Working with Text Data

Research Computing Summer School 2019

Ian Percel

University of Calgary, Research Computing Services

May 27, 2019

Outline

1. Accessing ARC and Example Problem
2. Text as Arrays
   - Theory
   - Practice
3. Basic `re` functions
   - Theory
   - Practice
4. Literals, Anchors, and Character Classes
   - Theory
   - Practice

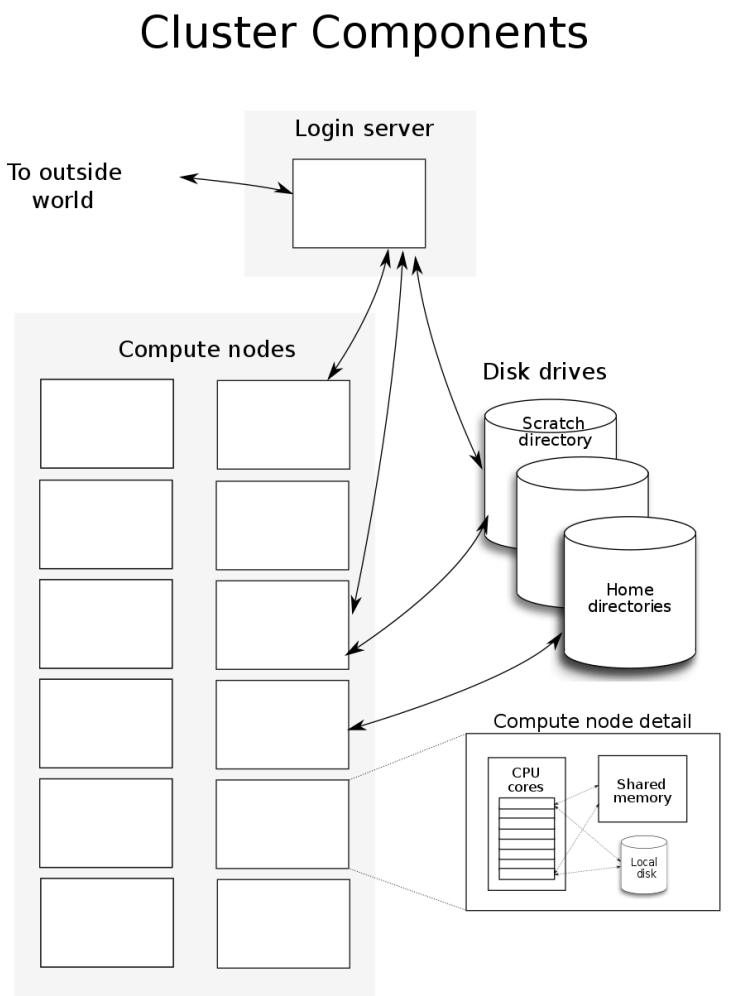
Outline

5 Mechanics of Regular Expressions: Scanning and Pattern Matching
- Theory
- Practice

6 Quantifiers
- Theory
- Practice

7 Grouping
- Theory
- Practice

Cluster Architecture

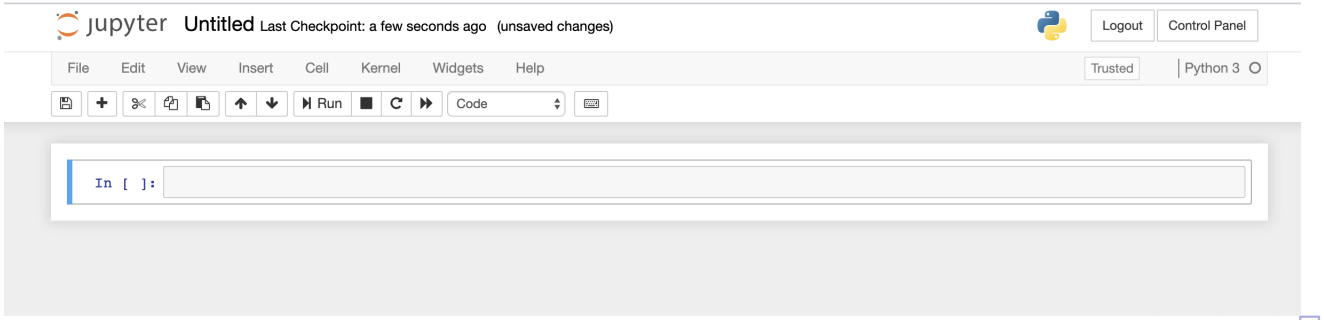


Jupyter Notebooks on ARC

- `https://jupyter.ucalgary.ca:8000/hub/login`
- Use your itusername and email password to login
- Upload any data files that you need to use with the upload button
- Create a new notebook using New > Notebook: Python 3

Jupyter Notebooks on ARC

- Rename notebook by double-clicking on the work Untitled and changing it in the provided field and clicking the rename button at the bottom right of the dialogue
- To run python code, enter it in the text box / cell and press the run button (pressing enter will just create a newline) try out 3+5
- The result will be printed below the cell
- A new cell will be automatically be created below the cell that was just run



Uploading Data to ARC

- Download data from the course abstract page
- Go to your browser tab for the Jupyter Hub page (the list of files) and click the Upload button
- Select the downloaded files (`Calgary_Business_Licences.csv` and `kv_form_calgary_buslicenses.txt`) from your computer and press Open
- The file should appear in the list of files for Jupyter Hub as `Calgary_Business_Licences.csv` and `kv_form_calgary_buslicenses.txt`

Where we are going

Data:

```
f = open("kv_form_calgary_buslicenses.txt", 'r')
x=f.readline()
f.close()
print(x)
```

```
CALL4CASH: {ADDRESS: 3908 17 AV SE, LICENCETYPES: PAYDAY LENDER, COMDISTNM: FOREST LAWN,
JOBSTATUSDESC: RENEWAL LICENSED, JOBCREATED: 2016/11/03, longitude: -113.977997740955,
latitude: 51.0381868455263, location: (51.0381868455263, -113.977997740955),
Count: 1, City Quadrants: 3, Ward Boundaries: 12, Calgary Communities: 85}
```

Code:

```
m3=re.match('^(?P<id>[a-zA-Z0-9_\s]+)\:\s*...
\{(?P<attdata>([a-zA-Z0-9_\s]+:([a-zA-Z0-9_\s\/\-\.]+|\s?\(\s?\-?\d+\.\d+\s?,\s?\-?\s?\d+\.\d+\s?\))...
,{0,1})+)\}$',x)

m4=re.findall('([a-zA-Z0-9_\s]+)\s?:...
\s?([a-zA-Z0-9_\s\/\-\.]+|\s?\(\s?\-?\d+\.\d+\s?,\s?\-?\s?\d+\.\d+\s?\)),?', ...
m3.group('attdata'))

print(m3.group('id'))
for z in m4:
    print(z[0].strip()+" is "+z[1])
```

Where we are going

Output:

```
CALL4CASH
ADDRESS is 3908 17 AV SE
LICENCETYPES is PAYDAY LENDER
COMDISTNM is FOREST LAWN
JOBSTATUSDESC is RENEWAL LICENSED
JOBCREATED is 2016/11/03
longitude is -113.977997740955
latitude is 51.0381868455263
location is (51.0381868455263, -113.977997740955)
Count is 1
City Quadrants is 3
Ward Boundaries is 12
Calgary Communities is 85
```

Storing text to a variable

- Text can be single or double quoted. This is called a string.
- Assign text to a variable with a simple statement
- Examine the contents of a text variable using print or just by calling the variable
- `type(x)` gives the data type,
- `len(x)` gives the number of letters in the string

```
x='The brown dog jumps over the lazy fox'
print(x)
The brown dog jumps over the lazy fox
```

Reading text from a file

- Encodings matter (ASCII, UTF-8)
- Line endings matter (carriage return and line feed are used as a pair in windows, line feed is used alone in linux or OS X)
- Three steps: (1) open the file, (2) `read()` or `readline()` or `readlines()` (3) close file
- For some tools we can process the whole file at once and the opening and closing is hidden `pd.read_csv(fileName)`

```
f = open("kv_form_calgary_buslicenses.txt", 'r')
x=f.readline()
f.close()
print(x)

import pandas as pd
from pandas import Series, DataFrame
df=pd.read_csv("Calgary_Business_Licences.csv")
df.head()
```

Elementary String Functions: formatting

```
In [1]: x='cat'
In [2]:x
Out[2]: 'cat'
In[3]: y="dog"
In[4]:x+" or "+y
Out[4]: 'cat or dog'
In[5]:x.upper()
Out[5]:'CAT'
In[6]:x.upper().lower()
Out[6]:'cat'
In[7]:'{2}, {1}, {0}'.format('a', 'b', 'c')
Out[7]:'c, b, a'
```

Elementary String Functions: parsing

```
In [8]: x[0]
Out [8]:'c'
In [9]: x[0:2]
Out [9]:'ca'
In [10]: x.find('a')
Out [10]: 1
In [11]: x.split('a')
Out [11]: ['c','t']
In[12]: x.replace('c','b')
Out[12]: 'bat'
```

Elementary String Functions: parsing 2

- `find` can also search for a string that is more than one character long
- `find` can accept a second argument that indicates the starting index for the search
- e.g. `x.find(':-',4)` will find the first instance of the colon hyphen after the 5th character in the string x
- `split` can also split on strings that are more than one character long
- A second parameter can specify the maximum number of splits desired

1. Read a line from `kv_form_calgary_buslicenses.txt` and assign it to a variable
2. Examine the new variable's type, length, and print its value
3. Find the index in the string of the first instance of a colon, using this value find the second instance of a colon
4. Find the first instance of a comma after the second colon
5. Use slices of strings (e.g. `x[0:9]`) with the information obtained above to extract the company name and then the first attribute and value pair
6. What characters are left over in the strings? Is there another way to split out this data?
7. Write a for loop the iteratively parses out key-value pairs

Generalizing string functions

- The string functions only work on "literals" and indexes
- How do we describe patterns instead of explicit characters?
- `re` is a module that provides Regular Expression functionality to Python
- The basic functions in `re` are conceptually very similar to those in the String class in Python but they support more complex pattern matching
- Essential operations include: Searching for matches (`re.search`, `re.match`, `re.findall`), splitting text based on some delimiting pattern (`re.split`), and replacing text (`re.sub`)

String function and re function usage looks different

- The string functions all look like `x.someFunction(parameters)` where `x` is the string you want to do things to
- The `re` functions all look like `re.someFunction(parameters, x)`
- These are both common forms in Python generally

match and search usage

- match will start from the start of the string you are searching through
- search will search everywhere in the string (more on this later)

```
import re
f = open("kv_form_calgary_buslicenses.txt", 'r')
x=f.readline()
f.close()

m1=re.search(':',x)
m2=re.match(':', x)
m3=re.match('CALL4CASH', x)
print(m1.group(0))
print(m2)
print(m3.group(0))

:
None
CALL4CASH
```

Indexes v Match Objects

- Python string functions return indexes as a result of search
- `re` search and match functions return Match objects (a sort of description of what was found)
- Match Object functions: group(groupNumber), start(groupNumber), end(groupNumber)
- If no match is found, `None` is returned
- `m.group(0)` is the whole matching string was found
- `m.start(0)` is the starting index of the matching string that was found
- `m.end(0)` is the ending index of the matching string that was found

findall and finditer usage

- findall returns a list of all matching strings
- finditer returns an iterator of all match objects corresponding to matches
- finditer is more flexible once you are comfortable with how to use match objects as it also tells you where the match is by index

```
m1=re.findall(':',x)
print(m1)
[':', ':', ':', ':', ':', ':', ':', ':', ':', ':', ':', ':', ':']
m2=re.finditer(':',x)
for z in m2:
    print(z.start(0))
9
19
48
74
102
...
```

split usage

- split is almost identical to the string function in its usage
- The main difference is the way it is called

```
m4=re.split(':',x)
print(m4)
['CALL4CASH', ' {ADDRESS', ' 3908 17 AV SE, LICENCETYPES',
' PAYDAY LENDER, COMDISTNM', ' FOREST LAWN, JOBSTATUSDESC',
' RENEWAL LICENSED, JOBCREATED', ' 2016/11/03, longitude',
' -113.977997740955, latitude', ' 51.0381868455263, location',
' (51.0381868455263, -113.977997740955), Count',
' 1, City Quadrants', ' 3, Ward Boundaries',
' 12, Calgary Communities', ' 85}\n']
```

sub usage

- sub is almost identical to replace except for the way it is called

```
m4=re.sub(':','-',x)
print(m4)
CALL4CASH- {ADDRESS- 3908 17 AV SE, LICENCETYPES- PAYDAY LENDER,
COMDISTNM- FOREST LAWN, JOBSTATUSDESC- RENEWAL LICENSED,
JOBCREATED- 2016/11/03, longitude- -113.977997740955,
latitude- 51.0381868455263, location- (51.0381868455263, -113.97
Count- 1, City Quadrants- 3, Ward Boundaries- 12, Calgary Commun
```

1. Read a line from `kv_form_calgary_buslicenses.txt` and assign it to a variable
2. Replace all curly braces '{'and '}'with pipes '|'
3. Find all pipes in one command
4. Split over pipes, then split the attribute-value pairs part of the resulting list over commas
5. Iterative over comma divided segments and parse out the two parts (attribute and value). What went wrong?

What do we mean by Pattern Matching?

- We often want to describe strings that have identifiable patterns without being explicit about which exact strings we are looking to find
- For example, one standard date format is YYYY/MM/DD
- How do we find all dates in a document that have this structure without knowing the dates in advance?
- With standard string functions, there is no easy way to do this

Literals

Literals are explicit strings

```
ABCD...XYZ
abcd...xyz
0123456789
:  ,  ;  / ! @ # %  &
```

Some characters need to be escaped because they have special meanings

```
\\  \?  \-  \+  \*  \.  \$  \|  \^
\(  \)  \[  \]  \{  \}
```

Anchors

- It is important to be able to specify where in the string a given symbol should match
- ^ indicates the location of the start of the string
- $ indicates the location of the end of the string

```
import re
m=re.match('^CALL','CALL4CASH: {ADDRESS: 3908 17 AV SE}')
m.group(0)
CALL
m=re.match('17 AV SE$','CALL4CASH: {ADDRESS: 3908 17 AV SE}')
m
None
```

Character Classes

- Character classes are a way of escaping the rigidity of literals
- List the characters that you want to include inside square brackets
- Ordered ranges can be indicated with a hyphen
- ^ will match anything except the listed characters
- Special characters don't need to be escaped

```
[abc] matches a or b or c
[a-f] matches a or b or c or d or e or f
[A-Za-z] matches any roman letter, capital or lower case
[0-4] matches 0 or 1 or 2 or 3 or 4
[3-7 ] matches 3 or 4 or 5 or 6 or 7 or a space
[^12] matches any character except the numbers 1 or 2
[.,/-] matches a period, a comma, a slash, or a hyphen
```

Character Classes 2

- How do we handle invisible or unprintable characters?
- There are special escape sequences for representing these
- There are also escape sequences for common groups of characters

```
\d is the same as [0-9] plus some other harder to print digits
\w matches word characters (roughly [a-zA-Z0-9_])
\s matches any white space character
\t matches a tab
\n matches a new line
\r matches a carriage return
```

Date Matching

- We are now in a position to solve our date finding problem
- YYYY is four digits
- MM is two digits
- DD is two digits
- if we want to get clever we can us custom character classes to exclude impossible dates

```
import re
x='Something happened on 2019/05/16 but not on 15/05/2019'
m=re.findall('\d\d\d\d/\d\d/\d\d', x)
m
['2019/05/16']
x='Something happened on 2019/05/16 but not on 1513/05/2019'
m=re.findall('\d\d\d\d/\d\d/\d\d', x)
m
['2019/05/16', '1513/05/20']
```

1. Rewrite the date format to match DD/MM/YYYY
2. Rewrite the date format to match only years before 2000 and after 1000, test this out on some sample data to check it
3. Examine a few rows of data from `kv_form_calgary_buslicenses.txt` and make a guess about the contents of a valid attribute name
4. Define a character class that includes all of the characters that appear in attribute names (but no others) and put a plus sign after the character class (this makes it possible to match one or more of these symbols consecutively)
5. Find all matches for this and examine them. Is it any good? What might have gone wrong
6. Can you extend this by a character to capture attribute names only?

Scanning

- How does a RegEx engine work?
- It reads a character at a time from the string (starting from the left-most character and proceeding right) and compares it to the pattern.
- This process is called scanning.
- If the scanner reaches a character that cannot possibly fit the pattern, it rejects the match and starts over from later in the string.
- There may be multiple ways of matching the pattern.

Scanning Example 1

```
x='12-ab-bc/123'
r='[0-9][0-9]-[a-z][a-z]'
1 is in the first  [0-9]
2 is in the second [0-9]
- matches -
a is in the first [a-z]
b is in second [a-z]
match = '12-ab'
```

Scanning Example 2

```
x='12-ab-bc/123'
r='[a-z][a-z]-[a-z][a-z]/'
1 is not in [a-z]
2 is not in [a-z]
- is not in [a-z]
a is in the first [a-z]
b is in second [a-z]
- matches -
b is in the third [a-z]
c is in the fourth [a-z]
/ matches /
match = 'ab-bc/'
```

Scanning Example 3

```
x='12-ab-bc/123'
r='[0-9a-z][0-9a-z/-][0-9a-z/-][0-9a-z/-][0-9a-z/-][0-9a-z/-]'
1,2,a,b, and b all match the first  [0-9a-z]
every subsequent character matches [0-9a-z/-]
the first match that will be found is
match = '12-ab-'
However, the following are also valid matches
(and would be reported by re.findall(r,x))
['12-ab-','2-ab-b', 'ab-bc/','b-bc/1','bc/123']
```

We see that there are multiple routes through the pattern matching different characters in each element of the pattern.

Scanning Puzzles

```
x='123.429 on 23/07/2016'
r1='^\d\d\d\d\d\d'
r2='^[\d.][\d.][\d.][\d.][\d.][\d.][\d.] \d\d/\d\d/\d\d\d\d'
r3='\d\d/\d\d/\d\d\d\d'
r4='\d\d [a-z][a-z] \d\d'
```

- Examine 4 different regular expressions `r1` to `r4` and the test string `x`. Assume that we are using the `re.search` function
- For `r1` describe which characters symbols would match which characters and when it would fail.
- What changes would fix this? check this by changing the expression and re-running it
- Repeat this analysis for `r2, r3, and r4`.

Solutions

```
x='123.429 on 23/07/2016'
r1='^\d\d\d\d\d\d'
```

- `r1` would start from the string beginning only and read in the digits 123 successfully before reaching a period and failing. It would be unable to restart from later in the string because of the `^` It can be fixed by adding a `_` after the third digit.

Solutions

```
x='123.429 on 23/07/2016'
r2='^[\d.][\d.][\d.][\d.][\d.][\d.][\d.] \d\d/\d\d/\d\d\d\d'
```

- `r2` would start from the string beginning and match '123.429 ' as each of the character classes would match either a digit or a decimal point and the space would match. when the 'o' was reached, it would fail to match a digit and the whole pattern would be rejected.
- Since there is no leading ^ the search would restart from 2 and 23.429 would match before a space would be unable to match the last digit pattern. This would happen again for each subsequent element of the decimal string. The match would fail immediately on both of the spaces and the 'on'. It would attempt again to match on 23 before failing on the slash. It would attempt on 3 before failing on the slash. It would fail on the first slash. It would attempt on 07 before failing on the slash. It would attempt on 7 before failing on the second slash. It would attempt on 2016 before running out of characters.
- It can be fixed by adding an 'on' and another space in the middle.

Solutions

```
x='123.429 on 23/07/2016'
r3='\d\d/\d\d/\d\d\d\d'
r4='\d\d [a-z][a-z] \d\d'
```

- `r3` would attempt several matches at the beginning of the string (failing on the slash pattern) before reaching the date string which would succeed in matching.
- `r4` would attempt several matches at the beginning of the string (failing on the space pattern) before reaching '29 on 23' which would match the pattern.

Quantifiers

- In general, we will know something about *which* characters might be present and we will know something about *how many* characters might be present
- For example, words consist of 1 or more lower case letters, upper case letters, and hyphens
- However, words can have many lengths. How would we capture the whole next word regardless of what it is?
- So far we have only used the most basic *quantification*, which is "one character of this type (and exactly one) must be present to match"
- How do we describe more complex allowances for variable length?

Quantifiers

? means 0 or 1 (preferably more)
+ means 1 or more (preferably more)
* means 0 or more (preferably more)
{n} means match exactly n repetitions
{n,m} means match anywhere between n and m repetitions
(but match as many as you can)
{n,} makes m infinite
{,m} makes n=0
adding ? to any of these quantifiers will cause them to match as few repetitions as possible
(while remaining faithful to the underlying rule)

Quantifier Examples

'a/?b' matches 'ab' and 'a/b'
'a/+b' matches 'a/b' and 'a//b' and 'a///b' and so on
'a/*b' matches 'ab' and 'a/b' and 'a//b' and so on
'a/{3}b' only matches 'a///b'
'a/{1,2}b' only matches 'a/b' and 'a//b'
'a/{3,}b' will match 'a///b' or any greater number of /
all of these will try to match as many slashes as possible
before proceeding to attempt to match 'b'

Greedy and Non-Greedy Quantifiers 1

```
x='a/b'
r='a[ab/]+'
a matches a
/ matches [ab/]+
b matches [ab/]+
match='a/b

x='a/b'
r='a[ab/]+?'
a matches a
/ matches [ab/]+
match='a/'
```

Greedy and Non-Greedy Quantifiers 2

```
x='a/b'
r='a[ab/]?[b]?'
a matches a
/ matches [ab/]?
b matches [b]?
match='a/b

x='a/b'
r='a[ab/]??[b]??'
a matches a
match='a'
```

Quantification Puzzles

1. Write a single regular expression that can match YYYY/MM/DD or DD/MM/YYYY using only the / and . character classes but with quantifiers
2. Write a regular expression that can match any length of positive decimal number (e.g. 12341212.123124124242425)
3. Write a regular expression that can match any single word
4. Test this against a sentence that you write using `re.findall()`
5. Extend the regular expression to only match words surrounded by white space
6. Test this against a sentence that you write using `re.findall()`

Grouping

- One of the most useful (and subtle) concepts in regular expressions is that of groups
- Groups allow us to identify a segment of the match distinct from the rest
- This can be useful for parsing, quantifying complex expressions, allowing explicit alternatives, and referencing other parts of the match for later repetition
- We will tackle each of these strategies in turn.

Basic Grouping Syntax

- A group is formed by enclosing a part of the pattern in parentheses
- Once matched, this pattern segment can by later accessed from the match object

```
x='Address:1234 12 Av NW'
r='[a-zA-Z\s]+:[0-9a-zA-Z\s.-/_,;]+'
m=re.match(r,x)
m.group(0)
'Address:1234 12 Av NW'
len(m.groups())
0
r2='([a-zA-Z\s]+):([0-9a-zA-Z\s.-/_,;]+)'
m2=re.match(r2,x)
len(m2.groups())
2
```

Grouping 1: Parsing

- The subgroups that have been saved to the match object can be used to parse data from text without slicing on indexes
- It suffices to know the order in which the groups will be created and how to interpret that
- `m.group(k)` where m is the match object and k is the subgroup number

```
x='Address:1234 12 Av NW'
r2='([a-zA-Z\s]+):([0-9a-zA-Z\s.-/_,;]+)'
m2=re.match(r2,x)
m2.group(0)
'Address:1234 12 Av NW'
m2.group(1)
'Address'
m2.group(2)
'1234 12 Av NW'
```

Grouping 2: Quantifying Groups

- Quantifiers can be applied directly to groups
- In this case, only repetitions of the entire group will be accepted

```
r='[NW]{1,3}'
x='NNN'
re.match(r,x).group(0)
'NNN'
r2='(NW){1,3}'
re.match(r2,x)
None
x2='NWNW'
re.match(r2,x2).group(0)
'NWNW'
```

Grouping 3: Alternation

- The pipe character (shift+\ ) can be used in a group to indicate different possibilities
- `(NW|SW)` will match NW or SW
- Quantification allows mixing of matches

```
r='(NW|SW|NE|SE)+'
x='NW'
x2='NWSE'
re.match(r,x).group(0)
'NW'
re.match(r,x2).group(0)
'NWSE'
re.match(r,x2).group(1)
'SE'
```

Grouping 4: Named Groups

- Groups can be assigned names using the for `(?P<name>SomePattern)`
- This will apply to the way that the group is recorded in the `m.group()` function and how it is referenced in the rest of the pattern
- If no name is given, a group can always be referenced by its capture order

```
r='(?P<stNum>[0-9]{4})\s(?P<stName>[0-9]{1,3})\s(?P<stType>St|Av)'
x='1234 42 St'
m=re.match(r,x)
m.groups()
('1234', '42', 'St')
m.group('stNum')
'1234'
m.group(2)
'42'
m.group('stType')
'St'
```

Grouping 4: Named Groups

- (?P=name) or $\backslash n$ (for the nth group) can be used to reference a group later in the pattern
- The example below demonstrates searching for records where two parts of a record agree on some field
- The first example matches, while the second example fails to match

```
r='Add1:(?P<stNum>[0-9]{4})[\s0-9A-Za-z]*; Add2:(?P=stNum)[0-9a-zA-Z\s]*'
x='Add1:1234 40 ST Other Words; Add2:1234 34 ST'
x2='Add1:1234 40 ST Still Other Words; Add2:1235 34 ST'
m=re.match(r,x)
m.group(0)
'Add1:1234 40 ST Other Words; Add2:1234 34 ST'
m.group(1)
'1234'
m2=re.match(r,x2)
m2
None
```

Problems 1

1. Define an expression for matching an address number followed by a street (numbered form e.g. 42nd). Name some of the character groups. You may want to start by testing you code against a string that only includes a simple sample address like '1234 14'. If you need a hint, start from the first Named Groups example and attempt to generalize it by modifying the quantifiers.
2. Test your expression against the data in the business license file using the test code below
3. Extend it to be followed by an appropriate string for matching avenue or street as it appears in the sample data and test it again

Problems 2

1. Extend it to allow a street direction
2. Use the named capture groups to extract exactly the Street Number and Direction only and print each of the those for every record
3. Search the file for any records where the street name and street number are the same. How many did you find?

Test Code

```
import re
f = open("kv_form_calgary_buslicenses.txt", 'r')
data=f.readlines()
f.close()
r=
matches=[]
for x in data:
    m=re.search(r,x)
    matches.append(m)
#new cell: look though the matches by changing the index k below
#and running that second cell repeatedly to view different results
#note that the argument to group can be changed to your group names
#to access the parsed data
k=0
print(matches[k].group(0))
```

Solutions

```
r='(?P<stNum>[0-9]{1,4}) (?P<stName>[0-9]{1,3}) (?P<stType>ST|AV)'
will match the number, street name, and street type
r2=r+' (?P<Dir>NW|SW|NE|SE)'
will match the number, street name, street type, and direction
for record in matches:
    if record:
        print(record.group('stNum')+' '+record.group('Dir'))
    else:
        print('no address found')
will produce the formatted output
r3='(\s|:)(?P<stNum>[0-9]{1,4}) (?P=stNum)'
r4=r3+' (?P<stType>ST|AV) (?P<Dir>NW|SW|NE|SE)'
can be used to find record with matching street name and number
the result is ' 50 50 AV SW'
```