Python's Other Collection Types and Algorithms

Andrew Dalke

dalke@dalkescientific.com

dalke scientific
More science. Less time.

list - stack - deque

bisect - heapq

tuple - namedtuple

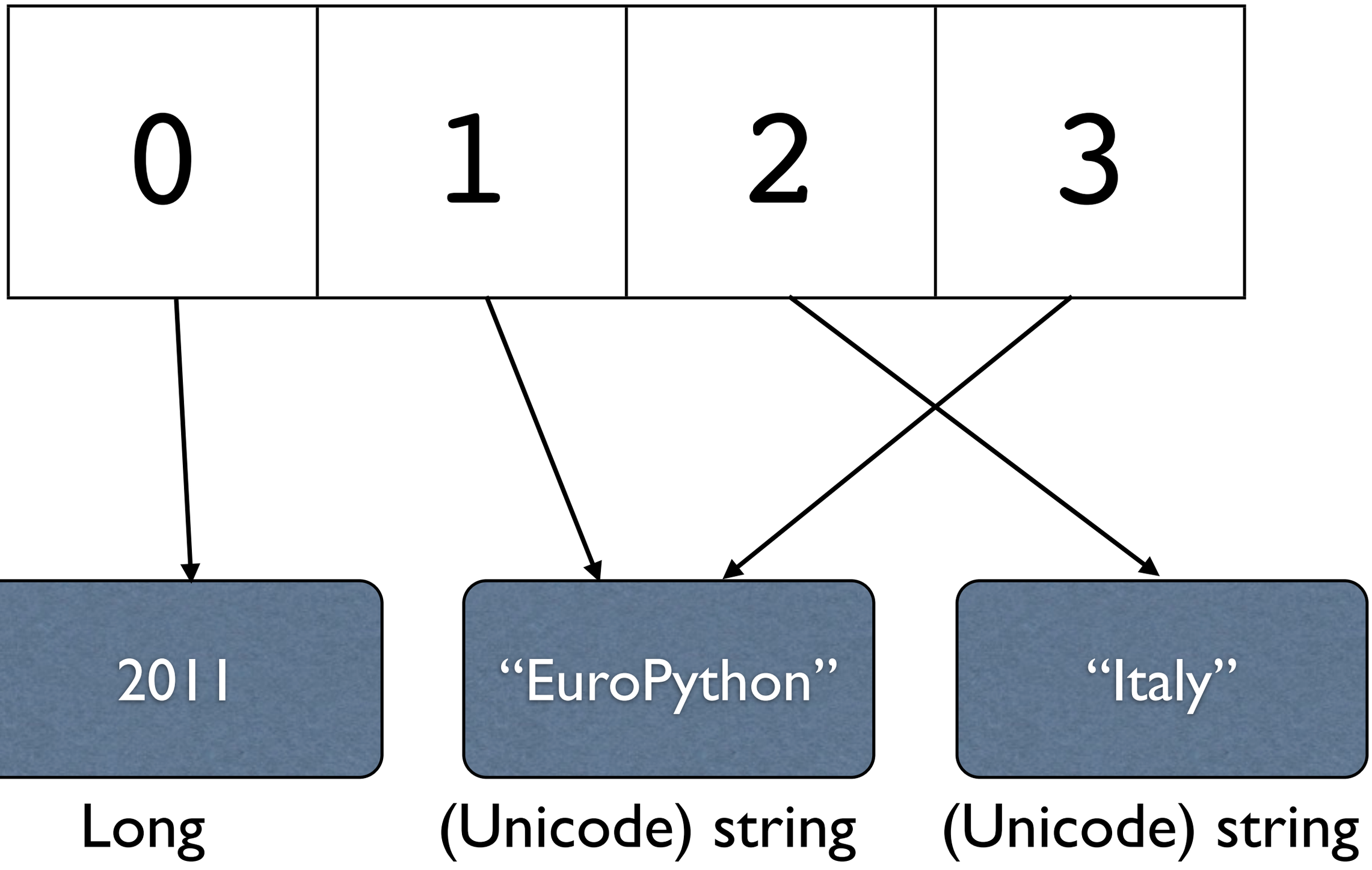
set - frozenset

dict - defaultdict - Counter - OrderedDict

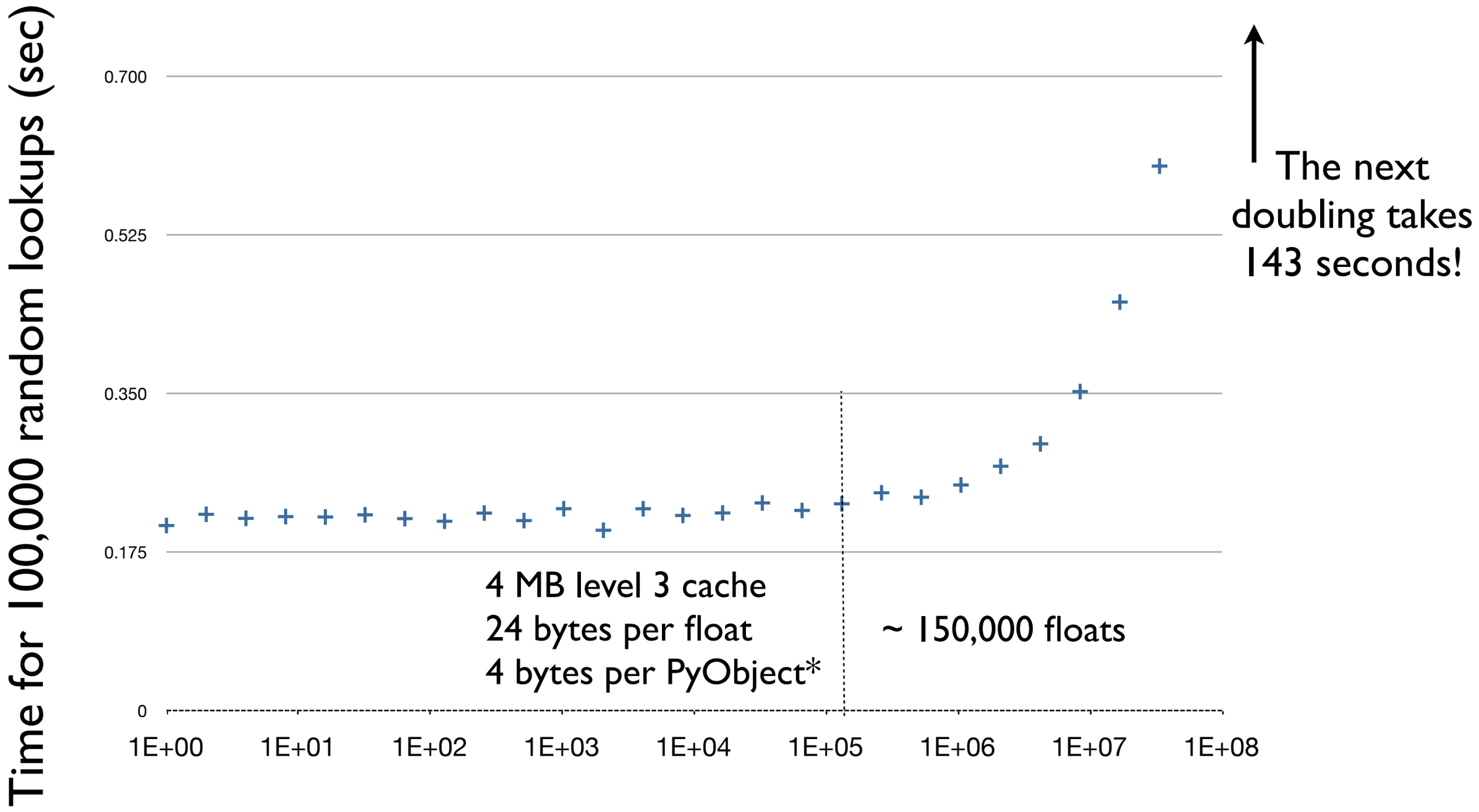
List

```
>>> year = 2011
>>> event = "EuroPython"
>>> where = "Italy"
>>> data = [year, event, where, event]
>>> data
[2011, 'EuroPython', 'Italy', 'EuroPython']
>>>
```

Contiguous block of `PyObject *`



100,000 random lookups in an list of length N



Append to a list

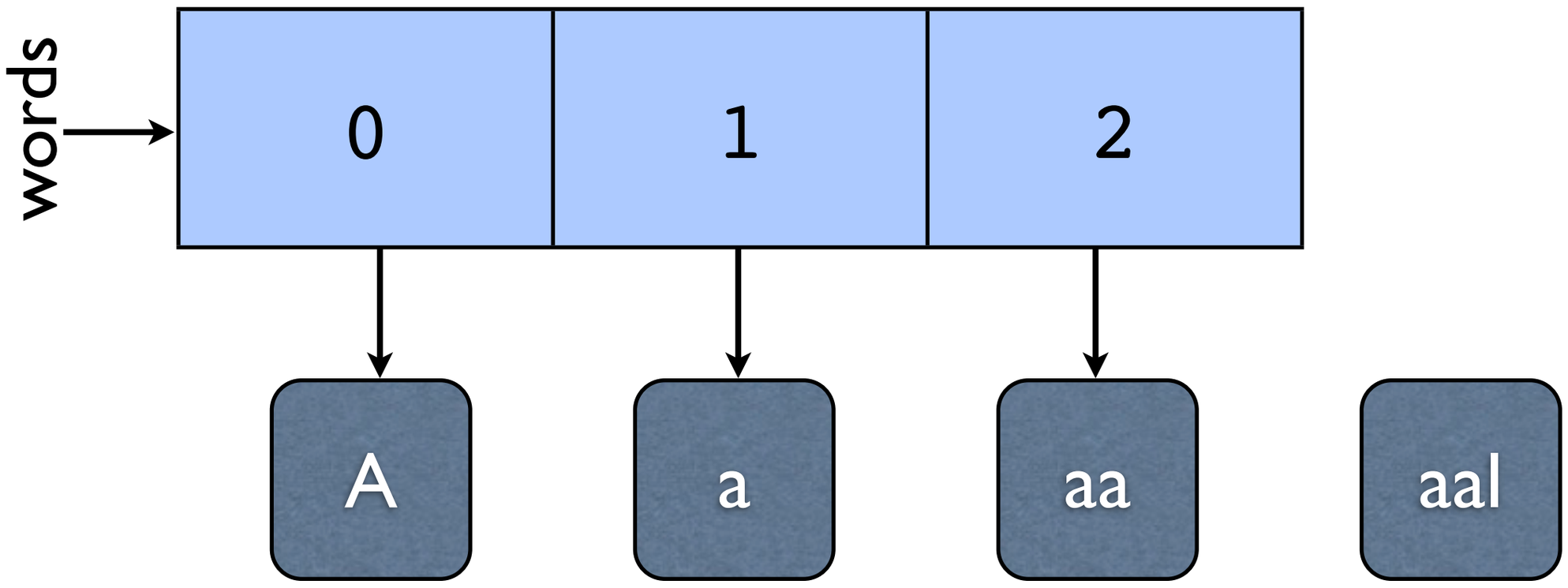
```
>>> words = []
>>> for line in open("/usr/share/dict/words"):
...     words.append(line.strip())
...
>>> len(words)
234936
>>> words[:4]
['A', 'a', 'aa', 'aal']
```

How does append work?

```
>>> words = ['A', 'a', 'aa']
>>> words.append('aal')
```

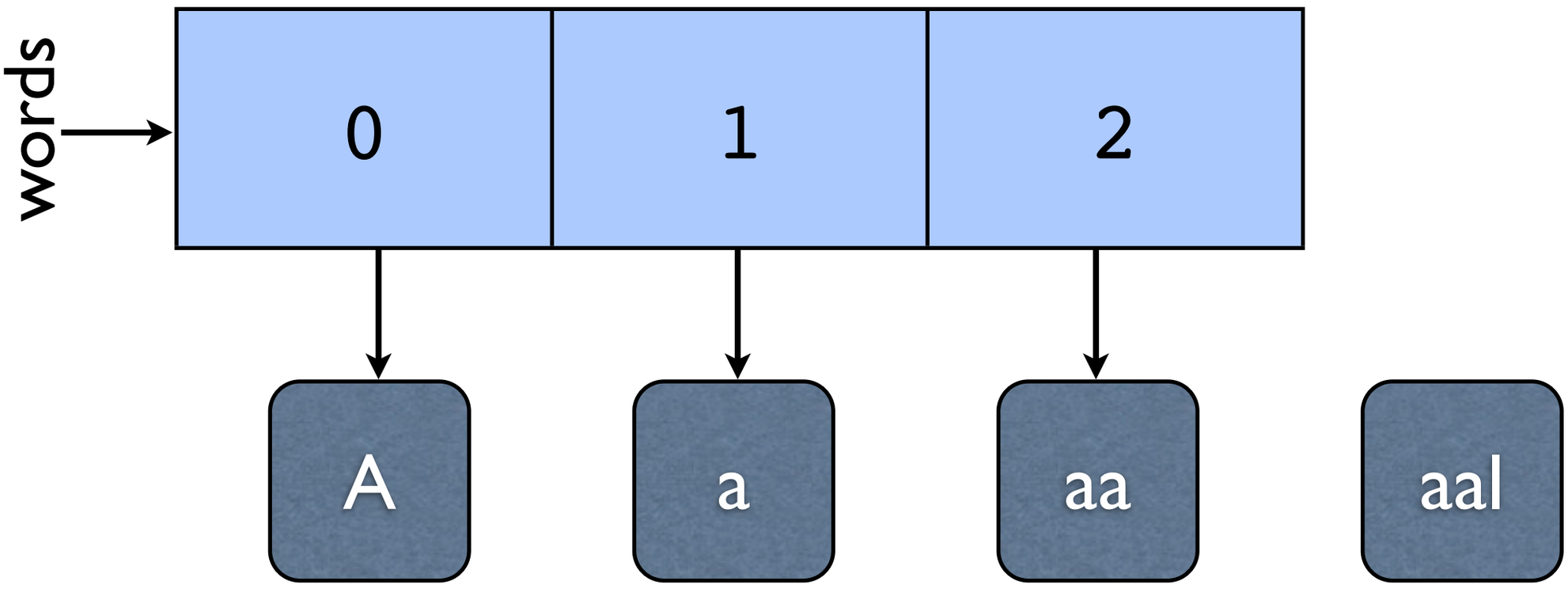
Slow algorithm is $O(n^2)$

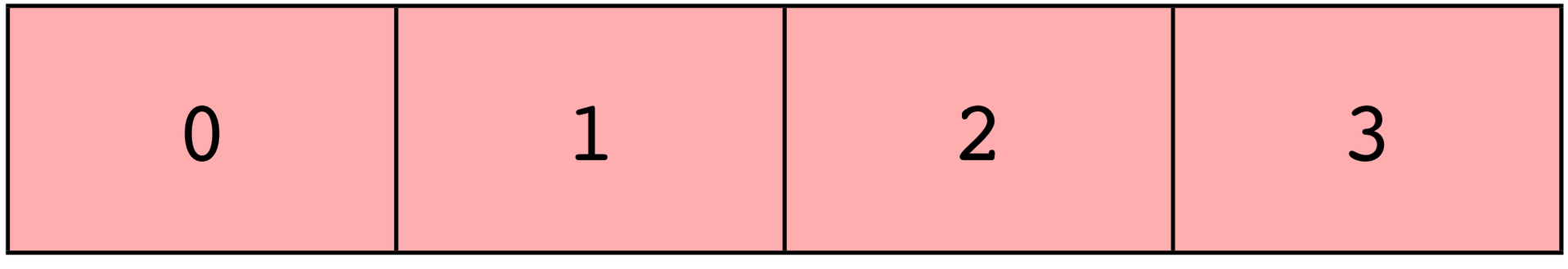
"words" list implementation points to a block of Python references



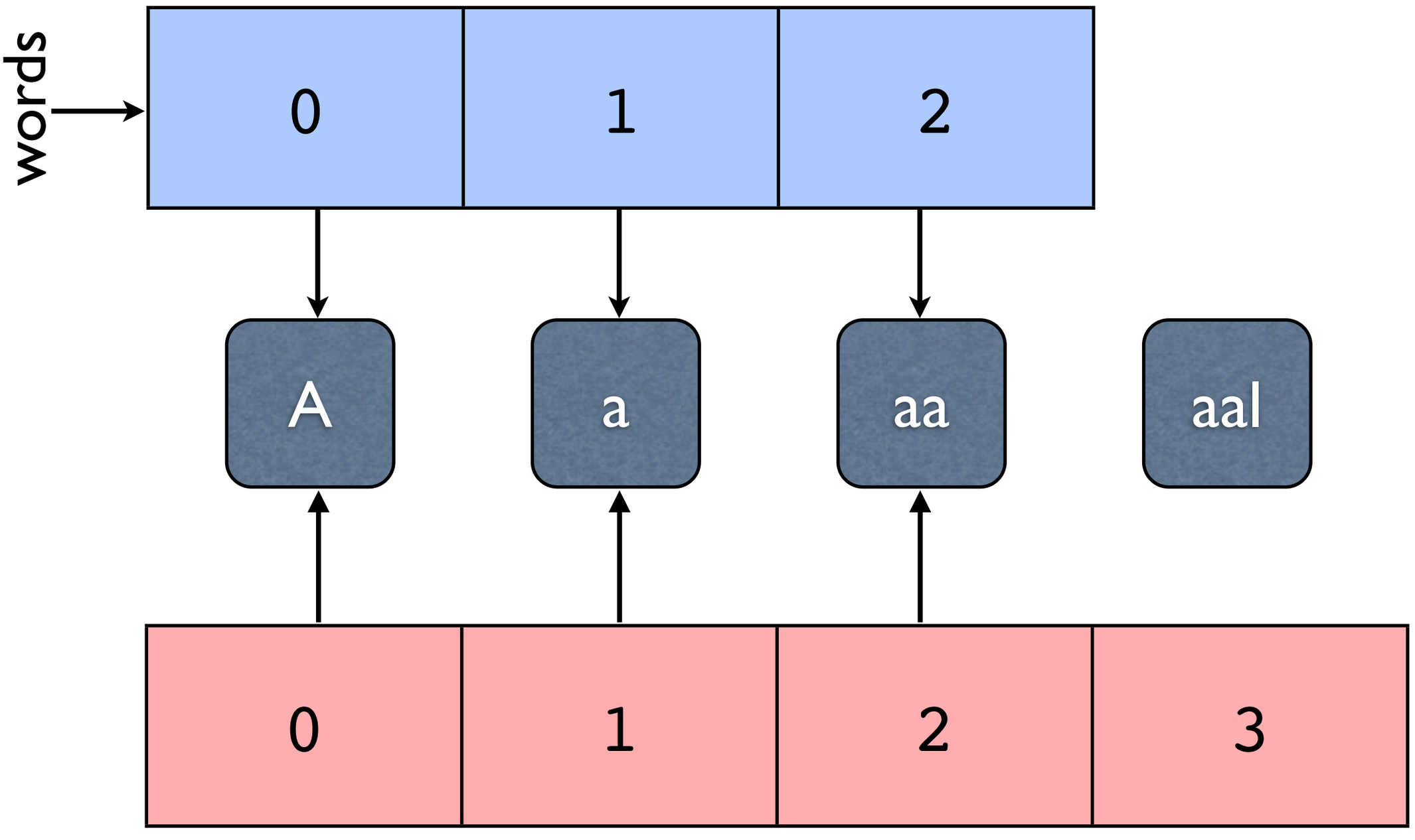
append a reference to "aal"

Make a new list of length 3+1 = 4

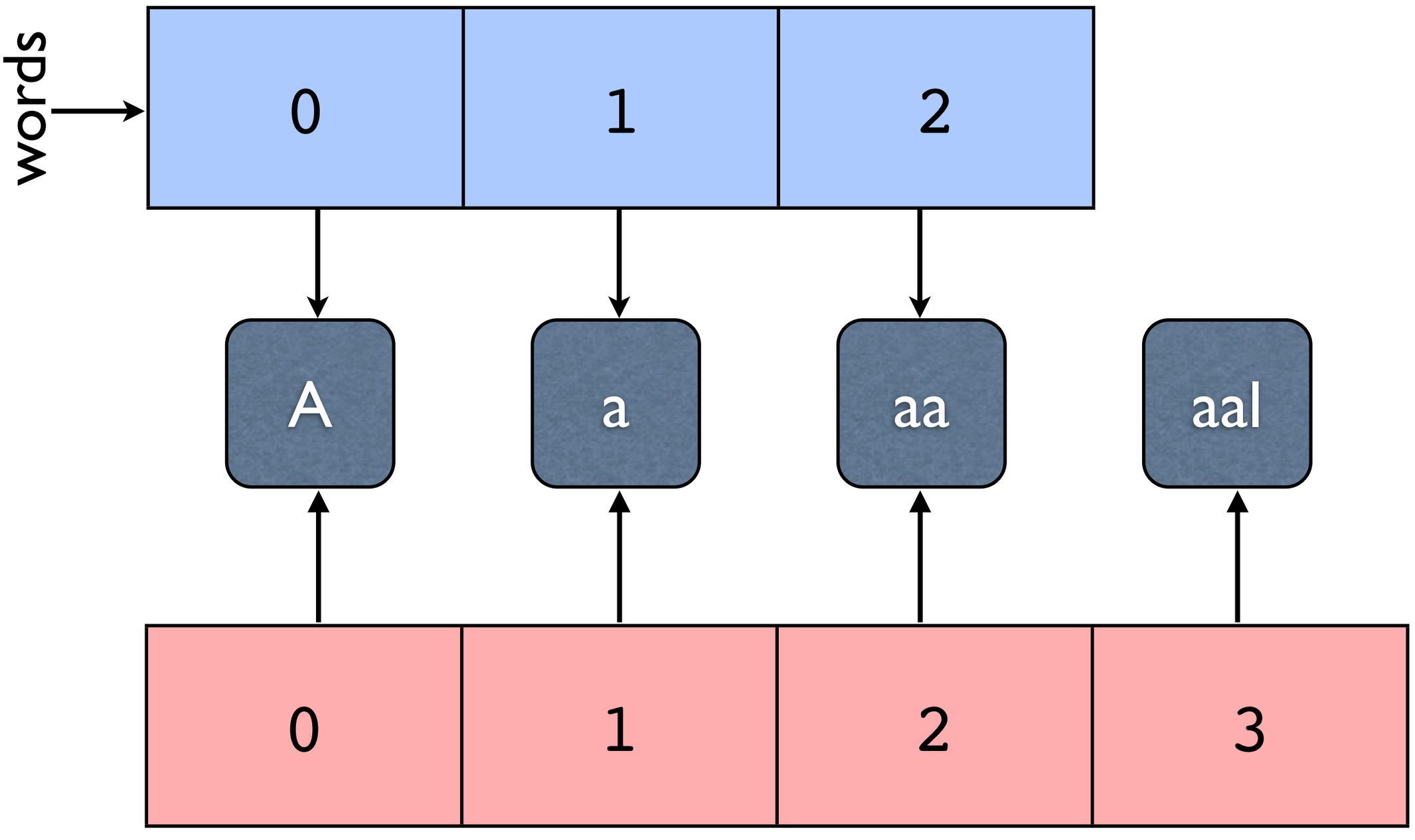




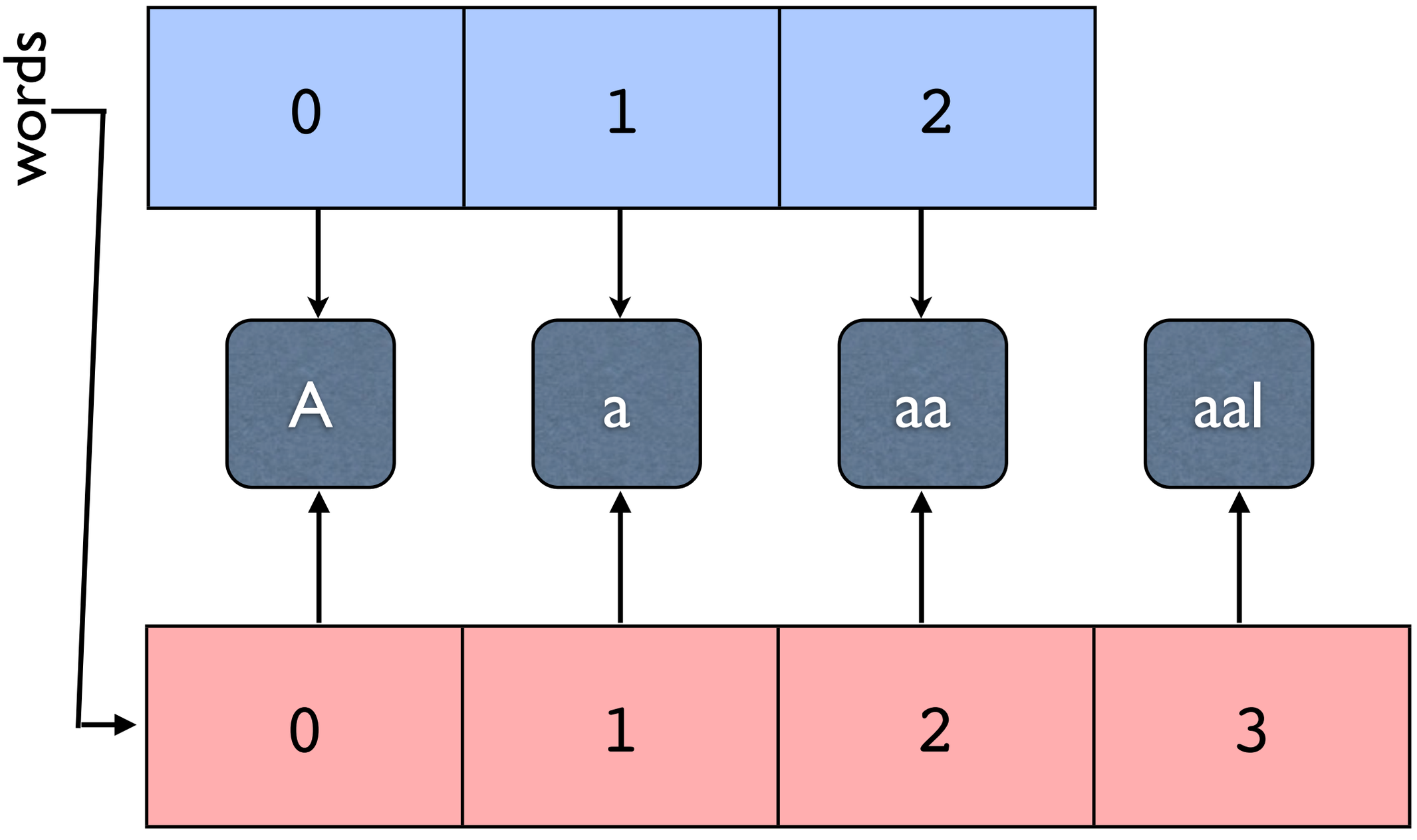
Make references to objects in the old list



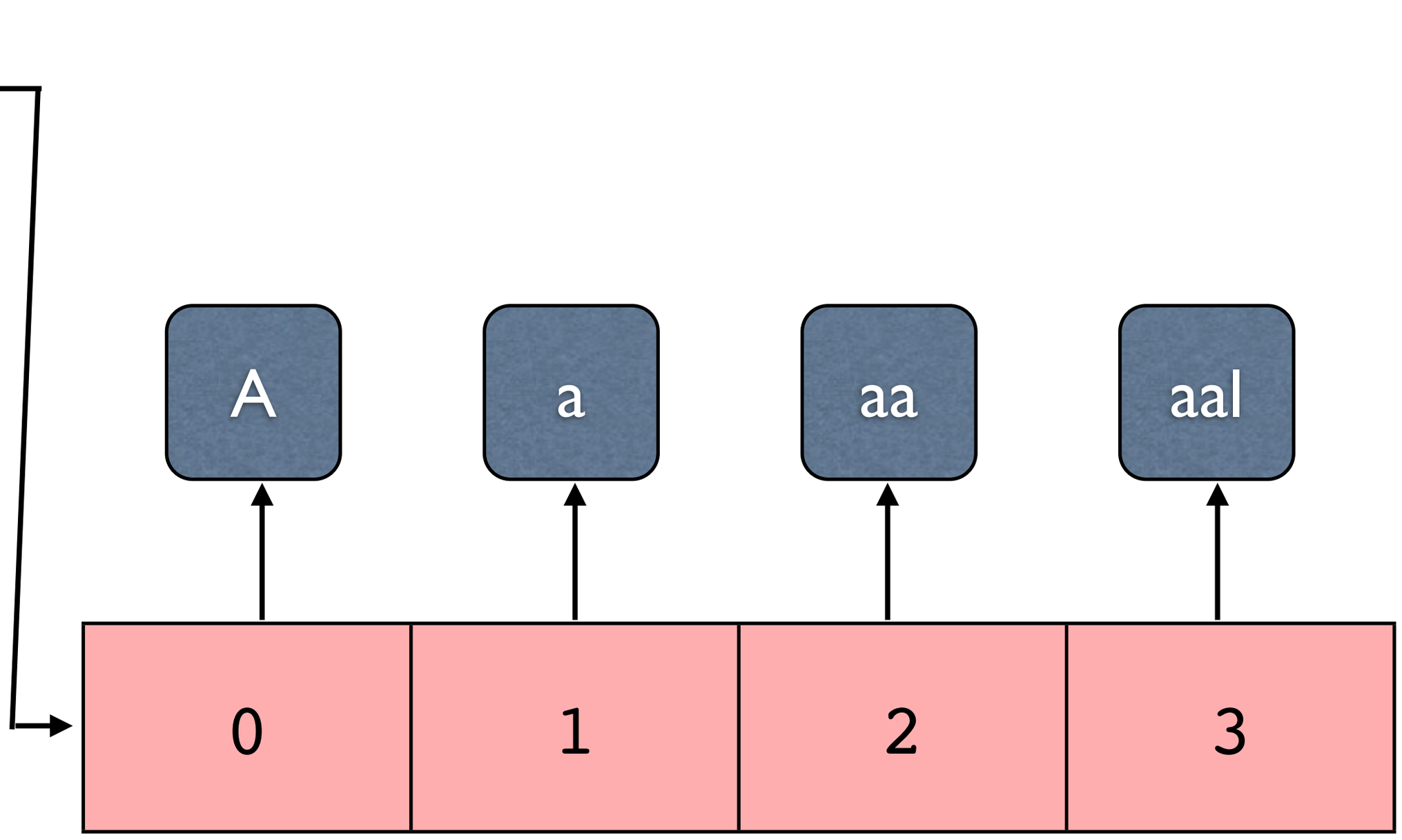
Add the reference to “aal”



Update “words” to use the new contiguous block



Free the old block



This append algorithm takes $O(n^2)$ time

An append to a list of size n takes n copies.

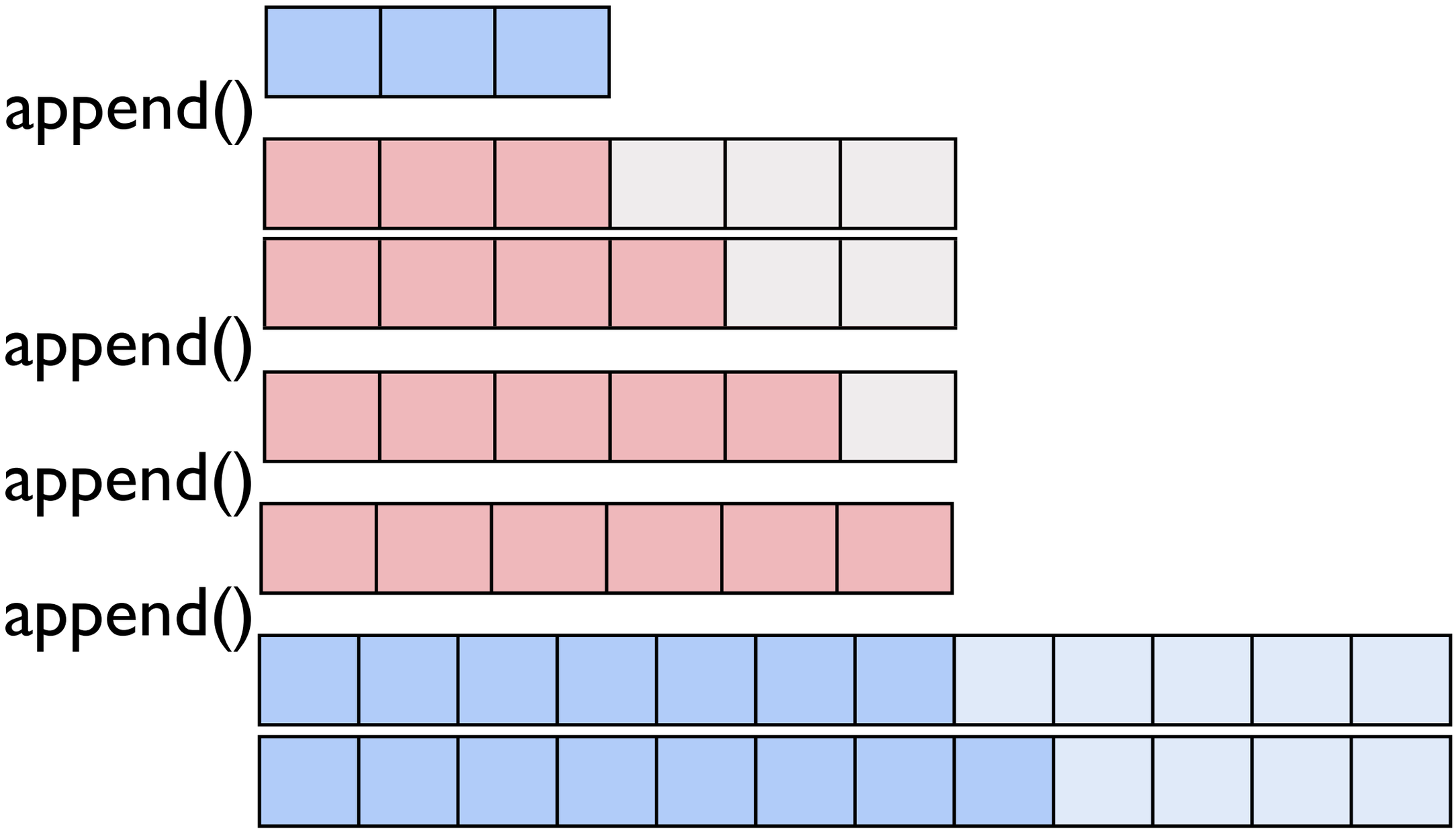
Appending 10 elements to an empty list does

0+1+2+3+4+5+6+7+8+9 = 45 pointer copies

1000 elements would do 499500 copies

The fix is simple ...

Preallocate empty space proportional to the list size (about 13%)



Appending 1000 elements needs fewer than 9000 copies

"Amortized linear append"

Deletion (pop(), del, and remove())
does a C realloc when the used size
is less than the preallocated size

In other word, pop() is also “amortized linear” time

list.insert(0, item)

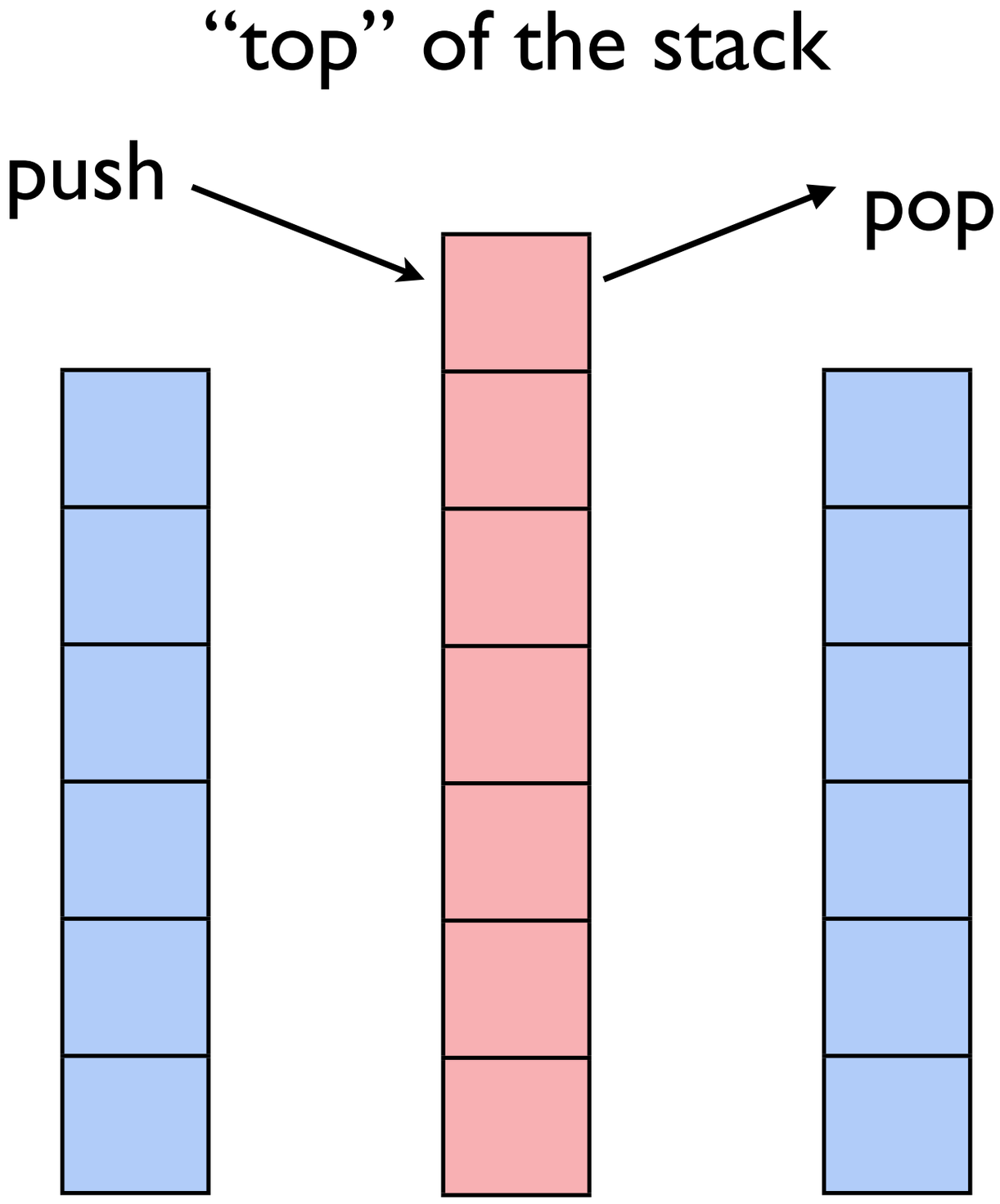
This still has quadratic scaling

Okay to use when small or rarely used
sys.path.insert(0,“/path/to/my/library”)

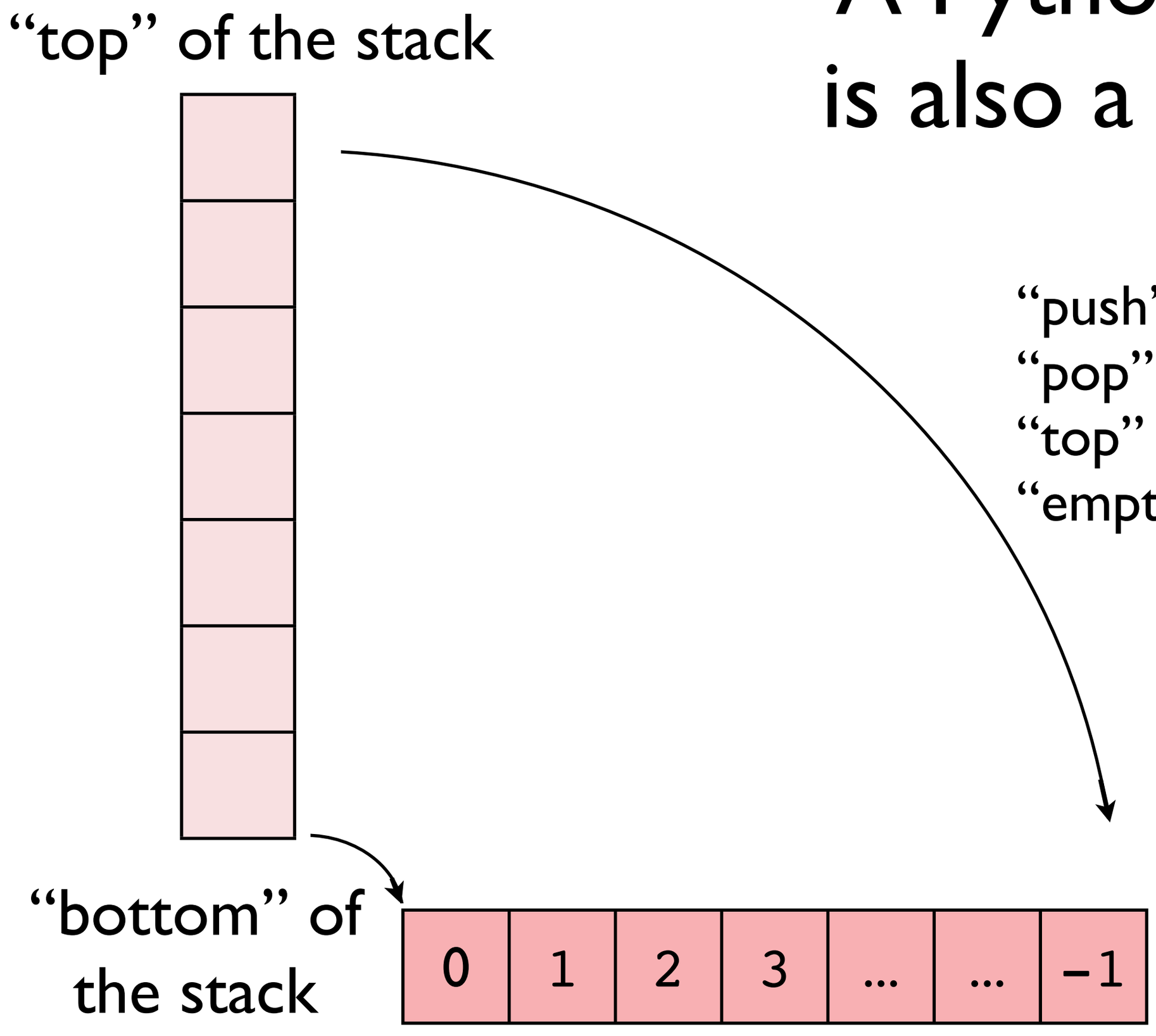
Otherwise, probably want to use deque

Stack

"Stack"
Abstract Data Type



A Python list is also a stack



"stack" is an "abstract data type"

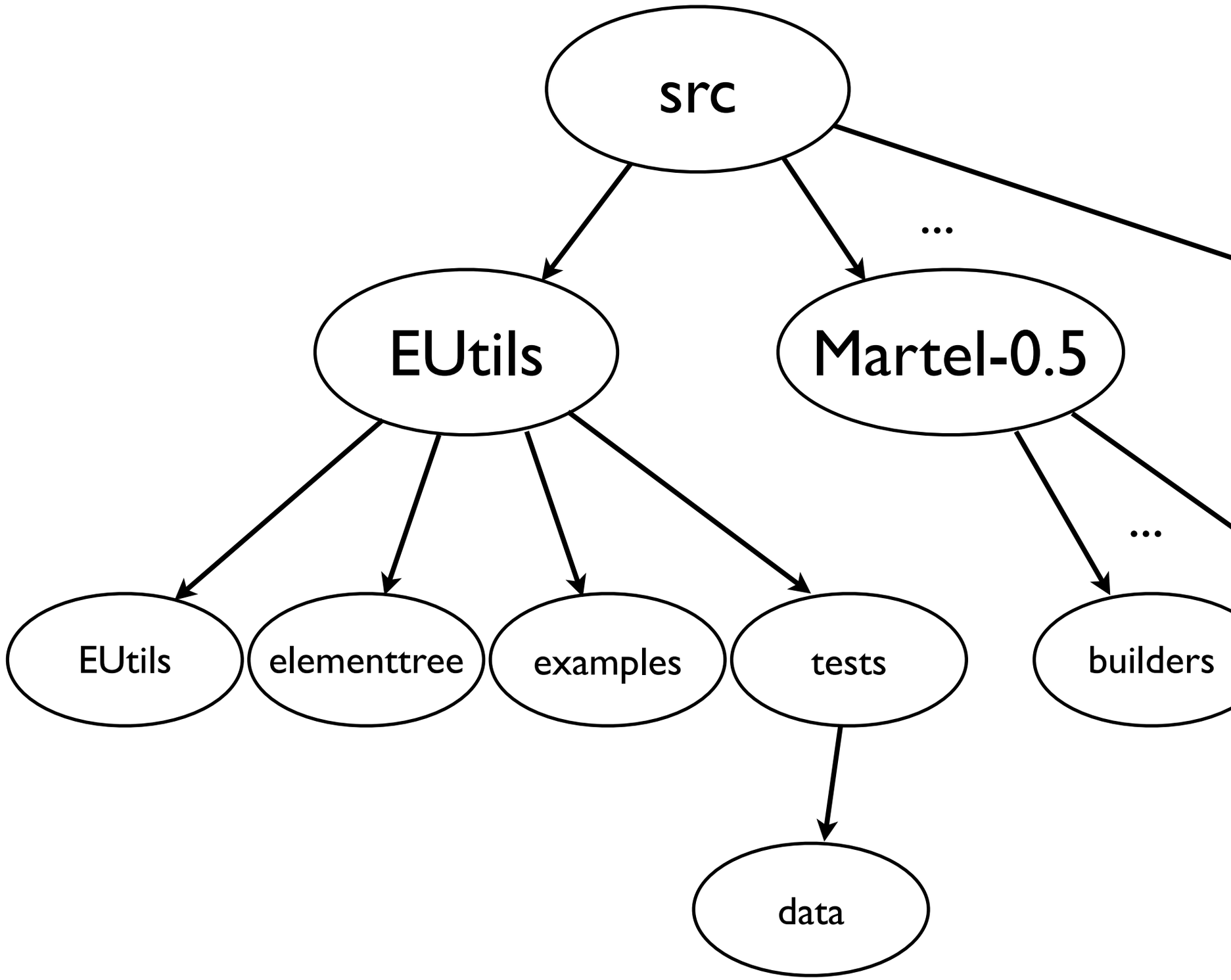
ADTs can map directly to a single class

Sometimes multiple ADTs map to the same class

Some ADTs are synthesized from existing data structures and other functions and exist only by convention

Stacks are often used to process tree structures

```
src
|-EUtils
|---EUtils
|---elementtree
|---examples
|---tests
|-----data
|-Martel-0.5
|---builders
|---doc
|---examples
|---formats
|---test
|-PyRSS2Gen
|---CVS
|---build
|-----lib
|---dist
|-lolpython
|-mysql_chem
|-mysql_oechem
 ...
```



Search the directory tree to find a given name

I'm looking for a file named "listobject.c"
I know it's somewhere in my Python distribution.

```
>>> find_filename("/Users/dalke/python-live", "listobject.c")
'/Users/dalke/python-live/Objects/listobject.c'
>>>
```

Ignore "find" and os.walk() and other tools.
Ignore cycles, unreadable directories, etc.

Standard depth-first/recursive solution

```
def find_filename_recursive(dirname, target_filename):
  # Check all filenames in the directory.
  subdirs = []
  for filename in os.listdir(dirname):
    path = os.path.join(dirname, filename)
    if filename == target_filename:
      return path
    # Keep track of subdirectories for later processing.
    if os.path.isdir(path):
      subdirs.append(path)

# Processed all of the names in this directory.

# Recursively search each of the subdirectories
for path in subdirs:
  found_filename = find_filename_recursive(path, target_filename)
  if found_filename:
    return found_filename

# Not found
return None
```

```
def find_filename_recursive(dirname, target_filename):
  # Check all filenames in the directory.
  subdirs = []
  for filename in os.listdir(dirname):
    path = os.path.join(dirname, filename)
    if filename == target_filename:
      return path
    # Keep track of subdirectories for later processing.
    if os.path.isdir(path):
      subdirs.append(path)

# Processed all of the names in this directory.

# Recursively search each of the subdirectories
for path in subdirs:
  found_filename = find_filename_recursive(path, target_filename)
  if found_filename:
    return found_filename

# Not found
return None
```

There's the stack

Manage the stack myself - simpler!

```
def find_filename_dfs(root, target_filename):
    # Keep track of the directories to search
    search_stack = [root]

    while search_stack:
        # Pop the top item from the stack
        dirname = search_stack.pop()

        for filename in os.listdir(dirname):
            path = os.path.join(dirname, filename)

            # Does the filename exist in the directory?
            if filename == target_filename:
                return path

            # If it's a directory, add it to the set
            # of directories I need to search
            if os.path.isdir(path):
                search_stack.append(path)

    # Not found
    return None
```

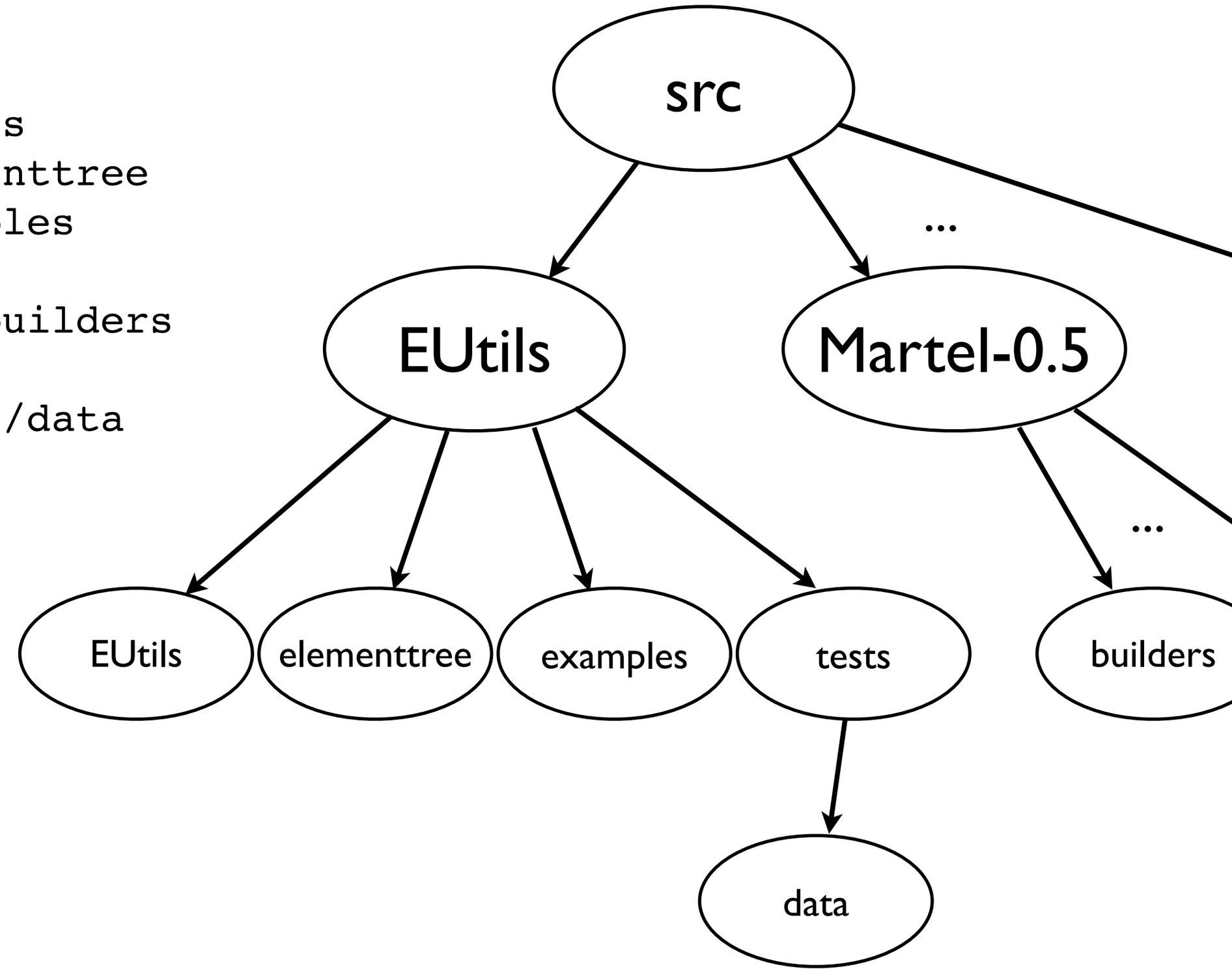
People tend to make shallow trees

Software tends to make deep trees

A breadth-first search might be better than depth-first

Breadth-first search

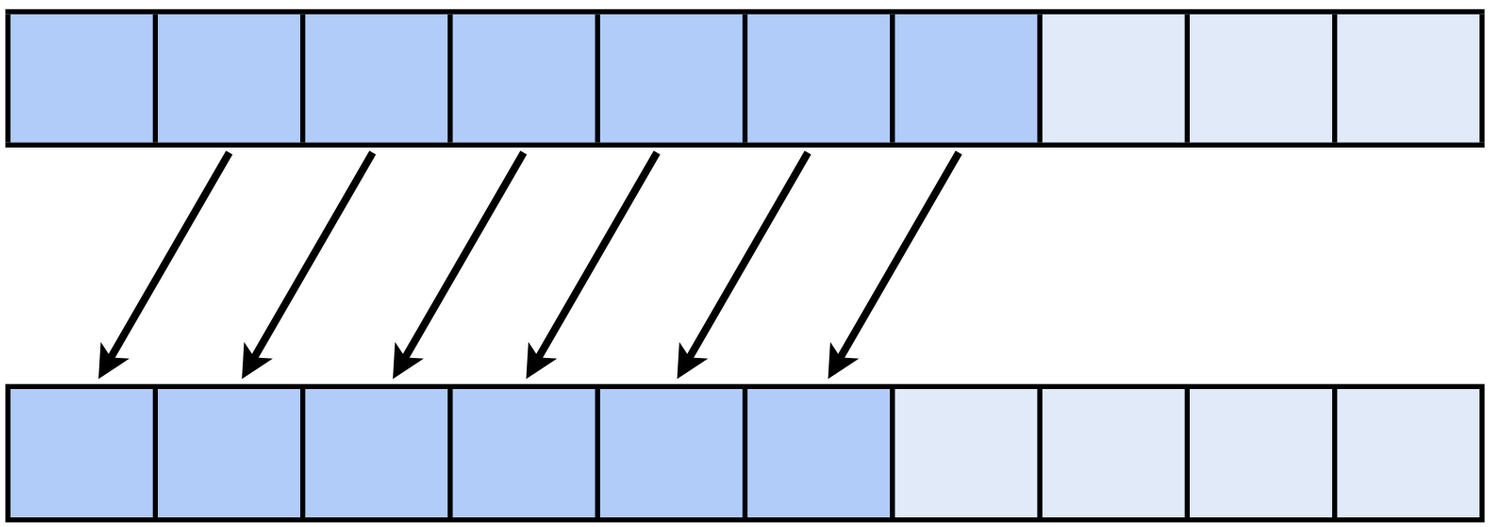
```
src
src/EUtils
src/Martel-0.5
  ...
src/EUtils/EUtils
src/EUtils/elementtree
src/EUtils/examples
src/EUtils/tests
src/Martel-0.5/builders
  ...
src/EUtils/tests/data
  ...
```



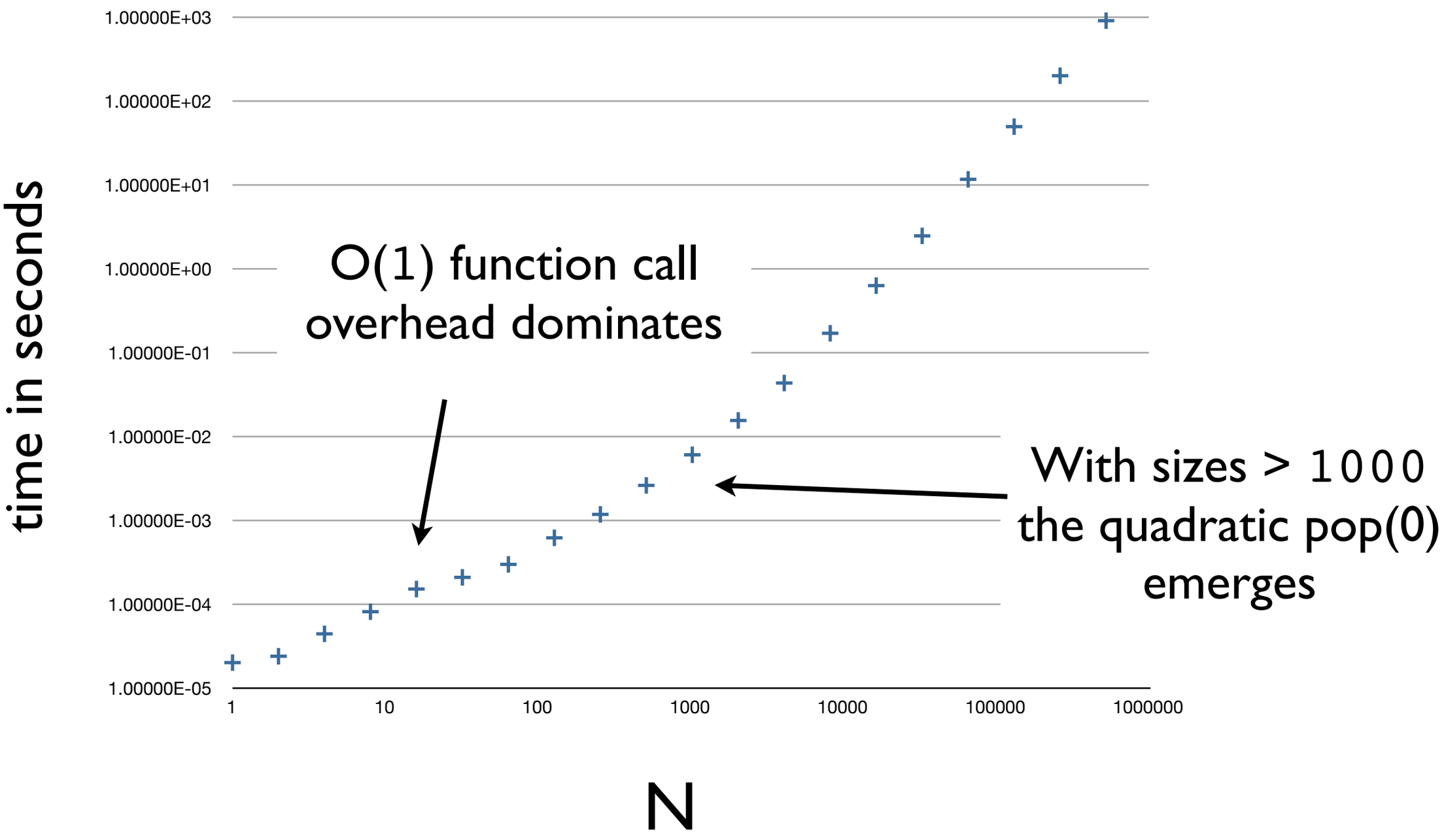
Change for a breadth-first search

```
def find_filename_bfs(root, target_filename):
    # Keep track of the directories to search
    search_stack = [root]

    while search_stack:
        # Pop the bottom item from the stack
        dirname = search_stack.pop(0)

        for filename in os.listdir(dirname):
            path = os.path.join(dirname, filename)

            # Does the filename exist in the directory?
            if filename == target_filename:
                return path

            # If it's a directory, add it to the set
            # of directories I need to search
            if os.path.isdir(path):
                search_stack.append(path)

    # Not found
    return None
```

Remember, list.pop(0) takes O(N) time

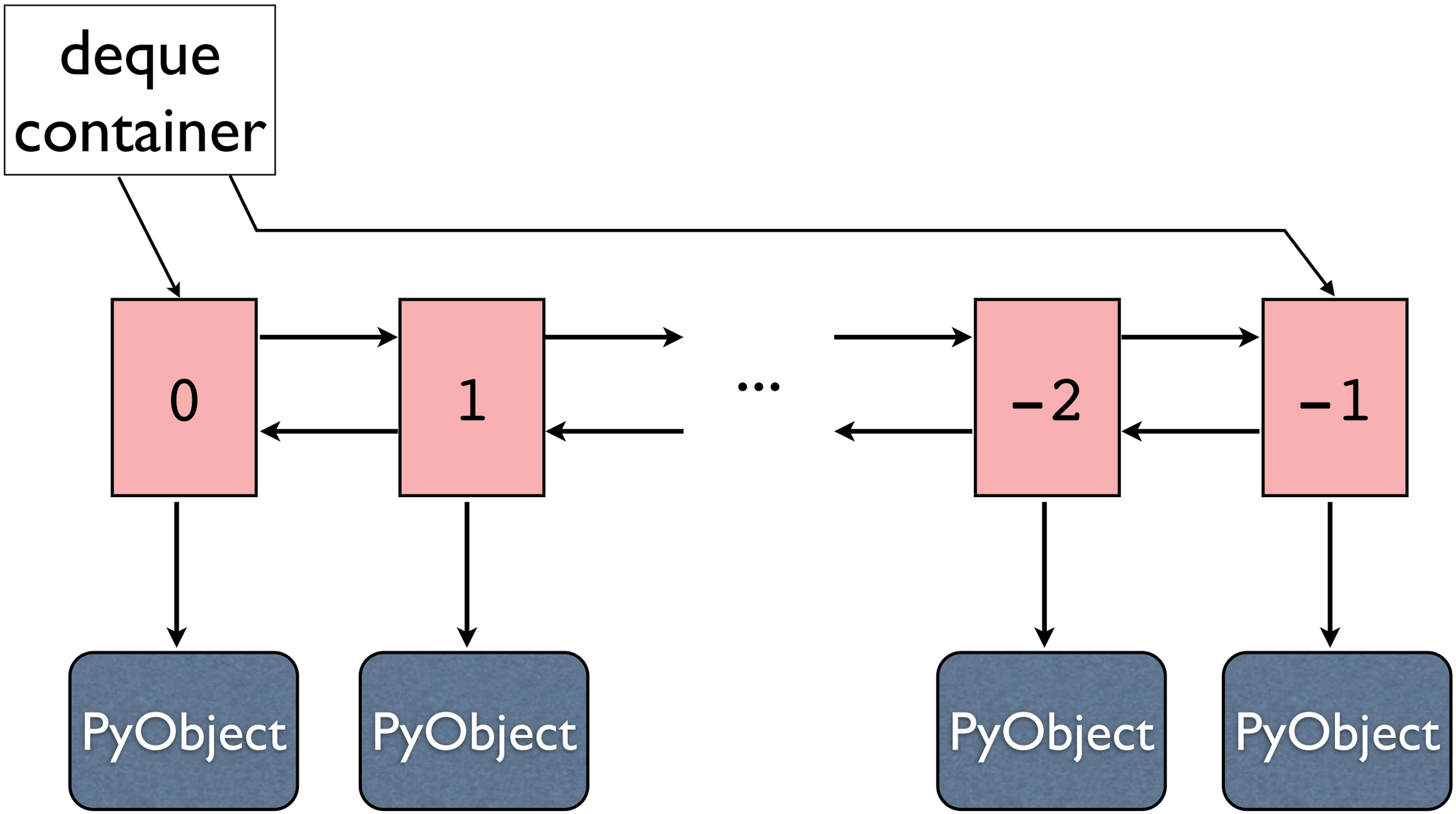


Append N items followed by N pop(0) calls

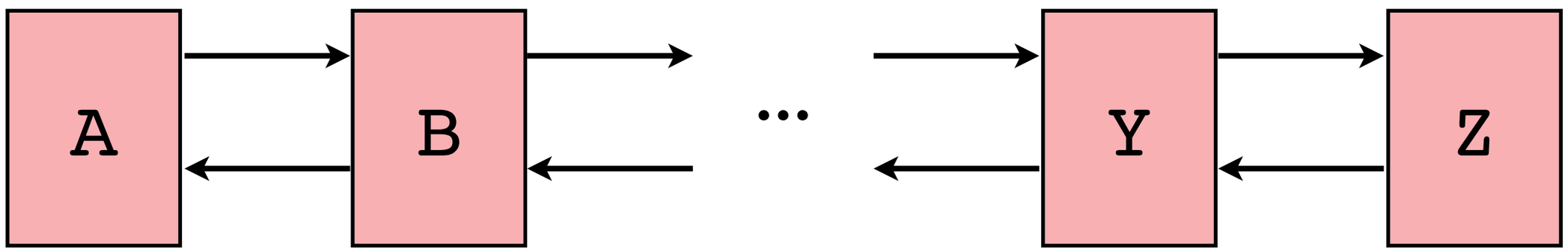


collections.deque

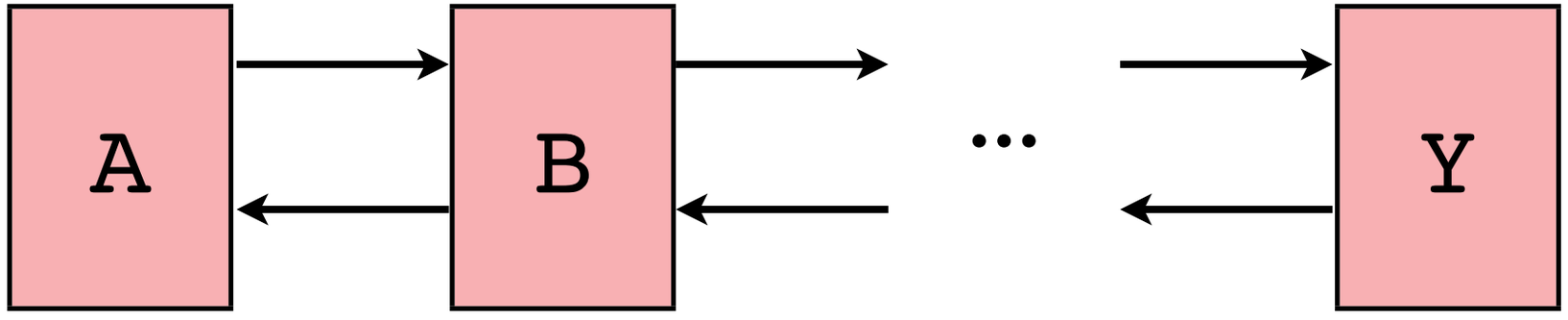
deque is a double-ended queue



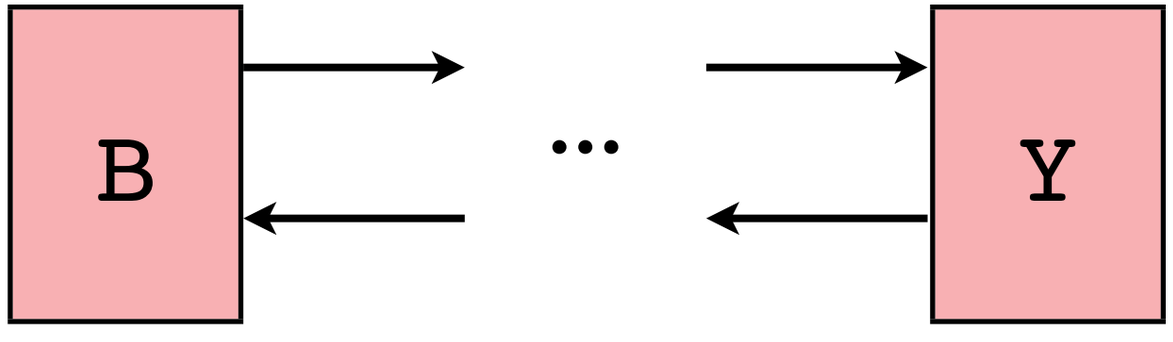
collections.deque(string.ascii_uppercase)



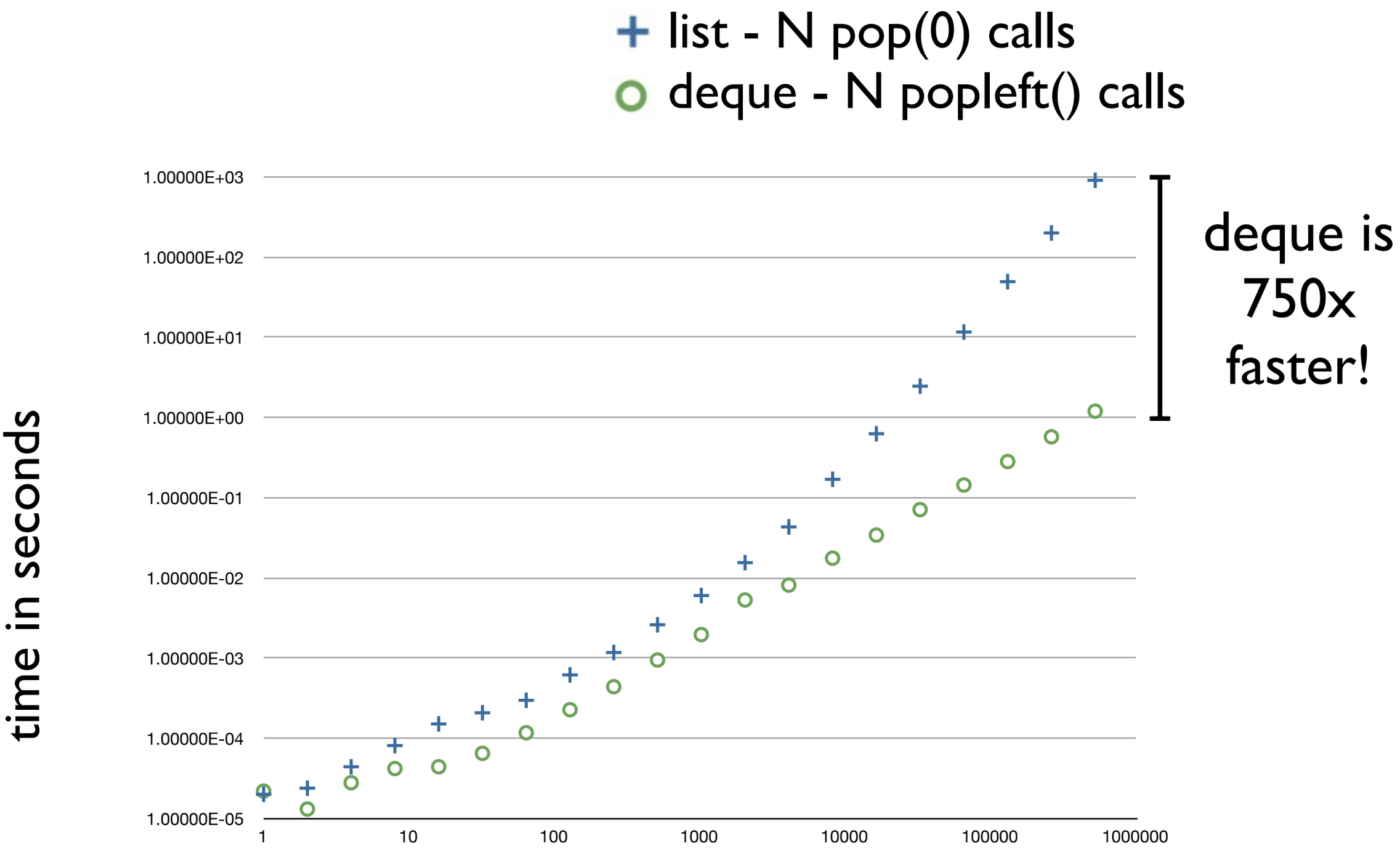
pop()



popleft()



Append N items followed by:



Even with N=1
deque is 25% faster

deque.pop() is also faster than list.pop()

Using a deque for breadth-first search

```
def find_filename_bfs(root, target_filename):
    # Keep track of the directories to search
    search_queue = collections.deque([root])

    while search_queue:
        # Pop the bottom item from the stack
        dirname = search_queue.popleft()

        for filename in os.listdir(dirname):
            path = os.path.join(dirname, filename)

            # Does the filename exist in the directory?
            if filename == target_filename:
                return path

            # If it's a directory, add it to the set
            # of directories I need to search
            if os.path.isdir(path):
                search_queue.append(path)

    # Not found
    return None
```

(Why do I use "_stack" or "_queue" in the variable name?)

deques and FIFO task queues

```
tasks = collections.deque([first_task])

while tasks:
    running_task = tasks.popleft()
    new_tasks = running_task.run()
    tasks.extend(new_tasks)
```

deques support a maximum size

“Keep track of the last N things”

```
from collections import deque
from itertools import chain, islice

history = deque([], 3)
with open("/usr/share/dict/words") as f:
    for line in f:
        if line != "Miami\n":
            history.append(line)
        else:
            print("===== Start =====")
            for line in chain(history, [line], islice(f, 0, 3)):
                print(line, end="")
            print("===== End =====")
            break
```

```
===== Start =====
mho
mhometer
mi
Miami
miamia
mian
Miao
===== End =====
```

Working with sorted data

What if I want to find an object?

Prerequisite: objects implement `__eq__`

```
def index(container, value):
    for i, item in enumerate(container):
        if item == value:
            return i
raise ValueError(value)
```

This is the list.index() algorithm

What if elements support `__lt__`?

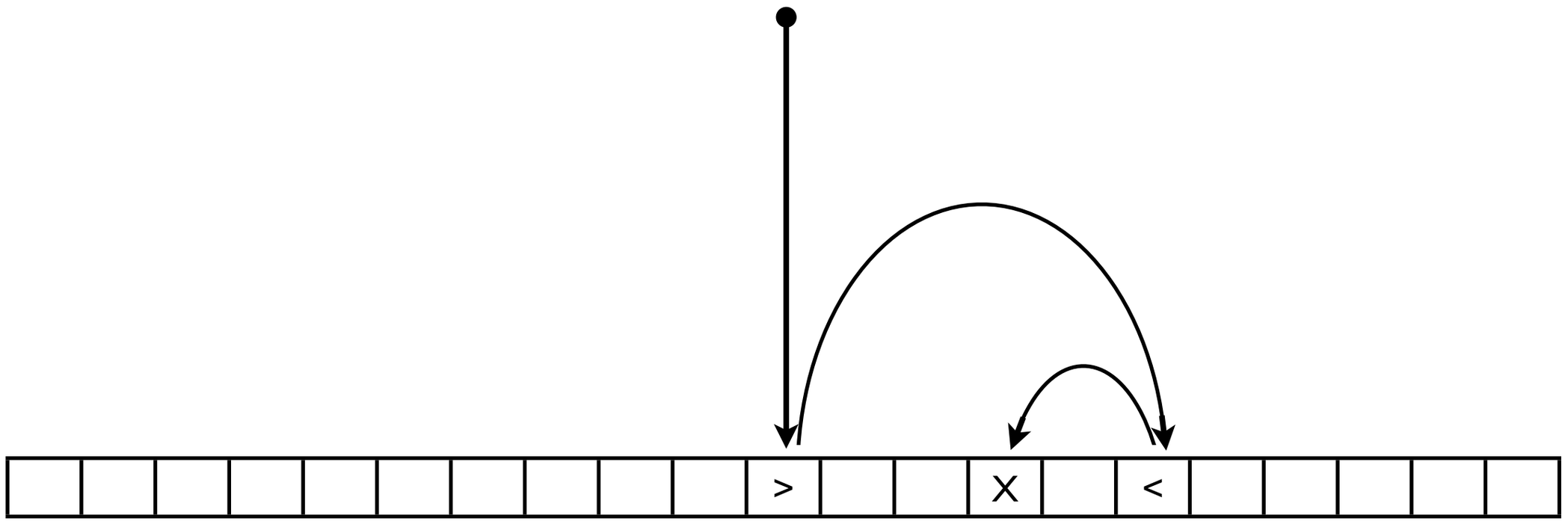
Then we can sort them!

```
>>> f = open("/usr/share/dict/words")
>>> words = [line.strip() for line in f]
>>> words[:4]
['A', 'a', 'aa', 'aal']
>>> words.sort()
>>> words[:4]
['A', 'Aani', 'Aaron', 'Aaronic']
>>>
```

Or: `words = sorted(line.strip() for line in f)`

Searching a sorted list takes O(log(N)) time

Use a binary search



Binary search is notoriously hard to get right.

Use Python's bisect module.

(plus the helper functions in the documentation)

Based on the documentation

```
import bisect

def index(container, value):
    i = bisect.bisect_left(container, value)
    if i != len(container) and container[i] == value:
        return i
    raise ValueError(value)
```

```
>>> index(words, "hello")
97803
>>> index(words, "hello2")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in index
ValueError: hello2
```

Linear search vs. bisect search of an ordered list of words

index	word	list.index	bisect index
0	A	0.3 µs	1.3 µs
46	Abigail	1.2	1.2
1000	Amazonian	19	1.2
117468	liang	2,900	1.1
234935	zythum	5,400	1.2

The abstract data type is
"binary searchable list"

The concrete implementation uses

- a Python list,
- the bisect module,
- convention on how
  they work together

What is the longest word in the list of words?

```
>>> max((len(word), word) for word in words)
(24, 'thyroparathyroidectomize')
>>>
```

What are the top five longest word?

```
>>> sorted(words, reverse=True, key=len)[:5]
['formaldehydesulphoxylate', 'pathologicopsychological',
'scientificophilosophical', 'tetraiodophenolphthalein',
'thyroparathyroidectomize']
>>>
```

Sorting the entire list takes O(N log(N)) time!

Workaround solution

```
def find_longest(words, count=1):
    largest = []
    for word in words:
        # Use "-len(word)" so the shortest word
        # has the largest (least negative) number
        largest.append( (-len(word), word) )

        # Sort so the smallest word is last
        largest.sort()

        if len(largest) > count:
            largest.pop()

    return [word for (negsize, word) in largest]
```

O(N) in the number of words, but large overhead

heapq module

Priority Queue

Elements must support comparison (`__lt__`)

You can "push" and "pop", like a stack or queue

"pop" removes and returns the smallest item

"push" and "pop" take O(log(N)) time

(Note: many people say that elements have a "priority", and pop returns the item with the highest priority. Python's terminology reflects the implementation similarities to a sorted list.)

Priority queues are built from:

- a Python list (the concrete data type)
- functions from the heapq module
- convention to not break heap invariants

```
>>> import heapq
>>> data = []
>>> heapq.heappush(data, 5)
>>> heapq.heappush(data, 3)
>>> heapq.heappush(data, 7)
>>> data[0]
3
>>> heapq.heappop(data)
3
>>> data[0]
5
```

Longest "count" words using a priority queue

```
from heapq import heappush, heappop

def find_longest(words, count=1):
    largest = []
    for word in words:
        # Use len(word) so the shortest word
        # has the smallest value (it gets popped first)
        heappush(largest, (len(word), word) )

        if len(largest) > count:
            heappop(largest)

largest.sort(reverse=True)
return [word for (n, word) in largest]
```

(there are ways to make this a bit faster)

heapq.nsmallest / heapq.nlargest

The heapq module has a better version of this algorithm

```
>>> heapq.nlargest(5, words, key=len)
['formaldehydesulphoxylate', 'pathologicopsychological',
'scientificophilosophical', 'tetraiodophenolphthalein',
'thyroparathyroidectomize']
```

heapq also implements a merge sort given sorted iterables

Scheduling with a Priority Queue

Need to schedule tasks to run in the future.

```
import time
import itertools
from heapq import heappush, heappop

counter = itertools.count(1)  # unique identifier

tasks = []

def add_task(delay, task):
    heappush(tasks, (time.time() + delay, next(counter), task) )

def process_tasks():
    while tasks:
        t, uid, task = heappop(tasks)
        dt = t - time.time()
        if dt >= 0:
            time.sleep(dt)
        task()
```

```
def stretch():
    print("Stretch!")
    add_task(3, stretch)

def save():
    print("Save your code!")
    add_task(5, save)

add_task(1, stretch)
add_task(1, save)
process_tasks()
```

Tuple

Two common uses of a tuple

"a read-only list" ("frozenlist")

"a light-weight object"

Both are supported, but strongly weighted to the right.

Tuples are for heterogeneous data, list are for homogeneous data.
Tuples are *not* read-only lists.

Guido van Rossum March 12, 2003 in python-dev

Don’t treat Guido’s statement as a mandate!

Yes, taking Guido too seriously can have that effect on people <1.1 wink>
The trick is in knowing when Guido is proscribing dogmatic law, and when he is describing his intent when he designed feature(tte)s of Python, or his point of view at this moment, or how he wished people would see it, or reacting to what he had for lunch. He's just human, you know, and no more infallible than the best of us.

Thomas Wouters March 12, 2003 in python-dev

```
- Issue #2025: Add tuple.count() and tuple.index()
    methods to comply with the collections.Sequence API.
```

NEWS in Python-2.6

```
[count() and index()] shouldn't be a burden for
implementers who use real inheritance from Sequence,
as they are concrete methods there. And it doesn't
make sense to move them to MutableSequence, because
there's nothing in them that requires the sequence to
be mutable.
```

[Python-3000] ABC method mismatch, Feb 6 2008

Raymond Hettinger, *Guido van Rossum*, Fred Drake

Tuples as dictionary keys

```
color_names = {
   (255,0,0): "red",
   (128,128,0): "olive",
   (255,255,0): "yellow",
   (0,128,128): "teal",
}
def to_hex(color):
   return "#%02x%02x%02x" % color
```

```
>>> color = (255, 0, 0)
>>> color_names.get(color) or to_hex(color)
'red'
>>> color = (0, 0, 0)
>>> color_names.get(color) or to_hex(color)
'#000000'
```

Tuples as light-weight objects

```
color = (255,   0,   0)
```

red green blue

Just have everyone agree that color[0] means red, color[1] means green, color[2] means blue

Just like we might agree that color.red means red, color.green means green, and color.blue means blue.

It's easier to understand attributes

Python 2.5

```
>>> import time
>>> time.gmtime()
(2011, 6, 17, 22, 31, 25, 4, 168, 0)
```

Python 2.6

```
>>> import time
>>> time.gmtime()
time.struct_time(tm_year=2011, tm_mon=6,
tm_mday=17, tm_hour=22, tm_min=31, tm_sec=42,
tm_wday=4, tm_yday=168, tm_isdst=0)
>>> time.gmtime()[0]
2011
>>> time.gmtime().tm_year
2011
```

Migrating to “heavy-weight” objects is boring

```
class Color(tuple):
    def __new__(cls, red, green, blue):
        return tuple.__new__(cls, (red, green, blue))
    @property
    def red(self):
        return self[0]
    @property
    def green(self):
        return self[1]
@property
def blue(self):
    return self[2]
```

... and surprisingly tricky!

```
>>> color = Color(255, 127, 0)
>>> color.red, color.green, color.blue
(255, 127, 0)
>>> periwinkle = Color(red=142,
...                    green=130, blue=254)
>>> periwinkle.blue
254
```

collections.namedtuple

```
>>> from collections import namedtuple
>>> Color = namedtuple("Color", "red green blue")
>>> Color(142, 130, 254).red
142
>>> Color(red=142, green=130, blue=254).green
130
>>>
```

Special “_asdict” and “_replace” methods

```
>>> orange = Color(0xf9, 0x73, 0x06)
>>> orange
Color(red=249, green=115, blue=6)
>>> orange._asdict()
OrderedDict([('red', 249), ('green', 115), ('blue', 6)])
>>> orange._asdict()["blue"]
6
>>> orange._replace(green=99)
Color(red=249, green=99, blue=6)
```

Remember - it’s still a tuple!

```
>>> black = Color(0, 0, 0)
>>> black.count(0)
3
```

Even more list-like collections in Python

- array module for homogenous types

```
>>> x = array.array("i", (7,4,7))
>>> x.count(7)
2
>>>
```

- queue module for threaded programming

Queue, LifoQueue and PriorityQueue

(deque is also thread-safe)

Set

Sets are an unsorted collection of elements

Duplicates are ignored

⇒ elements must implement __eq__

Sets are implemented as a hash table.

⇒ elements must implement __hash__

```
>>> values = {3, 1, 4, 1, 5, 9}
>>> values
{9, 3, 4, 5, 1}
>>> if username in {"root", "admin", "dalke"}:
...     auth.grant_all()
```

Sets are great for uniqueness

How many unique words do I have if I ignore capitalization?

```
>>> len(words)
234936
>>> len(set(word.lower() for word in words))
233614
>>>
```

"Is that Polish?"

"Is that polish?"

"It was an August day."

"It was an august day."

Only print one word of each given length

```
>>> seen_length = set()
>>> for word in words:
...     n = len(word)
...     if n in seen_length:
...         continue
...     seen_length.add(n)
...     print(word)
...
```

```
A
Aani
Aaron
Aaronic
Aaronical
Aaronite
Ab
Ababua
Abdominales
Abe
Abencerrages
Aberdonian
Acanthocephala
Acanthocereus
Acanthopterygii
Achromobacterieae
Actinomycetaceae
Archaeopterygiformes
Australopithecinae
Chlamydobacteriales
Prorhipidoglossomorpha
Pseudolamellibranchia
Pseudolamellibranchiata
formaldehydesulphoxylate
>>>
```

Sets are great for uniqueness

(though not essential)

```
>>> len(dict.fromkeys(word.lower() for word in words))
233614
```

Sets are more than “value-less dictionaries”

Set operations

```
>>> hamlet = {"to","be","or","not","to","be"}
>>> hamlet
{'not', 'be', 'or', 'to'}


>>> hamlet - {"to","be","me"}
{'not', 'or'}

>>> hamlet & {"to","be","me"}
{'be', 'to'}

>>> hamlet ^ {"to","be","me"}
{'not', 'or', 'me'}

>>> hamlet.union("to be me".split())
{'not', 'be', 'or', 'me', 'to'}

>>> hamlet.issubset(
...           "to be me and not you or him".split())
True
```

Inverted Index

Make a mapping of each letter [a-z] to a set.
The set for “a” contains all words which have an “a” or “A”

```
inverted = {
  “a”:   {'fawn', 'cisandine', 'Ichthyoidea', 'unsupportable', 'unattackab
    ⋮
  “n”:   {'fawn', 'cisandine', 'nunnery', 'unsupportable', 'unattackable',
    ⋮
  “s”:   {'cisandine', 'Phasiron', 'unsupportable', 'amirship', 'smurry',
    ⋮
}
```

Make the Inverted Index

```
import string

inverted = {}

for c in string.ascii_lowercase:
    inverted[c] = set()

for word in words:
    for c in word.lower():
        inverted[c].add(word)
```

Which words contain a “q” but no “u”?

```
>>> inverted["q"] - inverted["u"]
{'qoph', 'shoq', 'qintar', 'qasida', 'qere', 'Iraq',
'Q', 'miqra', 'q', 'Iraqi', 'Qoheleth', 'nastaliq',
'qeri', 'Iraqian', 'Pontacq'}
>>>
```

Which words contain ‘q’, ‘x’, and ‘z’?

```
>>> inverted["q"] & inverted["x"] & inverted["z"]
{'extraquiz', 'benzoquinoxaline', 'benzofuroquinoxaline', 'quixotize'}

>>> inverted["q"].intersection(inverted["x"], inverted["z"])
{'extraquiz', 'benzoquinoxaline', 'benzofuroquinoxaline', 'quixotize'}

>>> set.intersection(*(inverted[c] for c in "qxz"))
{'extraquiz', 'benzoquinoxaline', 'benzofuroquinoxaline', 'quixotize'}
```

What are the 10 longest words which contain a ‘y’ but no other vowel?

```
>>> import heapq
>>> heapq.nlargest(10, inverted["y"] -
...     set.union(*(inverted[c] for c in "aeiuo")),
...     key=len)
['symphysy', 'gypsyry', 'nymphly', 'gypsyfy', 'lymphy',
'rhythm', 'Flysch', 'syzygy', 'strych', 'sylphy']
>>>
```

Frozenset

"Constant" set you can use as keys in a dictionary.

I haven't found an example of where I should use a frozenset instead of a tuple with ordered elements.

Dictionary

Dictionaries are an unsorted mapping of keys to values

Dictionaries are implemented as a hash table

⇒ keys must implement __eq__ and __hash__

Brandon Craig Rhodes “The Mighty Dictionary” at PyCon 2010

http://python.mirocommunity.org/video/1591/pycon-2010-the-mighty-dictiona

__missing__ method

```
import socket

class DNSLookup(dict):
    def __missing__(self, hostname):
        print("Looking up", hostname)
        try:
            addr = socket.gethostbyname(hostname)
        except socket.error:
            addr = None
        self[hostname] = addr
        return addr
```

```
>>> dns_lookup = DNSLookup()
>>> dns_lookup["dalkescientific.com"]
Looking up dalkescientific.com
'66.39.47.217'
>>> dns_lookup["unknown.dalkescientific.com"]
Looking up unknown.dalkescientific.com
>>> dns_lookup["python.org"]
Looking up python.org
'82.94.164.162'
>>> dns_lookup["dalkescientific.com"]
'66.39.47.217'
>>> dns_lookup
{'unknown.dalkescientific.com': None, 'python.org': '82.94.164.162',
'dalkescientific.com': '66.39.47.217'}
```

```
class DefaultDict(dict):
    def __init__(self, callable):
        self.callable = callable
    def __missing__(self, name):
        item = self.callable()
        self[name] = item
        return item
```

```
>>> int()
0
>>> d = DefaultDict(int)
>>> d["a"]
0
>>> d["b"] += 1
>>> d["b"]
1
>>> d
{'a': 0, 'b': 1}
>>>
```

collections.defaultdict

```
import string
from collections import defaultdict

d = defaultdict(int)

with open("/usr/share/dict/words") as f:
    for line in f:
        for c in line.lower():
            d[c] += 1
```

```
>>> scale = max(d[c] for c in string.ascii_lowercase)
>>> for c in string.ascii_lowercase:
...     print(c,"="*(d[c]*75//scale+1))
...
```

```
a ==========================================================
b ============
c ==============================
d ======================
e ================================================================
f ========
g ===============
h ====================
i ==========================================================
j =
k ======
l ========================================
m ======================
n ==================================================
o ==================================================
p ========================
q ==
r ==================================================
s ========================================
t ==================================================
u ============================
v =======
w =====
x ===
y ================
z ===
```

Reverse a dictionary

The original dictionary may have duplicate values.
Result will map value to list of corresponding keys.

```
def reverse_dict(d):
    reversed = defaultdict(list)
    for k,v in d.items():
        reversed[v].append(k)
    return dict(reversed)
```

I return a normal dict!

```
>>> lengths = dict( (word, len(word))
...                  for word in "to be or not to be".split() )
>>> lengths
{'not': 3, 'to': 2, 'or': 2, 'be': 2}
>>> print(reverse_dict(lengths))
{2: ['to', 'or', 'be'], 3: ['not']}
>>>
```

collections.Counter

Counting elements is a very common task

```
>>> from collections import Counter
>>> letter_counter = Counter("bookkeeper")
>>> letter_counter
Counter({'e': 3, 'k': 2, 'o': 2, 'b': 1, 'p': 1, 'r': 1})
>>> isinstance(letter_counter, dict)
True

>>> letter_counter.most_common(3)
[('e', 3), ('k', 2), ('o', 2)]
>>> list(letter_counter.elements())
['b', 'e', 'e', 'e', 'k', 'k', 'o', 'o', 'p', 'r']

>>> letter_counter - Counter("beekeeper")
Counter({'o': 2, 'k': 1})
>>>
```

Counting all letters in the word list

```
from collections import Counter

letters = Counter()

with open("/usr/share/dict/words") as f:
    for line in f:
        letters.update(line.strip().lower())
```

```
>>> letters.most_common(10)
[('e', 234803), ('i', 200613), ('a', 198938), ('o',
170392), ('r', 160491), ('n', 158281), ('t', 152570), ('s',
139238), ('l', 130172), ('c', 103307)]
>>> letters.most_common()[-1]
('j', 3075)
>>>
```

OrderedDict

Dictionaries are unordered

```
#FPS1
#num_bits=166
#type=ChemFP-RDMACCS-RDKit/1
#source=Compound_007700001_007725000.sdf.gz
#date=2011-05-26T23:28:07
00008002003083606068 40a03705405bb2e1abea1f 7700001
00008000100400000c0b 2007fa17275e89dfaf7ff1f 7700003
...
```

```
headers = {}

with open(input_filename) as f:
  assert next(f) == "#FPS1\n"
  for line in f:
    if line.startswith("#"):
      key, value = line[1:-1].split("=", 1)
      headers[key] = value
    else:
      break
```

```
#FPS1
#num_bits=166
#type=ChemFP-RDMACCS-RDKit/1
#source=Compound_007700001_007725000.sdf.gz
#date=2011-05-26T23:28:07
```

I can't reproduce the input order

```
>>> headers
{'date': '2011-05-26T23:28:07', 'source':
'Compound_007700001_007725000.sdf.gz', 'num_bits': '166', 'type':
'ChemFP-RDMACCS-RDKit/1'}
>>>
>>> for k, v in headers.items():
...   print("#", k, "=", v, sep="")
...
#date=2011-05-26T23:28:07
#source=Compound_007700001_007725000.sdf.gz
#num_bits=166
#type=ChemFP-RDMACCS-RDKit/1
>>>
```

OrderedDict to the rescue

```
>>> from collections import OrderedDict
>>>
>>> d = OrderedDict()
>>> d["first"] = 1
>>> d["second"] = 2
>>> d["third"] = 3
>>> d
OrderedDict([('first', 1), ('second', 2), ('third', 3)])
>>> for k,v in d.items():
...     print(k,v)
...
first 1
second 2
third 3
```

compare to a regular dict

```
>>> d = {}
>>> d["first"] = 1
>>> d["second"] = 2
>>> d["third"] = 3
>>> for k,v in d.items():
...     print(k,v)
...
second 2
third 3
first 1
>>>
```

“cast in order of appearance”

Julius Caesar in XML form

```
...
<SPEAKER>FLAVIUS</SPEAKER>
<LINE>Hence! home, you idle creatures get you home:</LINE>
<LINE>Is this a holiday? what! know you not,</LINE>
<LINE>Being mechanical, you ought not walk</LINE>
<LINE>Upon a labouring day without the sign</LINE>
<LINE>Of your profession? Speak, what trade art thou?</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>First Commoner</SPEAKER>
<LINE>Why, sir, a carpenter.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>MARULLUS</SPEAKER>
<LINE>Where is thy leather apron and thy rule?</LINE>
<LINE>What dost thou with thy best apparel on?</LINE>
<LINE>You, sir, what trade are you?</LINE>
</SPEECH>
...
```

FLAVIUS
First Commoner
MARULLUS
…

Speaking roles in Julius Caesar in order

```
from xml.etree import ElementTree
from collections import OrderedDict

tree = ElementTree.parse("j_caesar.xml")

speakers = tree.findall("//SPEAKER")
ordered_speakers = OrderedDict.fromkeys(
                    node.text for node in speakers)

for speaker in ordered_speakers:
    print(speaker)
```

(Should there be an OrderedSet?)

FLAVIUS
First Commoner
MARULLUS
Second Commoner
CAESAR
CASCA
CALPURNIA
ANTONY
Soothsayer
BRUTUS
CASSIUS
CICERO
CINNA
LUCIUS
DECIUS BRUTUS
METELLUS CIMBER
TREBONIUS
PORTIA
LIGARIUS
Servant
PUBLIUS
ARTEMIDORUS
POPILIUS
Citizens
First Citizen
Second Citizen
Third Citizen
All
Fourth Citizen
Several Citizens
CINNA THE POET
OCTAVIUS
LEPIDUS
LUCILIUS
PINDARUS
First Soldier
Second Soldier
Third Soldier
Poet
MESSALA
TITINIUS
VARRO
GHOST
CLAUDIUS
Messenger
CATO
CLITUS
DARDANIUS
VOLUMNIUS
STRATO

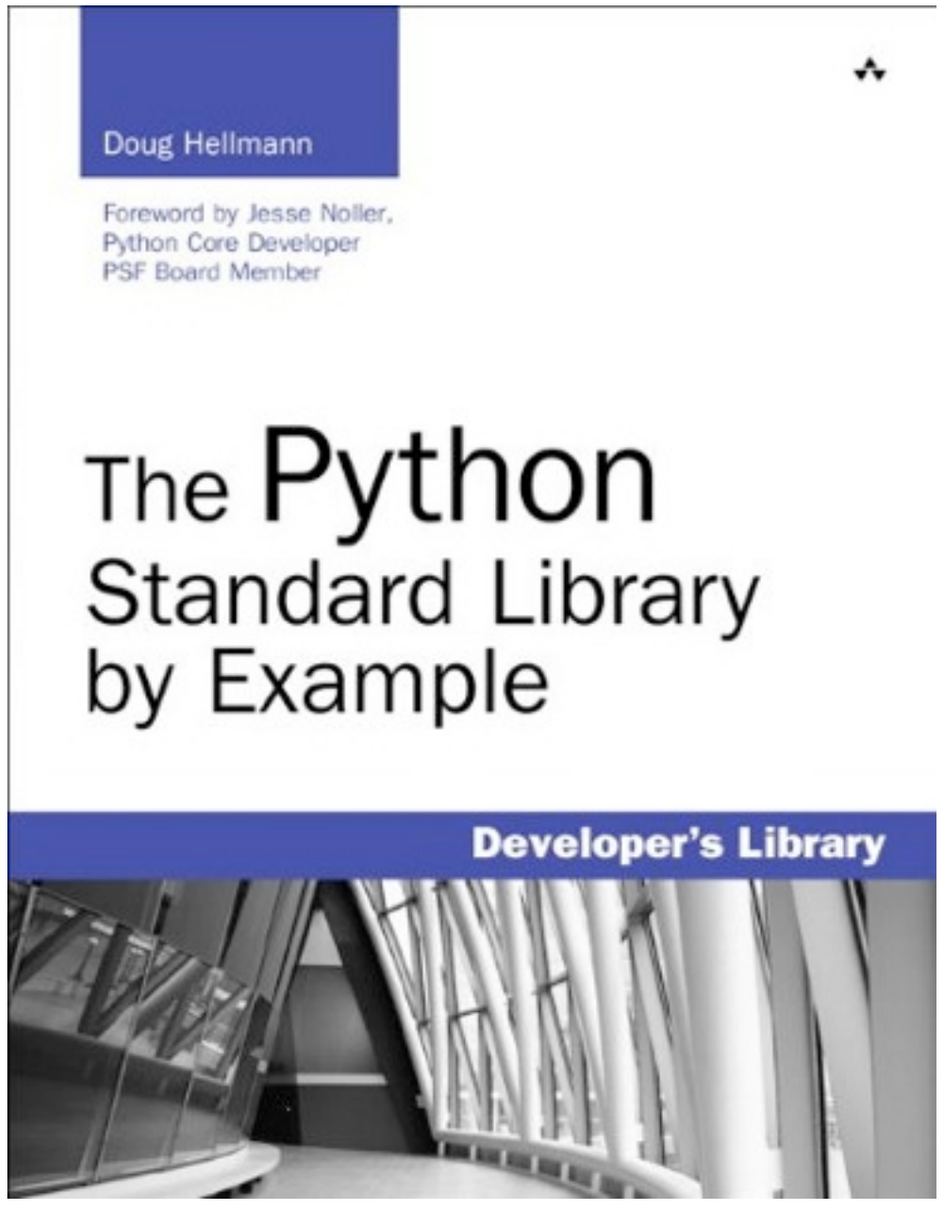
More details found in:

⇒ the Python documentation

⇒ Doug Hellmann's new book

Read the preview chapter

(then go ahead and buy it)



⇒ Raymond Hettinger's presentation

"Fun with Python's Newer Tools" at PyCon 2011

Python's Other Collection Types and Algorithms

Questions?

Andrew Dalke
dalke@dalkescientific.com

dalke scientific
More science. Less time.