# EDA & Working with Text (Reading: 5.3-5.7, 8)

**Learning goals:**

- Understand the goals of EDA: Data Types, Granularity, Scope, Temporality, and Faithfulness
- Learn and practice using regular expressions

**UC Berkeley Data 100 Summer 2019**
**Sam Lau**

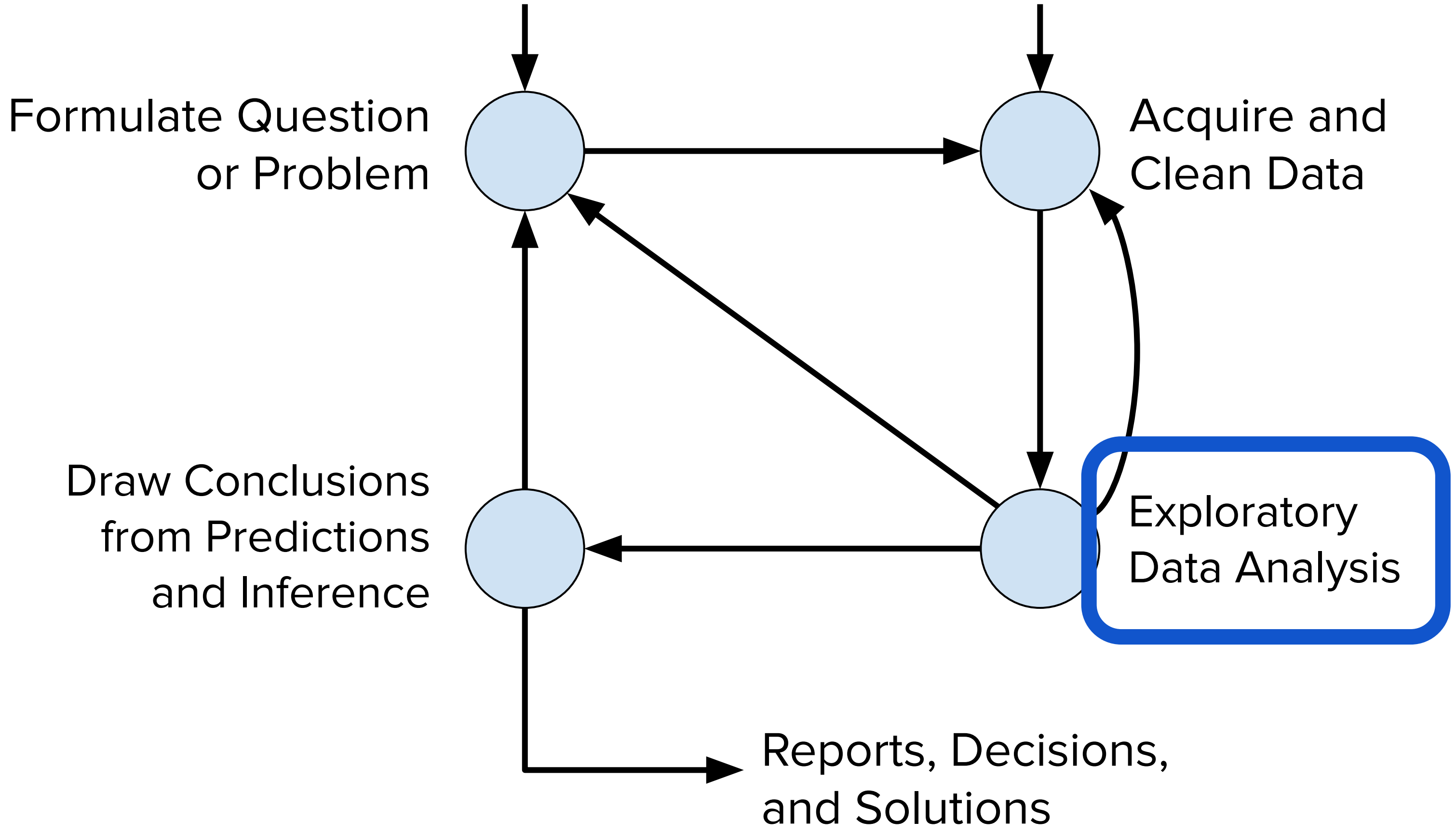
(Slides adapted from John DeNero, Josh Hug, Joey Gonzalez, and Fernando Perez)

## Announcements

- HW2 out today
  - Will be “officially” due Friday but we will take submissions without penalty until Tuesday (July 9)
- HW3 out Friday
  - Due the following Friday (July 12)
- **For deadline extension requests, contact your TA and cc Sam in the email.**
- Congrats on finishing Project 1!
  - Will use Project 1 as case study today

## Exploratory Data Analysis (EDA)

## Remember the Data Science Lifecycle?


Formulate Question or Problem
Acquire and Clean Data
Draw Conclusions from Predictions and Inference
Exploratory Data Analysis
Reports, Decisions, and Solutions

## How Do We Understand the Data?

- Iterative process of visualizing and transforming data to:
  - Understand patterns
  - Identify issues
  - Inform subsequent analysis
  - Discover potential hypotheses (careful with this one!)
- Have an open-ended attitude
  - Look for what you believe is there and what you believe is not there (Tukey, 1965)

## Goals of EDA

We want to determine the following:

| | |
|---|---|
| **Data Types** | What kinds of data do we have? |
| **Granularity** | How fine/coarse is each datum? |
| **Scope** | How (in)complete are the data? |
| **Temporality** | How are the data situated in time? |
| **Faithfulness** | How accurately do the data describe the world? |

## Data Types

## Data Types: What kinds of data do we have?

- In Data 100:
  - Nominal Data: categories without natural ordering
  - Ordinal Data: categories with natural ordering
  - Numerical Data: amounts or quantities
- Note that these are **statistical data types**
  - Since these are more common, just say **data types**
  - These are distinct from **computational data types...**

## Statistical vs. Computational Data Types

- Computational data types: int, float, string, boolean, etc.
- Each column in pandas has a computational data type
  - But does not imply a statistical data type
- Example: Sex sometimes coded as 0 for male, 1 for female, 2 for missing
  - Will be number in pandas, but should not be treated as numeric data!

# Condensed Dataset from Project 1:

**Computational**

```
>>> df.dtypes
```

```
id              int64
lat           float64
phone          object
dtype: object
```

(“object” in pandas usually means string)

| id | lat | phone |
|---:|---:|---:|
| 66 | 37.76 | +14152427970 |
| 1085 | 37.79 | +14152410256 |
| 1103 | 37.76 | +14152410256 |
| 1116 | 37.80 | +14159214049 |
| 1122 | 37.80 | +14156736781 |
| 1127 | 37.79 | +14159551940 |
| 1265 | 37.76 | +14158540888 |

**Statistical**

What do you think?

## Condensed Dataset from Project 1:

**Computational**

```
>>> df.dtypes
```

```
id            int64
lat         float64
phone        object
dtype: object
```

("object" in pandas usually means string)

| id | lat | phone |
|---:|---:|---:|
| 66 | 37.76 | +14152427970 |
| 1085 | 37.79 | +14152410256 |
| 1103 | 37.76 | +14152410256 |
| 1116 | 37.80 | +14159214049 |
| 1122 | 37.80 | +14156736781 |
| 1127 | 37.79 | +14159551940 |
| 1265 | 37.76 | +14158540888 |

**Statistical**

id: nominal

lat: numeric

phone: nominal

## Granularity

## Granularity: What does each record represent?

Do all records capture granularity at the same level?

- Some data will include summaries as records

If the data are coarse, how were they aggregated?

- Sampling, averaging, etc.

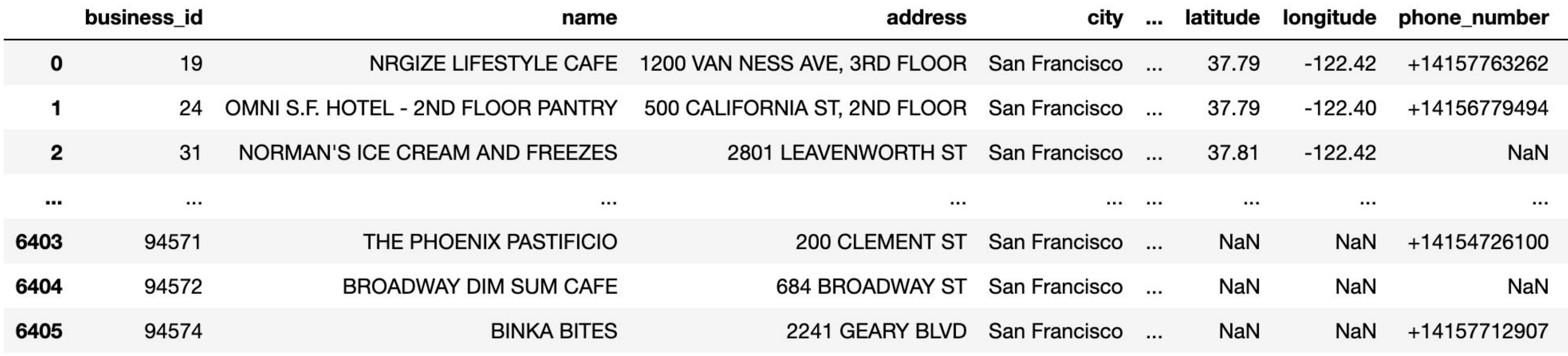
What kinds of aggregation is possible?

- From individuals to categories
- From individual events to totals across time or space

How might we adjust the granularity?

# Project 1 Question 2b

- What does each record represent (e.g., a business, a restaurant, a location, etc.)?
- What is the primary key?
- What would you find by grouping by the following columns: \`business_id\`, \`name\`, \`address\` each individually?



| | business_id | name | address | city | ... | latitude | longitude | phone_number |
|---|---|---|---|---|---|---|---|---|
| **0** | 19 | NRGIZE LIFESTYLE CAFE | 1200 VAN NESS AVE, 3RD FLOOR | San Francisco | ... | 37.79 | -122.42 | +14157763262 |
| **1** | 24 | OMNI S.F. HOTEL - 2ND FLOOR PANTRY | 500 CALIFORNIA ST, 2ND FLOOR | San Francisco | ... | 37.79 | -122.40 | +14156779494 |
| **2** | 31 | NORMAN'S ICE CREAM AND FREEZES | 2801 LEAVENWORTH ST | San Francisco | ... | 37.81 | -122.42 | NaN |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... |
| **6403** | 94571 | THE PHOENIX PASTIFICIO | 200 CLEMENT ST | San Francisco | ... | NaN | NaN | +14154726100 |
| **6404** | 94572 | BROADWAY DIM SUM CAFE | 684 BROADWAY ST | San Francisco | ... | NaN | NaN | NaN |
| **6405** | 94574 | BINKA BITES | 2241 GEARY BLVD | San Francisco | ... | NaN | NaN | +14157712907 |

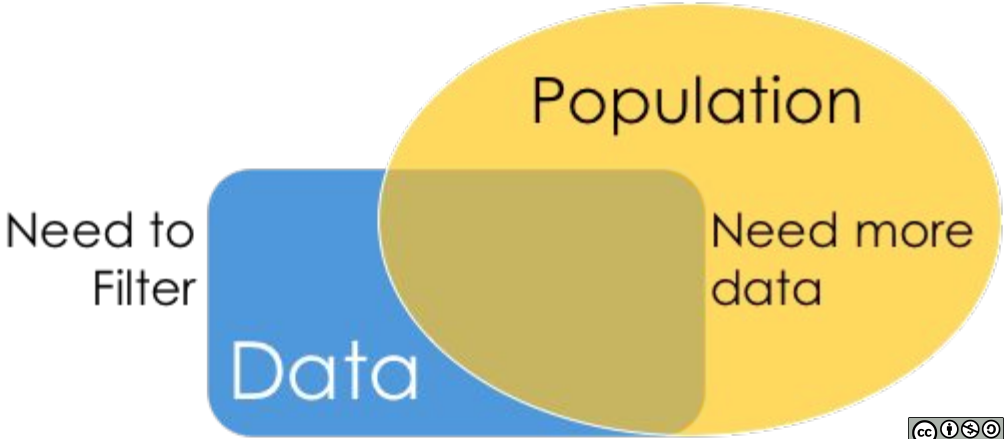
## Scope

## Scope: What do the data describe?

Do my data cover my area of interest?

Example: I am interested in studying crime in California, but I only have Berkeley + Seattle crime data.

In general, can only make statistical inferences about **probability samples** from **population of interest**!

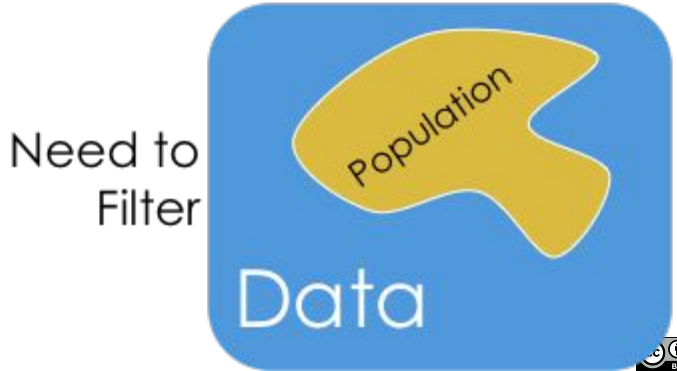


## Scope: What do the data describe?

Do my data cover more than my area of interest?

> Example: I am interested in studying Berkeley, but I have data for all of CA.

A sample over a large scope may have poor coverage for a narrow scope.



## Scope: What do the data describe?

What about the businesses dataset in project 1?

- Is it a sample? Or census?
- What time frame?

## Temporality

## Temporality: When were the data collected?

What is the meaning of each date/time field?

- When the described event happened?
- When the data were collected or entered into the system?
- When the data were copied?

## Temporality: When were the data collected?

Time and time formatting depend on location!

- Time zones
- Daylight savings (which transitions on different days in different regions)
- Regional formatting (06/07 vs 07/06)

## Temporality: When were the data collected?

Are there strange zero/null values?

- In Unix / POSIX systems, time is measured from January 1st 1970 Coordinated Universal Time (UTC): 0 degrees latitude w/o daylight savings
- Be wary if you see a lot of 1969 or 1970 timestamps

## Faithfulness

## Faithfulness: Should I trust the data?

Do the data contain unrealistic or incorrect values?

- Dates in the future for events in the past
- Locations that don’t exist
- Negative counts
- Misspellings of names
- Incorrectly typed numbers that are out of scale

## Faithfulness: Should I trust the data?

Do the data violate obvious dependencies?

- E.g., age and birthday don’t match

Were the data entered by hand?

- E.g., spelling errors, fields shifted

Did the data entry form provide default values?

Are there signs of deliberate data falsification?

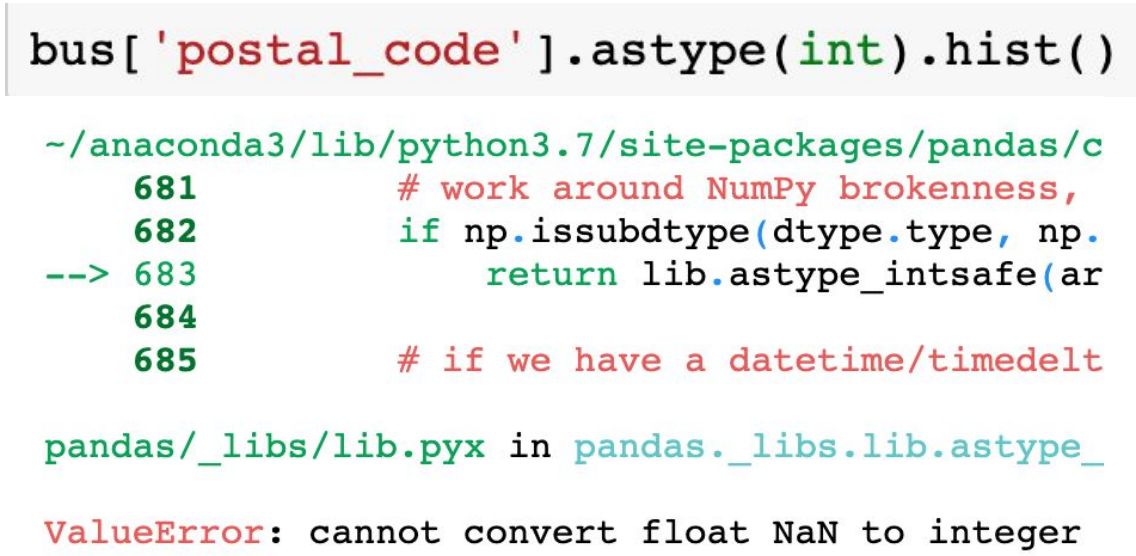
- Repeated names, fake email addresses, repeated use of uncommon names.

## EDA and Data Cleaning are an Iterative Cycle

# Project 1 Zip Codes

Let's plot the zip codes.

Missing values! Let's just drop for now and replot.

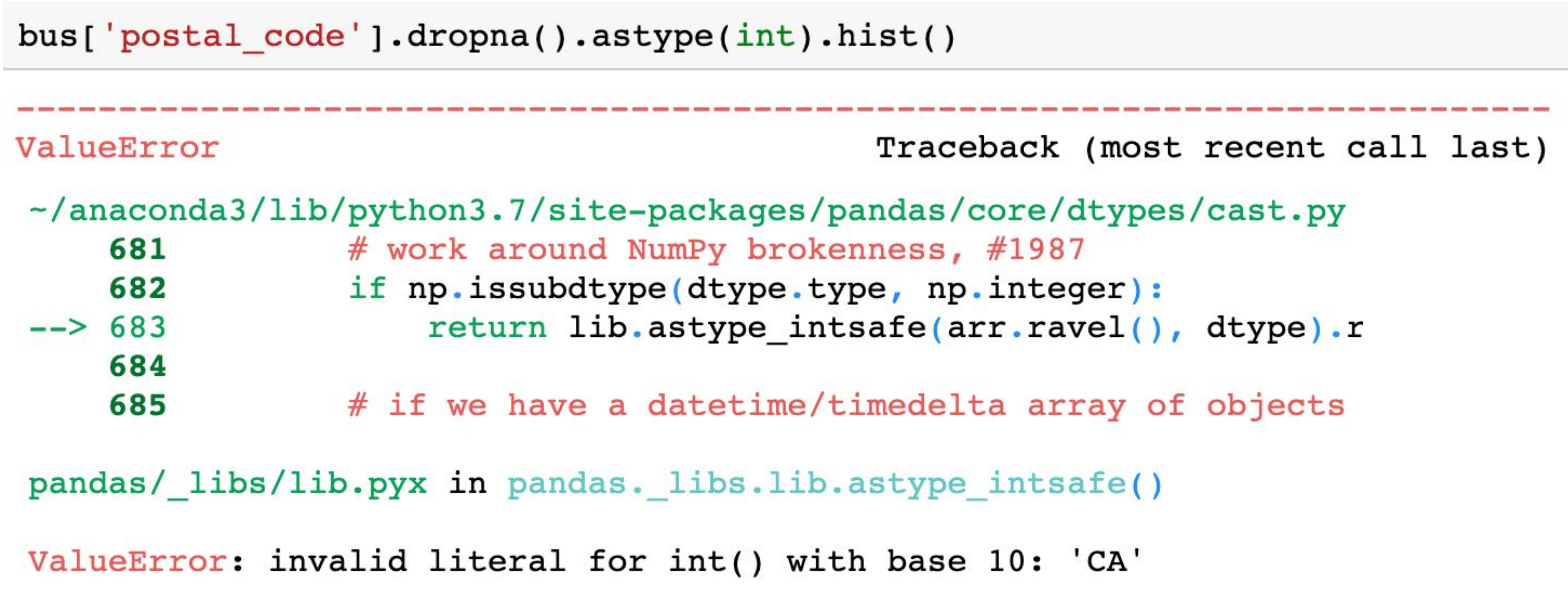


```
bus['postal_code'].astype(int).hist()
```

```
~/anaconda3/lib/python3.7/site-packages/pandas/c
    681             # work around NumPy brokenness,
    682             if np.issubdtype(dtype.type, np.
--> 683                 return lib.astype_intsafe(ar
    684 
    685             # if we have a datetime/timedelt

pandas/_libs/lib.pyx in pandas._libs.lib.astype_

ValueError: cannot convert float NaN to integer
```

## Project 1 Zip Codes

Weird zip codes! We'll only keep zip codes in SF.



```
bus['postal_code'].dropna().astype(int).hist()
```

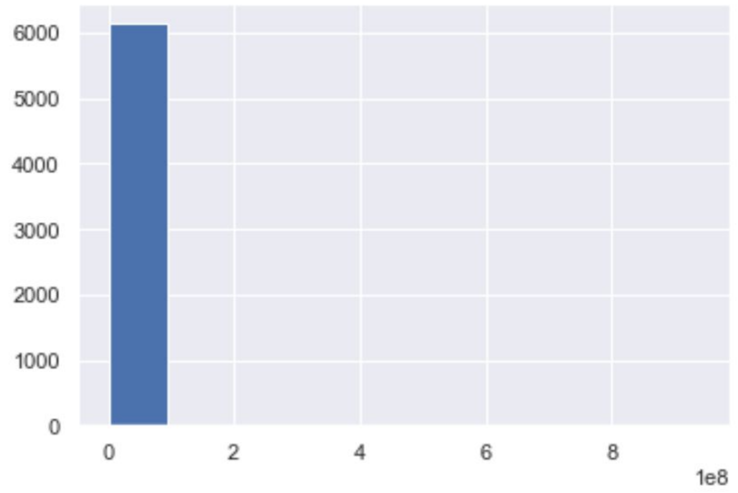
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

~/anaconda3/lib/python3.7/site-packages/pandas/core/dtypes/cast.py
    681         # work around NumPy brokenness, #1987
    682         if np.issubdtype(dtype.type, np.integer):
--> 683             return lib.astype_intsafe(arr.ravel(), dtype).r
    684 
    685         # if we have a datetime/timedelta array of objects

pandas/_libs/lib.pyx in pandas._libs.lib.astype_intsafe()

ValueError: invalid literal for int() with base 10: 'CA'
```

## Project 1 Zip Codes

Was this what we expected for a histogram of zip codes?

```
(bus.loc[bus['postal_code_5'].isin(all_sf_zip_codes), 'postal_code']
 .dropna()
 .astype(int)
 .hist()
)
```

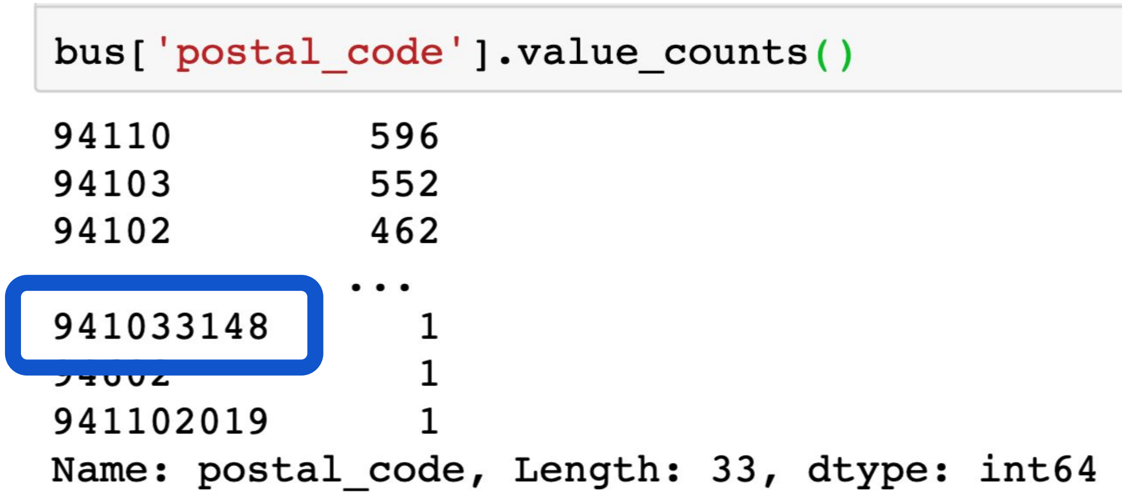
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a2b60f278>
```



## Project 1 Zip Codes

Looks like there are malformed zip codes!

We'll clean those to be five digits.



```
bus['postal_code'].value_counts()
```

```
94110        596
94103        552
94102        462
                ...
941033148      1
94602          1
941102019      1
Name: postal_code, Length: 33, dtype: int64
```

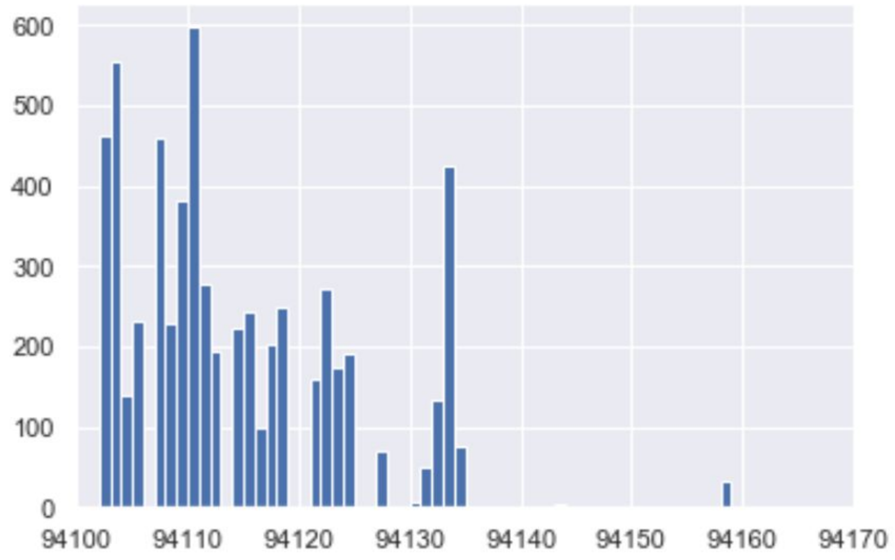
## Project 1 Zip Codes

Now our zip codes look reasonable.

Now we turn back to missing zip codes, etc.

```
(bus
 .assign(postal_code_5=bus['postal_code'].str[:5])
 .loc[bus['postal_code_5'].isin(all_sf_zip_codes), 'postal_code_5']
 .dropna()
 .astype(int)
 .hist(bins=np.arange(94100, 94170))
)
plt.xlim(94100, 94170)
```

```
(94100, 94170)
```



**Break!**

**Fill out Attendance:**

**http://bit.ly/at-d100**

## Text Fields

## Text Fields and Data Cleaning / EDA

Extract quantitative values from text: dates, times, positions, etc.

Determine if missing values are denoted with a special string: try value_counts

```
169.237.46.168 - - [26/Jan/2014:10:47:58
-0800] "GET /stat141/Winter04/ HTTP/1.1" 200
2585 "http://anson.ucdavis.edu/courses/"
```

**(demo)**

## Regular Expressions

## Regular Expressions

A *formal language* is a set of strings, typically described implicitly.

A *regular language* is a language that can be described by a *regular expression*.

- What’s “regular” about it? Repeated and optional parts of a string, but no nested structures such as checking for balanced parentheses.
- In practice, many regular expression implementations can recognize more than just regular languages, including nested structures.

## Regular Expressions

Example: `[0-9]{3}-[0-9]{2}-[0-9]{4}`

3 of any digit, then a dash, then 2 of any digit, then a dash, then 4 of any digit.

```
text = "My social security number is 123-45-6789.";
pattern = "[0-9]{3}-[0-9]{2}-[0-9]{4}"
re.findall(pattern, text)
```

## Regular Expression Syntax

The four basic operations for regular expressions.

| operation | order | example | matches | does not match |
|---|---|---|---|---|
| concatenation | 3 | AABAAB | AABAAB | any other string |
| or | 4 | AA\|BAAB | AA BAAB | any other string |
| closure (zero or more) | 2 | AB*A | AA ABBBBBBA | AB ABABA |
| parentheses | 1 | A(A\|B)AAB | AAAAB ABAAB | any other string |
|  |  | (AB)*A | A ABABABABA | AA ABBA |

## Regular Expression Syntax

AB*: A then zero or more copies of B: A, AB, ABB, ABBB

(AB)*: Zero or more copies of AB: ABABABAB, ABAB, AB

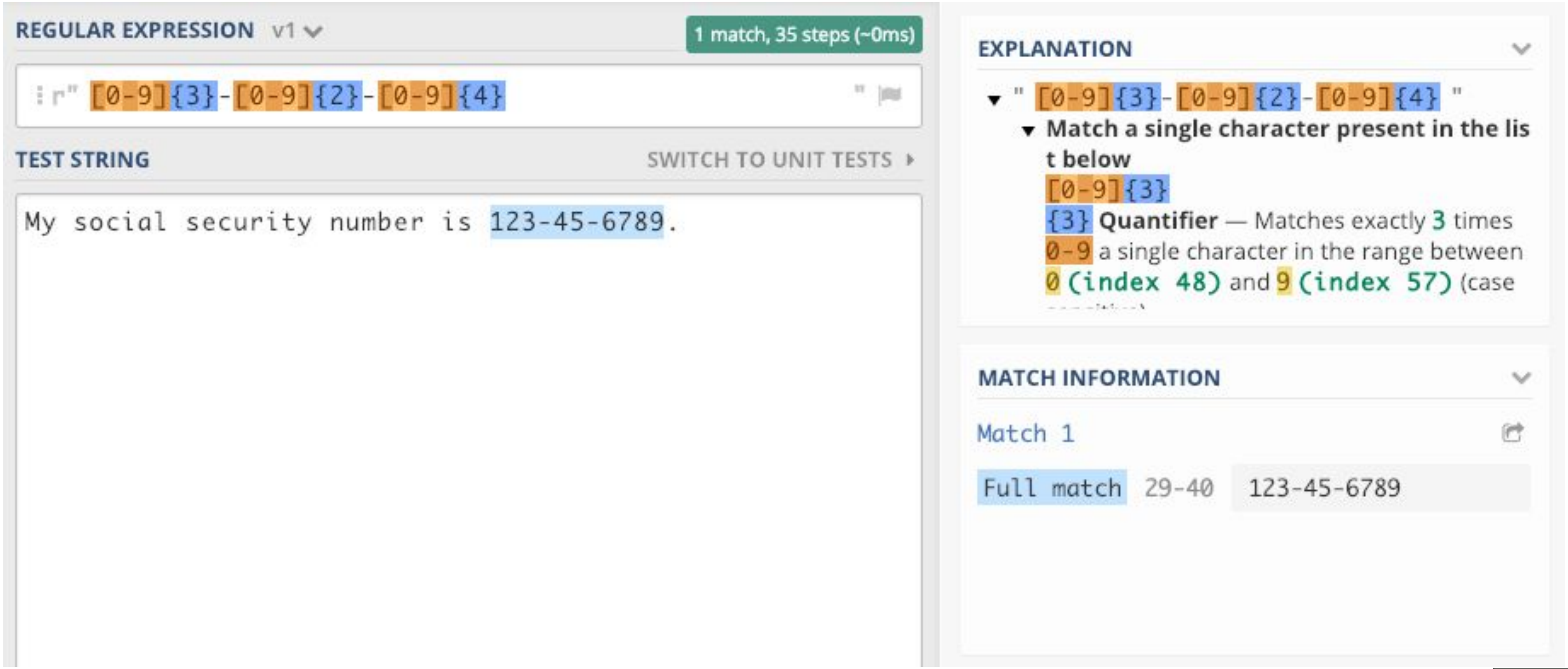
| operation | order | example | matches | does not match |
|---|---|---|---|---|
| concatenation | 3 | `AABAAB` | `AABAAB` | `any other string` |
| or | 4 | `AA\|BAAB` | `AA` `BAAB` | `any other string` |
| closure (zero or more) | 2 | `AB*A` | `AA` `ABBBBBBA` | `AB` `ABABA` |
| parentheses | 1 | `A(A\|B)AAB` | `AAAAB` `ABAAB` | `any other string` |
| | | `(AB)*A` | `A` `ABABABABA` | `AA` `ABBA` |

# Expanded Regex Syntax

| operation | example | matches | does not match |
|---|---|---|---|
| any character (except newline) | `.U.U.U.` | `CUMULUS` `JUGULUM` | `SUCCUBUS` `TUMULTUOUS` |
| character class | `[A-Za-z][a-z]*` | `word` `Capitalized` | `camelCase` `4illegal` |
| at least one | `jo+hn` | `john` `joooooohn` | `jhn` `jjohn` |
| zero or one | `joh?n` | `jon` `john` | `any other string` |
| repeated exactly {a} times | `j[aeiou]{3}hn` | `jaoehn` `jooohn` | `jhn` `jaeiouhn` |
| repeated from a to b times: {a,b} | `j[ou]{1,2}hn` | `john` `juohn` | `jhn` `jooohn` |

# regex101.com

https://regex101.com/r/fiHpRx/1



REGULAR EXPRESSION v1

1 match, 35 steps (~0ms)

```
r" [0-9]{3}-[0-9]{2}-[0-9]{4} "
```

TEST STRING

SWITCH TO UNIT TESTS

```
My social security number is 123-45-6789.
```

EXPLANATION

- " [0-9]{3}-[0-9]{2}-[0-9]{4} "
  - Match a single character present in the list below
    [0-9]{3}
    {3} Quantifier — Matches exactly 3 times
    0-9 a single character in the range between 0 (index 48) and 9 (index 57) (case

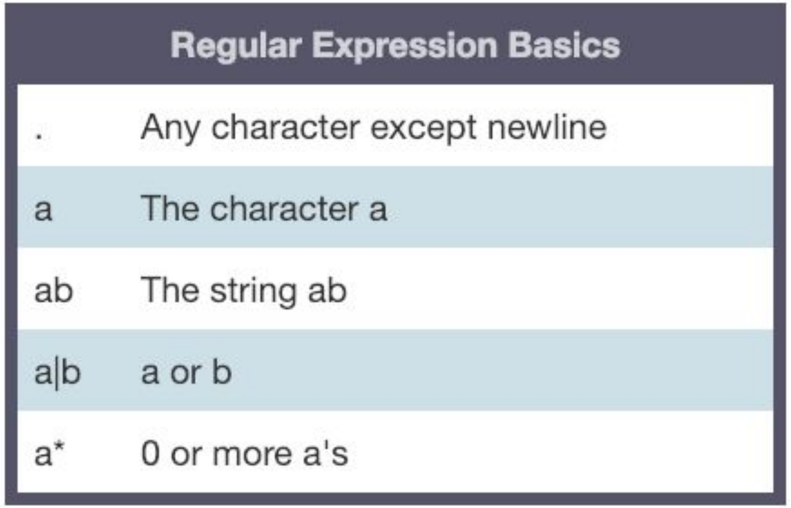
MATCH INFORMATION

Match 1

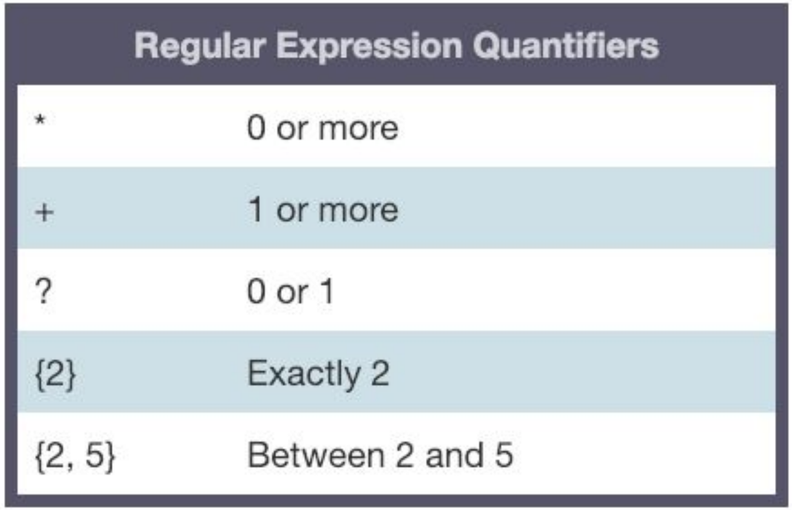
| | | |
|---|---|---|
| Full match | 29-40 | 123-45-6789 |

## Puzzle: Use regex101.com to test!

https://www.debuggex.com/cheatsheet/regex/python



| Regular Expression Basics | |
|---|---|
| . | Any character except newline |
| a | The character a |
| ab | The string ab |
| a\|b | a or b |
| a* | 0 or more a's |



| Regular Expression Quantifiers | |
|---|---|
| * | 0 or more |
| + | 1 or more |
| ? | 0 or 1 |
| {2} | Exactly 2 |
| {2, 5} | Between 2 and 5 |

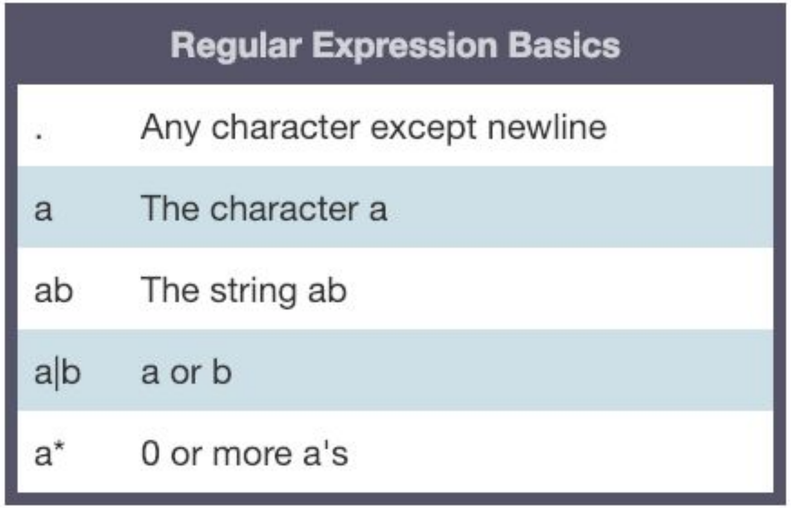
Give a regular expression that matches “moon”, “moooon”, “moooooon”, etc.

Your expression should match any **positive**, **even** number of o’s.

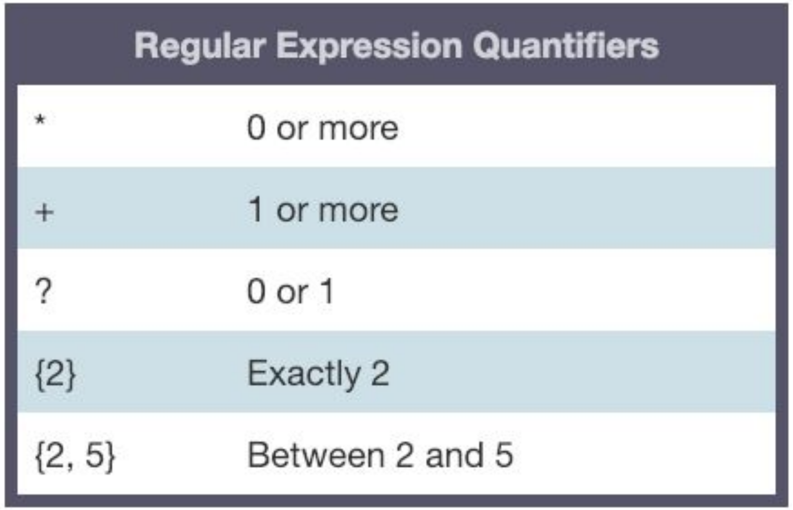
Solution: https://regex101.com/r/dj0nqD/1

## Puzzle: Use regex101.com to test!

https://www.debuggex.com/cheatsheet/regex/python



| Regular Expression Basics | |
|---|---|
| . | Any character except newline |
| a | The character a |
| ab | The string ab |
| a\|b | a or b |
| a* | 0 or more a's |



| Regular Expression Quantifiers | |
|---|---|
| * | 0 or more |
| + | 1 or more |
| ? | 0 or 1 |
| {2} | Exactly 2 |
| {2, 5} | Between 2 and 5 |

Give a regex that matches muun, muuuun, moon, mooooon, etc. Your expression should match any positive, even number of either u's or o's, but not a u/o mix.

Solution: https://regex101.com/r/4CMWjj/1

## Order of Operations in Regexes

`m((uu)+|(oo)+)n`

- Matches starting with m and ending with n, with either of the following in the middle:
  - `(uu)+`
  - `(oo)+`

Match examples:

```
muun
muuuun
moon
moooon
```

`m((uu)+)|((oo)+)n`

- Matches either of the following
  - `m` followed by `(uu)+`
  - `(oo)+` followed by n

Match examples:

```
muu
muuuu
oon
oooon
```

## Limitations of Regular Expressions

Writing regular expressions is like writing a program.

- Need to know the syntax well.
- Can be easier to write than to read.
- Can be difficult to debug.

- Regular expressions are sometimes jokingly referred to as a write only language.
- For parsing a hierarchical structure, such as JSON, use a parser, not a regex.

"Some people, when confronted with a problem, think ‘I know, I'll use regular expressions.’ Now they have two problems." - Jamie Zawinski (Source)

## Email Address Regular Expression

The regular expression for email addresses (for the Perl programming language):

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?: \r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\ ](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?: (?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n) ?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t] )*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])* )(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*) *:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r \n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t ]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\]( ?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(? :\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)? [ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]| \\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|" (?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[
\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|( ?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[
\t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([ ^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\ ]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\ r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\] |\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\ .|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(? :[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\". \[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]
]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)(?:,\s*(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[
\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t
])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|
\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\ ]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[
\t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[" ()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[
\t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<> @,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@, ;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\ ".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\". \[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[
"()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])
+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*))*)?;\s*)
```

From: [http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html](http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html)

# Even More Regular Expression Syntax

| operation | example | matches | does not match |
|---|---|---|---|
| built-in character classes | `\w+`<br>`\d+` | `fawef`<br>`231231` | `this person`<br>`423 people` |
| character class negation | `[^a-z]+` | `PEPPERS3982`<br>`17211!↑å` | `porch`<br>`CLAmS` |
| escape character | `cow\.com` | `cow.com` | `cowscom` |

Suppose you want to match one of the reserved characters, such as . or [ or ]

- In these cases, you must *escape* the character using a backslash.

## Regular Expressions Puzzle: tinyurl.com/reg913a

| operation | example | matches | does not match |
|---|---|---|---|
| built-in character classes | `\w+`<br>`\d+` | `fawef`<br>`231231` | `this person`<br>`423 people` |
| character class negation | `[^a-z]+` | `PEPPERS3982`<br>`17211!↑å` | `porch`<br>`CLAmS` |
| escape character | `cow\.com` | `cow.com` | `cowscom` |

Create a regular expression that matches the red portion below.

```
169.237.46.168 - - [26/Jan/2014:10:47:58
-0800] "GET /stat141/Winter04/ HTTP/1.1" 200
2585 "http://anson.ucdavis.edu/courses/"
```

# Even More Regular Expression Features

| operation | example | matches | does not match |
|---|---|---|---|
| beginning of line | `^ark` | `ark two`<br>`ark o ark` | `dark` |
| end of line | `ark$` | `dark`<br>`ark o ark` | `ark two` |
| non-greedy qualifier | `5.*?5` | `5005`<br>`55` | `5005005` |

5*5 would match this!

A few additional common regex features are listed above.

- Won’t discuss these in class, but might come up in discussion or hw.
- There are even more out there!

The official guide is good! [https://docs.python.org/3/howto/regex.html](https://docs.python.org/3/howto/regex.html)

## Matching Regular Expression Patterns in Python

## `re.findall` in Python

In Python, `re.findall(pattern, text)` will return a list of all matches.

```
text = "My SSN is 456-76-4295, or maybe 456-67-4295."
pattern = "[0-9]{3}-[0-9]{2}-[0-9]{4}"
m = re.findall(pattern, text)
print(m)
```

```
['456-76-4295', '456-67-4295']
```

## Regular Expression Groups

Earlier we used parentheses to specify the order of operations.

Parenthesis have another meaning:

- Every set of parentheses in the pattern corresponds to a group
- Regular expression matchers (e.g. re.findall, regex101.com) will return matches organized by groups.

```
s = """Observations: 03:04:53 - Horse awakens.
03:05:14 - Horse goes back to sleep."""
pattern = "(\d\d):(\d\d):(\d\d) - (.*)"
matches = re.findall(pattern, s)
```

```
[('03', '04', '53', 'Horse awakens.'),
 ('03', '05', '14', 'Horse goes back to sleep.')]
```

## re.sub in Python

In Python, `re.sub(pattern, repl, text)` will return text with all instances of `pattern` replaced by `repl`.

```
text = '<div><td valign="top">Moo</td></div>'
pattern = "<[^>]+>"
cleaned = re.sub(pattern, '', text)
print(cleaned)
```

```
    'Moo'
```

## Optional (but Handy) Regex Concepts

These regex features aren't going to be on an exam, but they are useful:

- **Lookaround**: match "good" if it's not preceded by "not":
  `(?<!not )good`
- **Backreferences**: match HTML tags of the same name:
  `<(\w+)>.*</\1>`

## Optional (but Handy) Regex Concepts

These regex features aren't going to be on an exam, but they are useful:

- **Named groups**: match a vowel as a named group:

  ```
  (?P<vowel>[aeiou])
  ```

- **Free Space**: Allow free space and comments in a pattern:

  ```
  # Match a 20th or 21st century date in yyyy-mm-dd format
  (19|20)\d\d                 # year (group 1)
  [- /.]                      # separator
  (0[1-9]|1[012])             # month (group 2)
  [- /.]                      # separator
  (0[1-9]|[12][0-9]|3[01])    # day (group 3)
  ```

## Use Case: Text Features

# Top 20 Violations

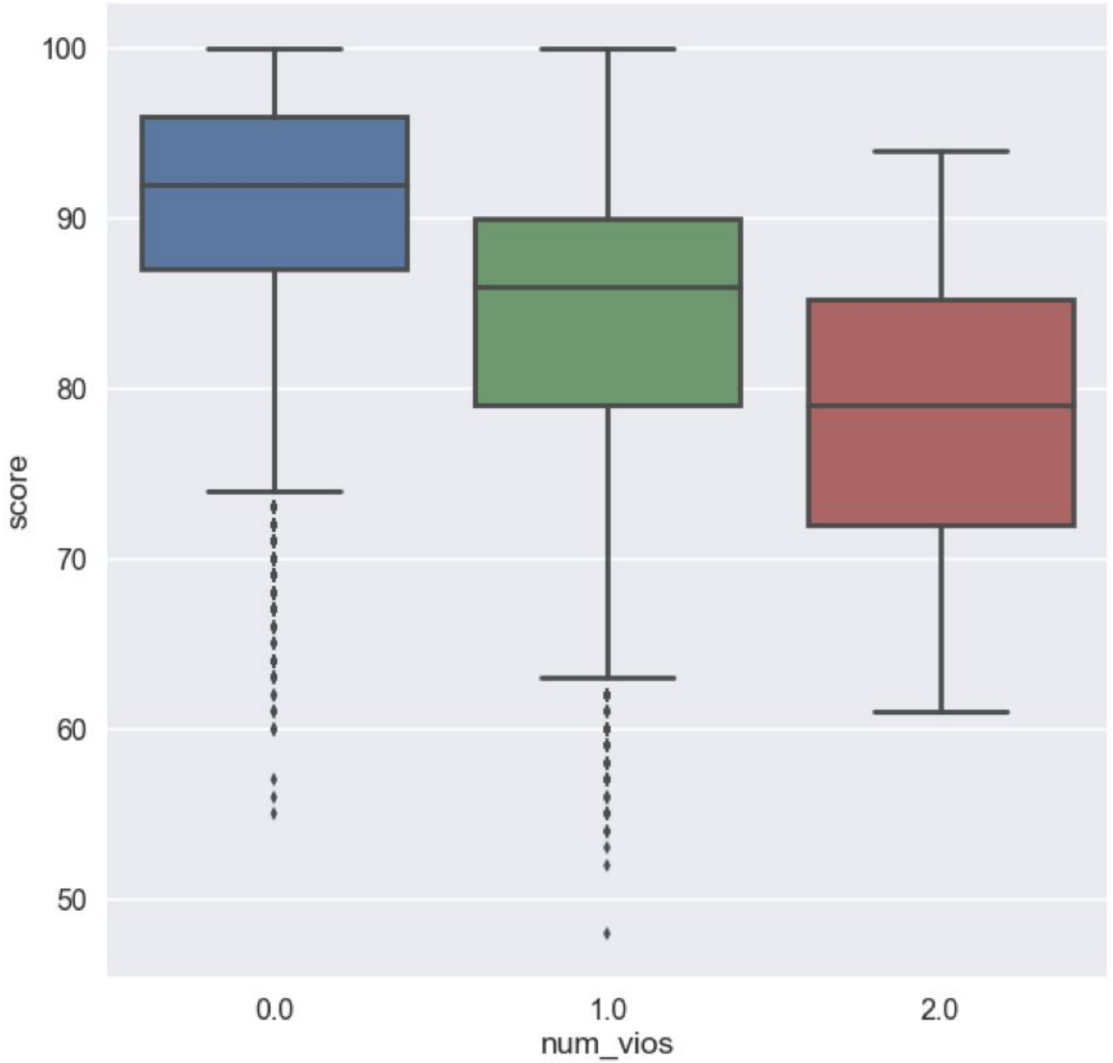
- unclean or degraded floors walls or ceilings
- moderate risk food holding temperature
- inadequate and inaccessible handwashing facilities
- unapproved or unmaintained equipment or utensils
- inadequately cleaned or sanitized food contact surfaces
- wiping cloths not clean or properly stored or inadequate sanitizer
- improper food storage
- foods not protected from contamination
- moderate risk vermin infestation
- high risk food holding temperature
- unclean nonfood contact surfaces
- food safety certificate or food handler card not available
- unclean or unsanitary food contact surfaces
- inadequate food safety knowledge or lack of certified food safety manager
- improper storage of equipment utensils or linens
- low risk vermin infestation
- permit license or inspection report not posted
- improper cooling methods
- unclean hands or improper use of gloves
- improper or defective plumbing

## What Features to Examine?

| | |
|---|---|
| Cleanliness | `'clean\|sanit'` |
| High risk | `'high risk'` |
| Vermin | `'vermin'` |
| Surfaces | `'wall\|ceiling\|floor\|surface'` |
| Humans | `'hand\|glove\|hair\|nail'` |
| Permits and Certification | `'permit\|certif'` |

## Vermin Violations

Median safety score is 86 for restaurants with one vermin infestation violation!



## Summary

- EDA is a process used to understand a dataset
  - Investigate Data Types, Granularity, Scope, Temporality, and Faithfulness
- To extract features from text fields:
  - Use built-in Python string methods
  - Or regex if your pattern is complicated