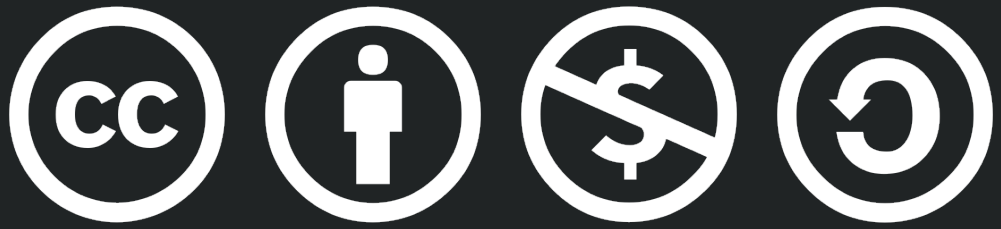
PyCon Canada 2016 Presentation

Quantifying the visual structure of written language

by Nick Anderegg



Quantifying the visual structure of written language

Nick Anderegg

Outline

- Research background
- My research
- Computational motivation
- Algorithm
- Take-aways

Background

Why study reading?

- Understand reading disabilities
- Improve natural language processing
- It's interesting

English reading

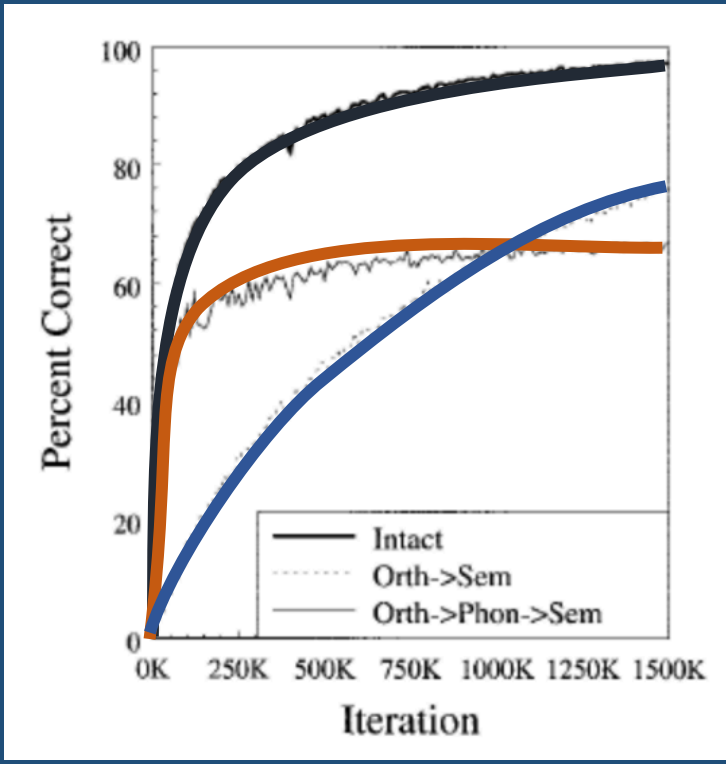
- Cooperative division of labor
- Depends on ease of computation on each route
- Neither route is perfect

Orthography

≈ The written form of a language

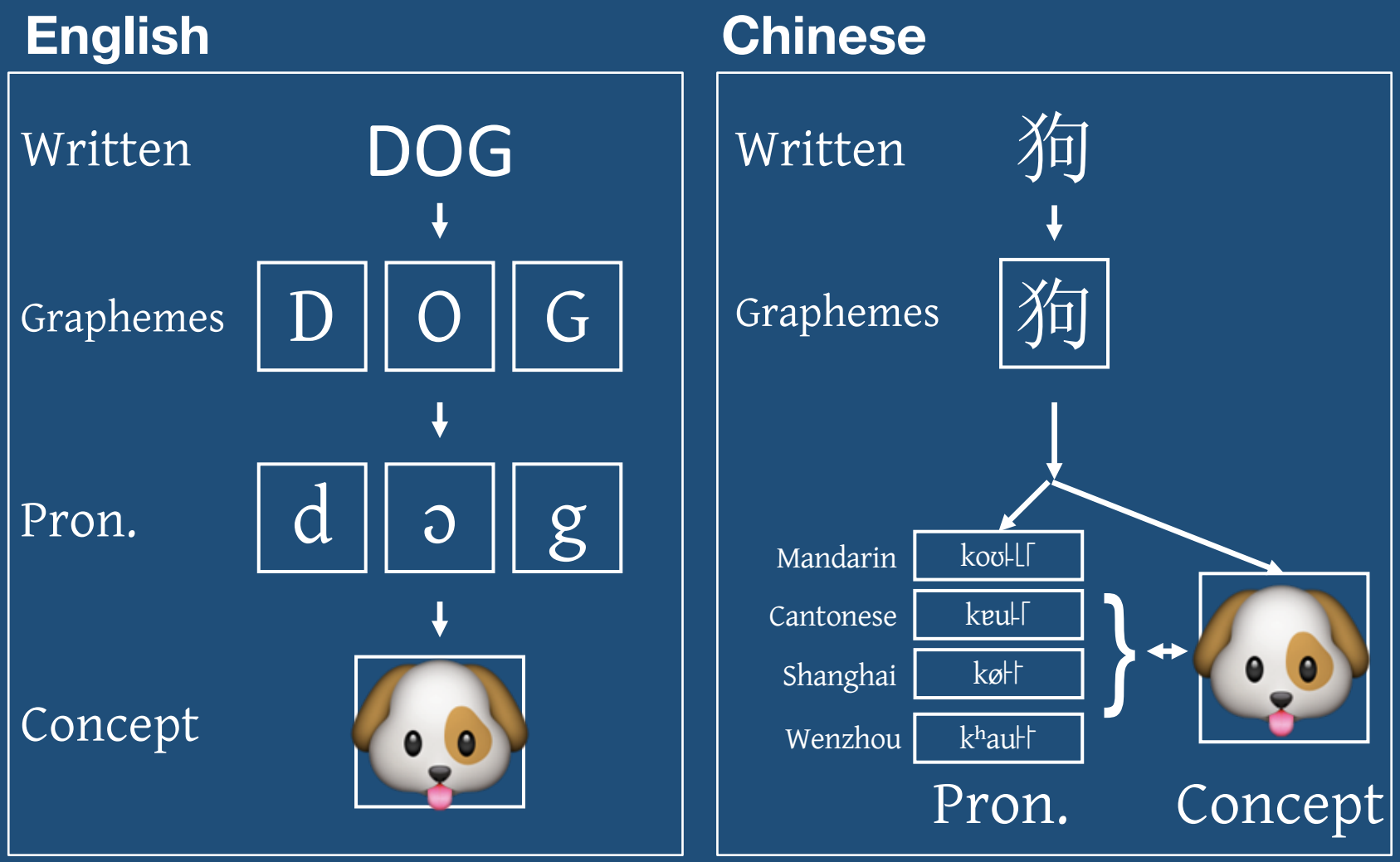
Phonology

≈ The sound system of a language



Harm and Seidenberg (2004)

Finding a universal model



Encoded information: Phono-semantic compounds

- Two indicators/radicals
- Sub-character phonology
  - Imperfect
  - Inconsistent

mō “to feel”

扌hand

莫 mò

mò “sword”

钅 metal

莫 mò

Why compare these?

English

- Computation of phonology from written form
- Does phonology mediate meaning?

Chinese

- Mixed evidence for sub-character phonology
- Directly encodes meaning information

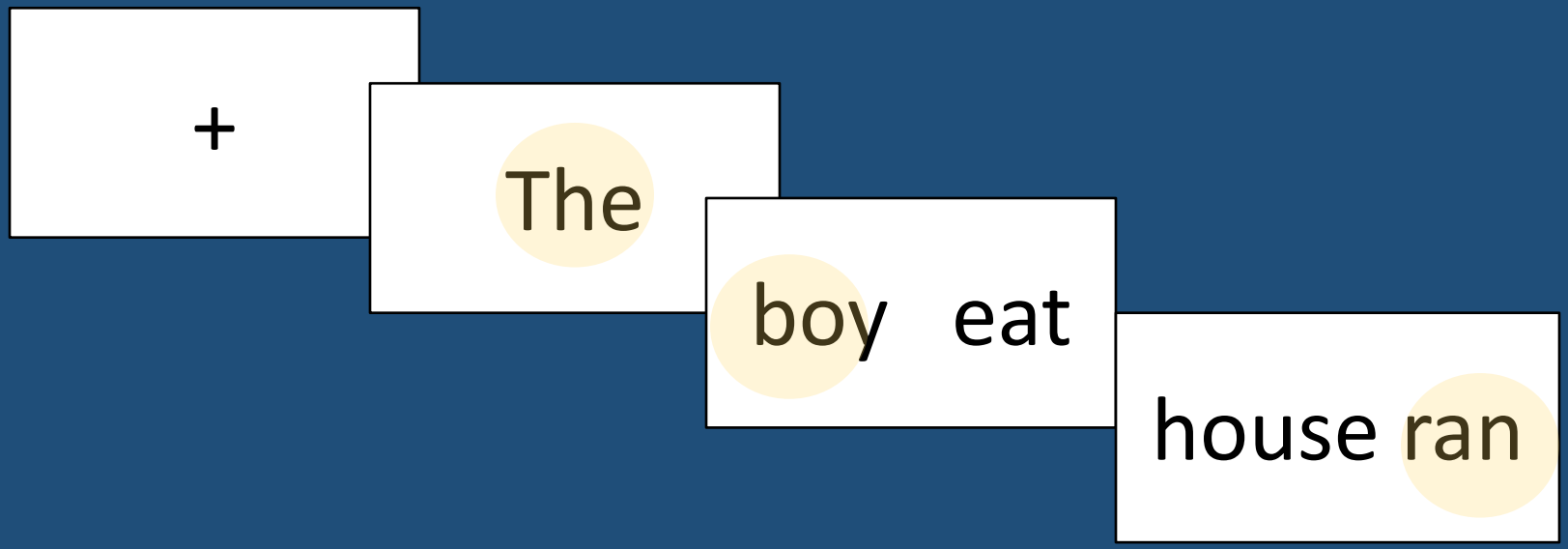
Both languages

- Relative activation of semantics by written form and phonological form
- “Division of labor”

Research

Current research

- Different methods find different processing times
  - Debate over the use of the retrieved phonological information
- Examine the specific role of phonology in word recognition
- Functionally isolate routes
  - Agreement about phonological activation…
  - …but disagreement about its role



The	boy	house	very	far	jump	school
	eat	ran	car	blue	after	read

SHE ###

RAN DOG

BED FOR

SEVEN JUMP

MILES POUNDS

Critical manipulation

- Orthographically similar / Phonological dissimilar

水 永

shuǐ yǒng

English illustration:

beach bench

Critical manipulation

- Orthographically similar / Phonological dissimilar

水 永

shuǐ yǒng

English illustration:

beach

The ###

man sit

red dug

in of

the out

sail soil

Motivation

Quantifying Visual Similarity

- A few different methods used in past
- Most common:
  - Same radical
  - Same number of strokes

An imperfect method

Radical: 立

Characters containing this radical (w/ 9 strokes):

亲 竑 竓
竚 竛 竖

An imperfect method

Radical: 立

Good pairs:

竑-讽

讼-竑

Bad pairs:

亲-讼

竑-竖

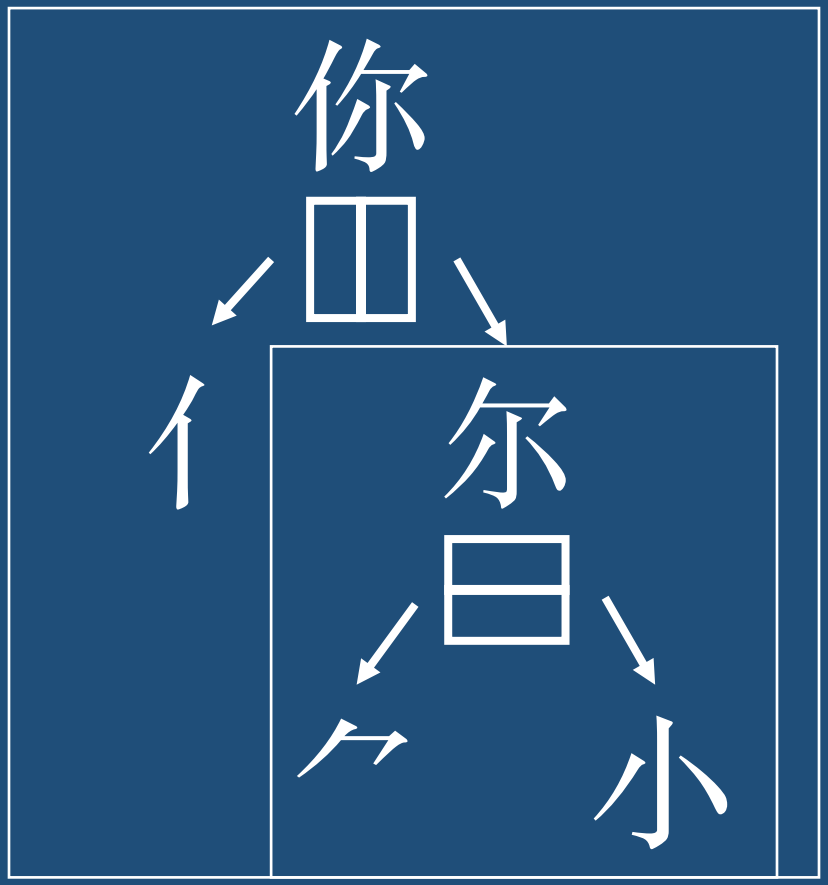
9 strokes

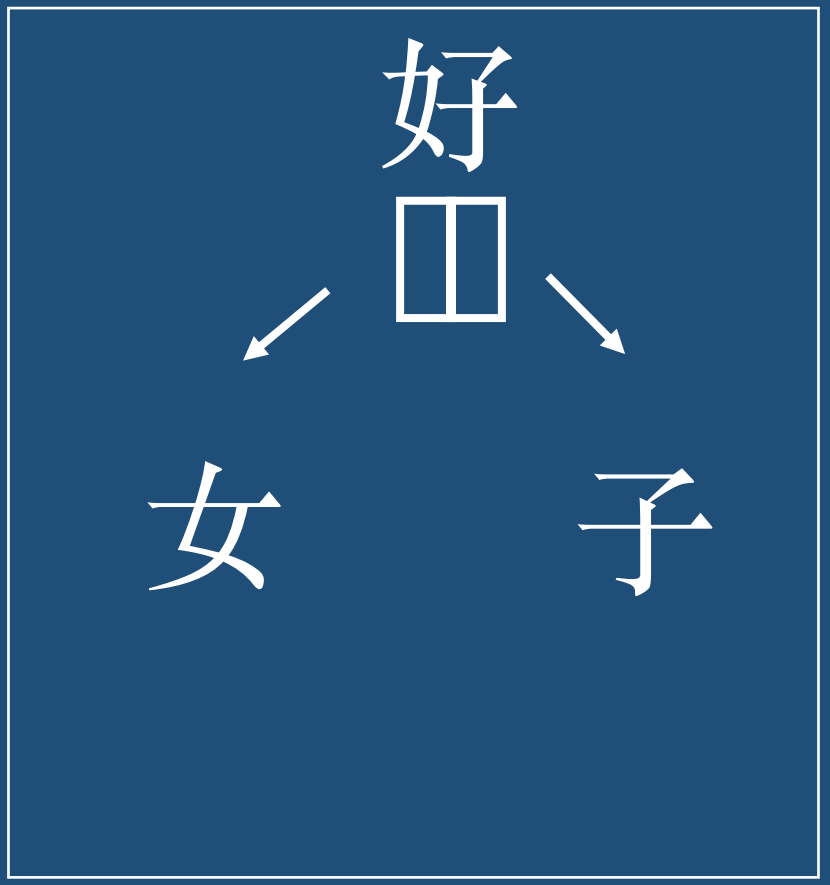
EAT

9 strokes???

Hierarchical structure

你好!





Ideographic Description Characters

- Unicode range U+2FF0 to U+2FFB
- Intended to act as a rough description of characters

	left to right		surround from below
	above to below		surround from left
	left to middle and right		surround from upper left
	above to middle and below		surround from upper right
	full surround		surround from lower left
	surround from above		overlaid

Ideographic Description Language

你 →  亻尔

尔 →  𠂉小

兹 →  𠂇  幺幺

Algorithm

Parsing IDS BNF

```
char      :: all valid CJK
binary    :: 2FF0 |2FF1| 2FF4..2FFB
ternary   :: 2FF2 |2FF3
sub       :: tergroup|bigroup|char
bigroup   :: binary sub sub
tergroup  :: ternary sub sub sub
expr      :: [ sub ]+
```

```
>>> import idsparser as ids
>>> ids.parse('丽', '⿱一⿰口⿰口冂丶⿰口冂丶')
['丽', ['⿱', ['一', ['⿰', [['⿰', ['冂', '丶']],
['⿰', ['冂', '丶']]]]]]]
>>> ids.pretty_parse('丽', '⿱一⿰口⿰口冂丶⿰口冂丶')
丽
└──⿱
    ├──一
    └──⿰
        ├──⿰
        │   ├──冂
        │   └──丶
        └──⿰
            ├──冂
            └──丶
```

Similarity algorithm

- Tree generation
- Structural weighting
- Root comparison
- Comparison reuse

Tree generation

```
class IDSTree():

def __init__(self, parse_tree, dictionary=None):
    self.head = parse_tree[0]
    self.ids = idsparser.unparse(parse_tree[1])
    self.tree = IDSNode(parse_tree[1], self._dict)

    self._dict = dictionary
    if self._dict:
        self._dict._nodes[self.head] = self.tree
        self._dict._nodes[self.ids] = self.tree
```

Tree generation

```
class IDSNode():

def __init__(self, parse_tree, dictionary=None):
    self.head = None
    self.ids = None
    self.children = None

    self._dict = dictionary
```

Tree generation

- If head of parse tree is IDS functor:
  - Assign functor to `self.head`
  - Assign flat sequence to `self.ids`
  - Assign tree children to `self.children` list as recursive *`IDSNode`*s
- If head of parse tree is not functor:
  - Leaf has been reached
  - Assign character to `self.head`
  - Assign character to `self.ids`

Structural weighting

```
self.weights = {
    '┐': (1/3, 2/3),
    '┘': (1/3, 2/3),
    '┌': (1/3, 2/3),
    '└': (1/3, 2/3),
    '┌': (1/3, 2/3),
    '┌': (1/3, 2/3),
    '└': (1/3, 2/3),
    '┐': (1/2, 1/2)
}
```

```
for x in functors.binary:
    self.weights[x] = (1/2, 1/2)
for x in functors.ternary:
    self.weights[x] = (1/3, 1/3, 1/3)
```

Root comparison

```
def compare_nodes(a, b):
    If a is b... #same tree
    If a.head == b.head...
        If a.head not in functors
            and b.head not in functors
        Else recursive comparison
    If a.head != b.head...
```

Comparison reuse

```
class IDSTree():

def __init__(self, parse_tree, dictionary=None):
    self.head = parse_tree[0]
    self.ids = idsparser.unparse(parse_tree[1])
    self.tree = IDSNode(parse_tree[1], self._dict)

    self._dict = dictionary
    if self._dict:
        self._dict._nodes[self.head] = self.tree
        self._dict._nodes[self.ids] = self.tree
```

Tweaking the algorithm

- Pure structural weighting resulting in many leaves with 0 similarity
- Radical similarity account for 1/3 of similarity
- Stroke proportion accounts for 1/3 of similarity

Examples

Before change:

After change:

0%

75%

Distributed processing

- Many individual cloud servers
- Elasticsearch database storing IDS comparisons
- 2000 chars = 1999000 comparisons
- 5000 chars = 12497500 comparisons

- At 10ms per comparison, 34 hours

Creating critical sets

- Pandas dataframe
- Sort by ortho sim, phono sim

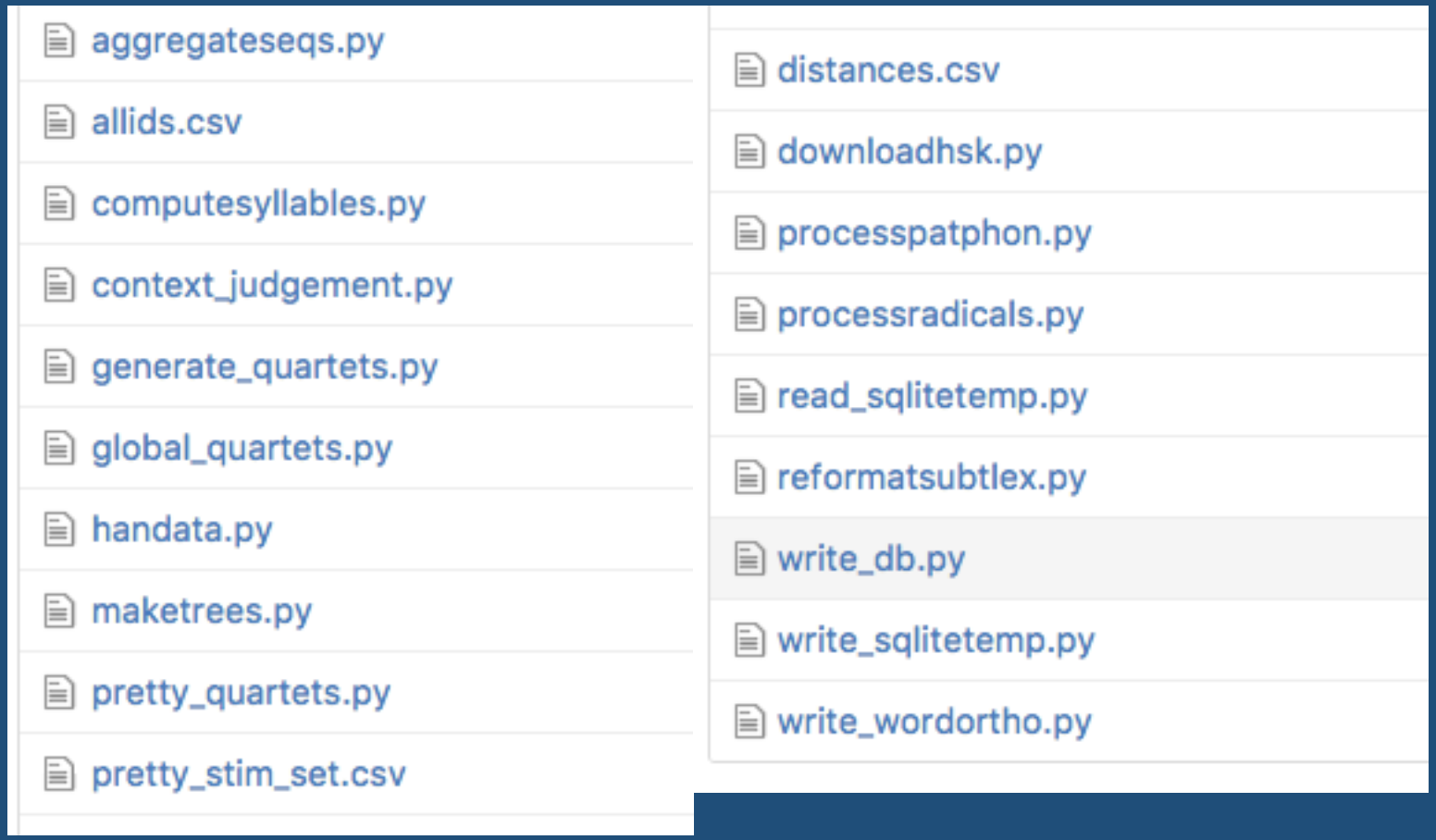
Critical:	治	zhi4	Aggregate: 10.402752739414652
Both sim:	沼	zhao3	Score: 78.4634
Ortho sim:	沿	yan2	Score: 78.4203
Phono sim:	置	zhi4	Score: 96.898
Both diff:	阉	yan1	Score: 78.0193

Thoughts

Python in Academia

- Why Python?
  - Easy to experiment
  - Lots of flexibility
  - Plan of attack isn't always clear when exploring a new idea

Python in Academia



Why do things programmatically?

- Reproducibility!
- A digital trail exists for all tools in the experiment design pipeline

- Flexibility!
- A small change in the pipeline doesn't require starting from scratch

Take away message

- Python can be used to explore new solutions to old problems
- Easily enables experimentation
- Tools exist for every sub-problem!