The intellect of man is forced to choose

Python for Poets

~~YOUR CODE IS BAD AND YOU SHOULD FEEL BAD~~

Your algorithm choice could be better

https://medium.com/capital-one-tech/heat-death-of-the-universe-and-faster-algorithms-using-python-dict-and-set-f31517e7fa76

Topics

- Floating point and integer numbers and their ranges/precisions
- Gross Orders of Complexity
- Important Rules
- Efficient structures for searching, counting, analytics
    - Building fast indices for searching
- Case Study: simple analytics using Python
    - AKA: "The Poetry Corpus"

Background — On Computing

Things that Matter

- Python is written in C
- NumPy is written in C (mostly)
  - Uses bare-metal data types
  - Imposes a number of limitations
- Pandas uses NumPy

Things that don't matter

Automatic Optimization

- Python doesn't optimize
- If it did debugging would be impossible

Tricks

- There are no secret "turbo-boost" coding tricks
- Use packages like numpy, pandas or numba

Integers

- Python: no limits — two internal representations
  - (fast) 64-bit values
  - (slow) Arrays of values (base 2**30) for numbers > 2**64
- Numpy: int8, int16, int32, int64

Float

$$f \approx \lfloor m_{10} \rfloor \times 10^{e_{10}} \approx \lfloor m_2 \rfloor \times 2^{e_2}$$

- Python: IEEE 754 compliant 64-bit
  - 64-bit values. Source text is decimal `6.0221409e+23`
  - Internal representation is binary $\dfrac{1{,}121{,}711{,}153{,}537{,}035}{2^{50}} \times 2^{79}$
  - Mantissa is ~48 bits = ~16 digits. Exponent is ±300
- Numpy: float32, float64

Float is an Approximation

Infinite Repeating “binary point” Values…
Are truncated

```
>>> 100+1/3-100
0.33333333333333286
>>> 100+1/3-100-1/3
-4.718447854656915e-15
```

Approximation doesn’t match Abstraction

This is going to involve work, right?

Orders of Complexity

$O(1)$	constant time
$O(n)$	scales linearly with the amount of data
$O(\log_2 n)$	scales with the log of the data. This is almost always because of some clever divide-and-conquer search
$O(n \log_2 n)$	sorting and similar algorithms that do repeated searches
$O(n^2)$	compare every item with every other item
$O(2^n)$	whoa! The powerset of all subsets
$O(n!)$	combinatoric explosion — all combinations of items

These are bad

Worst Case — Permutations

$O(n!)$ Permutations — All Possible Orderings

For n=10 items, there are 3,628,800 orderings

This is why we have sophisticated approximation-based algorithms

Optimial solutions would take centuries to find

Really Bad Case — Power Set

$O(2^n)$ powerset — set of all subsets

For a n=10 data set, there are 1,024 different subsets

Let's say you it takes 3 seconds to fetch one of the subsets

Overall? 51 minutes.

A Bad Case — Comparisons

$O(n^2)$ matrix — compare each item against every other item

For n=10,000,000 row dataset, that's 10^{12} operations

Let's say comparison each takes a looong 1 ms (10^{-3} sec)

So. 10^9 seconds

= 32 years

On a 64-core processor, it's only 6 months!

Not Too Bad — Sorting

$O(n \log_2 n)$ sort — each item does a $\log_2 n$ lookup

For a n=10,000,000 row dataset, that's 2.3×10^8 operations

In memory, and each operation takes 0.01 ms (10^{-5} sec)

= 40 minutes

Important Rules

First Rule of Optimization:

Don’t

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."

–Donald Knuth

Don’t optimize something that doesn’t work

- You have an app that’s slow
- You must have rock-solid unit test cases
- Until you have rock-solid unit test cases, DO NOT OPTIMIZE

"it's no sin for an optimizing compiler to make a wrong program worse" — Bill McKeenan

Don't Optimize Until You Profile

- The Pareto Principle
- Most of your program (80%) is fine
- Some small part (20%) of your program uses most of the resources
    - Hint: It's inside a loop somewhere
- Find the one function that's doing the most work; fix only that

Tools Summary

AFTER you get the algorithm right

Some Tools

- `pytest` — You must have unit test cases. You must have code coverage on the things you're going to optimize
- `logging` — Generally identify the likely location of problems
- `profile` — Pinpoint a "hot spot" where performance is really bad
- `timeit` — Explore alternatives to find one that's fastest
- `sys.getsizeof()` — Some sense of the size of an object. See https://docs.python.org/3/library/sys.html#sys.getsizeof
- `%prun` — IPython profiler (there are others)
- `%time` — IPython timing
- `%%timeit` — rich timing details

Algorithm and Data Structure

Or

How Do I Avoid O(n^2) ?

Two species of algorithms

Search & Sort — ∃ — There Exists

- This is where we often wind up with $O(n^2)$ (or worse) kinds of problems
- Algorithm and data structure matters

Everything Else — ∀ — For All (map and reduce)

- This is mostly bulk data movement — *should* be $O(n)$
- Memory matters

General Approaches

- Sets. O(1) Lookup. Size of set doesn't matter
- Dictionaries. O(1) Lookup. Size of dict doesn't matter
- The `bisect` module had $O(\log_2 n)$ search of a `sorted()` list

Some Examples

- Impossible $O(2^n)$ problem
- Case Study: Joins ("lookups") between two structures
  - We'll look at stop-words lookup
  - We'll look at bag-of-words vectorizing

Impossible Problem — Profile All You Want

$$\sum_{x:x \in S} = t$$

Search a set, S, for a subset, s, with a given total, t.

given $S = \{a, b, c, \ldots\}$

find $s \subset S$ where $\sum s = t$

Simple, Elegant, Unscalable

```python
from itertools import chain, combinations

def powerset(iterable):
    "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

def exact_sum(s, target):
    for subset in powerset(s):
        if sum(subset) == target:
            return subset
```

$$O(2^n)$$

Case Study

- Cleaning and Vectorizing Text
- Goal is a word vector for the “interesting” words
- Filter out stop words like “the” and “and”
- This reflects a common design pattern where there’s a lookup from one data structure to items in another
  - $O(n \times m)$ if we’re not careful
- There’s also a parsing aspect to decompose poetry lines to words.

The Poetry Data

https://github.com/aparrish/gutenberg-poetry-corpus

- http://static.decontextualize.com/gutenberg-poetry-v001.ndjson.gz
- https://raw.githubusercontent.com/nltk/nltk_data/gh-pages/packages/corpora/stopwords.zip

A Newline Delimited JSON Reader

```
poetry_path = Path.cwd()/"gutenberg-poetry-v001.ndjson.gz"
def poetry_line_iter(source_path: Path=poetry_path) -> Iterator[str]:
    """Read the GZIP file via a decompressor."""
    with gzip.open(source_path) as source:
        for line in source.readlines():
            json_line = json.loads(line)
            yield json_line['s']
```

Survey the File

```
for p in poetry_line_iter():
    print(p)
```

Reading the GZIP File

- Less I/O (fewer physical pages of data)
- More computation
- Is it worth it?

A Stopword Iterator

```
sw_path = Path.cwd()/"stopwords.zip"
def stopword_iter(source_path: Path=sw_path) -> Iterator[str]:
    """Read a tiny subset of the ZIP file."""
    with zipfile.ZipFile(source_path) as archive:
        with archive.open("stopwords/english") as words:
            for line in words:
                yield line.decode('ascii').rstrip()
    yield from (
        "thy", "thou", "thee", "thus", "oh", "hath", "tis", "us", "forth",
        "thus", “ye", "shall", "thine")
```

Survey the File

```
for s in stopword_iter():
    print(s)
```

Normalizing Words

- We'll strip almost all punctuation
- We can't strip all punctuation — we're vs. were
- Hyphenated words have single-word semantics
- Multiple apostrophes are rare (fo'c'sle, for example)

Words from Each Line

```
def word_iter(text: str) -> Iterator[str]:
    words = re.compile(
 r"[a-z]+[‘’’][a-z]+|[a-z]+(?:-[a-z]+)+|[a-z]+")
     for m in words.finditer(text.lower()):
         yield m.group(0)
```

The Regular Expression

- Kind of complicated
- A lot of computation
- Is it worth it?
  - It turns out, it's hard to do better than this
  - Feel free to try

Survey The File (again)

```
for line in (
        list(word_iter(text))          Second
        for text in poetry_line_iter()  First
):
    print(line)
```

- ['to', 'lonely', 'hamlet', 'and', 'to', 'stirring', 'town']
- ['cheering', 'the', 'wayworn', 'traveller', 'as', 'it', 'flows']
- ['when', 'all', 'the', 'fields', 'with', 'drought', 'are', 'parched', 'and', 'bare']

Remember The Intro?

Then nowise worship dusty deeds,
Nor seek, for this is also sooth,

- The biggest problem is Search
- Searching for Stopwords
- Searching for Vectorization words
- Search, Search, Search

Look for O(n)
Try to replace with O(log n) or O(1)

Removing Stopwords

```
def stopword_filter_1(stopwords: Iterable[str],
words: Iterable[str]) -> Iterable[str]:

sw_list = list(stopwords)

for w in words:

    stop = False

    for sw in sw_list:

        if w == sw:

            stop = True

            break

    if not stop:

        yield w
```

Explicitly

O(n poetry words × m stop words)

Can we do better?

Stopwords 2

```
def stopword_filter_2(stopwords: Iterable[str],
words: Iterable[str]) -> Iterable[str]:

    sw_list = list(stopwords)

     for w in words:

          if w not in sw_list:

              yield w
```

What's O(x) of `in`?

Stopwords 3

```
def stopword_filter_2(stopwords: Iterable[str],
words: Iterable[str]) -> Iterable[str]:

    sw_list = set(stopwords)

     for w in words:

         if w not in sw_list:

             yield w
```

What's O(x) of `in`?

set v. list

- `timeit.timeit("'a' in stop",`
  `    setup="from string import printable; stop=set(printable)")`
- Out[28]: 0.028795376998459687 ← O(1)?
- `timeit.timeit("'a' in stop",`
  `    setup="from string import printable; stop=list(printable)")`
- Out[29]: 0.1707196079987625 ← O(n)?

Hidden Alternative #4

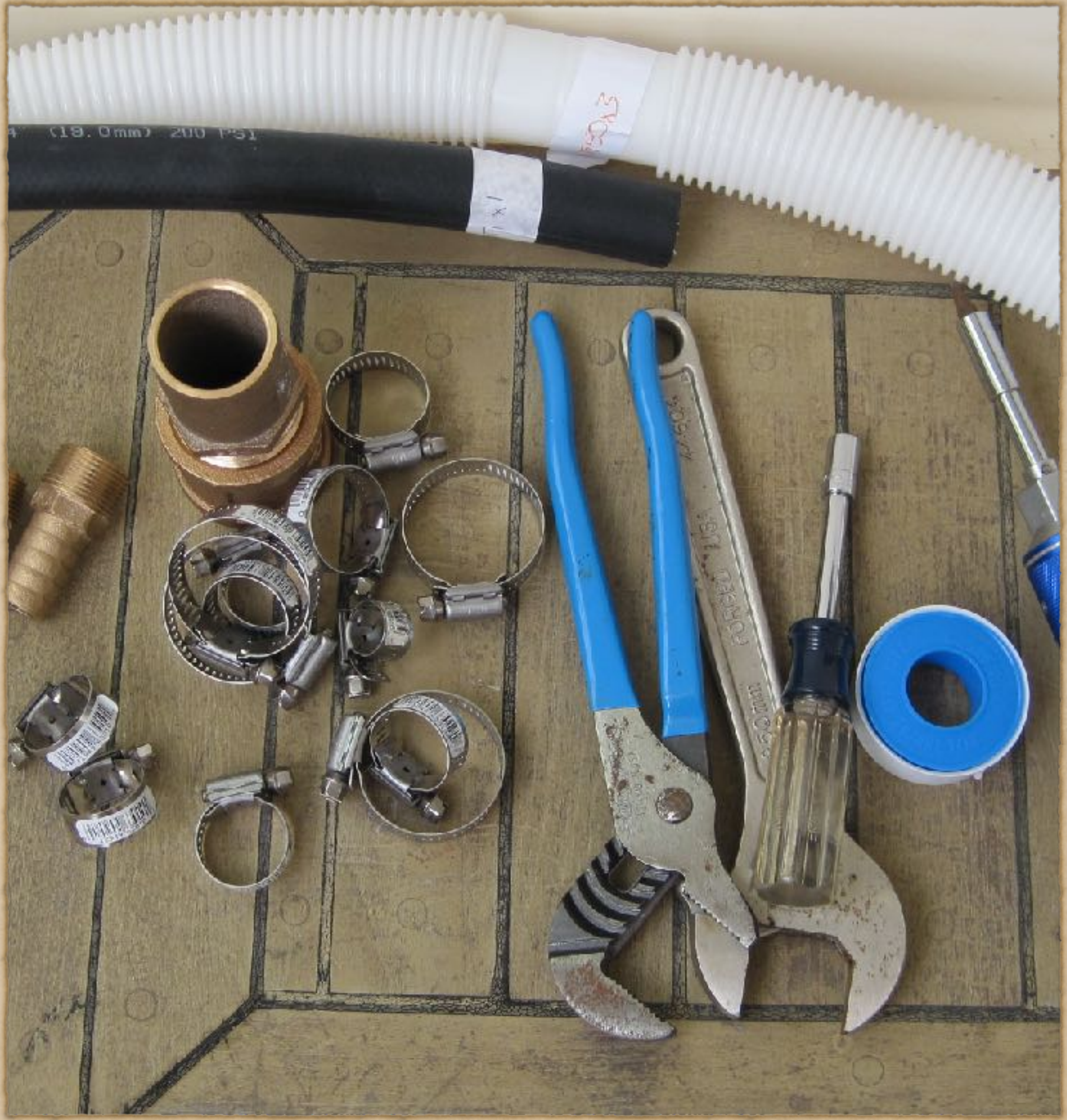
- The `bisect` module

```
array = sorted(items)

array[bisect.bisect(array, value) - 1] == value
```

How to Avoid Search

- Set does hash-based lookup O(1)
- Dict also hash-based O(1)
- bisect is tree-like O(log n)



Next Steps

Combining Things

```
word_bag = Counter(
    stopword_filter_1(
        stopword_iter(), First
        poetry_word_iter(poetry_line_iter())
    )            third             second
)
```

Survey

```
word_bag = Counter(
    stopword_filter_2(
        stopword_iter(),
        poetry_word_iter(poetry_line_iter())
    )
)

word_vector = sorted(
    k for k, v in word_bag.most_common(128))

print(word_vector)
```

Vectorize

```
def vectorize_3(word_vector: Sequence[str], line: str)
-> List[int]:

    index_iter = (
        (bisect(word_vector, w)-1, w)
        for w in word_iter(line))

    valid_index_set = set(i
        for i, w in index_iter if word_vector[i] == w)

    return [1 if i in valid_index_set else 0
        for i in range(len(word_vector))
    ]
```

Conclusion

- Know the access cost for your data structure
- Get to O(1) or O(n) whenever possible
- Avoid O(n×m)
- Compress Data, use Generator Expressions

Appendix

The Big Picture — Memory

less bulk means more cache

Tiers of Memory

Memory	Size	Speed	Applications
Network/Cloud	Vast	Glacial (s)	
Database	Large (Tb - Pb)	Very Slow (ms)	
Local Disk	Large (Tb - Pb)	Slow (μs to ms)	
RAM	Small (Gb - Tb)	Fast (ns to μs)	Large data structures
Cache	Tiny (Kb - Mb)	Superfast (ns)	Small data structures, numbers, small tuples

Not All RAM Is The Same

- Hardware RAM is the actual memory transistors
- Swap Space is RAM-like storage on a local disk drive
- On small systems (under 32Gb) there will often be swap
    - OS will shuffle pages in and out of physical RAM
    - Dirty pages must get re-written and are expensive
    - Code pages are read-only and are cheap
- Over 32 Gb of RAM? swap has few benefits

Consequence 1 — Compression

How much data are we talking about? Mb, Gb, Tb?

Where is the data? cache, RAM, file, database?

Compressed data (avro, parquet, etc.)

- The time decompress is (often) less than the time to transfer
- The cost to compress can be high
- You amortize a high cost to write against low cost to read many times

Consequence 2 — Cache-only Data

- A "lazy" processing pipeline
- Python's `map()` and `filter()`
- Generator expressions fit in cache naturally
  - Does not bring large volumes of data into RAM
  - May run mostly in cache
- You can't force this… But you can encourage it by keeping less in memory

Why this is hard

A common kind programming example reduces to

```
df = pandas.read-something()
struggle with df
```

This may not always work out well

The data is in RAM — which is better than disk or network — but isn't as good as cache

Reducing the volume of data is essential