Lecture 17: Computational Efficiency, OnDemand on CRC

LING 1340/2340: Data Science for Linguists

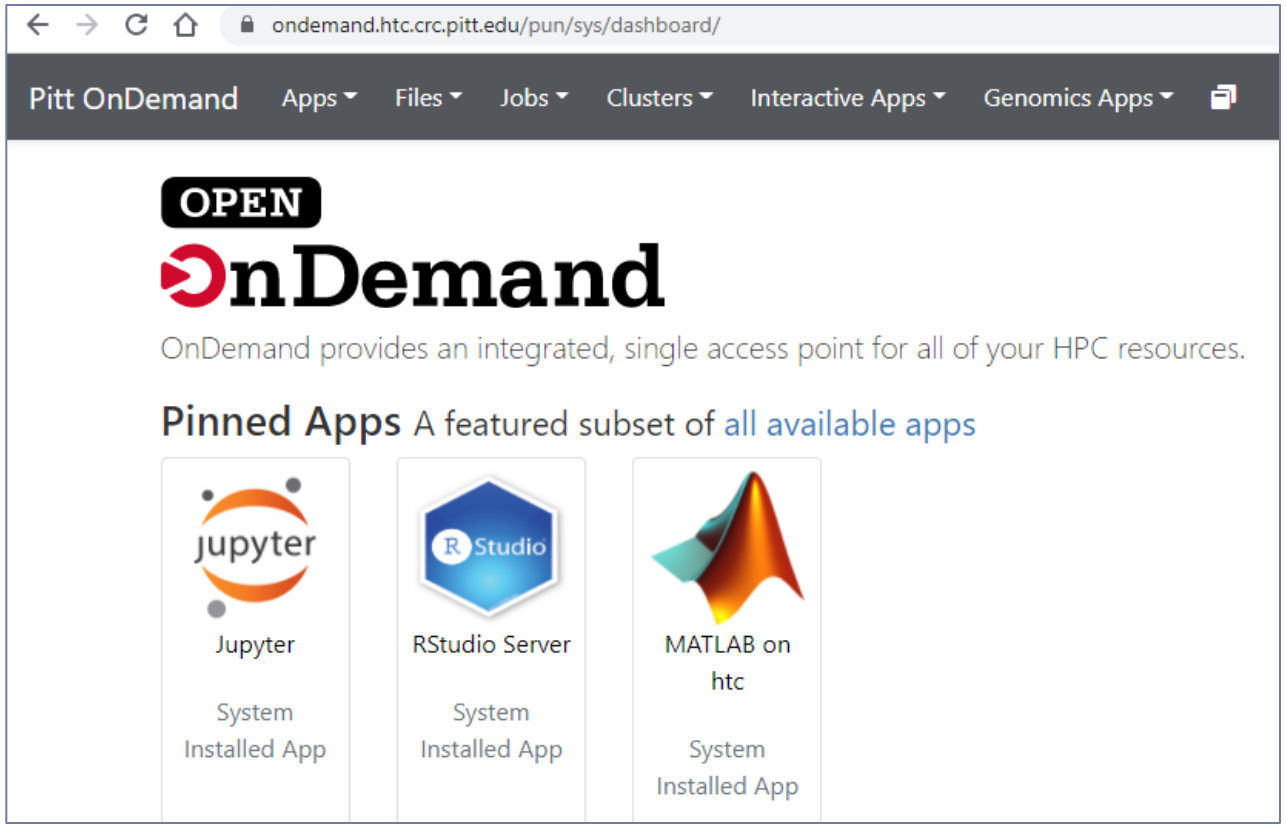
Na-Rae Han

Objectives

- OnDemand platform & JNB at CRC (GUI!)
- Computational efficiency
  - Memory vs. processing time
  - Algorithmic complexity
  - Big O notation

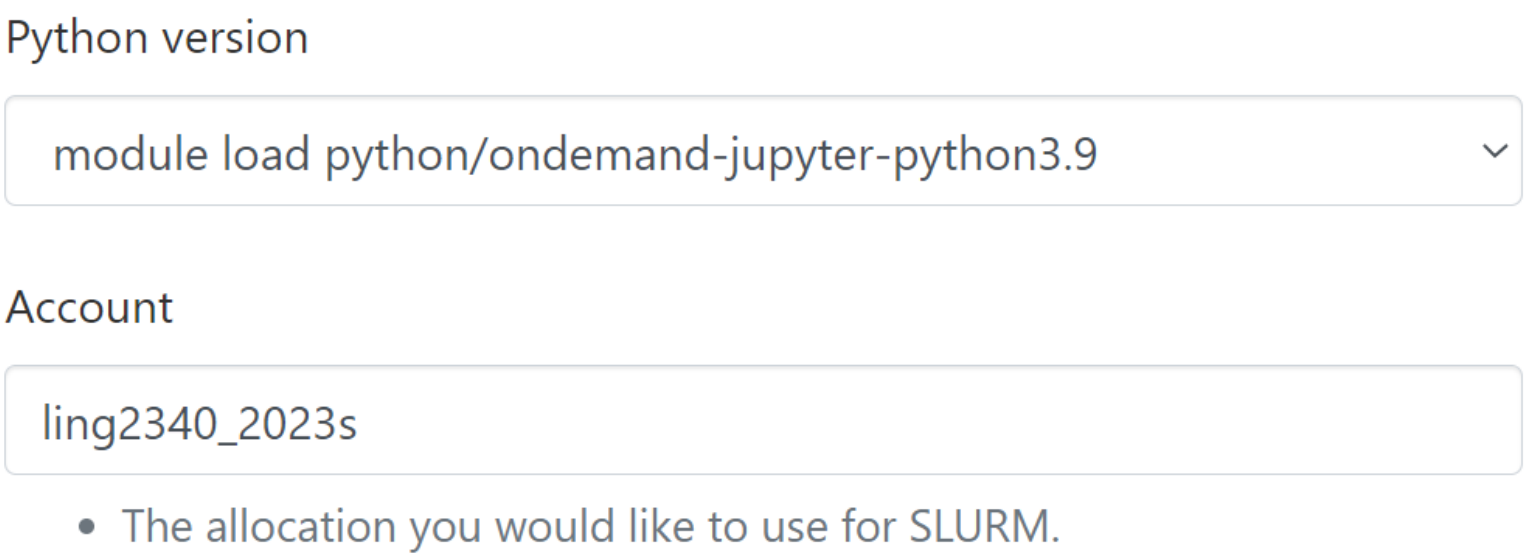
OnDemand on CRC!

- Browser-based gateway to CRC resources!
  - https://ondemand.htc.crc.pitt.edu/
- Jupyter Notebook (Lab) etc. are available!
- Help documentation is somewhat buried:
  - https://crc.pitt.edu/resources/htc-cluster/open-ondemand



Launching a session

- Python version: `module load python/ondemand-jupyter-python3.9`
- Account: `ling2340_2023s`



Jupyter (663715) 1 node | 1 core | Running

Host: >_htc-n16.crc.pitt.edu

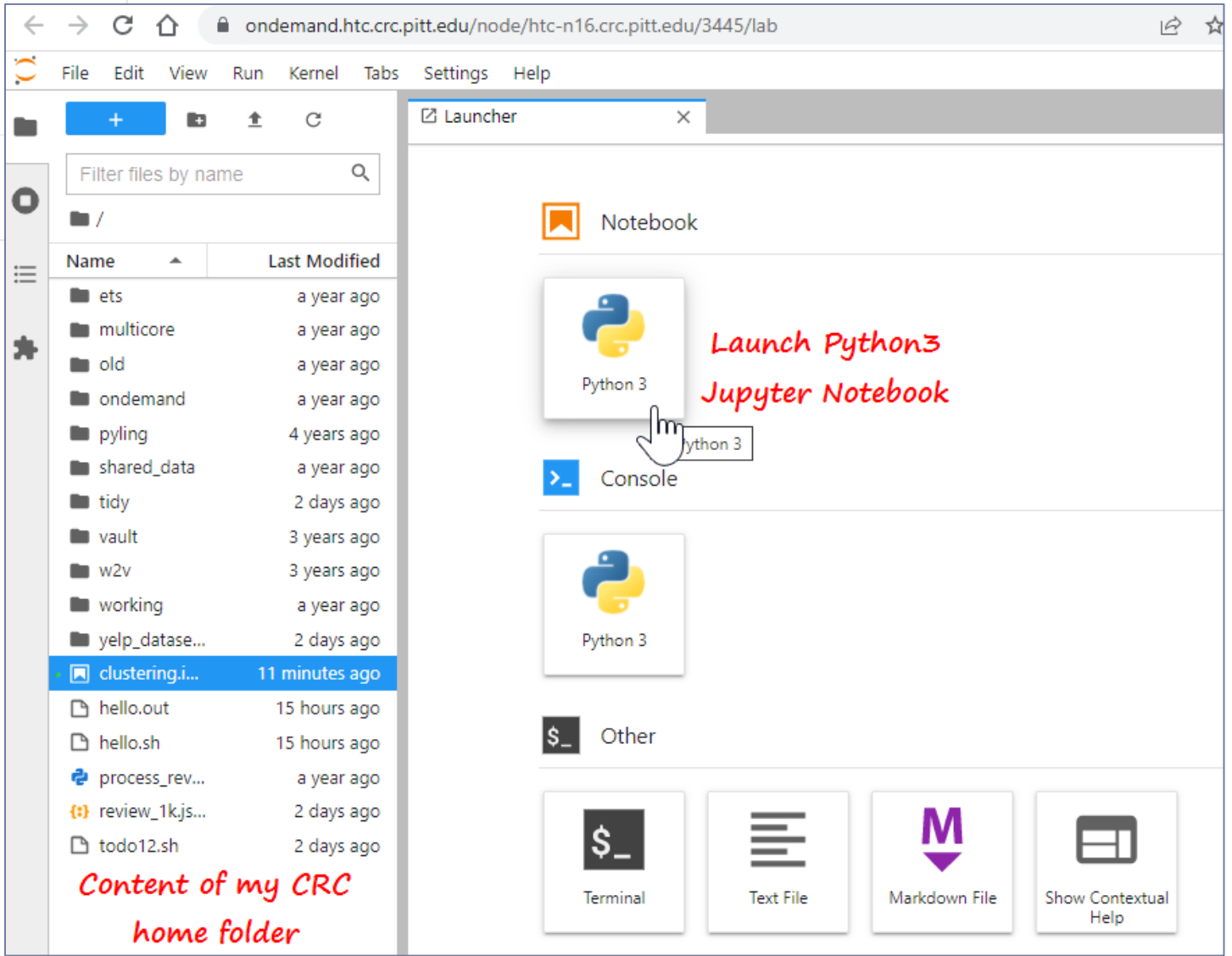
Delete

Created at: 2022-03-31 14:11:10 EDT

Time Remaining: 59 minutes

Session ID: 4f790652-695e-4043-9e38-27d4b9ace746

Connect to Jupyter



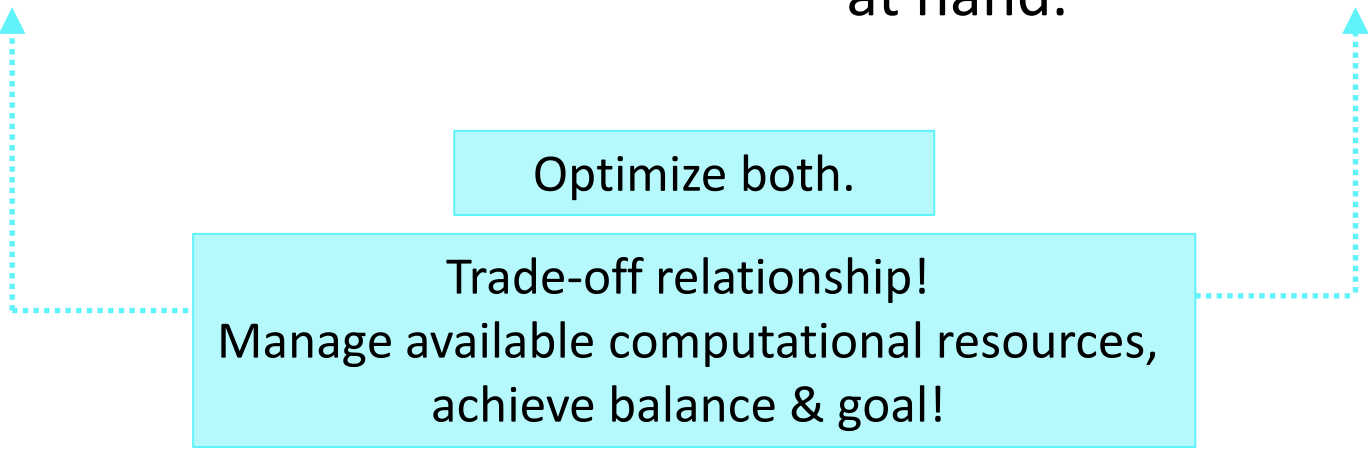
Computational efficiency: space vs. time

SPACE: **memory** footprint

- Do not create duplicate data objects.
- Avoid creating a data object that does not need to be stored in its entirety.
- Avoid creating interim, single-use data objects.

TIME: **processor** runtime

- Avoid duplicating an expensive processing step: process once, store result as an object, then reuse.
- Use an efficient algorithm.
- Use the data type optimal for the task at hand.



Data types and optimization

[1]:
```
import nltk
nltk.data.path.append('/zfs2/ling1340-2019s/shared_data/nltk_data')
```

[2]:
```
from nltk.corpus import gutenberg
%pprint
gutenberg.fileids()
```

```
Pretty printing has been turned OFF
```

[2]:
```
['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', 'bible-kjv.txt', 'blake-poems.txt', 'bryan
t-stories.txt', 'burgess-busterbrown.txt', 'carroll-alice.txt', 'chesterton-ball.txt', 'chesterton-brown.tx
t', 'chesterton-thursday.txt', 'edgeworth-parents.txt', 'melville-moby_dick.txt', 'milton-paradise.txt', 's
hakespeare-caesar.txt', 'shakespeare-hamlet.txt', 'shakespeare-macbeth.txt', 'whitman-leaves.txt']
```

[3]:
```
awords = gutenberg.words('carroll-alice.txt')
print(awords[:100])
print(len(awords))
```

```
['[', 'Alice', "'", 's', 'Adventures', 'in', 'Wonderland', 'by', 'Lewis', 'Carroll', '1865', ']', 'CHAPTE
R', 'I', '.', 'Down', 'the', 'Rabbit', '-', 'Hole', 'Alice', 'was', 'beginning', 'to', 'get', 'very', 'tire
d', 'of', 'sitting', 'by', 'her', 'sister', 'on', 'the', 'bank', ',', 'and', 'of', 'having', 'nothing', 't
o', 'do', ':', 'once', 'or', 'twice', 'she', 'had', 'peeped', 'into', 'the', 'book', 'her', 'sister', 'wa
s', 'reading', ',', 'but', 'it', 'had', 'no', 'pictures', 'or', 'conversations', 'in', 'it', ',', "'", 'an
d', 'what', 'is', 'the', 'use', 'of', 'a', 'book', ",'", 'thought', 'Alice', "'", 'without', 'pictures', 'o
r', 'conversation', "?'", 'So', 'she', 'was', 'considering', 'in', 'her', 'own', 'mind', '(', 'as', 'well',
'as', 'she', 'could', ',']
34110
```

[4]:
```
enable = open('/zfs2/ling1340-2019s/shared_data/enable1.txt').read().split()
print(enable[:30])
print(len(enable))
```

```
['aa', 'aah', 'aahed', 'aahing', 'aahs', 'aal', 'aalii', 'aaliis', 'aals', 'aardvark', 'aardvarks', 'aardwo
lf', 'aardwolves', 'aargh', 'aarrgh', 'aarrghh', 'aas', 'aasvogel', 'aasvogels', 'ab', 'aba', 'abaca', 'aba
cas', 'abaci', 'aback', 'abacterial', 'abacus', 'abacuses', 'abaft', 'abaka']
172820
```

"Alice in Wonderland", 34K tokens

"Enable" word list, 173K total words

Task: find Alice words that are not found in enable list

- Try 1: list-comprehend through awords (list), filter against enable (list)

```
[5]: %time notfound = [w for w in awords if w not in enable]
```

- Try 2: same, but filter against enable list as a SET

```
[6]: enable_set = set(enable)  # this one is a set data type
     print(len(enable_set))    # same size
```

```
172820
```

```
[7]: %time notfound = [w for w in awords if w not in enable_set]
```

- Try 3: both as SETS, compute the set difference

```
[8]: awords_set = set(awords)
     print(len(awords_set))    # now a set, smaller size
```

```
3016
```

```
[9]: %time notfound = awords_set.difference(enable_set)
```

- Try 1: list-comprehend through awords (list), filter against enable (list)

```
[5]: %time notfound = [w for w in awords if w not in enable]

CPU times: user 39.7 s, sys: 16.2 ms, total: 39.7 s
Wall time: 39.8 s
```

- Try 2: same, but filter against enable list as a SET

```
[6]: enable_set = set(enable)  # this one is a set data type
     print(len(enable_set))    # same size

172820

[7]: %time notfound = [w for w in awords if w not in enable_set]

CPU times: user 20.3 ms, sys: 4 ms, total: 24.3 ms
Wall time: 24.8 ms
```

much faster

List as a data type is NOT optimized for membership operations…

but **set** is!

- Try 3: both as SETS, compute the set difference

```
[8]: awords_set = set(awords)
     print(len(awords_set))    # now a set, smaller size

3016

[9]: %time notfound = awords_set.difference(enable_set)

CPU times: user 332 µs, sys: 2 µs, total: 334 µs
Wall time: 339 µs
```

blazing fast

Keep efficiency in mind, pick right combination of data structure and operation

Algorithmic complexity and the Big O

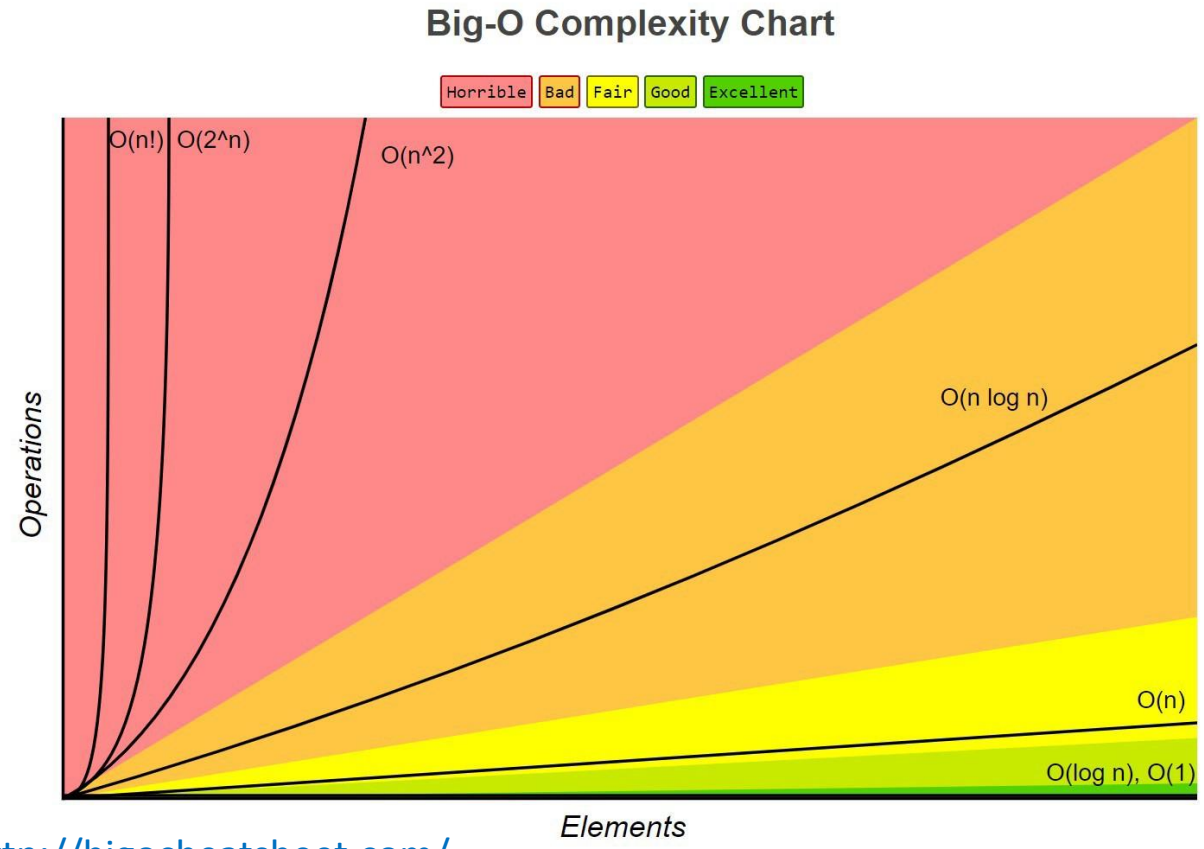
- https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/
- We have a **list of *n* items**. Imagine *n* is 100, 1000 or even 1 million.

1. Is the first element an even number?
   - Can be implemented in O(1): an algorithm that executes in a **constant** time regardless of the size of the input dataset.
2. Does the list contain value 42?
   - Can be implemented in O(n): an algorithm whose performance will grow **linearly** in proportion to the size of the input data.
3. Does the list contain duplicate values?
   - Can be implemented in $O(n^2)$:an algorithm whose performance is directly proportional to the square of the size of the input data set (**quadratic**).

Algorithmic complexity and the Big O

- https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/
- We have a **list of *n* items**. Imagine *n* is 100, 1000 or even 1 million.

4. Sort the list (ascending or descending)
   - Can be implemented in O(n log n): an algorithm that executes in **loglinear** time.
   - See: https://brilliant.org/wiki/sorting-algorithms/



http://bigocheatsheet.com/

Algorithmic efficiency: summary

- A problem can be implemented with varying degrees of algorithmic efficiency.
- A problem comes with its own inherent algorithmic complexity limit.
  - **Big O notation** is a mathematical notation that encapsulates the relationship between the processing time and the input data size.
  - Example: the most efficient known sorting algorithm bottoms out at O(n log n).
- In a nutshell…
  - Compose the most efficient algorithm that you can.
  - Understand the relationship between the data size growth and the processing time growth. O(n) has fair scalability, $O(n^2)$ becomes intractable.
  - Efficiency of an algorithm can lead to dramatic runtime difference when dealing with big data.

Wrap up

- Homework #4 out – don’t be too ambitious!
- Progress report #3, presentation up coming!