edX   GTx: CSE6040x
Introduction to Computing for Data Analysis

Help   akesarwani3 ▼

Sample solutions

🔖 Bookmark this page

---

part0 (Score: 8.0 / 8.0)

1. Test cell (Score: 1.0 / 1.0)
2. Test cell (Score: 2.0 / 2.0)
3. Test cell (Score: 2.0 / 2.0)
4. Test cell (Score: 3.0 / 3.0)

Important note! Before you turn in this lab notebook, make sure everything runs as expected:

- First, **restart the kernel** -- in the menubar, select Kernel→Restart.
- Then **run all cells** -- in the menubar, select Cell→Run All.

Make sure you fill in any place that says YOUR CODE HERE or "YOUR ANSWER HERE."

Part 0: Simple string processing review

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

```
In [1]: text = "sgtEEEr2020.0"
```

```
In [2]: text.isalpha()
```

Out[2]: False

```
In [3]: text.isdigit()
        text.isalpha()
        text.isspace()
        text.islower()
        text.isupper()
        text.isnumeric()
```

Out[3]: False

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]:    Student's answer                                                    (Top)

        def is_ssn (s):
            parts = s.split('-')
            correct_lengths = [3, 2, 4]
            if len(parts) != len(correct_lengths):
                return False
            for p, n in zip(parts, correct_lengths):
                if not (p.isdigit() and len(p) == n):
                    return False
            return True
```

In [5]:   Grade cell: `is_ssn_test`   Score: 1.0 / 1.0 (Top)

```
# 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')
print("\n(Passed!)")
```

```
(Passed!)
```

Regular expressions

Part 1 hints at the general problem of finding patterns in text. A handy tool for this problem is Python's regular expression module (https://docs.python.org/3/howto/regex.html).

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 (https://docs.python.org/3/howto/regex.html) for many more examples and details.

In [6]:   `import re`

Basics

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

In [7]:
```
pattern = 'fox'
pattern_matcher = re.compile (pattern)

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

```
<_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 ())
print (matches.span ())
```

```
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
```

```
assert matches_2 is not None
print ("Found", matches_2.group (), "@", matches_2.span ())
```

```
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 [10]:  # Make the above 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 ())
          print (re_names2.match ('Rich Salamander Vuduc').groups ())
```

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

Tagging pattern groups

```
In [11]:  # 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'))
          print (re_names3.match ('Rich Salamander Vuduc').group ('last'))
```

```
Rich
S
Vuduc
```

A regular expression debugger. There are several online tools to help you write and debug your regular expressions. See, for instance, regex101 (https://regex101.com/).

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 (https://tools.ietf.org/html/rfc5322#section-3.4.1), 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(s)` 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 (https://docs.python.org/3/tutorial/errors.html) in more detail, including raising `ValueError` objects (https://docs.python.org/3/tutorial/errors.html#raising-exceptions).

In [12]: Student's answer   (Top)

```python
def parse_email (s):
    """Parses a string as an email address, returning an (id, domain) pair."""
    pattern = '''
       ^
       (?P<user>[a-zA-Z][\w.\-+]*)
       @
       (?P<domain>[\w.\-]*[a-zA-Z])
       $
    '''
    matcher = re.compile(pattern, re.VERBOSE)
    matches = matcher.match(s)
    if matches:
        return (matches.group('user'), matches.group('domain'))
    raise ValueError("Bad email address")
```

In [13]: Grade cell: `parse_email_test`   Score: 2.0 / 2.0 (Top)

```python
# 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  ')
```

```
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!
```

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 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 [14]:

Student's answer   (Top)

```python
def parse_phone1 (s):
    pattern = '''
        \s*
        \((?P<area>\d{3})\) # Area code
        \s*
        (?P<local3>\d{3}) # Local prefix (3 digits)
        -
        (?P<local4>\d{4}) # Local suffix (4 digits)
    '''
    matcher = re.compile(pattern, re.VERBOSE)
    matches = matcher.match(s)
    if matches:
        return (matches.group('area'), matches.group('local3'), matches.group('local4')
)
    raise ValueError("Invalid phone number? '{}'".format(s))
```

In [15]:

Grade cell: `parse_phone1_test`   Score: 2.0 / 2.0 (Top)

```python
# 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 sampl
e 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)121-2121")
fail_phone("(abc) def-ghij")
```

```
Should pass: '(404) 121-2121'
Should pass: '(404)121-2121'
Should pass: '     (951) 891-7903'
Should pass: ' (440)   972-0741'
Should pass: '    (681)    564-2896    '
Should pass: '(000)765-1184 '
Should pass: '    (519)      185-3702      '
Should fail: '404-121-2121'
==> 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 [16]:

Student's answer   (Top)

```
def parse_phone2(s):
    pattern = '''
        ^\s*                # Leading spaces
        (?P<areacode>
            \d{3}-?         # "xxx" or "xxx-"
            | \(\d{3}\)\s*  # OR "(xxx) "
        )
        (?P<prefix>\d{3})   # xxx
        -?                  # Dash (optional)
        (?P<suffix>\d{4})   # xxxx
        \s*$                # Trailing spaces
    '''
    matcher = re.compile(pattern, re.VERBOSE)
    matches = matcher.match(s)
    if matches is None:
        raise ValueError("'{}' is not in the right format.".format(s))
    areacode = re.search('\d{3}', matches.group('areacode')).group()
    prefix = matches.group('prefix')
```

```
                       suffix = matches.group ('suffix')
                       return (areacode, prefix, suffix)
```

In [17]:

| Grade cell: parse_phone2_test | Score: 3.0 / 3.0 (Top) |

```python
# 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 sampl
e 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'
                 ]
for s in failure_cases:
    fail_phone2(s)

print("\n(Passed!)")
```

```
Should pass: '  (404)   555-1212  '
Should pass: '(404)555-1212  '
Should pass: '  404-555-1212 '
```

```
Should pass: '   404-5551212  '
Should pass: ' 4045551212'
Should pass: '    0901755474   '
Should pass: '  4788310980  '
Should pass: '  642-622-9402  '
Should pass: '   129-8039427'
Should pass: '    589816-5713      '
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.

(Passed!)
```

In [18]:

---

part1 (Score: 5.0 / 5.0)

1. Test cell (Score: 5.0 / 5.0)

Important note! Before you turn in this lab notebook, make sure everything runs as expected:

- First, **restart the kernel** -- in the menubar, select Kernel→Restart.
- Then **run all cells** -- in the menubar, select Cell→Run All.

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE."

Part 1: Processing an HTML file

One of the richest sources of information is the Web (http://www.computerhistory.org/revolution/networking/19/314)! In this notebook, we ask you to use string processing and regular expressions to mine a web page, which is stored in HTML format.

The data: Yelp! reviews. The data you will work with is a snapshot of a recent search on the Yelp! site (https://yelp.com) for the best fried chicken restaurants in Atlanta. That snapshot is hosted here: https://cse6040.gatech.edu/datasets/yelp-example (https://cse6040.gatech.edu/datasets/yelp-example)

If you go ahead and open that site, you'll see that it contains a ranked list of places:

Your task. In this part of this assignment, we'd like you to write some code to extract this list.

Getting the data

First things first: you need an HTML file. The following Jupyter "magic" commands will download a particular web page that we've prepared for this exercise and store it locally in a file.

> If the file exists, this command will not overwrite it. By not doing so, we can reduce accesses to the server that hosts the file. Also, if an error occurs during the download, this cell may report that the downloaded file is corrupt; in that case, you should try re-running the cell.

```
In [1]: !if ! test -f yelp.htm ; then echo "...Downloading 'yelp.htm' ..." ; curl -O https://cse6
        040.gatech.edu/datasets/yelp-example/yelp.htm ; fi
        !if ! test x"`md5sum yelp.htm | awk '{print $1;}'`" = x"4a74a0ee9cefee773e76a22a52d45a8e"
        ; then echo "*** Downloaded file may be corrupt; please re-run this cell. ***" ; rm -f ye
        lp.htm ; else echo "=== File 'yelp.htm' is available locally and appears to be ready for
        use. ===" ; fi
```

```
=== File 'yelp.htm' is available locally and appears to be ready for use. ===
```

Viewing the raw HTML in your web browser. The file you just downloaded is the raw HTML version of the data described previously. Before moving on, you should go back to that site and use your web browser to view the HTML source for the web page. Do that now to get an idea of what is in that file.

> If you don't know how to view the page source in your browser, try the instructions on this site (http://www.wikihow.com/View-Source-Code).

Reading the HTML file into a Python string. Let's also open the file in Python and read its contents into a string named, `yelp_html`.

```
In [2]: with open('yelp.htm') as yelp_file:
            yelp_html = yelp_file.read()

        # Print first few hundred characters of this string:
        print("*** type(yelp_html) == {} ***".format(type(yelp_html)))
        n = 1000
        print("*** Contents (first {} characters) ***\n{} ...".format(n, yelp_html[:n]))
```

```
*** type(yelp_html) == <class 'str'> ***
*** Contents (first 1000 characters) ***
<!DOCTYPE html>
<!-- saved from url=(0079)https://www.yelp.com/search?find_desc=fried+chicken&find_loc=At
lanta%2C+GA&ns=1 -->
```

```
<html xmlns:fb="http://www.facebook.com/2008/fbml" class="js gr__yelp_com" lang="en"><!--
<![endif]--><head data-component-bound="true"><meta http-equiv="Content-Type" content="te
xt/html; charset=UTF-8"><link type="text/css" rel="stylesheet" href="./Best Fried chicken
in Atlanta, GA - Yelp_files/css"><style type="text/css">.gm-style .gm-style-cc span,.gm-s
tyle .gm-style-cc a,.gm-style .gm-style-mtc div{font-size:10px}
</style><style type="text/css">@media print {  .gm-style .gmnoprint, .gmnoprint {    disp
lay:none  }}@media screen {  .gm-style .gmnoscreen, .gmnoscreen {    display:none  }}</st
yle><style type="text/css">.gm-style-pbc{transition:opacity ease-in-out;background-color:
rgba(0,0,0,0.45);text-align:center}.gm-style-pbt{font-size:22px;color:white;font-family:R
oboto,Arial,sans-serif;position:relative;margin:0;top:50%;-webkit-transform:translateY(-5
0%);-ms- ...
```

Oy, what a mess! It will be great to have some code read and process the information contained within this file.

Exercise (5 points): Extracting the ranking

Write some Python code to create a variable named `rankings`, which is a list of dictionaries set up as follows:

- `rankings[i]` is a dictionary corresponding to the restaurant whose rank is `i+1`. For example, from the screenshot above, `rankings[0]` should be a dictionary with information about Gus's World Famous Fried Chicken.
- Each dictionary, `rankings[i]`, should have these keys:
  - `rankings[i]['name']`: The name of the restaurant, a string.
  - `rankings[i]['stars']`: The star rating, as a string, e.g., `'4.5'`, `'4.0'`
  - `rankings[i]['numrevs']`: The number of reviews, as an **integer.**
  - `rankings[i]['price']`: The price range, as dollar signs, e.g., `'$'`, `'$$'`, `'$$$'`, or `'$$$$'`.

In [3]:
Student's answer   (Top)

```python
matchers = {
    'name': '''<a class="biz-name js-analytics-click" data-analytics-label="biz-name" h
ref="[^"]*" data-hovercard-id="[^"]*"><span>(.+)</span></a>''',
    'stars': '''title="([0-9.]+) star rating"''',
    'numrevs': '''(\d+) reviews''',
    'price': '''<span class="business-attribute price-range">(\$+)</span>'''
}

def get_field(s, key):
    from re import search
    assert key in matchers
    match = search(matchers[key], s)
    if match is not None:
        return match.groups()[0]
    return None

sections = yelp_html.split('<span class="indexed-biz-name">')
rankings = []
for i, section in enumerate(sections[1:]):
    rankings.append({})
    for key in matchers.keys():
        rankings[i][key] = get_field(section, key)

for r in rankings:
    r['numrevs'] = int(r['numrevs'])
```

In [4]:
Grade cell: `rankings_test`   Score: 5.0 / 5.0 (Top)

```python
# Test cell: `rankings_test`

assert type(rankings) is list, "`rankings` must be a list"
assert all([type(r) is dict for r in rankings]), "All `rankings[i]` must be dictionarie
s"

print("=== Rankings ===")
for i, r in enumerate(rankings):
    print("{}. {} ({}): {} stars based on {} reviews".format(i+1,
                                                             r['name'],
                                                             r['price'],
```

```
                                                           r['stars'],
                                                           r['numrevs']))

    assert rankings[0] == {'numrevs': 549, 'name': 'Gus's World Famous Fried Chicken', 'sta
    rs': '4.0', 'price': '$$'}
    assert rankings[1] == {'numrevs': 1777, 'name': 'South City Kitchen – Midtown', 'stars'
    : '4.5', 'price': '$$'}
    assert rankings[2] == {'numrevs': 2241, 'name': 'Mary Mac's Tea Room', 'stars': '4.0',
    'price': '$$'}
    assert rankings[3] == {'numrevs': 481, 'name': 'Busy Bee Cafe', 'stars': '4.0', 'price'
    : '$$'}
    assert rankings[4] == {'numrevs': 108, 'name': 'Richards' Southern Fried', 'stars': '4.
    0', 'price': '$$'}
    assert rankings[5] == {'numrevs': 93, 'name': 'Greens &amp; Gravy', 'stars': '3.5', 'pr
    ice': '$$'}
    assert rankings[6] == {'numrevs': 350, 'name': 'Colonnade Restaurant', 'stars': '4.0',
    'price': '$$'}
    assert rankings[7] == {'numrevs': 248, 'name': 'South City Kitchen Buckhead', 'stars':
    '4.5', 'price': '$$'}
    assert rankings[8] == {'numrevs': 1558, 'name': 'Poor Calvin's', 'stars': '4.5', 'price
    ': '$$'}
    assert rankings[9] == {'numrevs': 67, 'name': 'Rock's Chicken &amp; Fries', 'stars': '4
    .0', 'price': '$'}

    print("\n(Passed!)")
```

```
=== Rankings ===
1. Gus's World Famous Fried Chicken ($$): 4.0 stars based on 549 reviews
2. South City Kitchen – Midtown ($$): 4.5 stars based on 1777 reviews
3. Mary Mac's Tea Room ($$): 4.0 stars based on 2241 reviews
4. Busy Bee Cafe ($$): 4.0 stars based on 481 reviews
5. Richards' Southern Fried ($$): 4.0 stars based on 108 reviews
6. Greens &amp; Gravy ($$): 3.5 stars based on 93 reviews
7. Colonnade Restaurant ($$): 4.0 stars based on 350 reviews
8. South City Kitchen Buckhead ($$): 4.5 stars based on 248 reviews
9. Poor Calvin's ($$): 4.5 stars based on 1558 reviews
10. Rock's Chicken &amp; Fries ($): 4.0 stars based on 67 reviews
```

---

part2 (Score: 0.0 / 0.0)

1. Test cell (Score: 0.0 / 0.0)
2. Written response (Score: 0.0 / 0.0)
3. Written response (Score: 0.0 / 0.0)

Important note! Before you turn in this lab notebook, make sure everything runs as expected:

- First, **restart the kernel** -- in the menubar, select Kernel → Restart.
- Then **run all cells** -- in the menubar, select Cell → Run All.

Make sure you fill in any place that says YOUR CODE HERE or "YOUR ANSWER HERE."

Part 2: An extreme case of regular expression processing

> This part is **OPTIONAL**. That is, while there are exercises, they are worth 0 points each.

There is a beautiful theory underlying regular expressions, and efficient regular expression processing is regarded as one of the classic problems of computer science. In the last part of this lab, you will explore a bit of that theory, albeit by experiment.

In particular, the code cells below will walk you through a simple example of the potentially **hidden cost** of regular expression parsing. And if you really want to geek out, look at the article on which this example is taken: https://swtch.com/~rsc/regexp/regexp1.html

(https://swtch.com/~rsc/regexp/regexp1.html)

Quick review

Exercise 0 (ungraded) Let a^n be a shorthand notation for a string in which a is repeated n times. For example, a^3 is the same as aaa and a^6 is the same as $aaaaaa$. Write a function to generate the string for a^n, given a string a and an integer $n \geq 1$.

In [1]:
```
Student's answer                                                              (Top)

def rep_str (a, n):
    """Returns a string consisting of an input string repeated a given number of times.
"""
    assert type(a) is str and n >= 1
    return a * n
```

In [2]:
```
Grade cell: rep_str_test                                      Score: 0.0 / 0.0 (Top)

# Test cell: `rep_str_test`

def check_fixed(a, n, ans):
    msg = "Testing: '{}'^{} -> '{}'".format(a, n, ans)
    print(msg)
    assert rep_str(a, n) == ans, "Case failed!"

check_fixed('a', 3, 'aaa')
check_fixed('cat', 4, 'catcatcatcat')
check_fixed('', 100, '')

def check_rand():
    from random import choice, randint
    a = ''.join([choice([chr(k) for k in range(ord('a'), ord('z')+1)]) for _ in range(r
andint(1, 5))])
    n = randint(1, 10)
    msg = "Testing: '{}'^{}".format(a, n)
    print(msg)
    s_you = rep_str(a, n)
    for k in range(0, n*len(a), len(a)):
        assert s_you[k:(k+len(a))] == a, "Your result, '{}', is not correct at position
{} [{}].".format(s_you, k)

for _ in range(10):
    check_rand()

print("\n(Passed!)")
```

```
Testing: 'a'^3 -> 'aaa'
Testing: 'cat'^4 -> 'catcatcatcat'
Testing: ''^100 -> ''
Testing: 'f'^3
Testing: 'hnhh'^9
Testing: 'u'^8
Testing: 'evm'^8
Testing: 'iclup'^10
Testing: 'c'^6
Testing: 'jxmu'^3
Testing: 'bqhd'^10
Testing: 'ib'^10
Testing: 'wu'^3

(Passed!)
```

An initial experiment

Intuitively, you should expect (or hope) that the time to determine whether a string of length n matches a given pattern will be proportional to n. Let's see if this holds when matching simple input strings of repeated letters against a pattern designed to match such strings.

```
In [3]:  import re
```

```
In [4]:  # Set up an input problem
         n = 3
         s_n = rep_str ('a', n) # Input string
         pattern = '^a{%d}$' % n # Pattern to match it exactly

         # Test it
         print ("Matching input '{}' against pattern '{}'...".format (s_n, pattern))
         assert re.match (pattern, s_n) is not None

         # Benchmark it & report time, normalized to 'n'
         timing = %timeit -q -o re.match (pattern, s_n)
         t_avg = sum (timing.all_runs) / len (timing.all_runs) / timing.loops / n * 1e9
         print ("Average time per match per `n`: {:.1f} ns".format (t_avg))
```

```
Matching input 'aaa' against pattern '^a{3}$'...

Average time per match per `n`: 465.3 ns
```

Before moving on, be sure you understand what the above benchmark is doing. For more on the Jupyter "magic" command, %timeit, see:
http://ipython.readthedocs.io/en/stable/interactive/magics.html?highlight=magic#magic-magic
(http://ipython.readthedocs.io/en/stable/interactive/magics.html?highlight=magic#magic-magic)

Exercise 1 (ungraded) Repeat the above experiment for various values of n. To help keep track of the results, feel free to create new code cells that repeat the benchmark for different values of n. Explain what you observe. Can you conclude that matching simple regular expression patterns of the form ^a{n}$ against input strings of the form a^n does, indeed, scale linearly?

```
In [5]:  Student's answer                                                              (Top)

         # Use this code cell (and others, if you wish) to set up an experiment
         # to test whether matching simple patterns behaves at worst linearly
         # in the length of the input.

         def setup_problem (n):
             s_n = rep_str('a', n)
             p = "^a{%d}$" % n
             print ("\n[n={}] Matching pattern '{}'...".format(n, p))
             assert re.match(p, s_n) is not None
             return (p, s_n)

         N = [1000, 10000, 100000, 1000000]
         T = []
         for n in N:
             p, s_n = setup_problem (n)
             timing = %timeit -q -o re.match(p, s_n)
             T.append(sum(timing.all_runs) / len(timing.all_runs) / timing.loops / n * 1e9)
             print ("==> Average time per match per `n`: {:.1f} ns".format(T[-1]))
```

```
[n=1000] Matching pattern '^a{1000}$'...

==> Average time per match per `n`: 2.1 ns

[n=10000] Matching pattern '^a{10000}$'...

==> Average time per match per `n`: 0.8 ns

[n=100000] Matching pattern '^a{100000}$'...

==> Average time per match per `n`: 0.6 ns

[n=1000000] Matching pattern '^a{1000000}$'...

==> Average time per match per `n`: 0.6 ns
```

Student's answer   Score: 0.0 / 0.0 (Top)

Answer. To see asymptotically linear behavior, you'll need to try some fairly large values of n, e.g., a thousand, ten thousand, a hundred

thousand, and a million.

A more complex pattern

Consider a regular expression of the form:

$$(a?)^n(a^n)$$

For instance, $n = 3$, the regular expression pattern is `(a?){3}a{3}` == `a?a?a?aaa`. Start by convincing yourself that an input string of the form,

$$a^n = \underbrace{aa \cdots a}_{n \text{ occurrences}}$$

should match this pattern. Here is some code to set up an experiment to benchmark this case.

```
In [6]: def setup_inputs(n):
            """Sets up the 'complex pattern example' above."""
            s_n = rep_str('a', n)
            p_n = "^(a?){%d}(a{%d})$" % (n, n)
            print ("[n={}] Matching pattern '{}' against input '{}'...".format(n, p_n, s_n))
            assert re.match(p_n, s_n) is not None
            return (p_n, s_n)

        n = 3
        p_n, s_n = setup_inputs(n)
        timing = %timeit -q -o re.match(p_n, s_n)
        t_n = sum(timing.all_runs) / len(timing.all_runs) / timing.loops / n * 1e9
        print ("==> Time per run per `n`: {} ns".format(t_n))

        [n=3] Matching pattern '^(a?){3}(a{3})$' against input 'aaa'...

        ==> Time per run per `n`: 762.0489396681983 ns
```

Exercise 3 (ungraded) Repeat the above experiment but for different values of n, such as $n \in \{3, 6, 9, 12, 15, 18\}$. As before, feel free to use the code cell below or make new code cells to contain the code for your experiments. Summarize what you observe. How does the execution time vary with n? Can you explain this behavior?

```
In [7]:  Student's answer                                                                                    (Top)

         # Use this code cell (and others, if you wish) to set up an experiment
         # to test whether matching simple patterns behaves at worst linearly
         # in the length of the input.

         N = [3, 6, 9, 12, 15, 18]
         T = []
         for n in N:
             p_n, s_n = setup_inputs (n)
             timing = %timeit -q -o re.match (p_n, s_n)
             t_n = sum (timing.all_runs) / len (timing.all_runs) / timing.loops / n * 1e9
             print ("Time per run per `n`: {} ns".format (t_n))
             T.append (t_n)
```

```
[n=3] Matching pattern '^(a?){3}(a{3})$' against input 'aaa'...

Time per run per `n`: 665.3803299802045 ns
[n=6] Matching pattern '^(a?){6}(a{6})$' against input 'aaaaaa'...

Time per run per `n`: 1408.6322516636249 ns
[n=9] Matching pattern '^(a?){9}(a{9})$' against input 'aaaaaaaaa'...

Time per run per `n`: 5409.464725946437 ns
[n=12] Matching pattern '^(a?){12}(a{12})$' against input 'aaaaaaaaaaaa'...

Time per run per `n`: 29817.686138913916 ns
[n=15] Matching pattern '^(a?){15}(a{15})$' against input 'aaaaaaaaaaaaaaa'...

Time per run per `n`: 216116.05733374337 ns
[n=18] Matching pattern '^(a?){18}(a{18})$' against input 'aaaaaaaaaaaaaaaaaa'...
```

```
[n 18] matching pattern '(a.){18}(a[18})*' against input 'aaaaaaaaaaaaaaaaaa ...'
                    Time per run per `n`: 1394404.9611045218 ns
```

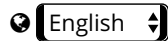Student's answer   Score: 0.0 / 0.0 (Top)

Answer. Here, you should observe something more like polynomial growth. Here are some results we collected, for instance.

n	t (ns)
3	945.8
6	1611.7
9	7040.1
12	41166.1
15	254927.4
18	1724843.9

In [8]:

© All Rights Reserved