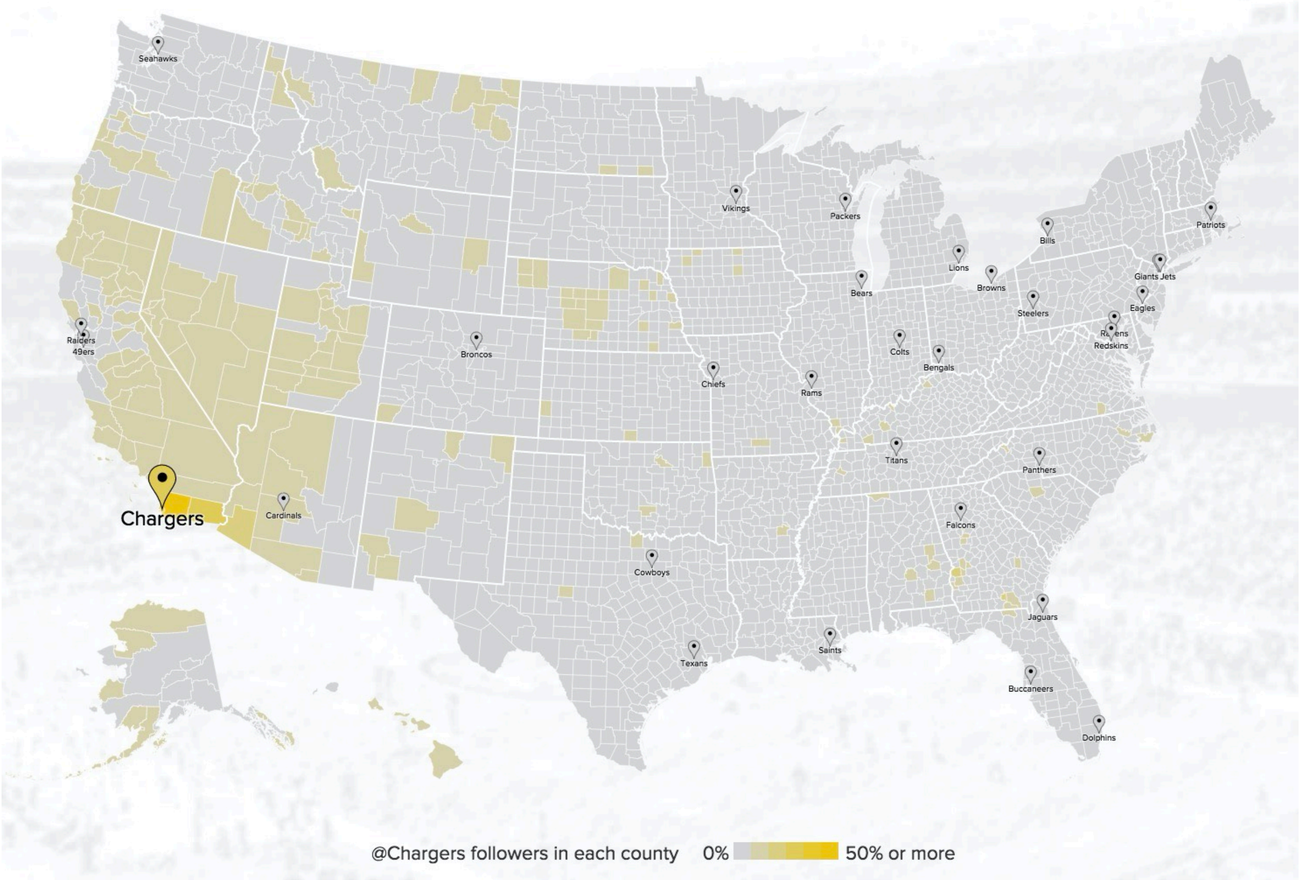
Bradley Voytek, Ph.D.

UC San Diego

Department of Cognitive Science
Halıcıoğlu Data Science Institute
Neurosciences Graduate Program

bvoytek@ucsd.edu
@bradleyvoytek

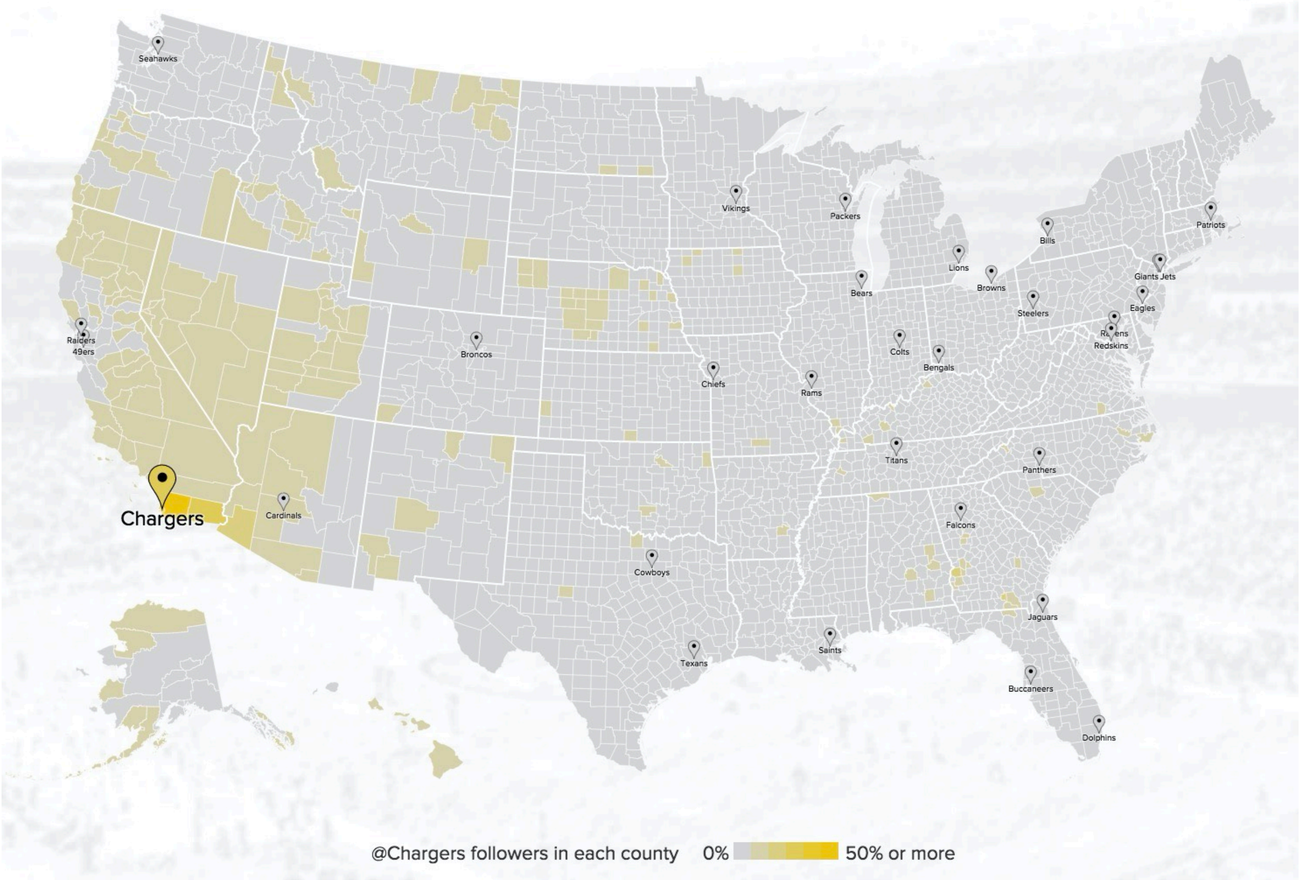
More NFL

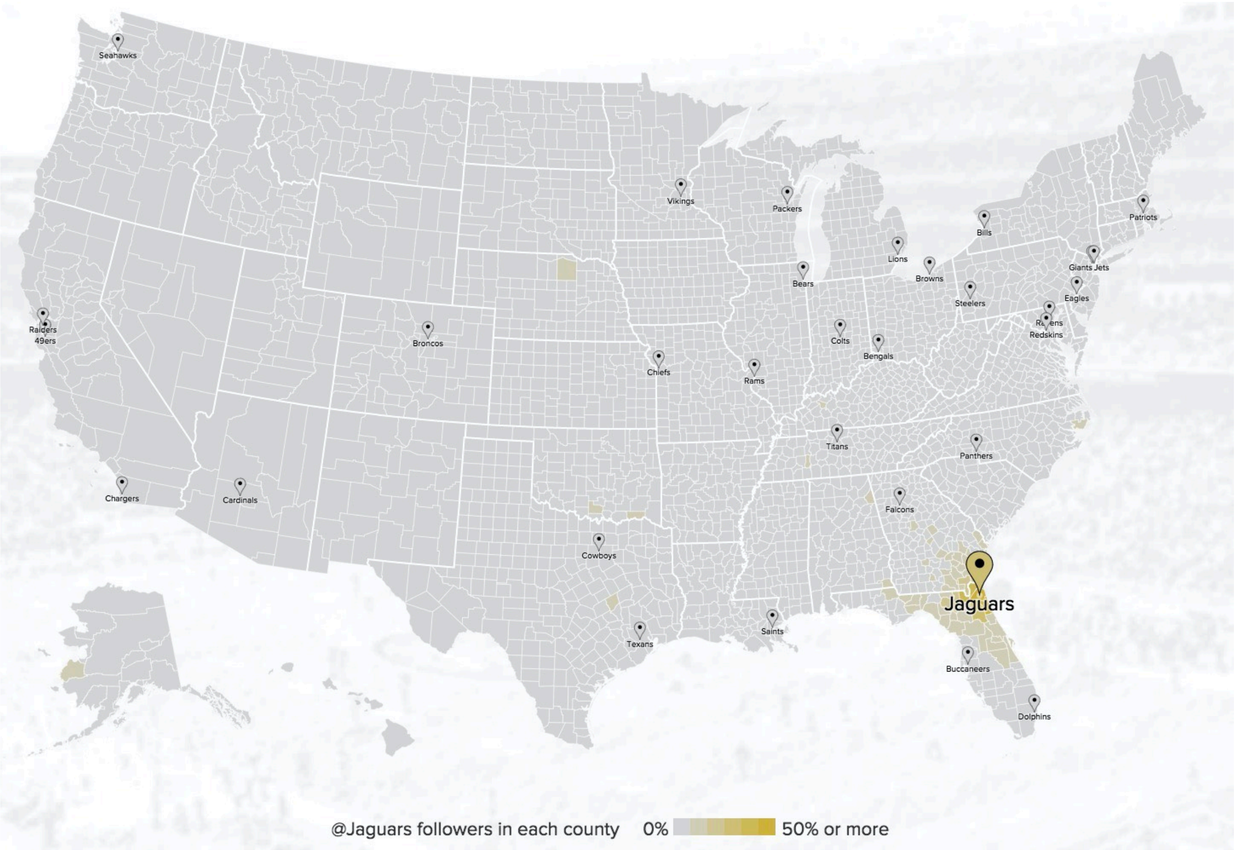


Source: https://interactive.twitter.com/nfl_followers2014

UC San Diego

More NFL

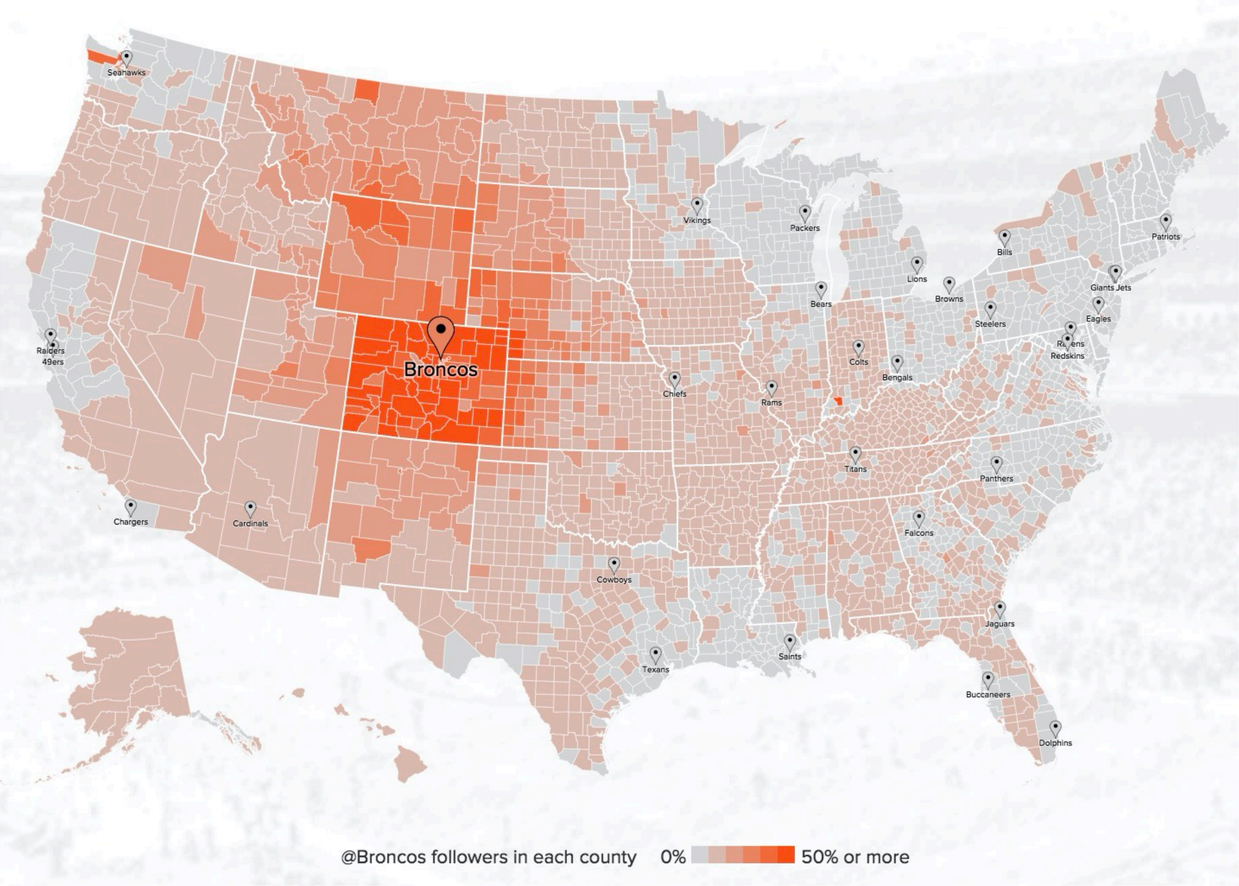


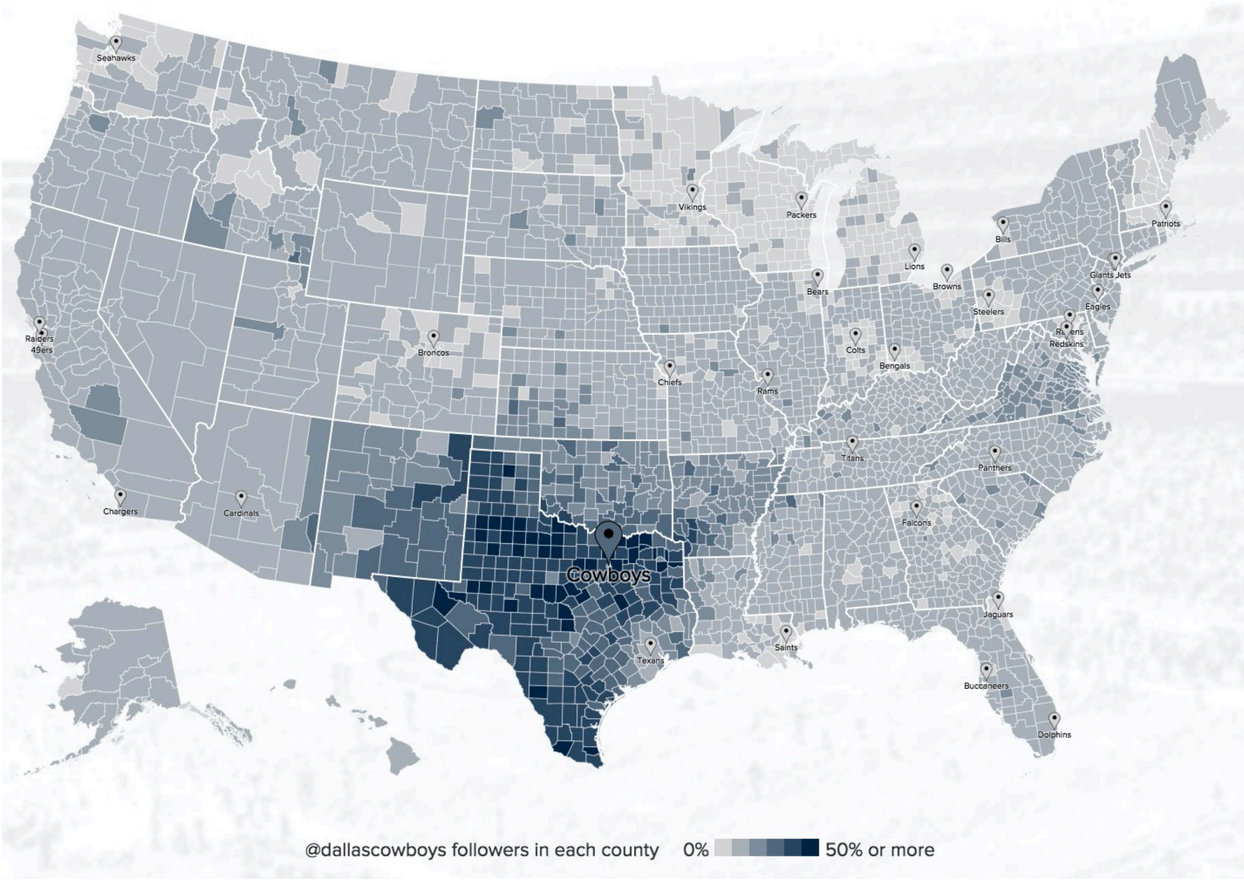


Source: https://interactive.twitter.com/nfl_followers2014

UC San Diego

More NFL

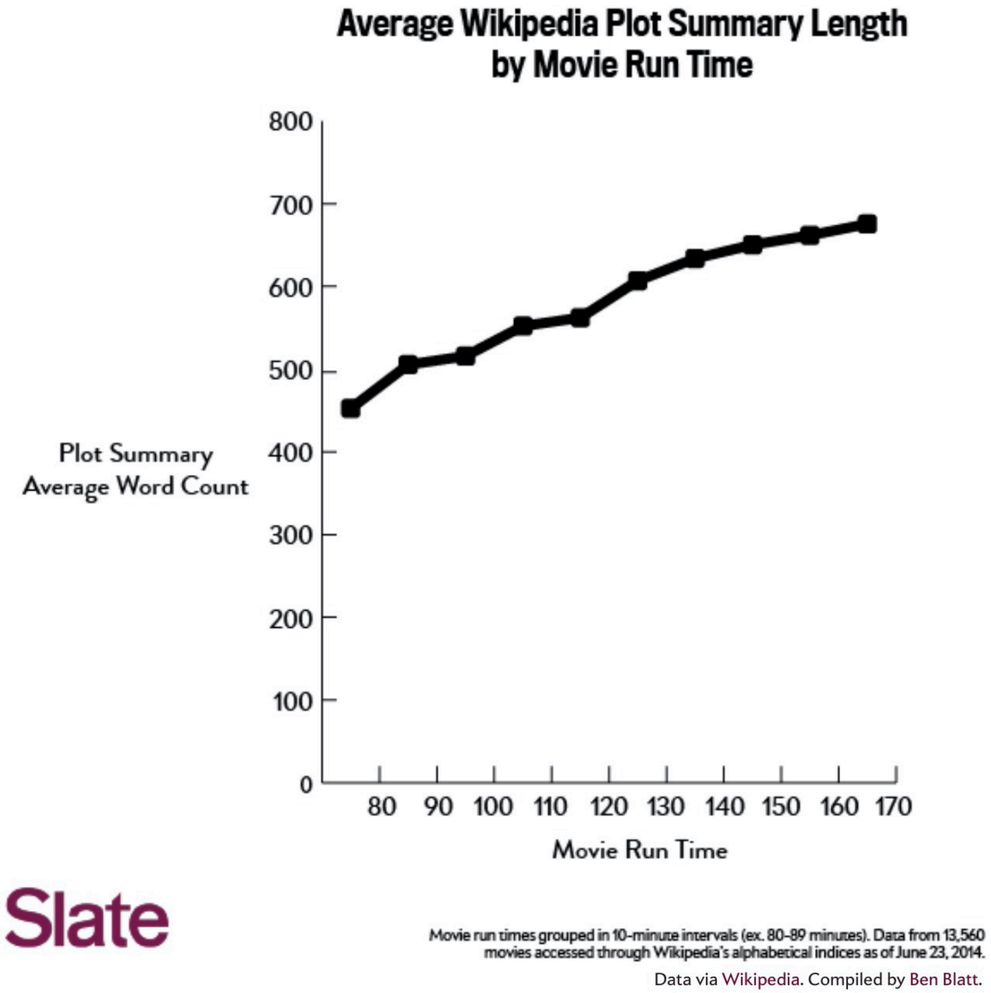




Source: https://interactive.twitter.com/nfl_followers2014

UC San Diego

Wikipedia



If we consider only films that enjoyed a theatrical release in the United States, the top 10 list looks like this:

1. ***Band of the Hand:*** 3,530 words
2. ***V/H/S/2***: 3,360 words
3. ***Premonition (2007)***: 3,130 words
4. ***Underground (1995)***: 2,824 words
5. ***The Insider***: 2,395 words
6. ***For Colored Girls***: 2,361 words
7. ***2046***: 2,346 words
8. ***Problem Child 2***: 2,282 words
9. ***K-19: The Widowmaker***: 2,046 words
10. ***The Good Girl***: 2,042 words

Many of these articles appear to have been stuffed with detail by superfans.

Source: Ben Blatt, http://www.slate.com/articles/arts/culturebox/2014/06/what_is_the_most_detailed_wikipedia_movie_plot_summary_of_all_time.html

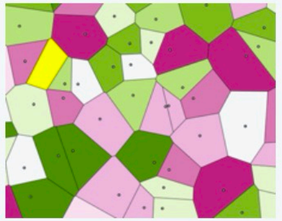
d3.js

Sunburst

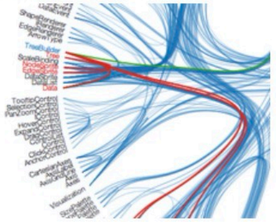
Node-Link Tree

Treemap

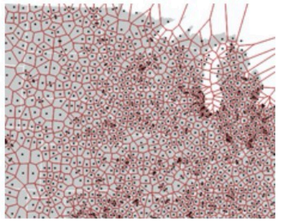
Voronoi Diagram



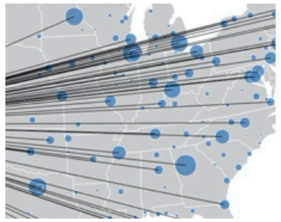
Hierarchical Edge Bundling



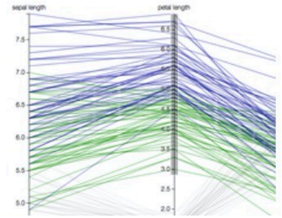
Voronoi Diagram



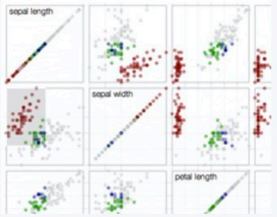
Symbol Map



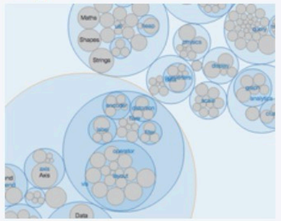
Parallel Coordinates



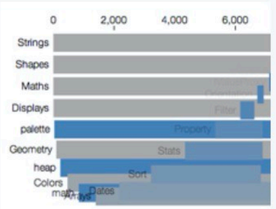
Scatterplot Matrix



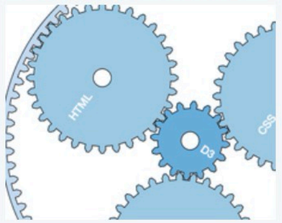
Zoomable Pack Layout



Hierarchical Bars



Epicyclical Gears



Source: https://d3js.org/

UC San Diego

COGS 108
Data Science in Practice

Python! (For great Data Science)

UC San Diego

Recommendations

- *Data Science from Scratch* - Joel Grus (Allen Institute for AI)
- *Python Data Science Handbook* - Jake VanderPlas (UW eScience Institute)

Why Python?

Python has emerged over the last couple decades as a first-class tool for scientific computing tasks, including the analysis and visualization of large datasets. This may have come as a surprise to early proponents of the Python language: the language itself was not specifically designed with data analysis or scientific computing in mind.

Source: VanderPlas, *Python Data Science Handbook*

Why Python?

The usefulness of Python for data science stems primarily from the large and active ecosystem of third-party packages: NumPy for manipulation of homogeneous array-based data, Pandas for manipulation of heterogeneous and labeled data, SciPy for common scientific computing tasks, Matplotlib for publication-quality visualizations, IPython [Jupyter] for interactive execution and sharing of code, Scikit-Learn for machine learning, and many more tools that will be mentioned in the following pages.

Source: VanderPlas, *Python Data Science Handbook*

Why Python?

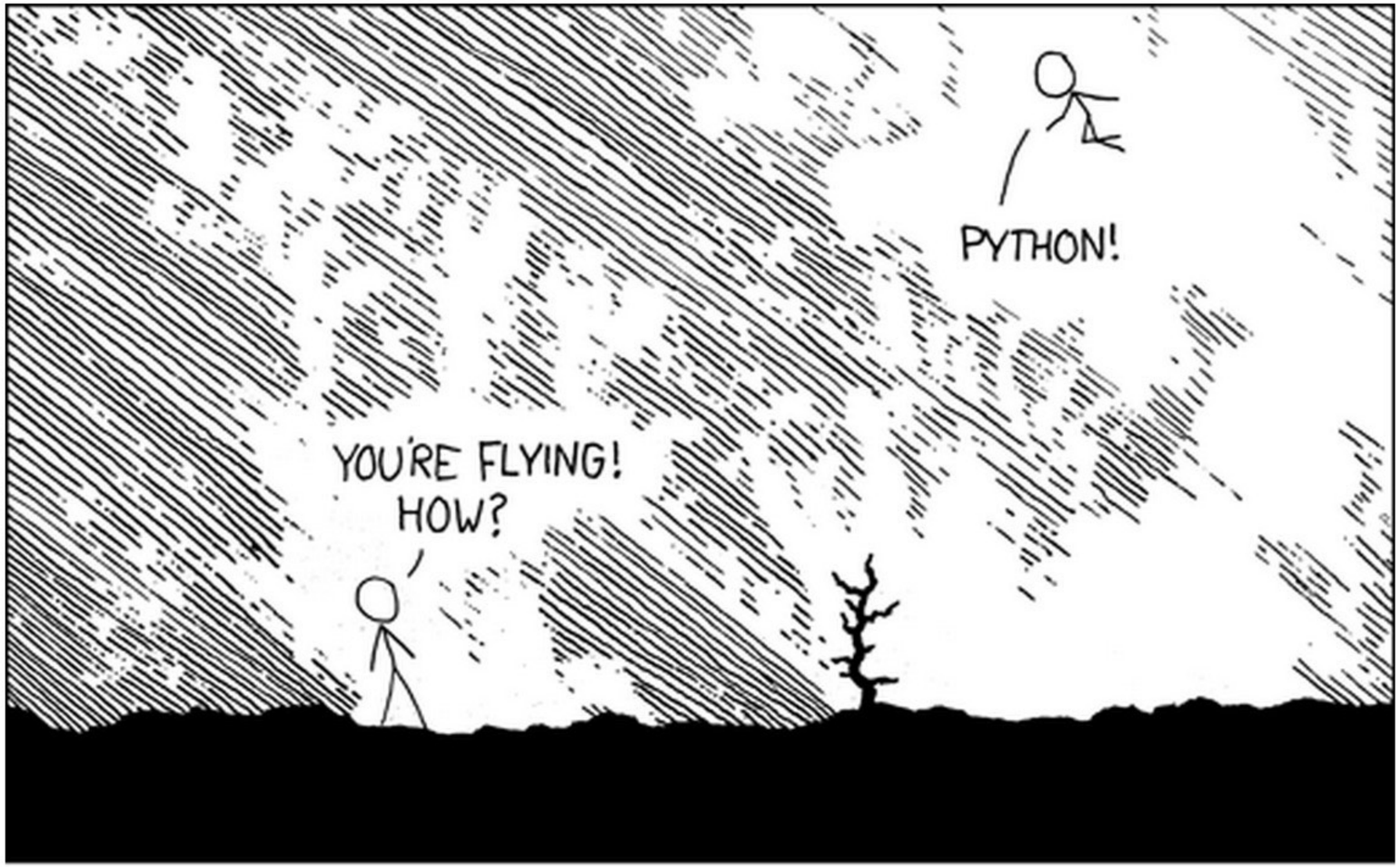
Python isn't the best at anything.

Why Python?

Python isn't the best at anything.

But it's the *second best* at EVERYTHING.

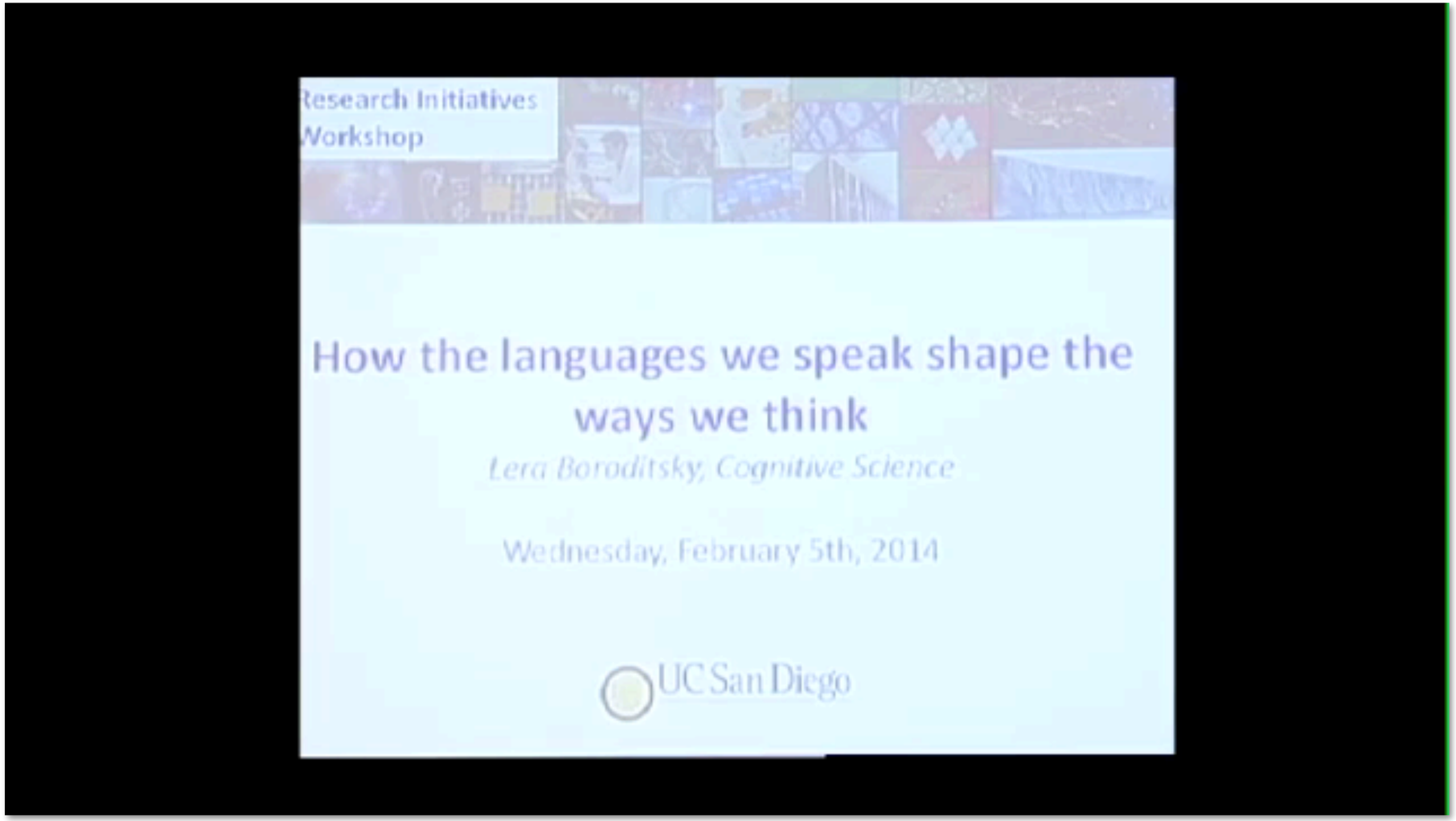
Why Python?



Source: *XKCD*

UC San Diego

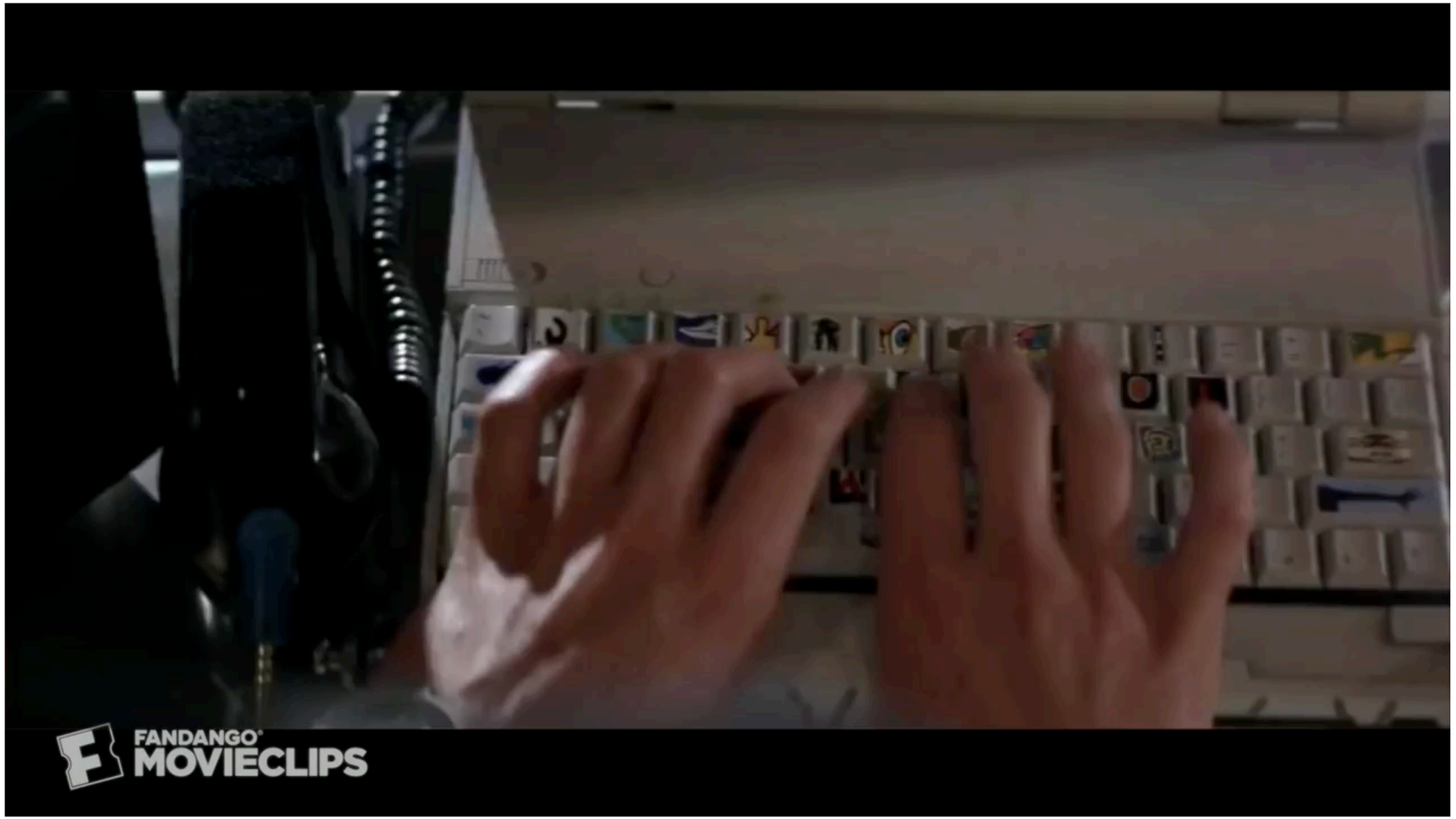
Languages shape thought



Source: Lera Boroditsky

UC San Diego

Getting computers to work for you



Source: *Hackers*

Getting computers to work for you

Source: *Hackers*

The future of programming

Source: Bret Victor, *Turing Complete* 2012

Jupyter - Magics

```
In [ ]: %quickref
```

```
IPython -- An enhanced Interactive Python - Quick Reference Card
================================================================

obj?, obj??      : Get help, or more help for object (also works as
                   ?obj, ??obj).
?foo.*abc*       : List names in 'foo' containing 'abc' in them.
%magic           : Information about IPython's 'magic' % functions.

Magic functions are prefixed by % or %%, and typically take their arguments
without parentheses, quotes or even commas for convenience.  Line magics take a
single % and cell magics are prefixed with two %%.

Example magic function calls:

%alias d ls -F   : 'd' is now an alias for 'ls -F'
alias d ls -F    : Works if 'alias' not a python name
alist = %alias   : Get list of aliases to 'alist'
cd /usr/share    : Obvious. cd -<tab> to choose from visited dirs.
%cd??            : See help AND source for magic %cd
%timeit x=10     : time the 'x=10' statement with high precision.
%%timeit x=2**100
```

Jupyter - Magics

```
In [46]: %pwd

Out[46]: '/Users/Voytek'
```

Jupyter - Magics

In [47]:
```
# print object details

x = 2
x?
```

```
Type:        int
String form: 2
Docstring:
int(x=0) -> integer
int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments
are given.  If x is a number, return x.__int__().  For floating point
numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string,
bytes, or bytearray instance representing an integer literal in the
given base.  The literal can be preceded by '+' or '-' and be surrounded
by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
Base 0 means to interpret the base from the string as an integer literal.
>>> int('0b100', base=0)
4
```

Jupyter - Magics

```
In [50]: # print object details

         my_string = 'hello world'
         my_string?
```

```
Type:        str
String form: hello world
Length:      11
Docstring:
str(object='') -> str
str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or
errors is specified, then the object must expose a data buffer
that will be decoded using the given encoding and error handler.
Otherwise, returns the result of object.__str__() (if defined)
or repr(object).
encoding defaults to sys.getdefaultencoding().
errors defaults to 'strict'.
```

Jupyter - Tab completion

```
In [ ]: # tab completion

        print(my_string.())
```

Jupyter - Tab completion

```
In [ ]: # tab completion

        print(my_string.())
```

```
In [ ]: # tab completion

        print(my_string.())
```

```
my_string.capitalize
my_string.casefold
my_string.center
my_string.count
my_string.encode
my_string.endswith
my_string.expandtabs
my_string.find
my_string.format
my_string.format_map
```

dropdown listing all the methods associated with my_string!

Jupyter - Tab completion

```
In [ ]: # tab completion

        print(my_string.())
```

```
In [ ]: # tab completion

        print(my_string.())
```

```
my_string.rstrip
my_string.split
my_string.splitlines
my_string.startswith
my_string.strip
my_string.swapcase
my_string.title
my_string.translate
my_string.upper
my_string.zfill
```

```
In [ ]: # mmul

        my_lis
            '
            '
            '
            '
        ]
        print(
```

Jupyter - Tab completion

```
In [ ]: # tab completion

        print(my_string.())
```

```
In [ ]: # tab completion

        print(my_string.|())
```

```
my_string.rstrip
my_string.split
my_string.splitlines
my_string.startswith
my_string.strip
my_string.swapcase
my_string.title
my_string.translate
my_string.upper
my_string.zfill
```

```
In [ ]: # mmul

        my_lis
            'c
            't
            't
            'f
        ]
        print(
```

```
In [51]: # tab completion

         print(my_string.upper())

HELLO WORLD
```

Jupyter - Multicursor support

In [52]:
```
# mmulticursor

my_list = [
    'one'
    'two'
    'three'
    'four'
]
print(my_list)
```

```
['onetwothreefour']
```

UC San Diego

Jupyter - Multicursor support

```
In [52]: # mmulticursor

         my_list = [
             'one'
             'two'
             'three'
             'four'
         ]
         print(my_list)

         ['onetwothreefour']
```

```
In [52]: # mmulticursor

         my_list = [
             'one'
             'two'
             'three'
             'four'
         ]
         print(my_list)
```

hold down *alt* while highlighting, then press *right arrow*

UC San Diego

Jupyter - Multicursor support

```
In [52]: # mmulticursor

         my_list = [
             'one'
             'two'
             'three'
             'four'
         ]
         print(my_list)

         ['onetwothreefour']
```

```
In [52]: # mmulticursor

         my_list = [
             'one',
             'two',
             'three',
             'four',
         ]
         print(my_list)
```

add a *comma* to the end of all lines, all at once!

Jupyter - Multicursor support

```
In [52]: # mmulticursor

         my_list = [
             'one'
             'two'
             'three'
             'four'
         ]
         print(my_list)

         ['onetwothreefour']
```

```
In [52]: # mmulticursor

         my_list = [
             'one',
             'two',
             'three',
             'four',
         ]
         print(my_list)
```

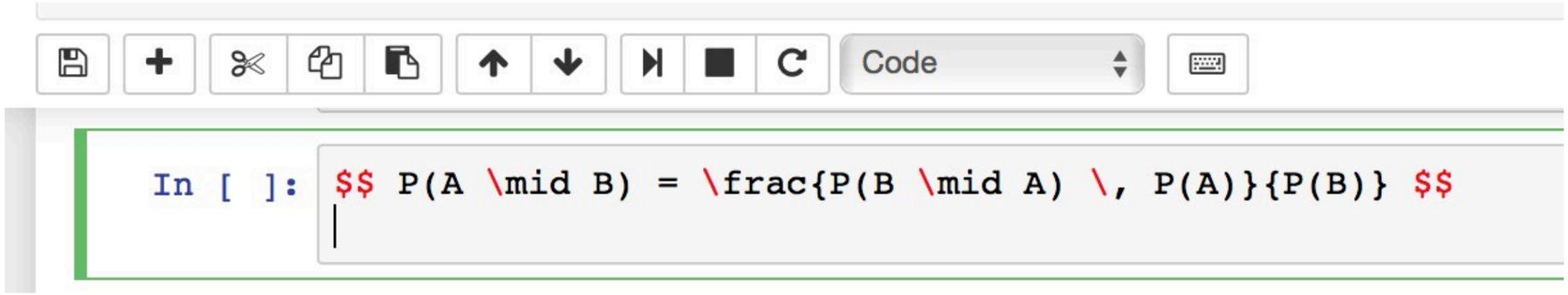
```
In [53]: # mmulticursor

         my_list = [
             'one',
             'two',
             'three',
             'four',
         ]
         print(my_list)

         ['one', 'two', 'three', 'four']
```

Jupyter - Magics

```
In [54]: %who

math      my_list          my_string
```

```
In [55]: %whos

Variable      Type       Data/Info
----------------------------------
math          module     <module 'math' from '/Use<...>h.cpython-35m-darwin.so'>
my_list       list       n=4
my_string     str        hello world
```

Jupyter - Markdown



Jupyter - Markdown

Code

In [ ]:
```
$$ P(A \mid B) = \frac{P(B \mid A) \, P(A)}{P(B)} $$
```

Markdown

```
$$ P(A \mid B) = \frac{P(B \mid A) \, P(A)}{P(B)} $$
```

Jupyter - Markdown

Code

```
In [ ]: $$ P(A \mid B) = \frac{P(B \mid A) \, P(A)}{P(B)} $$
```

Markdown

```
$$ P(A \mid B) = \frac{P(B \mid A) \, P(A)}{P(B)} $$
```

Markdown

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

Jupyter - Markdown

```
# Large-scale analysis of practice effects on interference across the lifespan

Behavioral data were collected from Lumosity, a web-based suite of games voluntarily played <em>ad libitum</em> by
users who pay a subscription fee to use the service. Anonymized data from the game "Lost in Migration"—a variant on
the traditional Eriksen Flanker task (see Eriksen & Eriksen, 1974)(Fig. 1)—were shared with the authors for purposes
of scientific research. Data were collected from <em>N</em> users aged 18-70, each of whom completed at least 24 game
sessions and one practice session. Lumosity's users assent to Terms of Service indicating that their anonymized data
may be used in aggregate for research purposes.

The "Lost in Migration" game (Fig. 1) is similar to the Eriksen Flanker task in that users respond to which of the
four possible directions a central bird is facing using the arrow keys on their computer keyboards. Four other birds,
each of which is facing in the same direction as one another, surround this central bird. There are two primary trial
types in this task: congruent and incongruent. In the congruent condition the central bird is facing the same
direction as the four surrounding birds; in the incongruent condition the central bird is facing in a different
direction. Each session lasts for 45 seconds. The within-subjects RT difference between incongruent and congruent
conditions was used to index interference.

<em>Note RT difference may not be right, I need help figuring out what to use here.</em>

Because of the relatively large sample size, even trivially small effects prove to be statistically significant so the
goal of this is largely model comparison and knowledge discovery.

<img src="./Lumos-TaskFigure.jpg", width=800/>
<strong>Fig. 1.</strong> Behavioral task. Examples of the two conditions included in the behavioral paradigm that
formed the basis of these analyses. In this task—a modified Flanker paradigm— subjects report the direction of the
central bird. On the left is an example of a congruent trial wherein the central target bird is facing in the same
direction as the flanking stimuli. On the right is an example of an incongruent trial wherein the central target bird
is facing in a different direction. The weighted percent difference between response times between the two trial types
gives an interference index, a measure of cognitive control.
```

Jupyter - Markdown

Large-scale analysis of practice effects on interference across the lifespan

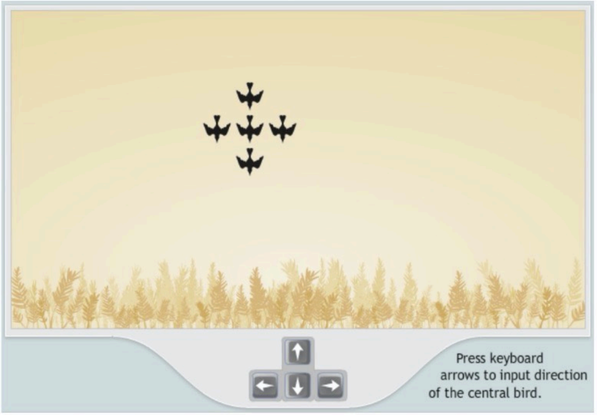
Behavioral data were collected from Lumosity, a web-based suite of games voluntarily played *ad libitum* by users who pay a subscription fee to use the service. Anonymized data from the game "Lost in Migration"—a variant on the traditional Eriksen Flanker task (see Eriksen & Eriksen, 1974)(Fig. 1)—were shared with the authors for purposes of scientific research. Data were collected from *N* users aged 18-70, each of whom completed at least 24 game sessions and one practice session. Lumosity's users assent to Terms of Service indicating that their anonymized data may be used in aggregate for research purposes.

The "Lost in Migration" game (Fig. 1) is similar to the Eriksen Flanker task in that users respond to which of the four possible directions a central bird is facing using the arrow keys on their computer keyboards. Four other birds, each of which is facing in the same direction as one another, surround this central bird. There are two primary trial types in this task: congruent and incongruent. In the congruent condition the central bird is facing the same direction as the four surrounding birds; in the incongruent condition the central bird is facing in a different direction. Each session lasts for 45 seconds. The within-subjects RT difference between incongruent and congruent conditions was used to index interference.

Note RT difference may not be right, I need help figuring out what to use here.

Because of the relatively large sample size, even trivially small effects prove to be statistically significant so the goal of this is largely model comparison and knowledge discovery.

congruent



incongruent

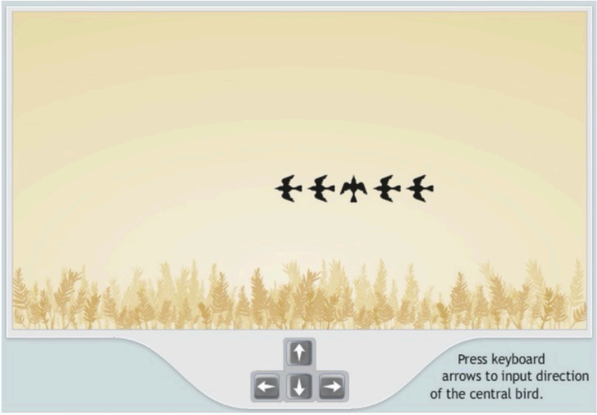


Fig. 1. Behavioral task. Examples of the two conditions included in the behavioral paradigm that formed the basis of these analyses. In this task—a modified Flanker paradigm— subjects report the direction of the central bird. On the left is an example of a congruent trial wherein the central target bird is facing in the same direction as the flanking stimuli. On the right is an example of an incongruent trial wherein the central target bird is facing in a different direction. The

Jupyter - Beginning an analysis

```
%reset
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import numpy as np
import scipy as sp
from scipy import signal
```

Jupyter - Beginning an analysis

magic to clear all variables

```
%reset
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import numpy as np
import scipy as sp
from scipy import signal
```

Jupyter - Beginning an analysis

```
%reset
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import numpy as np
import scipy as sp
from scipy import signal
```

magic to allow inline plotting

Jupyter - Beginning an analysis

magic for high resolution figures

```
%reset
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import numpy as np
import scipy as sp
from scipy import signal
```

Jupyter - retina resolution

ICK

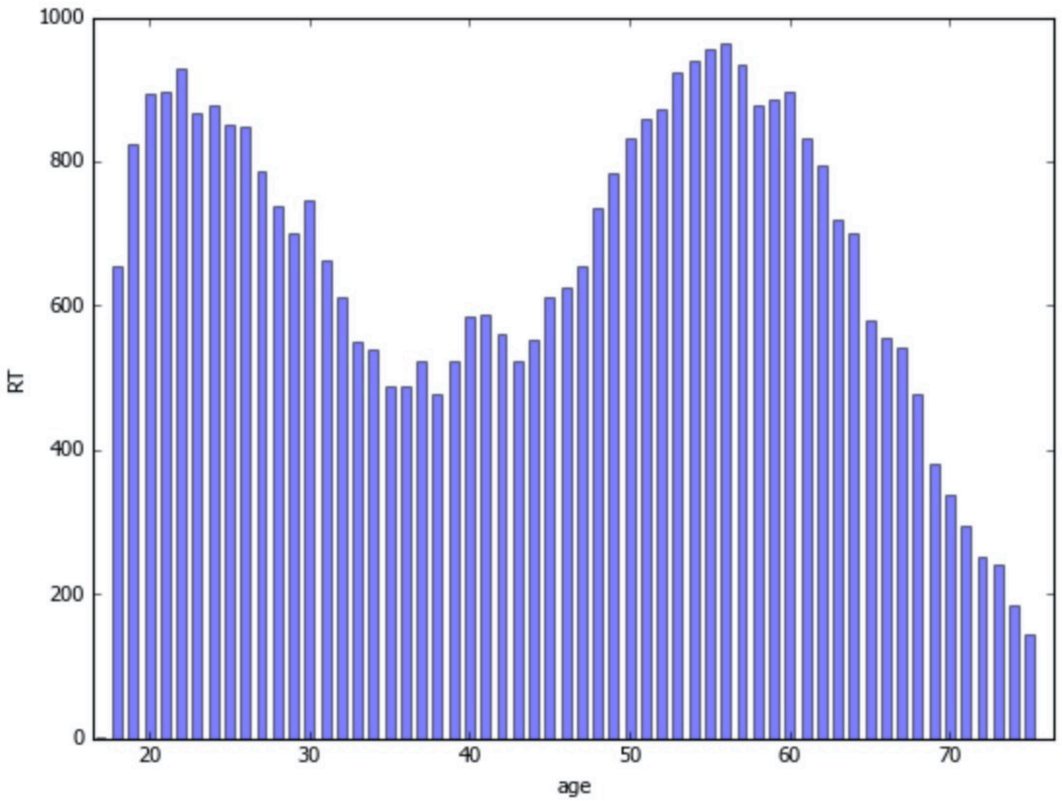


Fig. 2

Jupyter - retina resolution

ICK

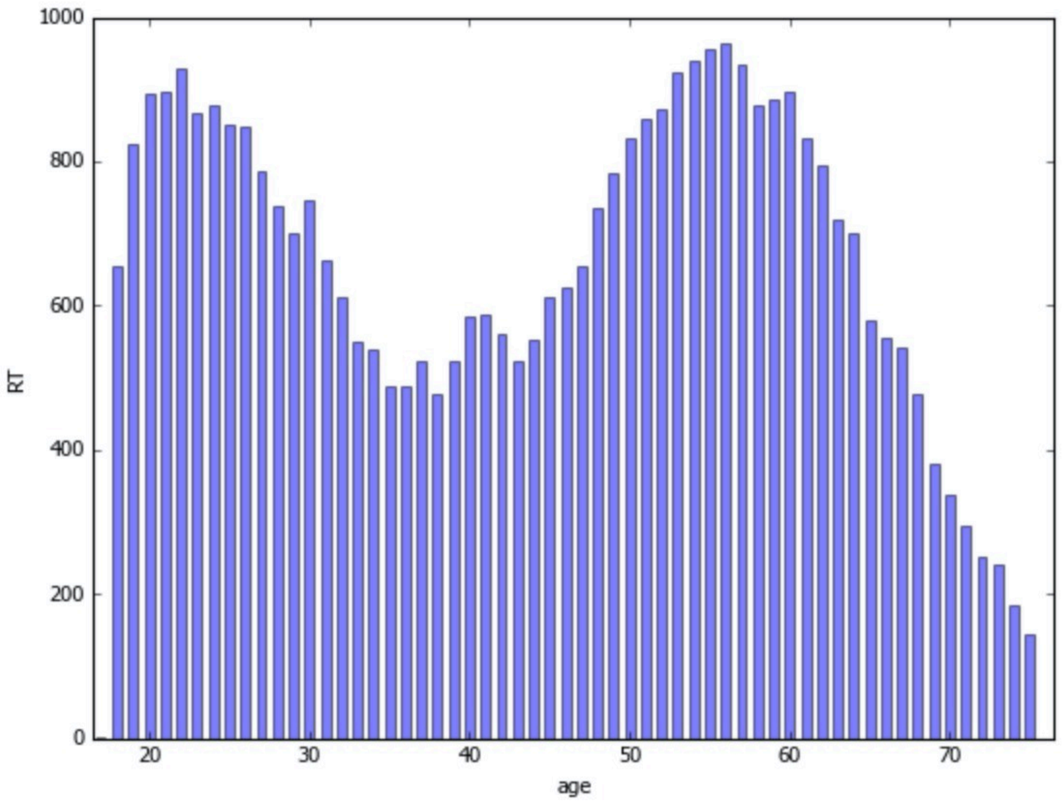


Fig. 2

YAY

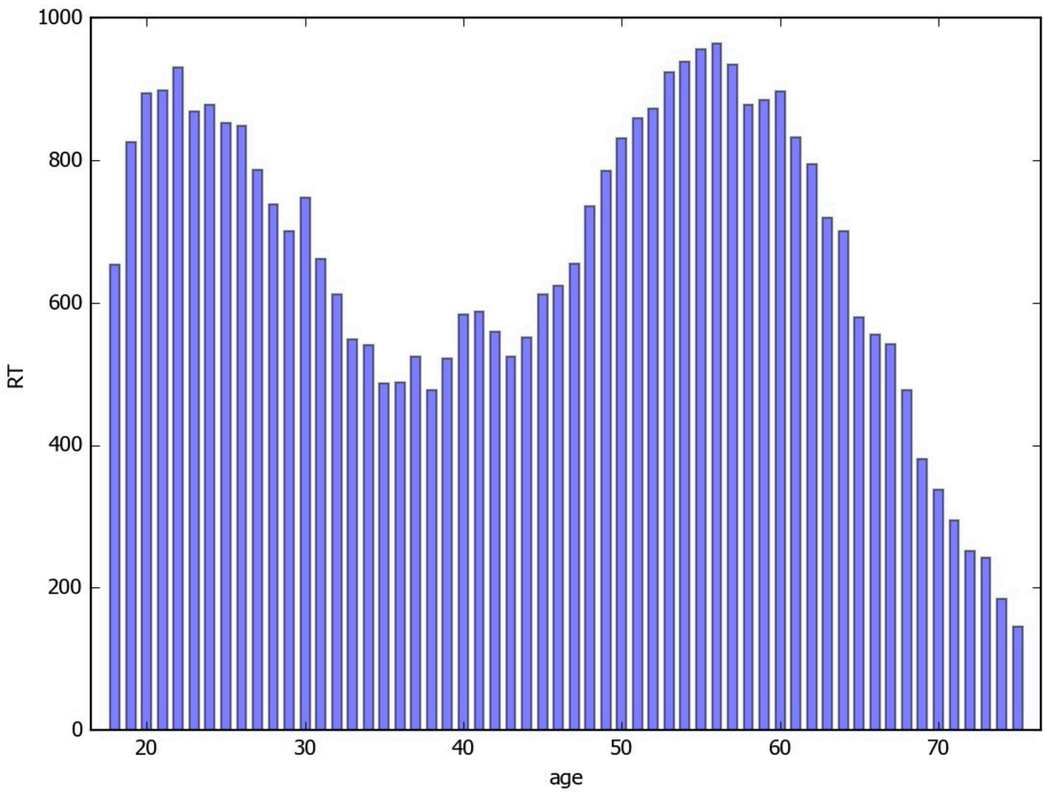


Fig. 2

UC San Diego

Jupyter - Plots in-line!

```
plt.plot(trials, rtc_by_age[0, :], 'sb', label='20-30 years (c)')
plt.plot(trials, rti_by_age[0, :], '.b', label='20-30 years (i)')
plt.plot(trials, rtc_by_age[1, :], 'sg', label='40-50 years (c)')
plt.plot(trials, rti_by_age[1, :], '.g', label='40-50 years (i)')
plt.plot(trials, rtc_by_age[2, :], 'sr', label='60-70 years (c)')
plt.plot(trials, rti_by_age[2, :], '.r', label='60-70 years (i)')
plt.title("RT by trial")
plt.xlabel("trial")
plt.ylabel("RT")
plt.legend(loc=1)
plt.figtext(.02, .02, "Fig. 3")
plt.show()
```

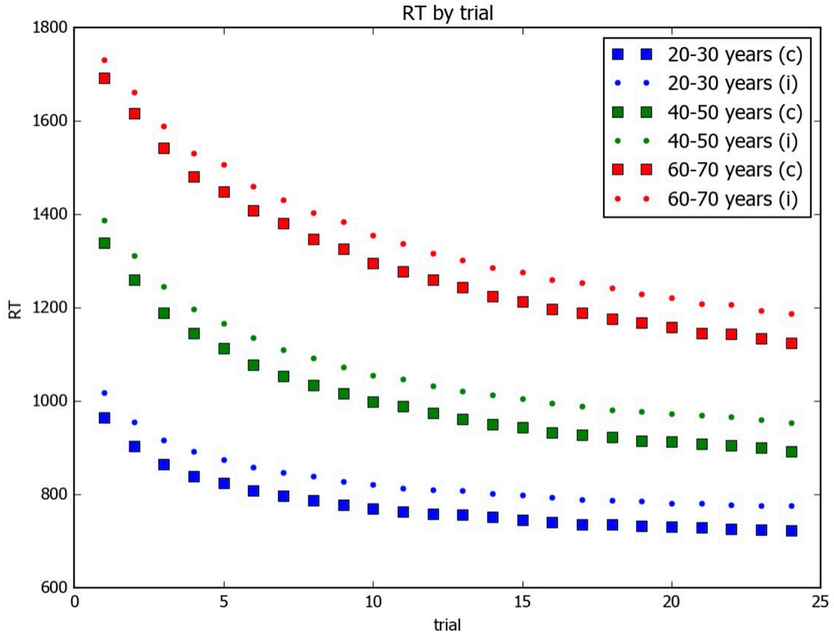


Fig. 3

Jupyter - Beginning an analysis

```
# plotting stuff
import seaborn as sns
sns.set_style('whitegrid')
sns.set_style("whitegrid", {'axes.grid' : False})

import matplotlib
import matplotlib.pyplot as plt

# plot settings
font = {'family' : 'Bitstream Vera Sans',
        'weight' : 'regular',
        'size'   : 13}
figure = {'figsize' : (16,8)}

matplotlib.rc('font', **font)
matplotlib.rc('figure', **figure)
```

plotting parameters

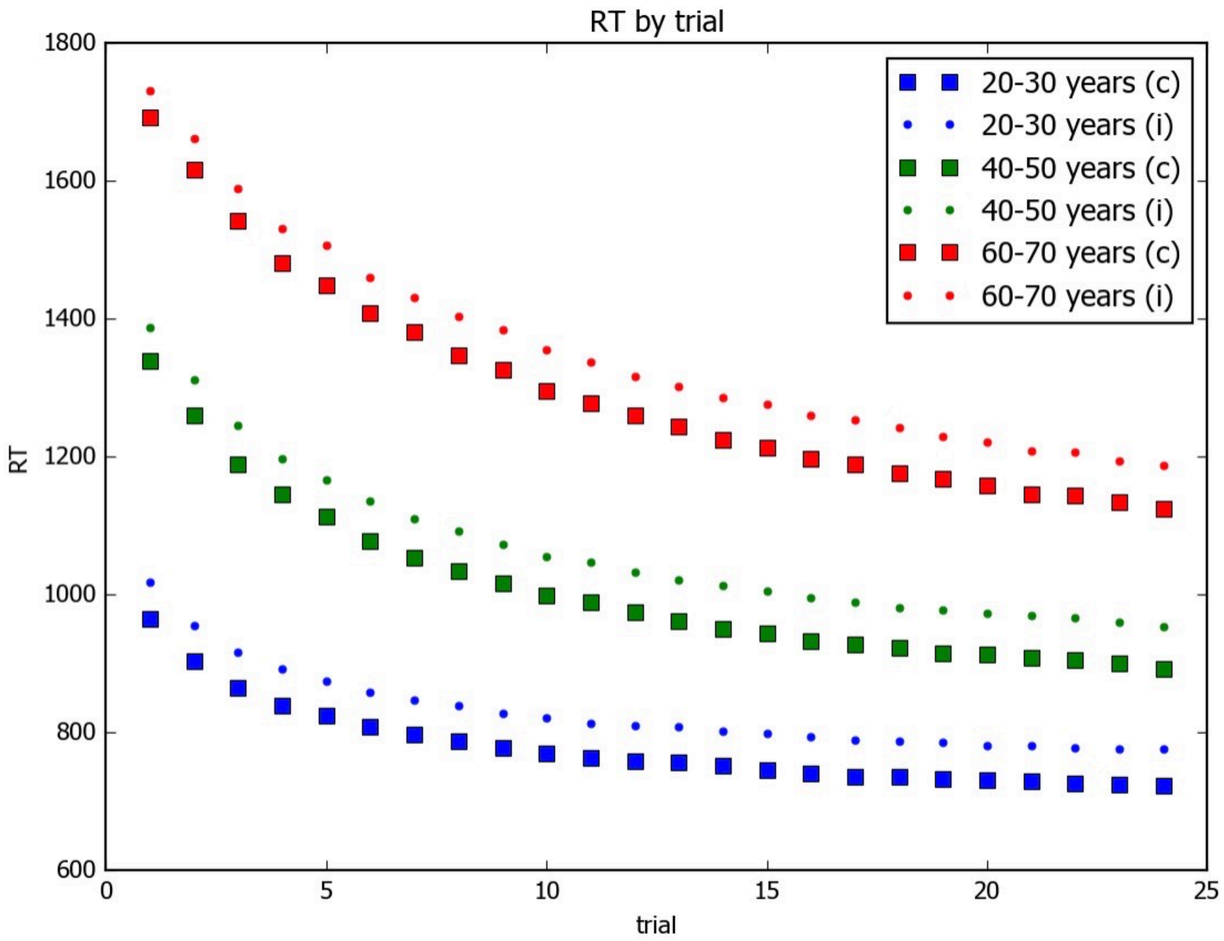
Jupyter - Beginning an analysis

```
# plotting stuff
import seaborn as sns
sns.set_style('whitegrid')
sns.set_style("whitegrid", {'axes.grid' : False})

import matplotlib
import matplotlib.pyplot as plt

# plot settings
font = {'family' : 'Bitstream Vera Sans',
        'weight' : 'regular',
        'size'   : 13}
figure = {'figsize' : (16,8)}

matplotlib.rc('font', **font)
matplotlib.rc('figure', **figure)
```

figure parameters

Jupyter - Figure parameters

```
rcParams['figure.figsize'] = 8, 6
rcParams['font.family'] = 'sans-serif'
rcParams['font.sans-serif'] = ['Tahoma']
```

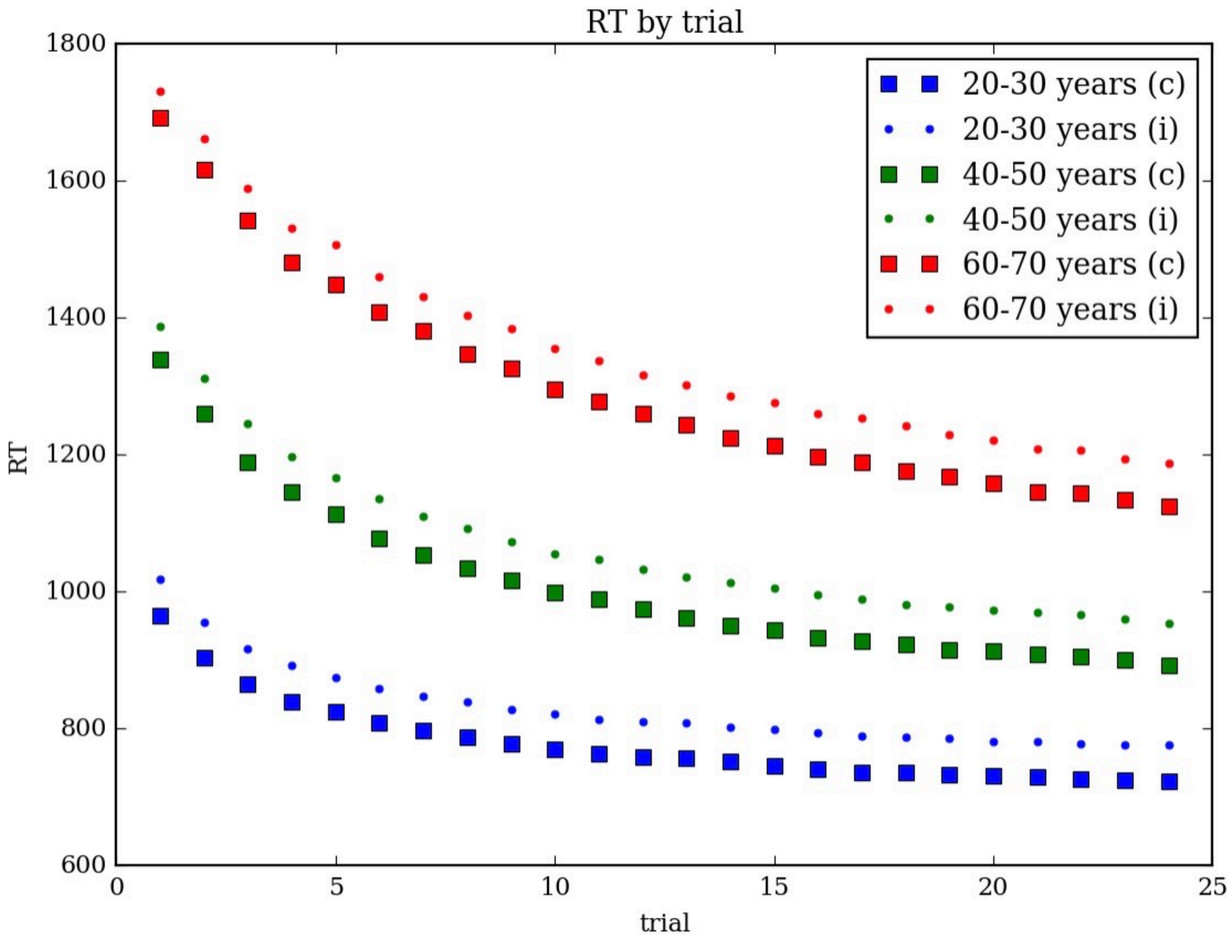


Jupyter - Figure parameters

```
rcParams['figure.figsize'] = 8, 6
rcParams['font.family'] = 'sans-serif'
rcParams['font.sans-serif'] = ['Tahoma']
```

```
rcParams['figure.figsize'] = 8, 6
rcParams['font.family'] = 'serif'
rcParams['font.sans-serif'] = ['Tahoma']
```

serif font now

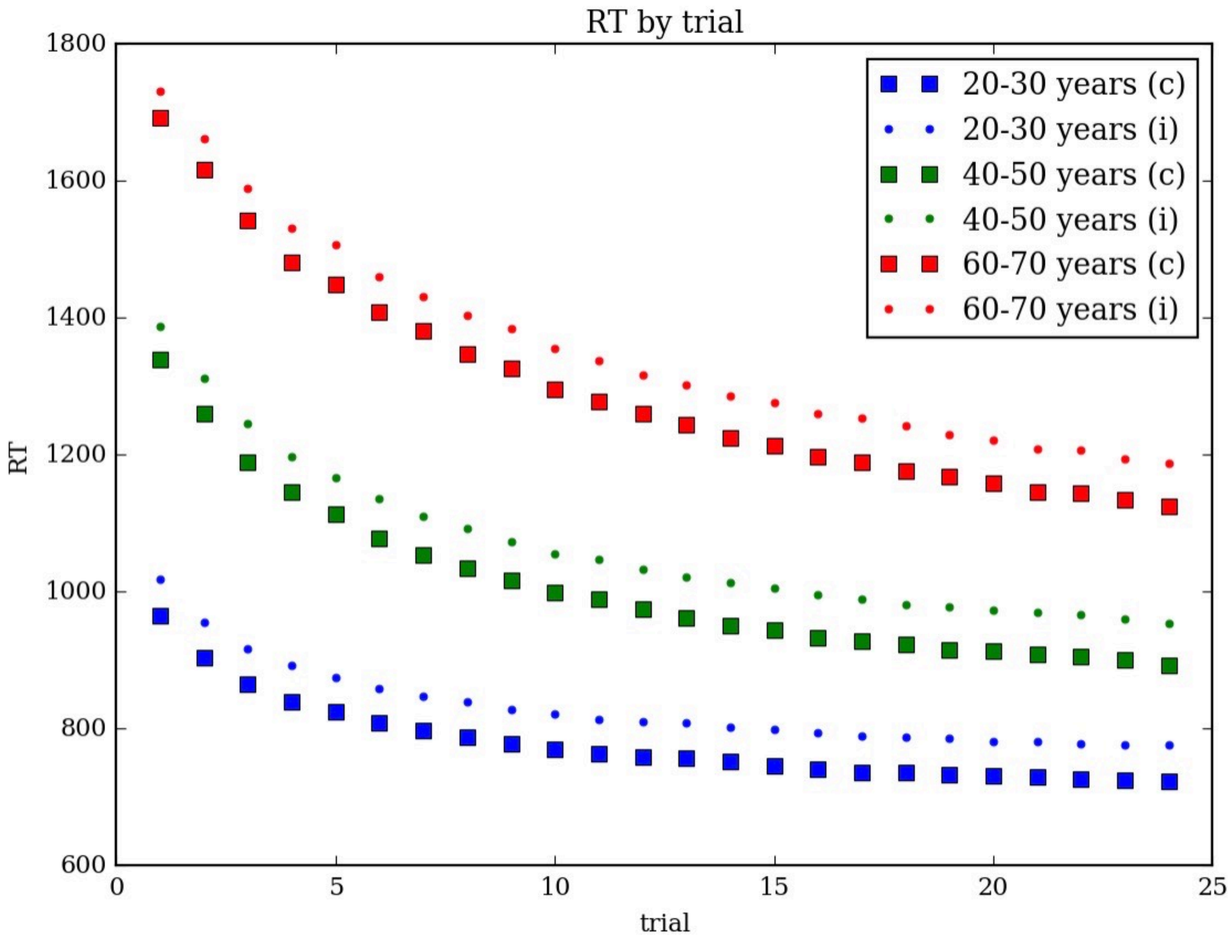


Jupyter - Figure parameters

```
rcParams['figure.figsize'] = 8, 6
rcParams['font.family'] = 'sans-serif'
rcParams['font.sans-serif'] = ['Tahoma']
```

```
rcParams['figure.figsize'] = 8, 6
rcParams['font.family'] = 'serif'
rcParams['font.sans-serif'] = ['Tahoma']
```

despite this saying sans serif



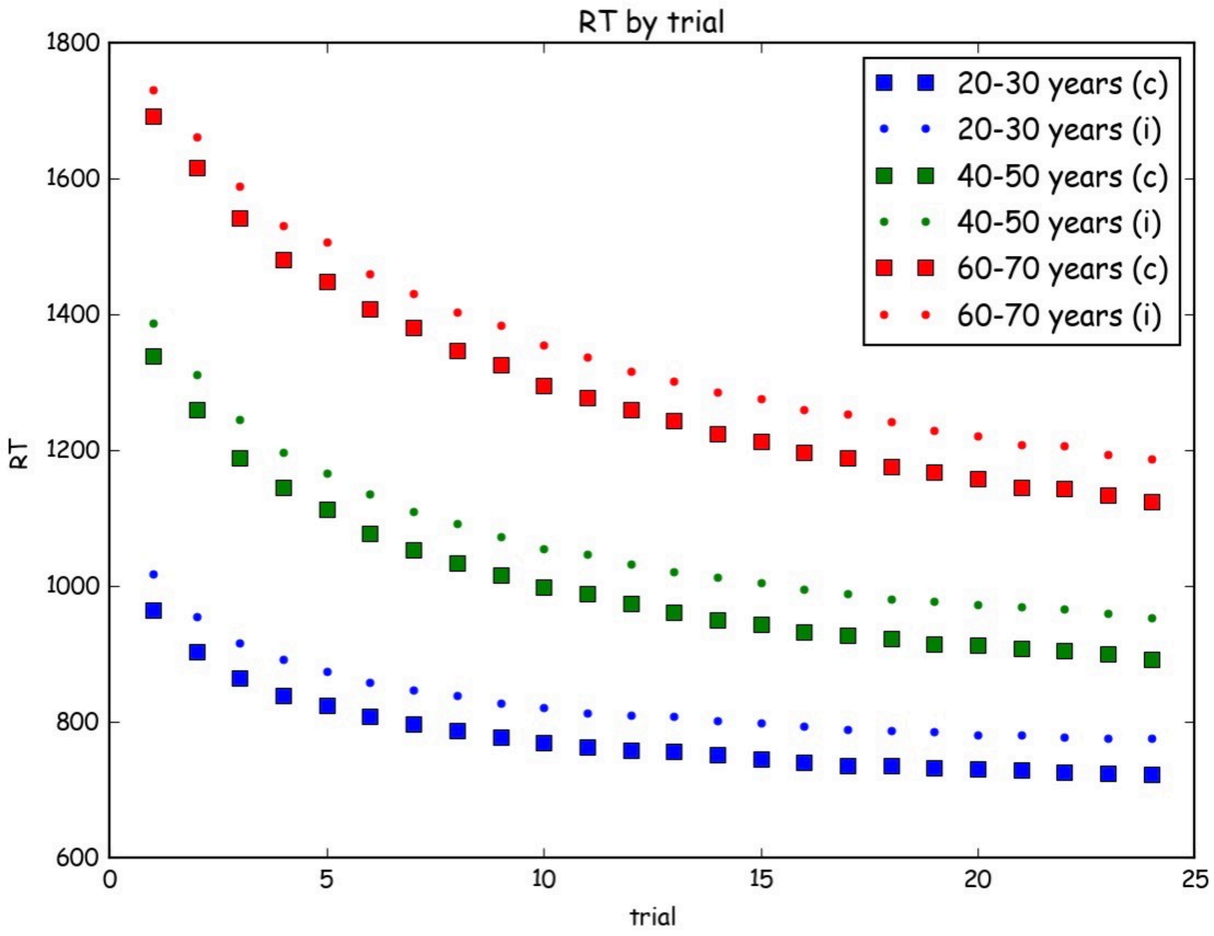
Jupyter - Figure parameters

```
rcParams['figure.figsize'] = 8, 6
rcParams['font.family'] = 'sans-serif'
rcParams['font.sans-serif'] = ['Tahoma']
```

```
rcParams['figure.figsize'] = 8, 6
rcParams['font.family'] = 'serif'
rcParams['font.sans-serif'] = ['Tahoma']
```

```
rcParams['figure.figsize'] = 8, 6
rcParams['font.family'] = 'sans-serif'
rcParams['font.sans-serif'] = 'Comic Sans MS'
```

comic sans ftw!



Jupyter - Figure parameters

NOTE! I didn't restart the jupyter kernel before plotting again, meaning it's still plotting in comic sans!

```
rcParams['figure.figsize'] = 8, 6
rcParams['font.family'] = 'sans-serif'
rcParams['font.sans-serif'] = 'Comic Sans MS'
```

```
rcParams['figure.figsize'] = 4, 3
rcParams['font.family'] = 'sans-serif'
rcParams['font.sans-serif'] = ['Tahoma']
```

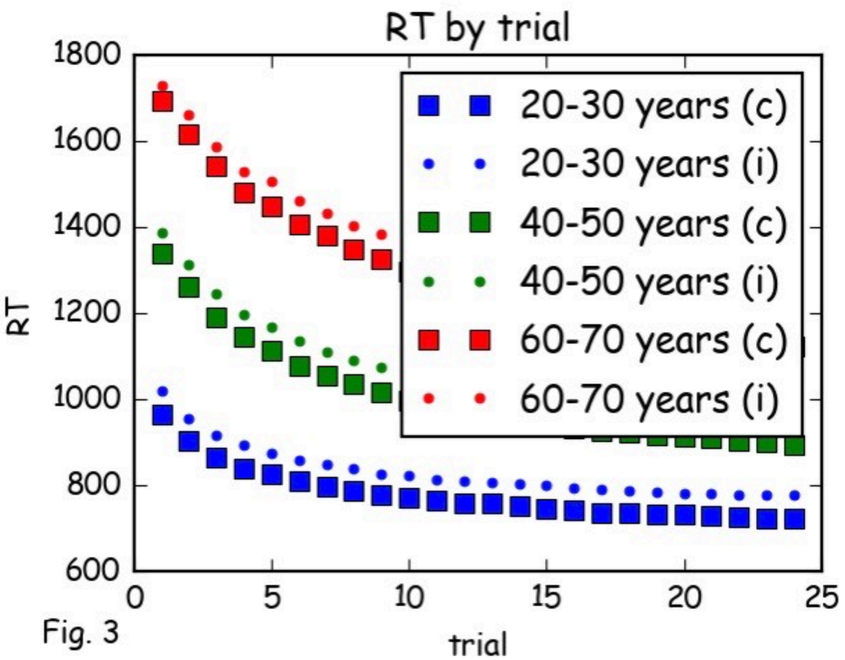


Fig. 3