String Manipulation

In [16]:
```
lst='ankit ,   kiio ,  summi'.split(',')
lst
```

Out[16]:
```
['ankit ', '   kiio ', '  summi']
```

In [17]:
```
lst1=[s.strip() for s  in lst]
lst1
```

Out[17]:
```
['ankit', 'kiio', 'summi']
```

In [18]:
```
s="@".join(lst1)
s
```

Out[18]:
```
'ankit@kiio@summi'
```

In [19]:
```
[1,2,3].index(3)
```

Out[19]:
```
2
```

In [20]:
```
"ankit".index('k')
```

Out[20]:
```
2
```

In [21]:
```
s.find('@')
```

Out[21]:
```
5
```

In [22]:
```
s.find('@@')
```

Out[22]:
```
-1
```

Note the difference between find and index is that index raises an exception if the string isn't found (versus returning –1)

In [23]:
```
s.count('@')
```

Out[23]:
```
2
```

In [24]:
```
s=s.replace('@','#')
s
```

Out[24]:
```
'ankit#kiio#summi'
```

Argument --> Description

count --> Return the number of non-overlapping occurrences of substring in the string.

endswith --> Returns True if string ends with suffix.

startswith --> Returns True if string starts with prefix.

join --> Use string as delimiter for concatenating a sequence of other strings.

index --> Return position of first character in substring if found in the string; raises ValueError if not found.

find --> Return position of first character of rst occurrence of substring in the string; like index, but returns –1 if not found.

rfind --> Return position of first character of last occurrence of substring in the string; returns –1 if not found.

replace --> Replace occurrences of string with another string.

strip,rstrip,lstrip --> Trim whitespace, including newlines; equivalent to x.strip() (and rstrip, lstrip, respectively) for each element.

split --> Break string into list of substrings using passed delimiter.

lower --> Convert alphabet characters to lowercase.

upper --> Convert alphabet characters to uppercase.

casefold --> Convert characters to lowercase, and convert any region-specific variable character combinations to a common comparable form.

ljust,rjust --> Left justify or right justify, respectively; pad opposite side of string with spaces (or some other fill character) to return a string with a minimum width.

In [25]:
```
'ankit'.count('nk')
```

Out[25]:
```
1
```

In [26]:
```
'xyxyxxxyy'.count('xy')
```

Out[26]:
```
3
```

In [27]:
```
s
```

Out[27]:
```
'ankit#kiio#summi'
```

In [28]:
```
s.endswith('summi')
```

Out[28]:
```
True
```

In [29]:
```
s.endswith('mmi')
```

Out[29]:
```
True
```

In [30]:
```
s.endswith('me')
```

Out[30]:
```
False
```

In [31]:
```
s.startswith('an')
```

Out[31]:
```
True
```

In [32]:
```
s.startswith('pp')
```

Out[32]:
```
False
```

In [34]:
```
s.rfind('k')
```

Out[34]:
```
6
```

In [35]:
```
'    ankit       '.rstrip()
```

Out[35]:
```
'    ankit'
```

In [36]:
```
'    ankit       '.lstrip()
```

Out[36]:
```
'ankit       '
```

In [37]:
```
'ankit'.upper()
```

Out[37]:
```
'ANKIT'
```

In [38]:
```
'ANkit'.lower()
```

Out[38]:
```
'ankit'
```

In [39]:
```
' AnkIT'.casefold()
```

Out[39]:
```
' ankit'
```

In [40]:
```
# Basic ljust examples
text = "Hello"
print(text.ljust(10))        # 'Hello     ' (spaces added)
print(text.ljust(10, '-'))   # 'Hello-----'
print(text.ljust(10, '*'))   # 'Hello*****'

# With specified fill character
print(text.ljust(8, '.'))    # 'Hello...'
```

```
Hello
Hello-----
Hello*****
Hello...
```

In [41]:
```
text = "Hi"

# Different padding methods
print(text.ljust(5, '-'))   # 'Hi---'
print(text.rjust(5, '-'))   # '---Hi'
print(text.center(5, '-'))  # '-Hi--'
print(text.zfill(5))        # '000Hi' (only zeros, right-aligned)
```

```
Hi---
---Hi
--Hi-
000Hi
```

Regular Expressions

The re module functions fall into three categories: pattern matching, substitution, and splitting. Naturally these are all related; a regex describes a pattern to locate in the text, which can then be used for many purposes.

suppose we wanted to split a string with a variable number of whitespace characters (tabs, spaces, and newlines). The regex describing one or more whitespace characters is \\s+

In [42]:
```
import re
text="ankit\n summi\t\t\r kiio"
re.split('\\s+',text)
```

Out[42]:
```
['ankit', 'summi', 'kiio']
```

When you call re.split('\s+', text), the regular expression is first compiled, and then its split method is called on the passed text.
You can compile the regex yourself with re.compile, forming a reusable regex object

In [43]:
```
regex=re.compile('\\s+')
regex.split(text)
```

Out[43]:
```
['ankit', 'summi', 'kiio']
```

If, instead, you wanted to get a list of all patterns matching the regex, you can use the findall method

In [44]:
```
regex.findall(text)
```

Out[44]:
```
['\n ', '\t\t\r ']
```

To avoid unwanted escaping with \ in a regular expression, use raw string literals like r'C:\x' instead of the equivalent 'C:\x'.

Creating a regex object with re.compile is highly recommended if you intend to apply the same expression to many strings; doing so will save CPU cycles.

In [46]:
```
text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com
"""

pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'

# re.IGNORECASE makes the regex case-insensitive
regex = re.compile(pattern, flags=re.IGNORECASE)
regex.findall(text)
```

Out[46]:
```
['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']
```

search returns a special match object for the first email address in the text. For the preceding regex, the match object can only tell us the start and end position of the pattern in the string

In [48]:
```
text
```

Out[48]:
```
'Dave dave@google.com\nSteve steve@gmail.com\nRob rob@gmail.com\nRyan ryan@yahoo.com\n'
```

In [47]:
```
m = regex.search(text)
m
```

Out[47]:
```
<re.Match object; span=(5, 20), match='dave@google.com'>
```

In [49]:
```
text[m.start():m.end()]
```

Out[49]:
```
'dave@google.com'
```

Relatedly, sub will return a new string with occurrences of the pattern replaced by the a new string

In [50]:
```
print(regex.sub('REDACTED', text))
```

```
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

Suppose you wanted to find email addresses and simultaneously segment each address into its three components: username, domain name, and domain suffix.

To do this, put parentheses around the parts of the pattern to segment

In [51]:
```
pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'
regex = re.compile(pattern, flags=re.IGNORECASE)
```

A match object produced by this modified regex returns a tuple of the pattern components with its groups method

In [52]:
```
m = regex.match('wesm@bright.net')
m.groups()
```

Out[52]:
```
('wesm', 'bright', 'net')
```

findall returns a list of tuples when the pattern has groups

In [53]:
```
regex.findall(text)
```

Out[53]:
```
[('dave', 'google', 'com'),
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]
```

sub also has access to groups in each match using special symbols like \1 and \2.

The symbol \1 corresponds to the first matched group, \2 corresponds to the second, and so forth

In [54]:
```
print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text))
```

```
Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

Regular expression methods

Argument ---> Description

findall ---> Return all non-overlapping matching patterns in a string as a list

finditer ---> Like findall, but returns an iterator

match ---> Match pattern at start of string and optionally segment pattern components into groups; if the pattern matches, returns a match object, and otherwise None

search ---> Scan string for match to pattern; returning a match object if so; unlike match, the match can be anywhere in the string as opposed to only at the beginning

split ---> Break string into pieces at each occurrence of pattern

sub, subn ---> Replace all (sub) or first n occurrences (subn) of pattern in string with replacement expression; use symbols \1, \2, ... to refer to match group elements in the replacement string

Vectorized String Functions in pandas

In [57]:
```
import numpy as np
import pandas as pd
data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com','Rob': 'rob@gmail.com', 'Wes': np.nan}
data = pd.Series(data)
data
```

Out[57]:
```
Dave     dave@google.com
Steve    steve@gmail.com
Rob        rob@gmail.com
Wes                  NaN
dtype: object
```

In [58]:
```
data.str.contains('gmail')
```

Out[58]:
```
Dave     False
Steve     True
Rob       True
Wes        NaN
dtype: object
```

In [59]:
```
data.str.findall(pattern, flags=re.IGNORECASE)
```

Out[59]:
```
Dave     [(dave, google, com)]
Steve    [(steve, gmail, com)]
Rob        [(rob, gmail, com)]
Wes                        NaN
dtype: object
```

In [60]:
```
matches = data.str.match(pattern, flags=re.IGNORECASE)
matches
```

Out[60]:
```
Dave     True
Steve    True
Rob      True
Wes       NaN
dtype: object
```

In [61]:
```
data.str[:5]
```

Out[61]:
```
Dave     dave@
Steve    steve
Rob      rob@g
Wes        NaN
dtype: object
```

Partial listing of vectorized string methods

Method --> Description

cat --> Concatenate strings element-wise with optional delimiter

contains --> Return boolean array if each string contains pattern/regex

count --> Count occurrences of pattern

extract --> Use a regular expression with groups to extract one or more strings from a Series of strings; the result will be a DataFrame with one column per group

endswith --> Equivalent to x.endswith(pattern) for each element

startswith --> Equivalent to x.startswith(pattern) for each element

findall --> Compute list of all occurrences of pattern/regex for each string

get --> Index into each element (retrieve i-th element)

isalnum --> Equivalent to built-in str.alnum

isalpha --> Equivalent to built-in str.isalpha

isdecimal --> Equivalent to built-in str.isdecimal

isdigit --> Equivalent to built-in str.isdigit

islower --> Equivalent to built-in str.islower

isnumeric --> Equivalent to built-in str.isnumeric

isupper --> Equivalent to built-in str.isupper

join --> Join strings in each element of the Series with passed separator

len --> Compute length of each string

lower, upper --> Convert cases; equivalent to x.lower() or x.upper() for each element

match --> Use re.match with the passed regular expression on each element, returning matched groups as list

pad --> Add whitespace to left, right, or both sides of strings

center --> Equivalent to pad(side='both')

repeat --> Duplicate values (e.g., s.str.repeat(3) is equivalent to x * 3 for each string)

replace --> Replace occurrences of pattern/regex with some other string

slice --> Slice each string in the Series

split --> Split strings on delimiter or regular expression

strip --> Trim whitespace from both sides, including newlines

rstrip --> Trim whitespace on right side

lstrip --> Trim whitespace on left side

In [ ]: