lower())
monty python

>>> print(s.upper())
MONTY PYTHON

>>> print(s.capitalize())
Monty python
```

Reformat (2)

- *str*.`lstrip(`[*chars*]`)`
  - Return a copy of the string with leading characters removed.
  - If omitted, the *chars* argument defaults to whitespace characters

- *str*.`rstrip(`[*chars*]`)`
  - Like `lstrip()`, but trailing characters are removed

- *str*.`strip(`[*chars*]`)`
  - *str*.`lstrip(`[*chars*]`)` + *str*.`rstrip(`[*chars*]`)`

```
>>> s = '-- monty python --'

>>> print(s.lstrip(' -'))
monty python --

>>> print(s.rstrip('- '))
--- monty python

>>> print(s.strip(' –mno'))
ty pyth
```

Split

- *str*.split(*sep*, *maxsplit*)
  - Return a list of the words in the string, using *sep* as the delimiter string
  - If *maxsplit* is given, at most *maxsplit* splits are done. Otherwise all possible splits are made
  - The *sep* argument may consist of multiple characters (None = whitespaces)
  - If *sep* is given, consecutive delimiters are NOT grouped together

```
>>> s = 'hi hello world'
>>> s.split()
['hi', 'hello', 'world']
>>> t = '1:2:3'
>>> t.split(':')
['1', '2', '3']
>>> t.split(':', 1)
['1', '2:3']
>>> t = '1:2::3'
>>> t.split(':')
['1', '2', '', '3']
>>> t.split('::')
['1:2', '3']
```

Join

- *str*.join(*iterable*)
  - Return a string which is the concatenation of the strings in *iterable*
  - *iterable*: List, Tuple, String, Dictionary, Set
  - The separator between elements is the string (*str*) providing this method

```
>>> menu = ['spam', 'ham', 'egg']
>>> ','.join(menu)
'spam,ham,egg'
>>> ' '.join(menu)
'spam ham egg'
>>> ' * '.join(menu)
'spam * ham * egg'
>>> '#'.join('spam')
's#p#a#m'
```