Part 0 of 2: Simple string processing review

This notebook accompanies the videos for Topic 5: Preprocessing unstructured text.

In [1]:

In [2]:
```
# Strings have methods for checking "global" string properties
print("1.", text.isalpha())

# These can also be applied per character
```

```
1. False
2. [True, True, True, True, True, True, True, False, False, False, False, False, False]
```

In [3]:
```
# Here are a bunch of additional useful methods
print("BELOW: (global) -> (per character)")
print(text.isdigit(), "-->", [c.isdigit() for c in text])
print(text.isspace(), "-->", [c.isspace() for c in text])
print(text.islower(), "-->", [c.islower() for c in text])
print(text.isupper(), "-->", [c.isupper() for c in text])
```

```
BELOW: (global) -> (per character)
False --> [False, False, False, False, False, False, False, True, True, True, True, False, True]
False --> [False, False, False, False, False, False, False, False, False, False, False, False, False]
False --> [True, True, True, False, False, False, True, False, False, False, False, False, False]
False --> [False, False, False, True, True, True, False, False, False, False, False, False, False]
False --> [False, False, False, False, False, False, False, True, True, True, True, False, True]
```

Exercise 0 (1 point). Create a new function that checks whether a given input string is a properly formatted social security number, i.e., has the pattern, `XXX-XX-XXXX`, *including* the separator dashes, where each `X` is a digit. It should return `True` if so or `False` otherwise.

In [4]:
```
def is_ssn(s):
    #
    # YOUR CODE HERE
    #
```

In [5]:
```
# Test cell: `is_snn_test`
assert is_ssn('832-38-1847')
assert not is_ssn('832 -38 -  1847')
assert not is_ssn('832-bc-3847')
assert not is_ssn('832381847')
assert not is_ssn('8323-8-1847')
assert not is_ssn('abc-de-ghij')
```

```
(Passed!)
```

Regular expressions

Exercise 0 hints at the general problem of finding patterns in text. A handy tool for this problem is Python's regular expression module.

A *regular expression* is specially formatted pattern, written as a string. Matching patterns with regular expressions has 3 steps:

1. You come up with a pattern to find.
2. You compile it into a *pattern object*.
3. You apply the pattern object to a string, to find *matches*, i.e., instances of the pattern within the string.

> What follows is just a small sample of what is possible with regular expressions in Python; refer to the regular expression documentation for many more examples and details.

In [1]:

Basics

Let's see how this scheme works for the simplest case, in which the pattern is an exact substring.

In [2]:
```
pattern = 'fox'
pattern_matcher = re.compile(pattern)

input = 'The quick brown fox jumps over the lazy dog'
matches = pattern_matcher.search(input)
```

```
<_sre.SRE_Match object; span=(16, 19), match='fox'>
```

You can also query matches for more information.

In [8]:
```
print(matches.group())
print(matches.start())
print(matches.end())
```

```
fox
16
19
(16, 19)
```

Module-level searching. For infrequently used patterns, you can also skip creating the pattern object and just call the module-level search function, `re.search()`.

In [9]:
```
matches_2 = re.search ('jump', input)
assert matches_2 is not None
```

```
Found jump @ (20, 24)
```

Other Search Methods

1. match() - Determine if the RE matches at the beginning of the string.
2. search() - Scan through a string, looking for any location where this RE matches.
3. findall() - Find all substrings where the RE matches, and returns them as a list.
4. finditer() - Find all substrings where the RE matches, and returns them as an iterator.

Creating pattern groups

In [3]:
```
# Make the expression more readable with a re.VERBOSE pattern
re_names2 = re.compile ('''^              # Beginning of string
                           ([a-zA-Z]+)    # First name
                           \s+            # At least one space
                           ([a-zA-Z]+\s)? # Optional middle name
                           ([a-zA-Z]+)    # Last name
                           $              # End of string
                        ''',
                        re.VERBOSE)
print (re_names2.match ('Rich Vuduc').groups ())
print (re_names2.match ('Rich S Vuduc').groups ())
```

```
('Rich', None, 'Vuduc')
('Rich', 'S ', 'Vuduc')
('Rich', 'Salamander ', 'Vuduc')
```

For more details on the re.VERBOSE pattern, see here.

Tagging pattern groups

In [4]:
```
# Named groups
re_names3 = re.compile ('''^
                           (?P<first>[a-zA-Z]+)
                           \s
                           (?P<middle>[a-zA-Z]+\s)?
                           \s*
                           (?P<last>[a-zA-Z]+)
                           $
                        ''',
                        re.VERBOSE)
print (re_names3.match ('Rich Vuduc').group ('first'))
print (re_names3.match ('Rich S Vuduc').group ('middle'))
```

```
Rich
S
Vuduc
```

A regular expression debugger. There are several online tools to help you write and debug your regular expressions. See, for instance, pythex, regexr, or regex101.

Email addresses

In the next exercise, you'll apply what you've learned about regular expressions to build a pattern matcher for email addresses.

Although there is a formal specification of what constitutes a valid email address, for this exercise, let's use the following simplified rules.

- We will restrict our attention to ASCII addresses and ignore Unicode. If you don't know what that means, don't worry about it---you shouldn't need to do anything special given our code templates, below.
- An email address has two parts, the username and the domain name. These are separated by an `@` character.
- A username **must begin with an alphabetic** character. It may be followed by any number of additional *alphanumeric* characters or any of the following special characters: `.` (period), `-` (hyphen), `_` (underscore), or `+` (plus).
- A domain name **must end with an alphabetic** character. It may consist of any of the following characters: alphanumeric characters, `.` (period), `-` (hyphen), or `_` (underscore).
- Alphabetic characters may be uppercase or lowercase.
- No whitespace characters are allowed.

Valid domain names usually have additional restrictions, e.g., there are a limited number of endings, such as `.com`, `.edu`, and so on. However, for this exercise you may ignore this fact.

Exercise 1 (2 points). Write a function `parse_email` that, given an email address `s`, returns a tuple, `(user-id, domain)` corresponding to the user name and domain name.

For instance, given `richie@cc.gatech.edu` it should return `(richie, cc.gatech.edu)`.

Your function should parse the email only if it exactly matches the email specification. For example, if there are leading or trailing spaces, the function should *not* match those. See the test cases for examples.

If the input is not a valid email address, the function should raise a `ValueError`.

> The requirement, "raise a `ValueError`" refers to a technique for handling errors in a program known as *exception handling*. The Python documentation covers exceptions in more detail, including raising `ValueError` objects.

In [5]:
```
def parse_email (s):
    """Parses a string as an email address, returning an (id, domain) pair."""
    #
    # YOUR CODE HERE
    #
```

In [6]:
```
# Test cell: `parse_email_test`

def pass_case(u, d):
    s = u + '@' + d
    msg = "Testing valid email: '{}'".format(s)
    print(msg)
    assert parse_email(s) == (u, d), msg

pass_case('richie', 'cc.gatech.edu')
pass_case('bertha_hugely', 'sampson.edu')
pass_case('JKRowling', 'Huge-Books.org')

def fail_case(s):
    msg = "Testing invalid email: '{}'".format(s)
    print(msg)
    try:
        parse_email(s)
    except ValueError:
        print("==> Correctly throws an exception!")
    else:
        raise AssertionError("Should have, but did not, throw an exception!")

fail_case('x @hpcgarage.org')
fail_case('   quiggy.smith38x@gmail.com')
fail_case('richie@cc.gatech.edu  ')
fail_case('4test@gmail.com')
```

```
Testing valid email: 'richie@cc.gatech.edu'
Testing valid email: 'bertha_hugely@sampson.edu'
Testing valid email: 'JKRowling@Huge-Books.org'
Testing invalid email: 'x @hpcgarage.org'
==> Correctly throws an exception!
Testing invalid email: '   quiggy.smith38x@gmail.com'
==> Correctly throws an exception!
Testing invalid email: 'richie@cc.gatech.edu  '
==> Correctly throws an exception!
Testing invalid email: '4test@gmail.com'
==> Correctly throws an exception!
Testing invalid email: 'richie@cc.gatech.edu7'
==> Correctly throws an exception!
```

Phone numbers

Exercise 2 (2 points). Write a function to parse US phone numbers written in the canonical "(404) 555-1212" format, i.e., a three-digit area code enclosed in parentheses followed by a seven-digit local number in three-hyphen-four digit format. It should also **ignore** all leading and trailing spaces, as well as any spaces that appear between the area code and local numbers. However, it should **not** accept any spaces in the area code (e.g., in '(404)') nor should it in the seven-digit local number.

It should return a triple of strings, `(area_code, first_three, last_four)`.

If the input is not a valid phone number, it should raise a `ValueError`.

In [7]:
```
def parse_phone1 (s):
    #
    # YOUR CODE HERE
    #
```

In [8]:
```
# Test cell: `parse_phone1_test`

def rand_spaces(m=5):
    from random import randint
    return ' ' * randint(0, m)

def asm_phone(a, l, r):
    return rand_spaces() + '(' + a + ')' + rand_spaces() + l + '-' + r + rand_spaces()

def gen_digits(k):
    from random import choice # 3.5 compatible; 3.6 has `choices()`
```

```
    DIGITS = '0123456789'
    return ''.join([choice(DIGITS) for _ in range(k)])

def pass_phone(p=None, a=None, l=None, r=None):
    if p is None:
        a = gen_digits(3)
        l = gen_digits(3)
        r = gen_digits(4)
        p = asm_phone(a, l, r)
    else:
        assert a is not None and l is not None and r is not None, "Need to supply sample solution."
    msg = "Should pass: '{}'".format(p)
    print(msg)
    p_you = parse_phone1(p)
    assert p_you == (a, l, r), "Got {} instead of ('{}', '{}', '{}')".format(p_you, a, l, r)

def fail_phone(s):
    msg = "Should fail: '{}'".format(s)
    print(msg)
    try:
        p_you = parse_phone1(s)
    except ValueError:
        print("==> Correctly throws an exception.")
    else:
        raise AssertionError("Failed to throw a `ValueError` exception!")


# Cases that should definitely pass:
pass_phone('(404) 121-2121', '404', '121', '2121')
pass_phone('(404)121-2121', '404', '121', '2121')
for _ in range(5):
    pass_phone()

fail_phone("404-121-2121")
fail_phone('(404)555 -1212')
```

```
Should pass: '(404) 121-2121'
Should pass: '(404)121-2121'
Should pass: '  (685)660-6830  '
Should pass: ' (865)  546-6361   '
Should pass: ' (927)    117-2683 '
Should pass: '   (175)    366-4921   '
Should pass: '(404)    801-4155'
Should fail: '404-121-2121'
==> Correctly throws an exception.
Should fail: '(404)555 -1212'
==> Correctly throws an exception.
Should fail: ' ( 404)121-2121'
==> Correctly throws an exception.
Should fail: '(abc) def-ghij'
==> Correctly throws an exception.
```

Exercise 3 (3 points). Implement an enhanced phone number parser that can handle any of these patterns.

- (404) 555-1212
- (404) 5551212
- 404-555-1212
- 404-5551212
- 404555-1212
- 4045551212

As before, it should not be sensitive to leading or trailing spaces. Also, for the patterns in which the area code is enclosed in parentheses, it should not be sensitive to the number of spaces separating the area code from the remainder of the number.

In [9]:
```
def parse_phone2 (s):
    #
    # YOUR CODE HERE
    #
```

In [10]:
```
# Test cell: `parse_phone2_test`

def asm_phone2(a, l, r):
    from random import random
    x = random()
    if x < 0.33:
        a2 = '(' + a + ')' + rand_spaces()
    elif x < 0.67:
        a2 = a + '-'
    else:
        a2 = a
    y = random()
    if y < 0.5:
        l2 = l + '-'
    else:
        l2 = l
    return rand_spaces() + a2 + l2 + r + rand_spaces()
```

```
def pass_phone2(p=None, a=None, l=None, r=None):
    if p is None:
        a = gen_digits(3)
        l = gen_digits(3)
        r = gen_digits(4)
        p = asm_phone2(a, l, r)
    else:
        assert a is not None and l is not None and r is not None, "Need to supply sample solution."
    msg = "Should pass: '{}'".format(p)
    print(msg)
    p_you = parse_phone2(p)
    assert p_you == (a, l, r), "Got {} instead of ('{}', '{}', '{}')".format(p_you, a, l, r)

pass_phone2("  (404)   555-1212  ", '404', '555', '1212')
pass_phone2("(404)555-1212  ", '404', '555', '1212')
pass_phone2("  404-555-1212 ", '404', '555', '1212')
pass_phone2("  404-5551212 ", '404', '555', '1212')
pass_phone2(" 4045551212", '404', '555', '1212')

for _ in range(5):
    pass_phone2()


def fail_phone2(s):
    msg = "Should fail: '{}'".format(s)
    print(msg)
    try:
        parse_phone2 (s)
    except ValueError:
        print ("==> Function correctly raised an exception.")
    else:
        raise AssertionError ("Function did *not* raise an exception as expected!")

failure_cases = ['+1 (404) 555-3355',
                 '404.555.3355',
                 '404 555-3355',
                 '404 555 3355',
                 '(404-555-1212'
                ]
for s in failure_cases:
    fail_phone2(s)
```

```
Should pass: '  (404)   555-1212  '
Should pass: '(404)555-1212  '
Should pass: '  404-555-1212 '
Should pass: '  404-5551212 '
Should pass: ' 4045551212'
Should pass: '  082-3199716 '
Should pass: '    422-174-1815   '
Should pass: '515-4896154 '
Should pass: '(009)   9498846 '
Should pass: ' 334376-8198'
Should fail: '+1 (404) 555-3355'
==> Function correctly raised an exception.
Should fail: '404.555.3355'
==> Function correctly raised an exception.
Should fail: '404 555-3355'
==> Function correctly raised an exception.
Should fail: '404 555 3355'
==> Function correctly raised an exception.
Should fail: '(404-555-1212'
==> Function correctly raised an exception.

(Passed!)
```

Fin! This cell marks the end of Part 0. Don't forget to save, restart and rerun all cells, and submit it. When you are done, proceed to Parts 1 and 2.