Lazy Coding

Memoization, Delayeds, and Almost-Rights

Dr. Scott Gorlin

Harvard University

Fall 2018

Agenda

- Optimization
- Memoization
- Cache Backends
- Hashing
- Sketching
- Final
- Science Fair
- Readings

Optimization

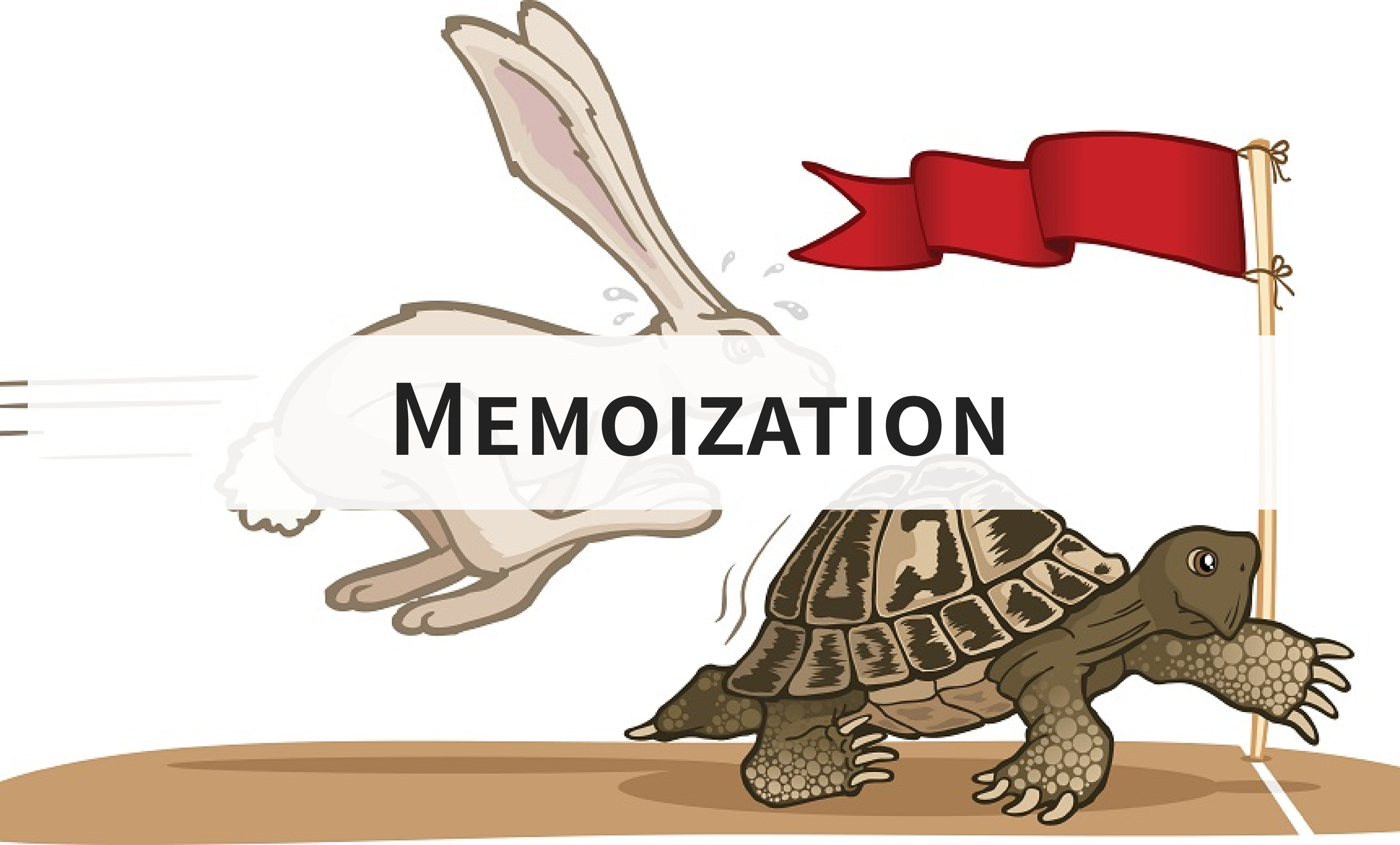
Speed Isn't Everything

Last week was about ***faster*** code…

…which, in general, is the ***worst*** kind to write

Today, we'll discuss how to be ***lazy***

Memoization



Memoization

> *Memoization … is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again*
>
> — *Wikipedia*

This is a form of ***caching***

... BY ANY OTHER NAME

Lazy Properties

```
class A:
    @cached_property
    def x(self):
        print('Initializing')
        return 5

>>> a = A()
>>> a.x
Initializing
5
>>> a.x
5
```

Global Caches

```
@lru_cache()
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

Instance Caches

```
class BiasedSum:
    def __init__(self, b=0):
        self.b = b

    @instance_lru()
    def sum(self, *args):
        print("Evaluating")
        return (
            self.b + sum(args)
        )
```

Caches differ on what they ***store by*** (singletons vs by args), ***where*** they store (global cache vs per instance), and ***if/how they purge***

Why memoize?

Time/Memory Tradeoff

We can exchange speed for memory overhead

Lazy is Simpler

A lazy implementation is often *simpler* to write and maintain

We can simplify calculation, eg, $O(N^2)$ to $O(N)$

We can avoid overloading `__init__` and relax assumptions about what code has run when the object is created, vs when a property is needed

Lazy Properties

Do it once!

```
def lazy_property(func):
    @property
    @wraps(func)
    def wrapped(self):
        attr = '_' + func.__name__
        try:
            return getattr(self, attr)
        except AttributeError:
            v = func(self)
            setattr(self, attr, v)
            return v
    return wrapped
```

```
class A:
    @lazy_property
    def x(self):
        return 5

>>> a = A()
>>> a.x
5
>>> a.__dict__
{'_x': 5}
```

Lazy Properties

```
class lazy_property:
    def __init__(self, func):
        self.func = func

    def __get__(self, instance, cls=None):
        attr = '_' + self.func.__name__
        try:
            return getattr(instance, attr)
        except AttributeError:
            v = self.func(instance)
            setattr(instance, attr, v)
            return v
```

```
class A:
    @lazy_property
    def x(self):
        return 5


>>> a = A()
>>> a.x
5
>>> a.__dict__
{'_x': 5}
```

Lazy Properties

Another recipe…

```
# django.utils.functional
class cached_property:
    def __init__(self, func, name=None):
        self.func = func
        self.__doc__ = getattr(func, '__doc__')
        self.name = name or func.__name__

    def __get__(self, instance, cls=None):
        # Replace descriptor with result!
        res = instance.__dict__[
            self.name
        ] = self.func(instance)
        return res
```

Here, the first call ***replaces*** the descriptor with the result; the function is never called again

```
class B:
    @cached_property
    def x(self):
        return 5

>>> b = B()
>>> b.x
5
>>> b.__dict__
{'x': 5}
```

This is slightly faster, but confusingly alters types IMHO.

Lazy Properties

Memoizing properties assumes ***immutability*** of the property, or otherwise indicates you wish to freeze it (eg a `created` timestamp)

It's a good paradigm for objects you don't want to create unnecessarily

```
class SomeTask(Task):
    @cached_property
    def logger(self):
        logger = getLogger('my_task_name')
        ...

>>> task = SomeTask()  # No logger yet
>>> task.logger.info('Starting!')
```

```
class TrainedModel(Task):
    @cached_property
    def model(self):
        with self.output().open() as f:
            return pickle.load(f)

>>> task = TrainedModel()
>>> task.model.predict(...)
```

Just be sure you don't ***actually*** want a method! (`task.get_model()`)

Global Cache

```
def cache(func):
    @wraps(func)
    def wrapped(*args):
        try:
            return wrapped._cache[args]
        except KeyError:
            r = func(*args)
            wrapped._cache[args] = r
            return r
    wrapped._cache = {}
    return wrapped
```

Note this changes `fib` from $O(1.618^N)$ to $O(N)$ with $M(N)$!

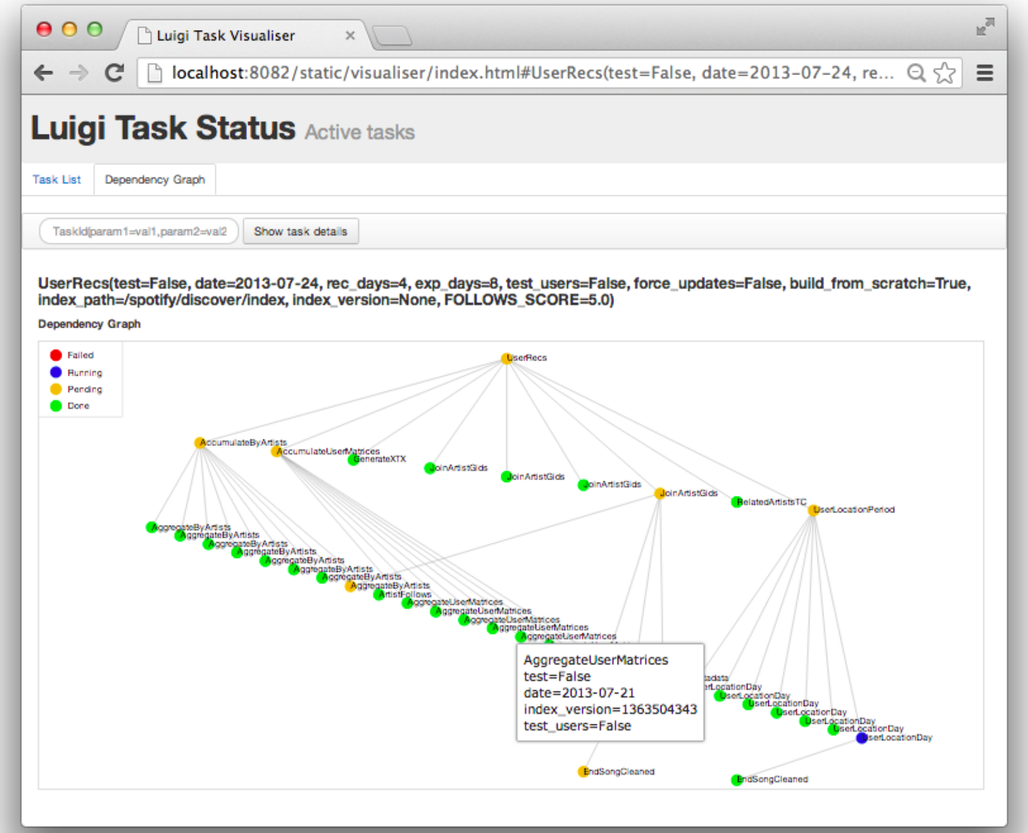
```
@cache
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> [fib(i) for i in range(10)]
```

```
>>> pprint(fib._cache)
{(0,): 0,
 (1,): 1,
 (2,): 1,
 (3,): 2,
 (4,): 3,
 (5,): 5,
 (6,): 8,
 (7,): 13,
 (8,): 21,
 (9,): 34}
```

Global Cache

```
@cache
def get_salted_id(task):
    ...
```

Reduce $O(N^2)$ to $O(N)$. Works because luigi Task's are singletons.



Global Cache

Instance edition:

```
class Poly:
    """ax^n + bx^(n-1) + ..."""
    def __init__(self, *coefs):
        self.coefs = coefs

    @cache
    def f(self, x):
        n = len(self.coefs) - 1
        return sum([
            c * x**(n - i)
            for i, c in enumerate(self.coefs)
        ])
```

```
>>> p = Poly(1, 1, 1)
>>> p.f(1), p.f(5)
(3, 31)
>>> Poly.f._cache
{(<__main__.Poly object at 0x10a418128>, 1): 3,
 (<__main__.Poly object at 0x10a418128>, 5): 31}
```

Because of `self` in the bound method, the cache includes the instance as the first cached arg. This is OK, but can prevent garbage collection of instances

Global Cache

Python objects are ***garbage collected*** and memory is freed when no variables point to the object any more (eg, they're out of scope)

By definition, a global cache will preserve any objects passed in as args!

A ***weak reference*** is a variable that doesn't prevent the object from being garbage collected. Consider wrapping objects in `weakref.ref` before calling a cached function.

Instance Cache

```
def instance_cache(func):
    cname = '_{}_cache'.format(func.__name__)
    @wraps(func)
    def wrapped(self, *args):
        c = get_or_create_cache(self, cname)
        try:
            return c[args]
        except KeyError:
            v = func(self, *args)
            c[args] = v
            return v
    return wrapped
```

```
def get_or_create_cache(obj, cname):
    try:
        c = getattr(obj, cname)
    except AttributeError:
        c = {}
        setattr(obj, cname, c)
    return c
```

```
class Poly:
    @instance_cache
    def f(self, x):
        ...

>>> p = Poly(1, 1, 1)
>>> p.f(5)
31

>>> p._f_cache
{(5,): 31}
```

Code sacrifices generality, but the instance owns its own cache

Instance Cache

```
def instance_lru(**lru_kwargs):
    def generate(func):
        def wrapped_cache(self):
            @lru_cache(**lru_kwargs)
            @wraps(func)
            def wrapped(*args, **kwargs):
                return func(self, *args, **kwargs)
            return wrapped
        return cached_property(wrapped_cache,
name=func.__name__)
    return generate
```

```
class A:
    @instance_lru()
    def f(self, x):
        ...
```

Key Equivalence

If the arg order doesn't matter, you should ignore it for caching!

```
def commutative(func):
    @wraps(func)
    def wrapped(*args):
        return func(*sorted(args))
    return wrapped

@commutative
@cache
def cached_sum(*args):
    print(args)
    return sum(args)
```

```
>>> cached_sum(6, 5, 4)
(4, 5, 6)
15

>>> cached_sum(5, 4, 6)
15
```

NB: order of the decorators is important!

Cache Backends

Backend Considerations

Metadata and Alg

- What is stored?
- What is purged?

Medium

- In memory, or persistent?
- Cost of lookup vs calc?
- Cost of hit vs miss?

Auto Purging

In general, caches are $M(N)$

Least Recently Used Cache

```
from functools import lru_cache


@lru_cache(maxsize=2**7)
def f(x):
    ...
```

Your go-to default!

Least Frequently Used Cache

```
from cachetools.func import lfu_cache


@lfu_cache(maxsize=2**7)
def g(x):
    ...
```

Also `ttl_cache` to expire ***old*** results

Persistent Caches

Persistent caches store values on-disk, but will be slower than in-mem

Django's Cache System is robust and can be backed in-memory, by files, in a DB or Memcached. It is designed for webpages, but works for any python data, and is dict-like

```
from django.core.cache import cache

model = cache.get('test_data')
if model is None:
    model = GLM()
    model.fit(*get_recent_data())
    cache.set('model', model, 30)  # TTL
model.predict(data)
```

Persistent Caches

shelve implements a persistent, dictionary-like object

```
with shelve.open('spam') as db:
    db['eggs'] = 'eggs'
```

Persistent Caches

Ensure that file IO is not more expensive than the result you're caching!

Try to ***batch*** multiple cache ops to minimize overhead (eg load shelf, cache multiple calls, then sync to disk). This trades off speed and chance of lost work.

Persistent Caches

Service	Notes
Memcached	Fast in-memory (but persistent to python proc)
Redis	Key-Value store
Hbase	Hadoop distirbuted KV Store
DynamoDB	AWS KV Store

Hashing

Hashing

Global caches must be keyed on arguments. That means the arguments must be ***hashable***

Hashable has a slightly different meaning here than we've used before

Hashing

Mapping ***data*** to a globally (nearly) unique ***integer***

Python hash

Python implements its own hashing algorithm:

```
>>> hash(1)
1
>>> hash(1.0)
1
>>> hash(1.01)
2305843009213696l
>>> hash(10)
10
```

```
>>> hash(9234324)
9234324
>>> hash('s')
3654731674154865774
>>> hash(object())
270709786
```

… which is designed for ***simplicity*** and ***speed***, not security

Hashing

There are many hash algorithms which vary by:

- Cryptographic strength
- Speed
- Memory
- Uniformity/bit independence
- Avalanche/Chaotic nature
  - or, locality-sensitive hashing

sometimes, ‘multiple hash functions’ also just means a different salt or different bits of the output

Hashing

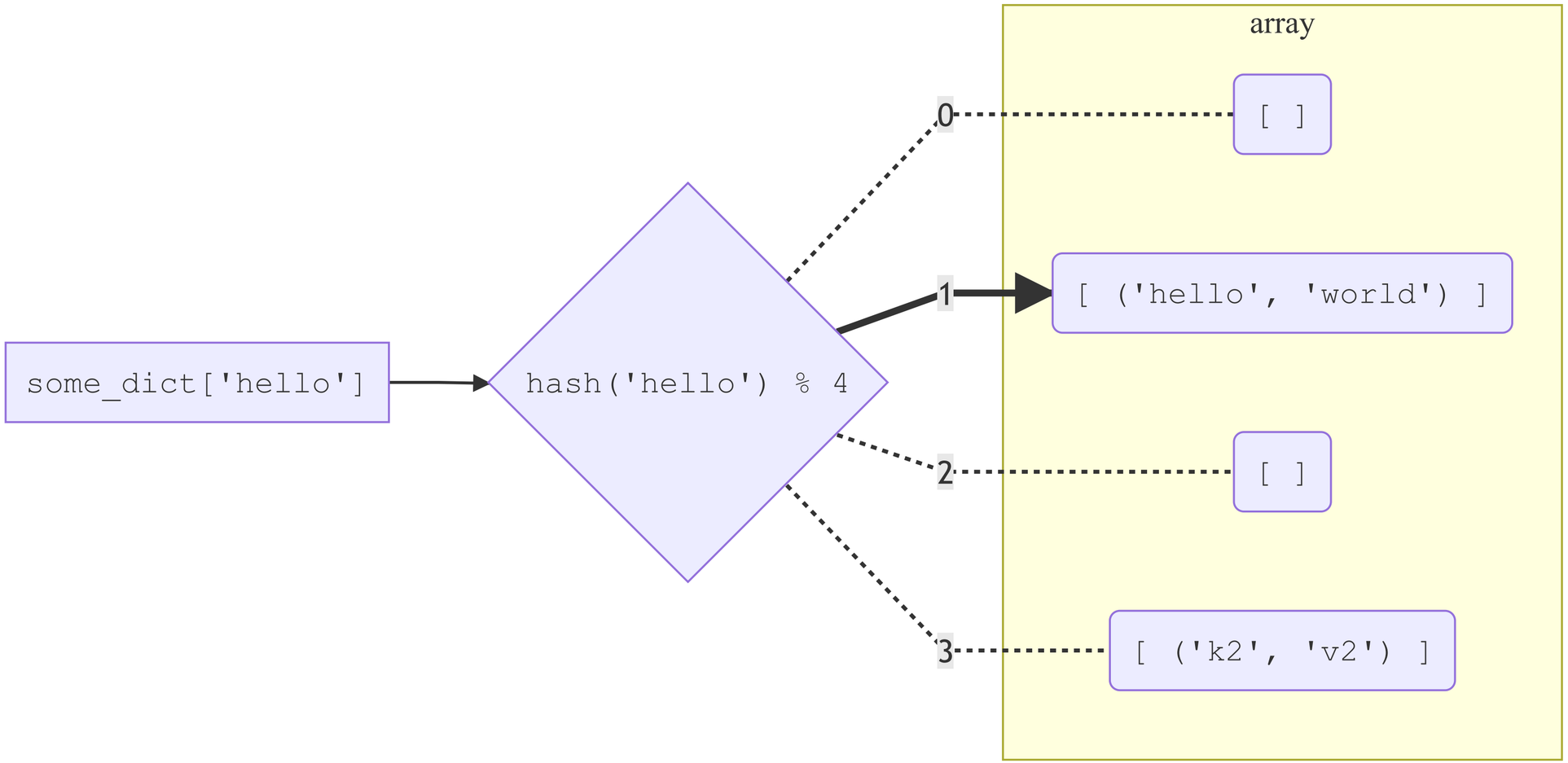
> *An object is **hashable** if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.*
>
> — *Python Glossary*

Hashing

Hashing is used to quickly collate or check (in)equality of arbitrary objects. It is used in `set` and `dict` (formally, Hash Collections and Hash Tables).

Hashing

Hash tables (eg, a `dict`): hash of key indexes array



Hashing

Arbitrary objects are hashable!

```
>>> hash(object())
270709790
```

… because the hash is related to the ***object pointer*** in memory, which never changes, and `__eq__()` by default ***tests identity***

```
>>> a = object(); b = object()
>>> hash(a) == hash(b)
False
```

Mutability

Some ***mutable*** objects explicitly disallow hashing, because they cannot simultaneously override `__eq__()` and avoid changing their hash

```
>>> [1, 2, 3] == [1, 2, 3]
True
>>> hash([1, 2, 3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

SKETCHING

$$\min(9, 4, 7, 4) = 4$$

Summary Stats

Many calculations can be reformulated from ***summary stats***: calculations on ***parts*** of the data, that can be rolled up, and used to compute the end result

```
def mean(stream):
    s = 0
    n = 0
    for v in stream:
        s += v
        n += 1
    return s / n
```

```
def sum_stats(stream):
    s = 0
    n = 0
    for v in stream:
        s += v
        n += 1
    return s, n

def chunked_mean(*streams):
    # M(N/chunks), can be parallel
    summaries = map(sum_stats, streams)
    totals, counts = zip(*summaries)
    return sum(totals) / sum(counts)
```

Summary Stats

...including `std` and `var`!

$$\sigma^2 = E[(X - \mu)^2] = E[X^2] - E[X]^2$$

```
def sum_stats(stream):
    """count, sum(x), sum(x^2)"""
    # NB: not most efficient
    return tuple(
        sum(v**coef for v in stream)
        for coef in range(3)
    )
```

```
def chunked_var(*streams):
    summaries = map(sum_stats, streams)
    count, sum_x, sum_x2 = map(sum, zip(*streams))
    return sum_x2/count - (sum_x/count)**2
```

... very useful to store $\sum X^2$ on fact tables!

Summary Stats

… but not every algorithm!

```
def n_uniques(stream):
    return len(set(stream))
```

```
def chunked_uniques(*streams):
    # This is wrong! Items appearing in different
    # streams will not be de-duped
    # A correct solution requires M(N),
    # which sucks!
    return sum(map(n_uniques, streams))
```

Sketches

A ***sketch*** is a data structure and algorithm for ***fast, approximate*** calculations

Sketches

I shoot hoops, and sink free throws with $p(swish = 1) = 0.5$

I'm shooting sets of 8. I just made 8 in a row!

$$E\left[sets|[1] * 8\right] = \frac{1}{p(swish = 1)^8} = 2^8 = 256$$

Summarizing Rarity

The summary statistic for rare evens is the minimum observed probability

`min` is an aggregateable, parallelizable operation

An estimate of how many things you've seen is inversely proportional to the rarest thing you observed

Measuring Rarity

An item's rarity can be defined as the rarity of its hash!!

$$sha(d) = 0b\mathbf{00000}10110..., p(d) := 2^{-5}$$

The number of uniques in a stream can be approximated from the rarest hash!

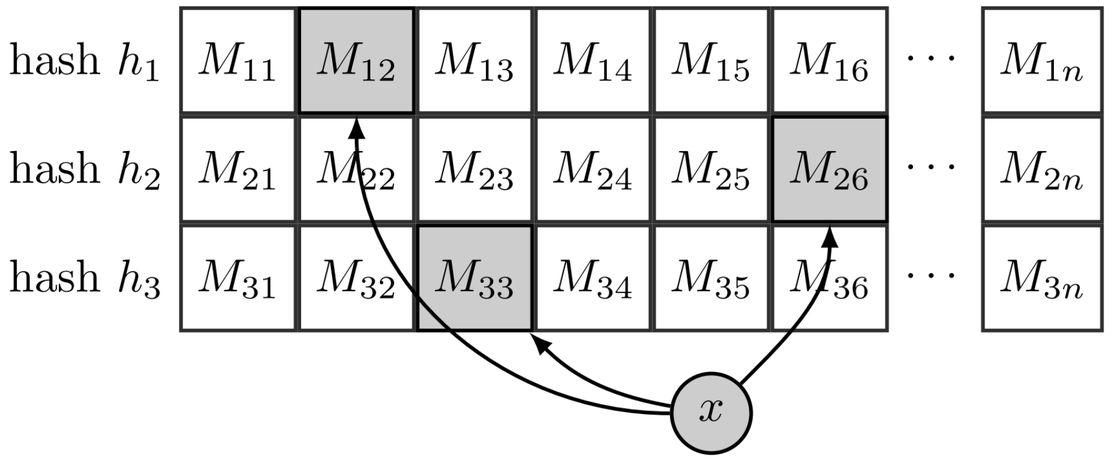
$$|S| \approx \frac{1}{\min p_i}$$

This is a ***count-min-sketch***

Cardinality Estimation

Count-Min-Sketch and HyperLogLog are two algorithms which measure the rarity of tokens in a stream, and yield good approximations to the number of uniques

HyperLogLog does a better job by using multiple independent hashes and averaging



$M_{i,j} == 1$ indicates we have observed a hash with $j + 1$ leading zeros on hash i. Elementwise $max(M^{(1)}, M^{(2)})$ summarizes multiple streams

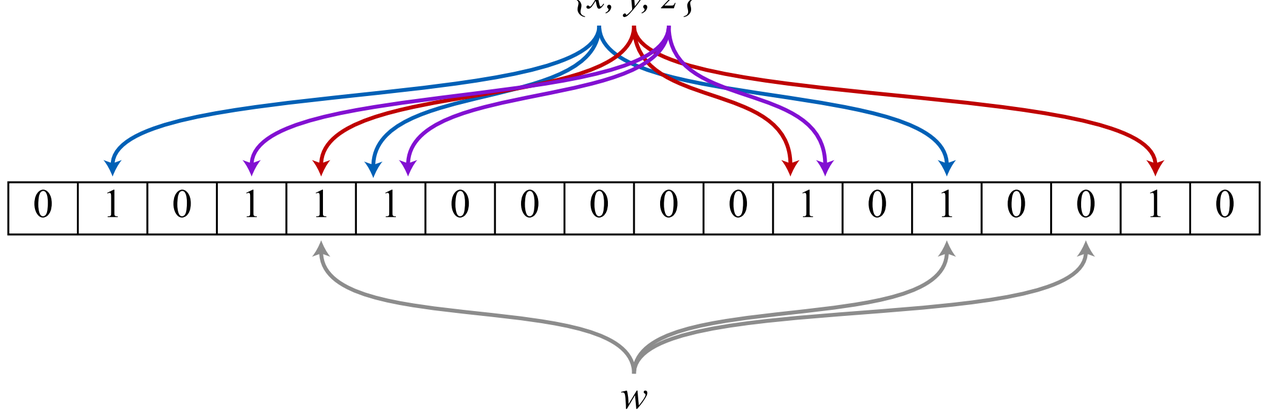
Set Testing

The hash gives us approximate identity

If we have seen a hash before, we have ***probably*** seen that item before!

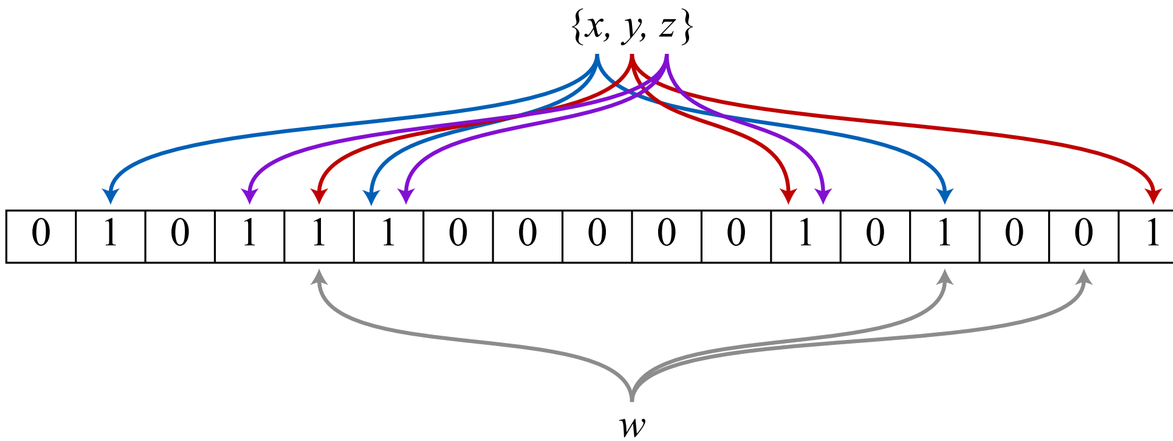
Bloom filters

A ***Bloom filter*** stores a 1 where it has seen some hash `%` `size` before



Bloom filters

With multiple hashes per item, we approximate set membership!



$$p(FP) = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^{k}$$

Final

Final Deets

- 12/19 In class
- Same format (github, open note, etc)
  - More MC to expedite test and grading
- Up to 33% from first half

Science Fair

Final Project

- 12/12 Projects due and science fair!
- You don’t need a poster (though it could be fun)
  - But you need to explain your work in ~10 min to your peers
  - Slides on laptop/printouts/web demo OK (no projectors)
  - Suggestions: architecture diagrams, code snippets, output plots, etc
- 10% of project grade based on fair presentation
- Please add comment to your proposal on canvas: final title and 1+ sentence abstract by next week to share with class
- Best project (by vote) gets extra credit

Readings

- Weak references
- cachetools
- Bloom Filters
- Poke around: Data Sketches
- Time Adaptive Sketches
- Choosing a good hash function
  - Part 1, Part 2, Part 3