# Python Data Handling: A Deeper Dive

David Beazley
@dabeaz
http://www.dabeaz.com

Pearson

# Python Fluency

- Mastery of Python's built-in types, useful libraries, and data handling idioms are a fundamental part of Python literacy
- You shouldn't even have to think twice about it in day-to-day coding
- This course is about reinforcing your skills
- Going beyond an introductory tutorial

# Beyond Frameworks

- You might be inclined to turn to libraries and frameworks to solve common data problems
- "Look up the command"
- But, Python provides useful building blocks
- You can quickly code a lot of things yourself if you know how to put them together

# Materials and Setup

- Supporting code and data for this course:

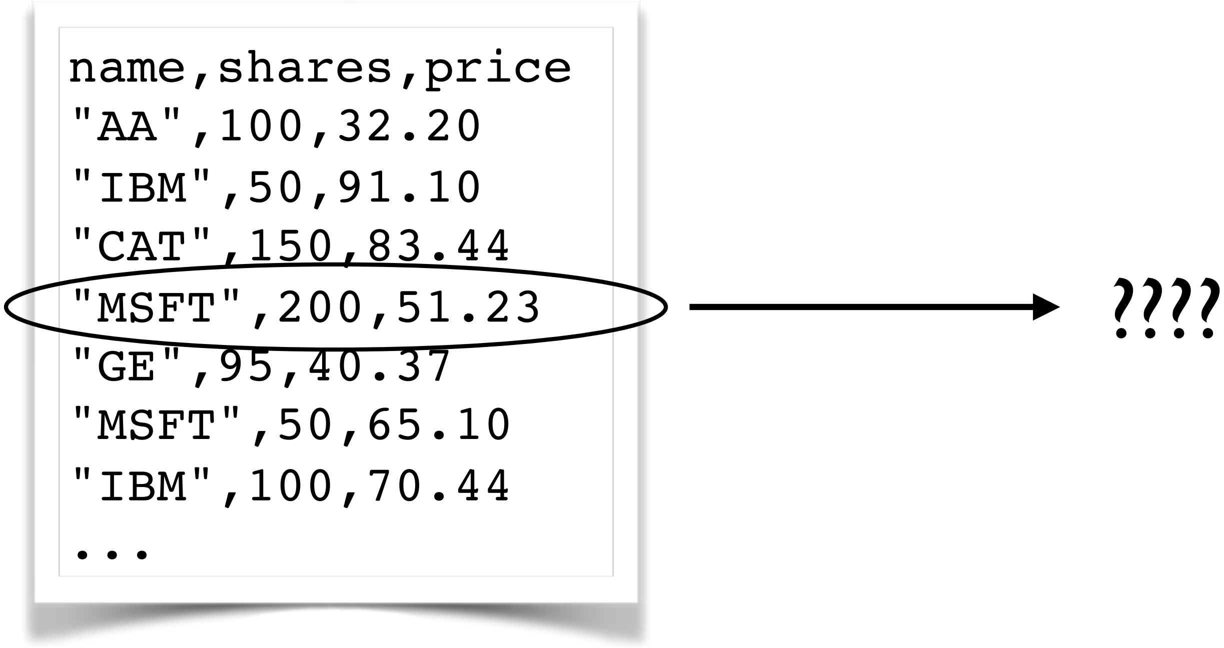
  http://www.dabeaz.com/datadeepdive

- Python 3.6+ is assumed
- Any operating system is fine
- Slides are merely a guide. Presentation will rely heavily on live-demos, examples.

# Part 1: Data Structure Shootout

# Problem

Some data ...



```
name,shares,price
"AA",100,32.20
"IBM",50,91.10
"CAT",150,83.44
"MSFT",200,51.23
"GE",95,40.37
"MSFT",50,65.10
"IBM",100,70.44
...
```

????

How do you "best" represent records/ structures in Python?

# Tuples

- A collection of values packed together

```
s = ('GOOG', 100, 490.1)
```

- Can use like an array

```
name = s[0]
cost = s[1] * s[2]
```

- Unpacking into separate variables

```
name, shares, price = s
```

- Immutable

```
s[1] = 75      # TypeError. No item assignment
```

# Dictionaries

- An unordered set of values indexed by "keys"

```
s = {
   'name'    : 'GOOG',
   'shares'  : 100,
   'price'   : 490.1
  }
```

- Use the key name to access

```
name = s['name']
cost = s['shares'] * s['price']
```

- Modifications are allowed

```
s['shares'] = 75
s['date'] = '7/25/2015'
del s['name']
```

# User-Defined Classes

- A simple data structure class

```
class Stock(object):
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

- This gives you the nice object syntax...

```
>>> s = Stock('GOOG', 100, 490.1)
>>> s.name
'GOOG'
>>> s.shares * s.price
49010.0
>>>
```

# Classes and Slots

- For data structures, consider adding __slots__

```
class Stock(object):
    __slots__ = ('name', 'shares', 'price')
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

- Slots is a performance optimization that is specifically aimed at data structures
- Less memory and faster attribute access

# Named Tuples

- namedtuple(*clsname*, *fieldnames*)

```
from collections import namedtuple

Stock = namedtuple('Stock',
                   ['name', 'shares', 'price'])
```

- It creates a class that you use to make instances

```
>>> s = Stock('GOOG',100,490.1)
>>> s.name
'GOOG'
>>> s.shares * s.price
49010.0
>>>
```

- Instances look like tuples

# Challenge

The file "ctabus.csv" is a CSV file containing ridership data from the Chicago Transit Authority bus system.

```
route,date,daytype,rides
3,01/01/2001,U,7354
4,01/01/2001,U,9288
6,01/01/2001,U,6048
8,01/01/2001,U,6309
```

15.7MB,
736000+ rows

What's the most efficient way to read it into a Python list so that you can work with it?

# Part 2: Collections

# Collecting Things

- Programs often have to work many objects
- And build relationships between objects
- There are some basic building blocks
  - Lists, tuples, sets, dicts
  - collections module
- Better to think about nature of the problem

# Keeping Things in Order

- Use lists when the <u>order</u> of data matters
- Example: A list of tuples

```
portfolio = [
    ('GOOG', 100, 490.1),
    ('IBM', 50, 91.1),
    ('CAT', 150, 83.44)
]
```

```
portfolio[0] ⟶ ('GOOG', 100, 490.1)
portfolio[1] ⟶ ('IBM', 50, 91.1)
```

- Lists can be sorted and rearranged

# Keeping Distinct Items

- Use a set for keeping unique/distinct objects

```
a = {'IBM','AA','AAPL' }
```

- Converting to a set will eliminate duplicates

```
names = ['IBM','YHOO','IBM','CAT','MSFT','CAT','IBM']
unique_names = set(names)
```

- Sets are useful for membership tests

```
members = set()

members.add(item)       # Add an item
members.remove(item)    # Remove an item

if item in members:     # Test for membership
   ...
```

# Building an Index/Mapping

- Use a dictionary (maps keys -> values)

```
prices = {
    'GOOG' : 513.25,
    'CAT'  : 87.22,
    'IBM'  : 93.37,
    'MSFT' : 44.12
    ...
}
```

- Usage

```
p = prices['IBM']              # Value lookup
prices['HPE'] = 37.42          # Assignment
if name in prices:             # Membership test
   ...
```

# Composite Keys

- Use tuples for keys

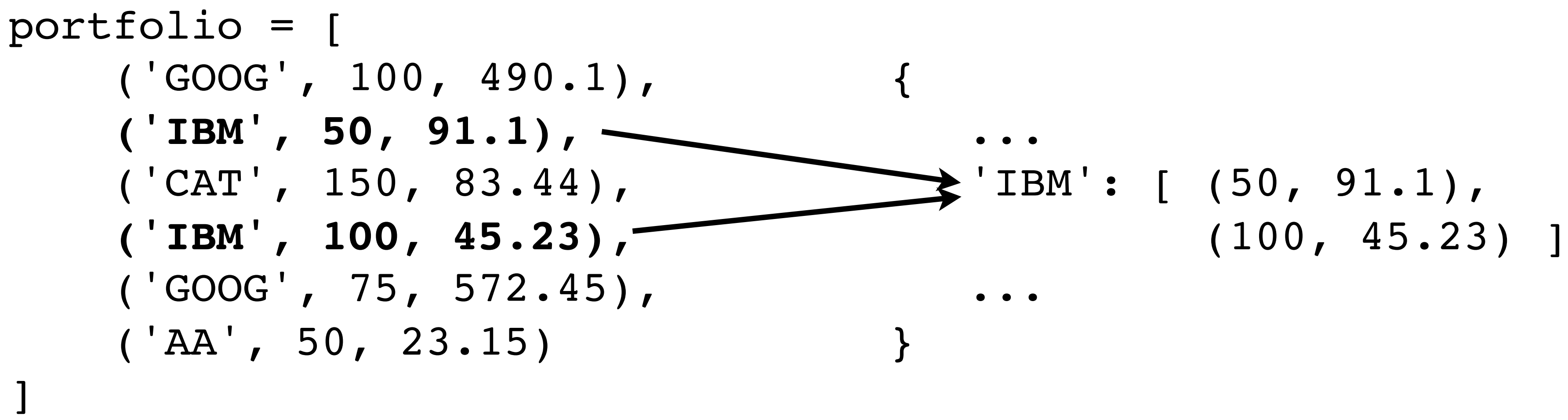
```
prices = {
    ('GOOG', '2017-02-01'): 517.20,
    ('GOOG', '2017-02-02'): 518.23,
    ('GOOG', '2017-02-03'): 518.71,
    ...
    ('IBM', '2017-02-01'): 92.50,
    ('IBM', '2017-02-02'): 92.72,
    ('IBM', '2017-02-03'): 91.92,
    ...
}
```

- Usage:

```
p = prices['IBM', '2017-02-01']
prices['IBM','2017-02-04'] = 92.3
```

# One-to-Many Mapping

- Problem: Map keys to multiple values



```
portfolio = [
    ('GOOG', 100, 490.1),
    ('IBM', 50, 91.1),
    ('CAT', 150, 83.44),
    ('IBM', 100, 45.23),
    ('GOOG', 75, 572.45),
    ('AA', 50, 23.15)
]
```

```
{
    ...
    'IBM': [ (50, 91.1),
             (100, 45.23) ]
    ...
}
```

- Strategy: Store multiple values in a container
- Make the value a list, set, dict, etc.

# One-to-Many Mapping

- Common solution: defaultdict(*initializer*)

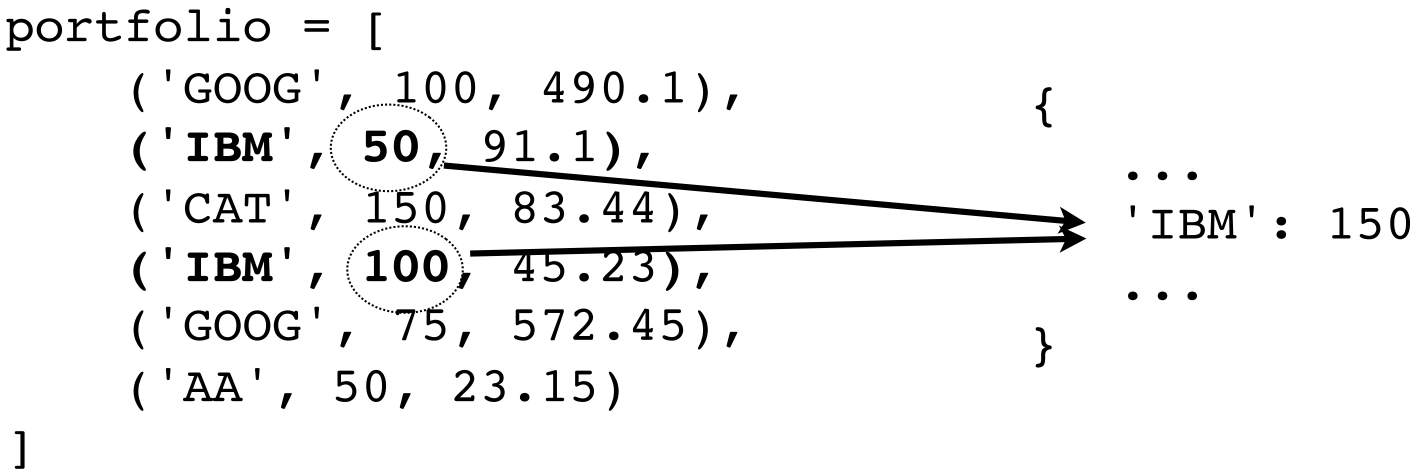
```
from collections import defaultdict
holdings = defaultdict(list)
for name, shares, price in portfolio:
    holdings[name].append((shares, price))

>>> holdings['IBM']
[ (50, 91.1), (100, 45.23) ]
>>>
```

- defaultdict automatically creates initial element

```
>>> d = defaultdict(list)
>>> d['x']
[]
>>> d
defaultdict(<class 'list'>, {'x': []})
>>>
```

# Counting Things

- Example: Tabulate total shares of each stock



```
portfolio = [
    ('GOOG', 100, 490.1),
    ('IBM', 50, 91.1),
    ('CAT', 150, 83.44),
    ('IBM', 100, 45.23),
    ('GOOG', 75, 572.45),
    ('AA', 50, 23.15)
]
```

```
{
    ...
    'IBM': 150
    ...
}
```

- Solution: Use a Counter

```
from collections import Counter
total_shares = Counter()
for name, shares, price in portfolio:
    total_shares[name] += shares

>>> total_shares['IBM']
150
>>>
```

# Challenge

Answer a few questions about the Chicago bus data...

1. How many bus routes exist?
2. How many people rode route 22 on 9-Apr-2007?
3. What are 10 most popular routes?
4. What are 10 most popular routes in 2016?
5. What 10 routes had greatest increase 2001-2016?

# Part 3: Python Object Model

# Everything is an Object

- Everything you use in Python is an "object"

```
a = None
b = 42
c = 4.2
d = "forty two"
e = [1,2,3]
f = ('ACME', 50, 91.5)

def g(x):           # Even functions are objects
    return 2*x
```

- Programs are based on manipulating objects

# Under the Covers

- All objects have an id, class and a reference count

**id**

class

refcount

...

```
class Something:
    def method_a(self):
        ...
    def method_b(self):
        ...
```

```
a = Something()
b = a
c = [..., a, ...]
```

- The id is the memory address
- The class is the "type"
- Reference count used for garbage collection

# Under the Covers

- You can investigate...

```
>>> a = "hello world"
>>> id(a)
4562360496
>>> type(a)
<class 'str'>
>>> import sys
>>> sys.getrefcount(a)
2
>>>
```

- Normally, you don't think about it too much

# Understanding Assignment

- Many operations in Python are related to "assigning" or "storing" values

```
a = value                # Assignment to a variable
s[n] = value             # Assignment to an list
s.append(value)          # Appending to a list
d['key'] = value         # Adding to a dictionary
```

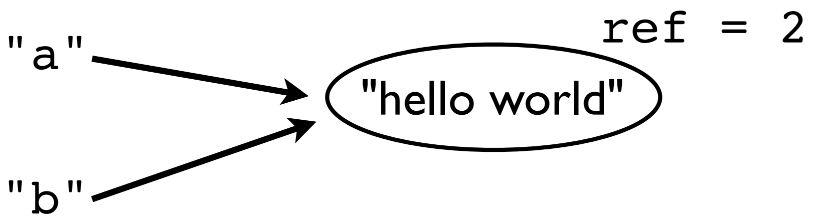
- A caution : assignment operations never make a copy of the value being assigned

- All assignments store the memory address only (object id). Increase the refcount.

# Assignment Example

- Consider this code fragment:

```
>>> a = "hello world"
>>> b = a
>>> id(a)
4562360496
>>> id(b)
4562360496
>>>
```



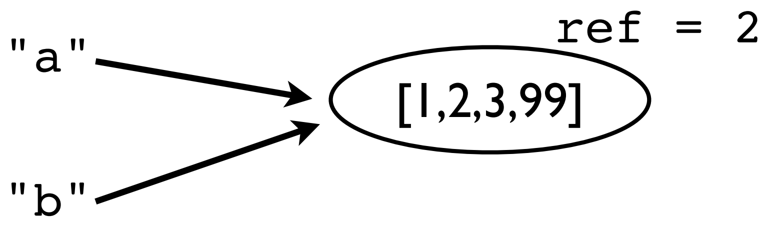
There is only one string. Two different names refer to it.

- This happens for all objects (ints, floats, etc.)
- You don't notice because of immutability

# Mutability Caution

- Consider this version:

```
>>> a = [1,2,3]
>>> b = a
>>> b
[1, 2, 3]
>>> b.append(999)
>>> b
[1, 2, 3, 999]
>>> a
[1, 2, 3, 999]
>>>
```



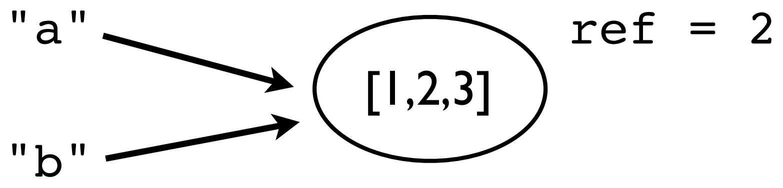
There is only one list object, but there are two references to it

- Both values change!

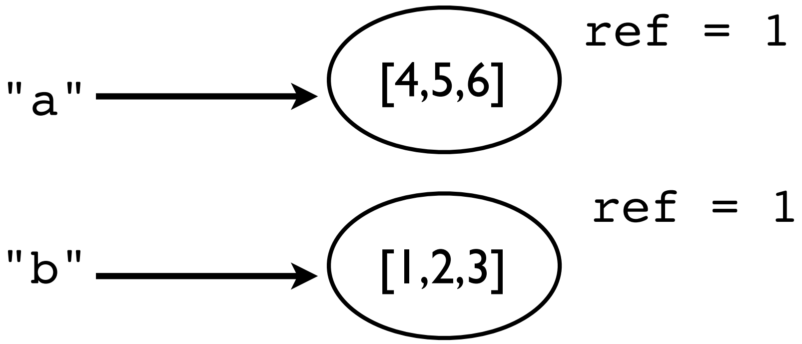
# Reassigning Values

- Assignment never overwrites an existing object

```
a = [1,2,3]
b = a
```



```
a = [4,5,6]
```



- Variables are names for objects
- Assignment moves the name elsewhere

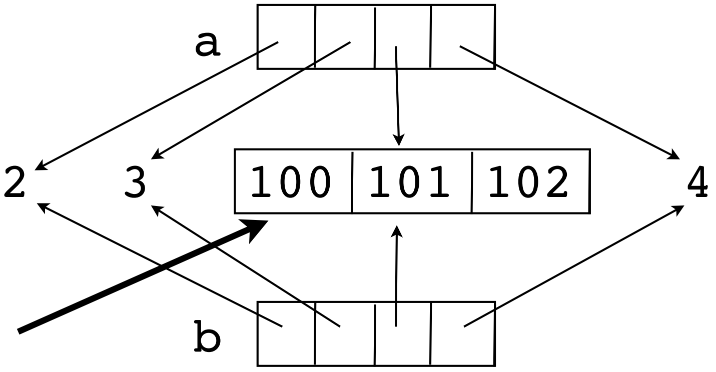
# Shallow Copies

- Containers have methods for copying

```
>>> a = [2,3,[100,101],4]
>>> b = list(a)                  # Make a copy
>>> a is b
False
```

- However, items are copied by reference

```
>>> a[2].append(102)
>>> b[2]
[100,101,102]
>>>
```



This inner list is still being shared

- Known as a "shallow copy"

# Deep Copying

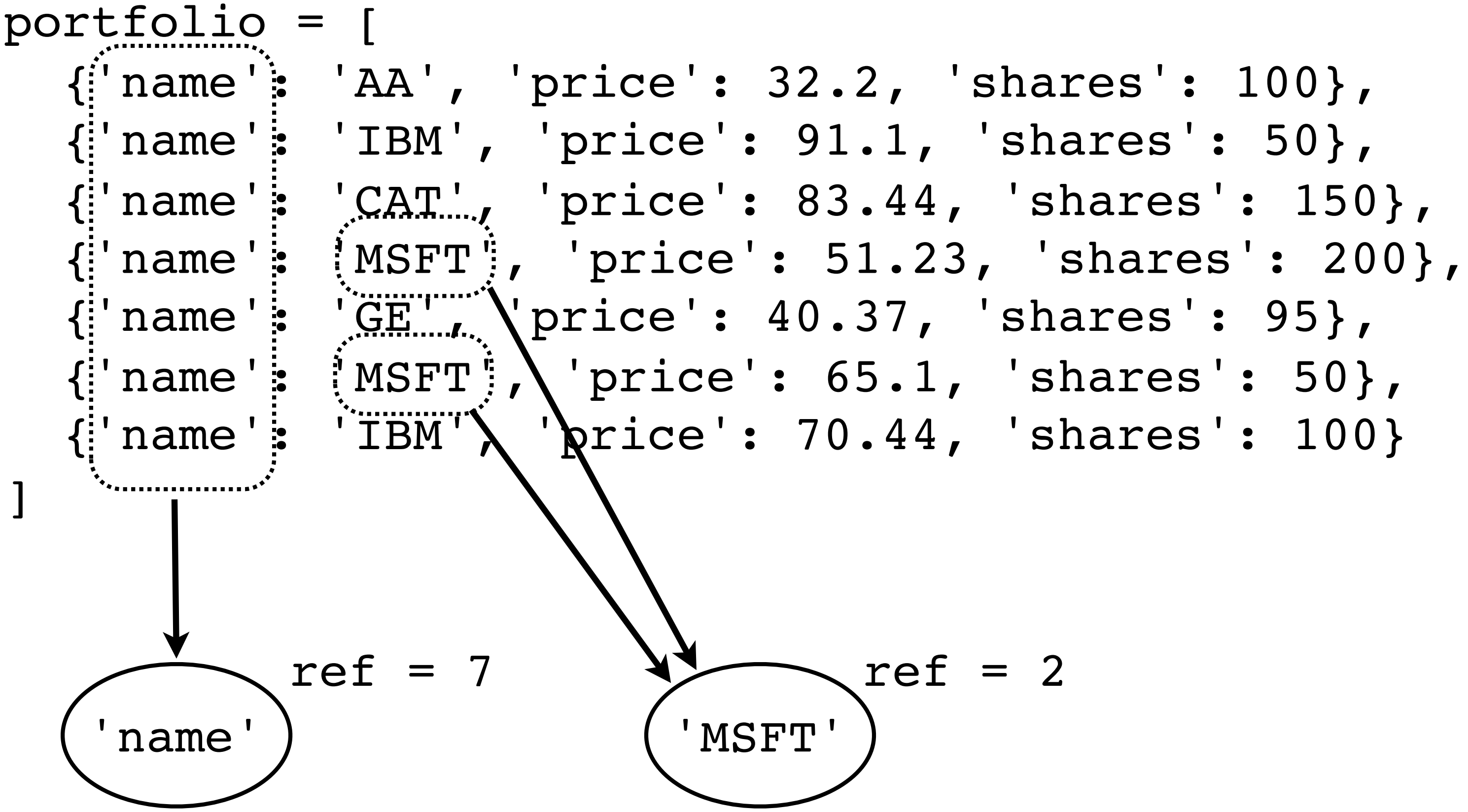
- Sometimes you need to makes a copy of an object and all objects contained within it
- Use the copy module

```
>>> a = [2,3,[100,101],4]
>>> import copy
>>> b = copy.deepcopy(a)
>>> a[2].append(102)
>>> b[2]
[100,101]
>>>
```

- This is the only safe way to copy something

# Exploiting Immutability

- Immutable values can be safely shared



```
portfolio = [
   {'name': 'AA', 'price': 32.2, 'shares': 100},
   {'name': 'IBM', 'price': 91.1, 'shares': 50},
   {'name': 'CAT', 'price': 83.44, 'shares': 150},
   {'name': 'MSFT', 'price': 51.23, 'shares': 200},
   {'name': 'GE', 'price': 40.37, 'shares': 95},
   {'name': 'MSFT', 'price': 65.1, 'shares': 50},
   {'name': 'IBM', 'price': 70.44, 'shares': 100}
]
```

- Sharing can save significant memory

# Challenge

Come up with some clever "hack" to save a lot of memory reading that CTA bus data (hint: look at the data with a hint of string caching)

# Builtin Representation

- None (a singleton)

type
refcount

(16 bytes)

- float (64-bit double precision)

type
refcount
value

(24 bytes)

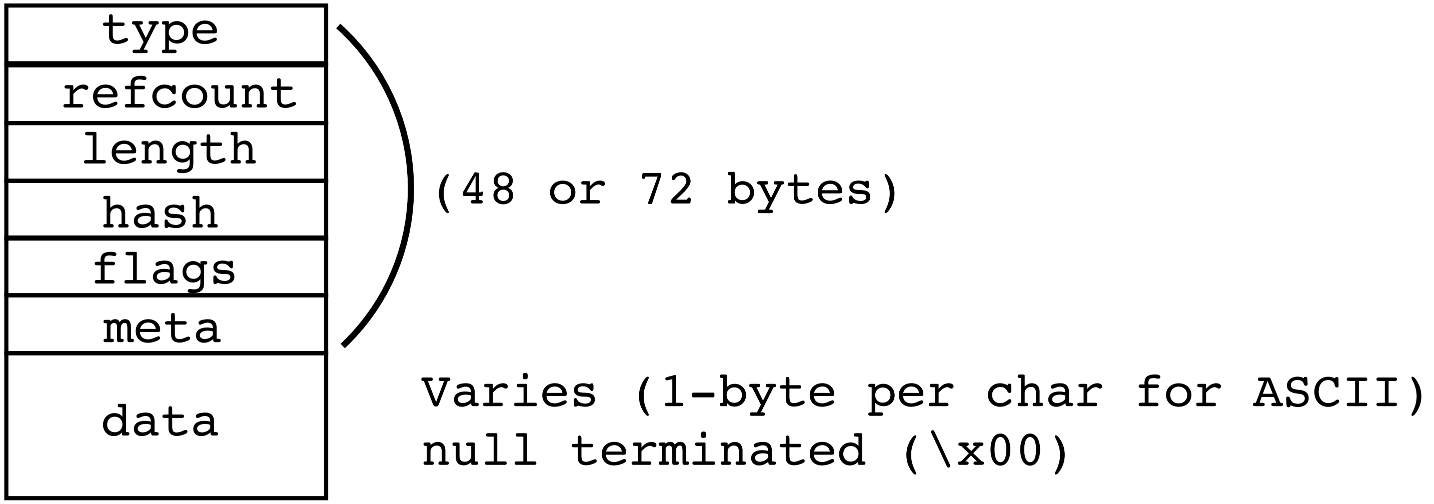
- int (arbitrary precision)

type
refcount
size
digits
...
digits

(28-??? bytes)

digits stored in
30-bit chunks

# String Representation



type
refcount
length
hash
flags
meta
data

(48 or 72 bytes)

Varies (1-byte per char for ASCII)
null terminated (\x00)

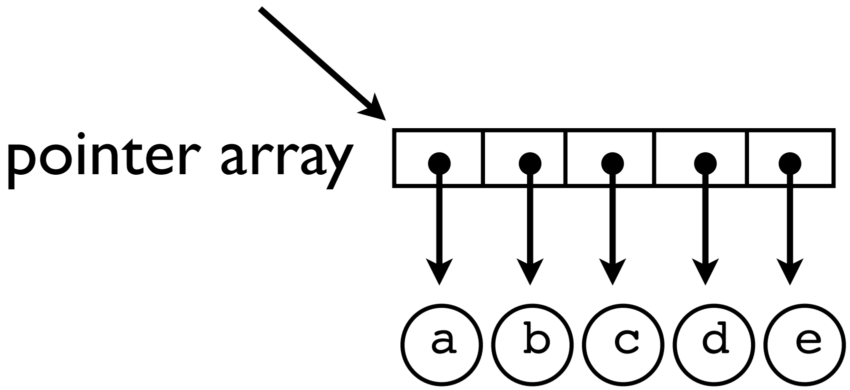
- Strings adapt to Unicode (size may vary)

```
>>> a = 'n'
>>> b = 'ñ'
>>> sys.getsizeof(a)
50
>>> sys.getsizeof(b)
74
>>>
```

# Container Representation

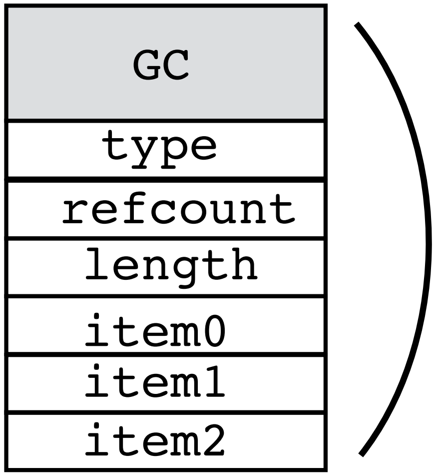
- Container objects only hold references (object ids) to their stored values

```
items = [a, b, c, d, e]         # A list
```



- All operations involving the container internals only manipulate the ids (not the objects)

# Tuple Representation



GC
type
refcount
length
item0
item1
item2

Garbage collection (internal)

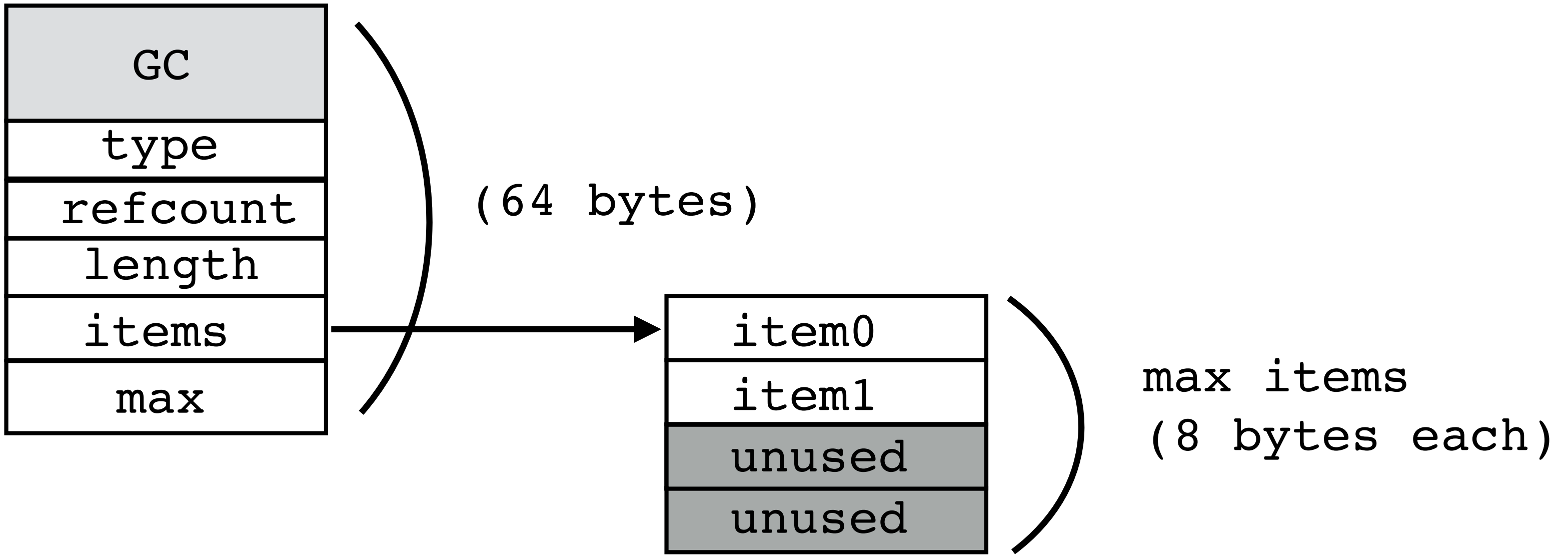
(48 bytes + 8 bytes/item)

- Examples:

```
>>> a = ()
>>> sys.getsizeof(a)
48
>>> b = (1,2,3)
>>> sys.getsizeof(b)
72
>>>
```

Note: size does not include the items themselves. It's just for the tuple part.

# List Representation



GC
type
refcount
length
items
max

(64 bytes)

item0
item1
unused
unused

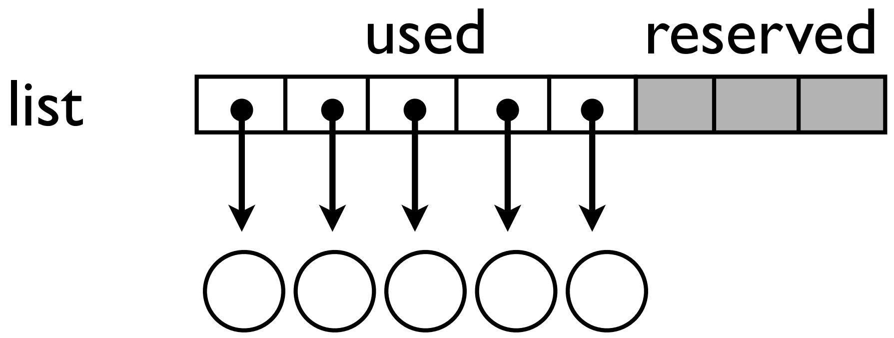
max items
(8 bytes each)

- Lists are resizable (storage space will grow)

```
>>> a = [1,2,3,4]
>>> sys.getsizeof(a)
96
>>> a.append(5)
>>> sys.getsizeof(a)
128
>>>
```

# Over-allocation

- All mutable containers (lists, dicts, sets) tend to over-allocate memory so that there are always some free slots available

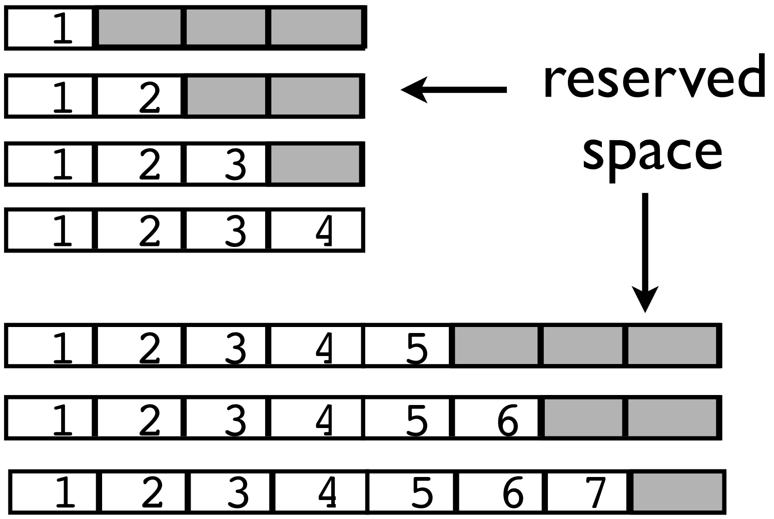


- This is a performance optimization
- Goal is to make appends, insertions fast

# Example : List Memory

- Example of list memory allocation

```
items = []
items.append(1)
items.append(2)
items.append(3)
items.append(4)

items.append(5)
items.append(6)
items.append(7)
```



- Extra space means that most append() operations are very fast (space is already available, no memory allocation required)

# Set/Dict Hashing

- Sets and dictionaries are based on hashing
- Keys are used to determine an integer "hashing value" (__hash__() method)

```
a = 'Python'
b = 'Guido'
c = 'Dave'

>>> a.__hash__()
-539294296
>>> b.__hash__()
1034194775
>>> c.__hash__()
2135385778
```

- Value used internally (implementation detail)

# Key Restrictions

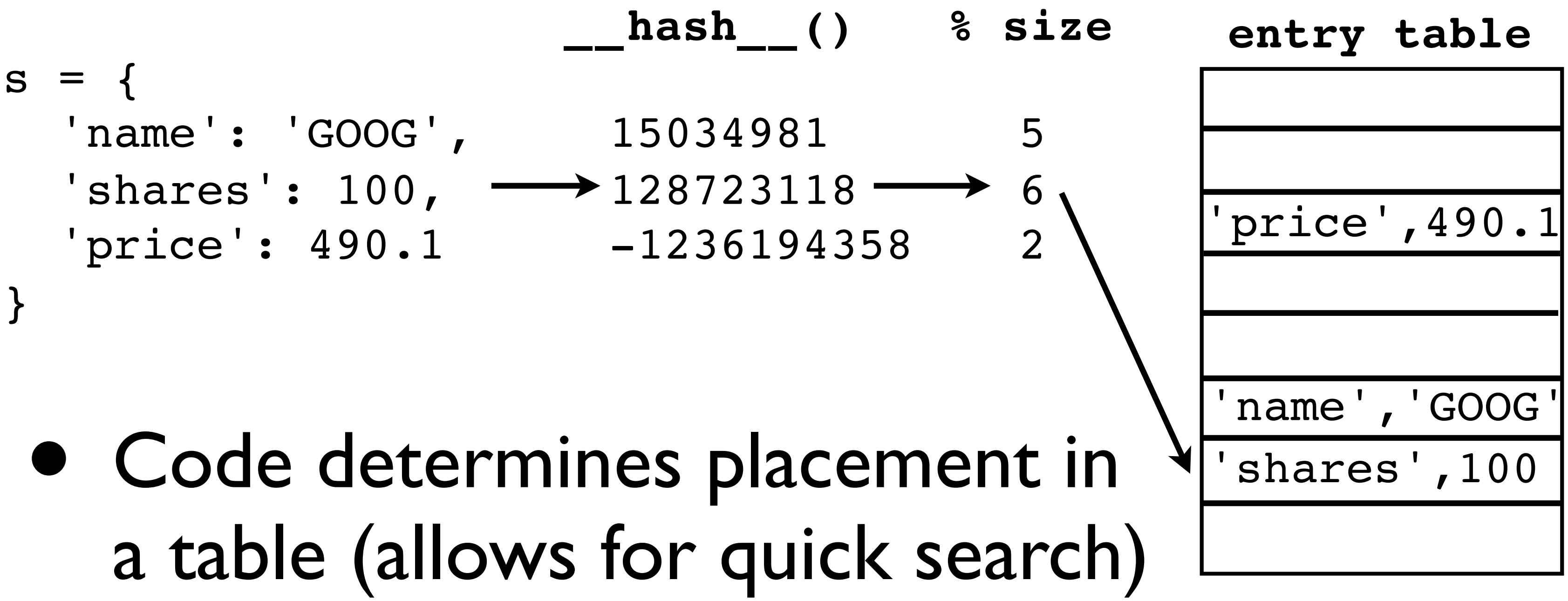
- Sets/dict keys restricted to "hashable" objects

```
>>> a = {'IBM','AA','AAPL'}
>>> b = {[1,2],[3,4]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>>
```

- This usually means you can only use strings, numbers, or tuples (no lists, dicts, sets, etc.)

# Item Placement

- Hashing in a nutshell....



```
                         __hash__()    % size
s = {
   'name': 'GOOG',        15034981        5
   'shares': 100,         128723118       6
   'price': 490.1        -1236194358      2
}
```

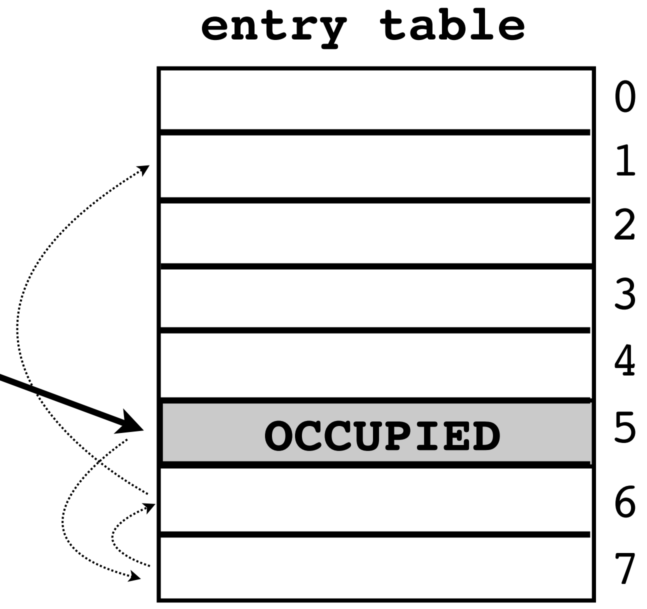
- Code determines placement in a table (allows for quick search)

- But there's an issue with collisions...

# Collision Resolution

- Hash index is perturbed until an open slot found

```
key='name'
h = key.__hash__() -> 15034981
i = h % size -> 5
```



- Recurrence

```
i, h = perturb(i, h, size)

i = 7, 6, 1, 4, 5, 2, 3, 0, ...
```

- Every slot is tried eventually
- Works better if many open slots available

# Set/Dict Representation

- You always start with space for 8 items

```
>>> a = { }
>>> sys.getsizeof(a)
240
>>> a = { 'a':1, 'b':2, 'c':3, 'd':4 }
>>> sys.getsizeof(a)
240
>>>

>>> b = set()
>>> sys.getsizeof(b)
224
>>> b = { 1, 2, 3, 4}
>>> sys.getsizeof(b)
224
>>>
```

- But there's a catch... you can't use all of it

# Set/Dict Overallocation

- Sets/dicts never fill up completely
- Increase their size if more than 2/3 full

```
>>> a = { 'a':1, 'b':2, 'c':3, 'd':4 }
>>> sys.getsizeof(a)
240
>>> a['e'] = 5
>>> sys.getsizeof(a)
240
>>> a['f'] = 6
>>> sys.getsizeof(a)
368
>>>
```

- A possible surprise if building data structures

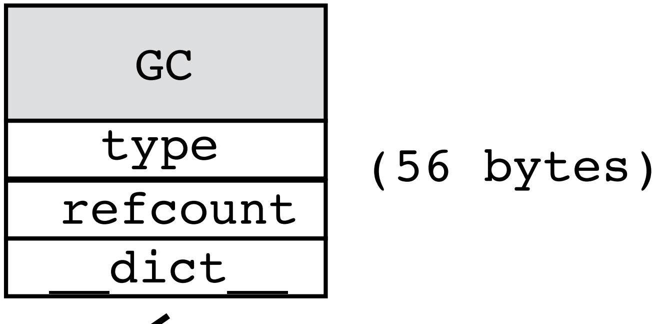
# Demo

In which Dave demonstrates the unusual fate that befalls a program that places more than 5 entries in a dictionary...

# Instance Representation

- Instances normally use dictionaries

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

>>> p = Point(2,3)
>>> p.__class__
<class '__main__.Point'>
>>> p.__dict__
{ 'x': 2, 'y': 3 }
>>>
```



- There are some optimizations

# Key Sharing Dicts

- Instances use a compact key-sharing dict

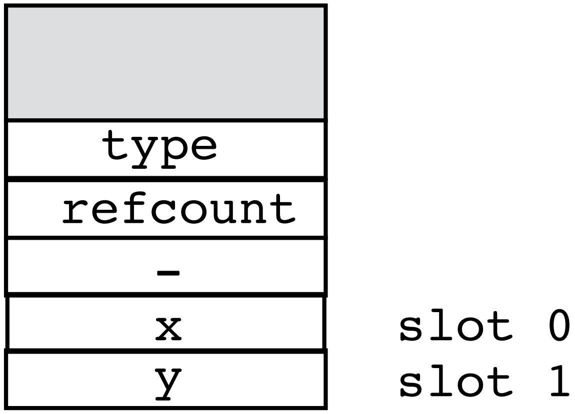
```
0 - 5  items  (112 bytes)
6 - 10 items  (152 bytes)
```

- Insight: All instances created will have exactly the same set of keys
- The keys can be shared across dicts
- So, a bit more efficient than a normal dict

# Instances w/slots

- Slots eliminate the instance dictionary

```
class Point:
    __slots__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

>>> p = Point(2,3)
>>> p.__class__
<class '__main__.Point'>
>>> p.__dict__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Point' object has no attribute '__dict__'
>>>
```



# Demo

- Tuples vs. slots
- Dicts vs. classes

# Part 4: Thinking in Functions

# Algebra Refresher

- Functions (from math)

$f(x) = 3x + 2$

- Essential Features
  - To evaluate, substitute the "x"
  - For each input, there is one output
  - Output is always the same for the same input
- Often a powerful way to think about coding

# Functions

- Functions are building blocks
- Example: Compute $\sum_{n=1}^{100} \frac{1}{n^2}$

```
def square(x):
    return x * x

def recip(x):
    return 1/x

def sum_invsquare(start, stop):
    total = 0
    for n in range(start, stop+1):
        total += recip(square(n))
    return total

result = sum_invsquare(1, 100)
```

# Higher-Order Functions

- Functions can accept other functions as input

```
def sum_terms(start, stop, term):
    total = 0
    for n in range(start, stop+1):
        total += term(n)
    return total

def invsquare(x):
    return 1.0/(x * x)

total = sum_terms(1, 100, invsquare)
```

- Functions are data just like numbers, strings, etc.

# Higher-Order Functions

- Functions can create new functions

```
def compose(f, g):
    def h(x):
        return f(g(x))
    return h

def recip(x):
    return 1/x

def square(x):
    return x * x

total = sum_terms(1, 100, compose(recip, square))
```

- Higher-order functions allow generalization and abstraction centered around functions

# List Processing

- Applying a function to elements of a list

```
def square(x):
    return x * x

data = [1, 2, 3, 4, 5, 6, 7]

squared_data = []
for x in data:
    squared_data.append(square(x))
```

- This is an extremely common task
- Transforming/filtering list data

# List Comprehensions

- Creates a list by mapping an operation to each element of an iterable

```
>>> a = [1, 2, 3, 4, 5]
>>> b = [2*x for x in a]
>>> b
[2, 4, 6, 8, 10]
>>>
```

- Another example:

```
>>> names = ['IBM', 'YHOO', 'CAT']
>>> a = [name.lower() for name in names]
>>> a
['ibm', 'yhoo', 'cat']
>>>
```

# List Comprehensions

- A list comprehension can also filter

```
>>> a = [1, -5, 4, 2, -2, 10]
>>> b = [2*x for x in a if x > 0]
>>> b
[2,8,4,20]
>>>
```

- Another example: lines containing a substring

```
>>> f = open('stockreport.csv', 'r')
>>> goog = [line for line in f if 'GOOG' in line]
>>>
```

# List Comprehensions

- General syntax

```
[expression for name in sequence if condition]
```

- What it means

```
result = []
for name in sequence:
    if condition:
       result.append(expression)
```

- Can be used anywhere a sequence is expected

```
>>> a = [1, 2, 3, 4]
>>> sum([x*x for x in a])
30
>>>
```

# List Comp: Examples

- List comprehensions are hugely useful
- Collecting the values of a specific field

```
stocknames = [s['name'] for s in portfolio]
```

- Performing database-like queries

```
a = [s for s in portfolio if s['price'] > 100
                          and s['shares'] > 50 ]
```

- Quick mathematics over sequences

```
cost = sum([s['shares']*s['price'] for s in portfolio])
```

# Historical Digression

- List comprehensions come from math

```
a = [x**2 for x in s if x > 0]     # Python
```

- Mathematical notation (set theory)

$$a = \{ x^2 \mid x \in s,\ x > 0 \}$$

- But most Python programmers would probably just view this as a "cool shortcut"

# Set/Dict Comprehensions

- List comprehension

```
>>> [ s['name'] for s in portfolio ]
[ 'AA', 'IBM', 'CAT', 'MSFT', 'GE', 'MSFT', 'IBM ' ]
>>>
```

- Set comprehension (eliminate duplicates)

```
>>> { s['name'] for s in portfolio }
{ 'GE', 'IBM', 'CAT', 'AA', 'MSFT' }
>>>
```

- Dict comprehension (makes a key:value mapping)

```
>>> { s['name']: 0 for s in portfolio }
{ 'GE': 0, 'IBM': 0, 'CAT': 0, 'AA': 0, 'MSFT': 0 }
>>>
```

# Reductions

- sum(s), min(s), max(s)

```
>>> s = [1, 2, 3, 4]
>>> sum(s)
10
>>> min(s)
1
>>> max(s)
4
>>>
```

- Boolean tests: any(s), all(s)

```
>>> s = [False, True, True, False]
>>> any(s)
True
>>> all(s)
False
>>>
```

# Map-Reduce

- Many problems fit into a "map-reduce" model

```
               data = [ ... ]
                      ↓
mapping = [op(x) for x in data if predicate(x) ]
                      ↓
          result = reduce(mapping)
```

- Conceptually simple
- Benefits for distributing work/performance

# Challenge

1. Rewrite the bus data code using list comprehensions and a functional programming style.

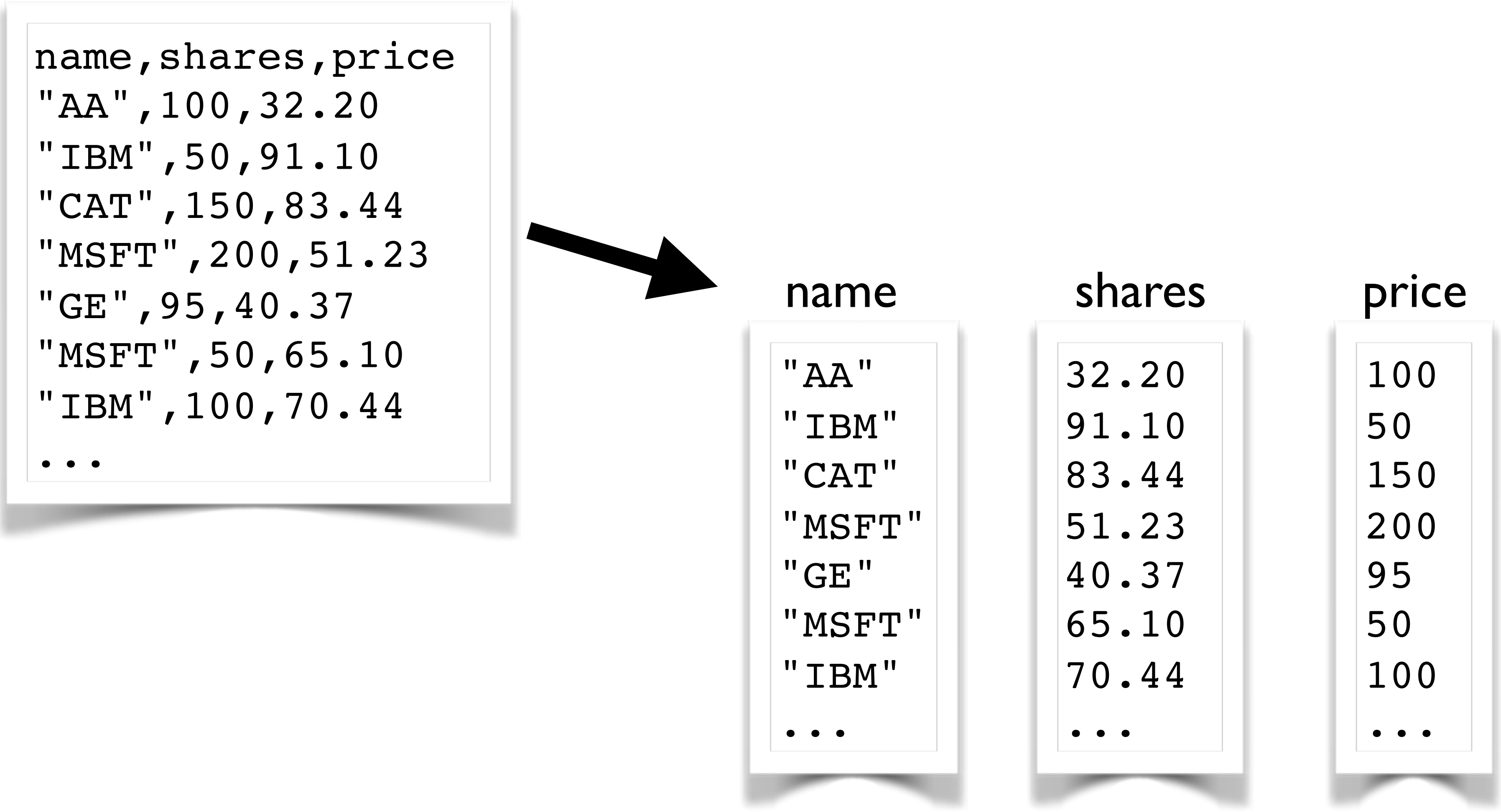
2. Find out on what day the route 22 bus had the highest ridership.

# Part 5: Thinking in Columns

# Story so Far

- Main focus has been "object oriented"
- Each row is an "object" or "record"
- Different representations (tuple, dict, etc.)
- But, that's not the only viewpoint

# Columns not Rows



```
name,shares,price
"AA",100,32.20
"IBM",50,91.10
"CAT",150,83.44
"MSFT",200,51.23
"GE",95,40.37
"MSFT",50,65.10
"IBM",100,70.44
...
```

| name | shares | price |
|---|---|---|
| "AA" | 32.20 | 100 |
| "IBM" | 91.10 | 50 |
| "CAT" | 83.44 | 150 |
| "MSFT" | 51.23 | 200 |
| "GE" | 40.37 | 95 |
| "MSFT" | 65.10 | 50 |
| "IBM" | 70.44 | 100 |
| ... | ... | ... |

Think spreadsheets...

# An Experiment

- List of tuples

```
rows = [
  ('AA', 100, 32.2),
  ('IBM', 50, 91.1),
  ('CAT', 150, 83.44),
  ('MSFT', 200, 51.23),
  ...
]
```

- A tuple of lists

```
columns = (
    ['AA', 'IBM', 'CAT', 'MSFT', ...],
    [100, 50, 150, 200, ...],
    [32.2, 91.1, 83.44, 51.23, ...]
)
```

- What are storage requirements?

# An Experiment

- List of tuples

```
rows = [
  ('AA', 100, 32.2),
  ('IBM', 50, 91.1),
  ('CAT', 150, 83.44),
  ('MSFT', 200, 51.23),
  ...
]
```

Per-record overhead:
72 bytes (tuples)

- A tuple of lists

```
columns = (
    ['AA', 'IBM', 'CAT', 'MSFT', ...],
    [100, 50, 150, 200, ...],
    [32.2, 91.1, 83.44, 51.23, ...]
)
```

Per-record overhead:
24 bytes (list items)

- What are storage requirements?

# Challenge

Read the bus data into separate lists representing columns. Does it make a difference? Can you still work with the data?

# Arrays

- numpy library provides support for arrays
- A collection of uniformly typed objects

```
>>> import numpy
>>> a = numpy.array([1,2,3,4], dtype=numpy.int64)
>>> a
array([1, 2, 3, 4])
>>>
```

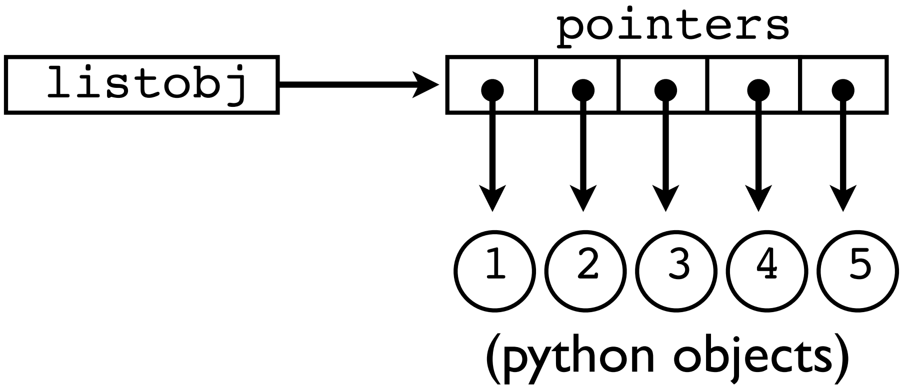
- Differs from a list (heterogenous items)

```
>>> b = [1,2,3,4]
>>> b[2] = 'hello'
>>> b
[1, 2, 'hello', 4]
>>> a[2] = 'hello'          # ValueError exception
```

# Arrays vs. Lists

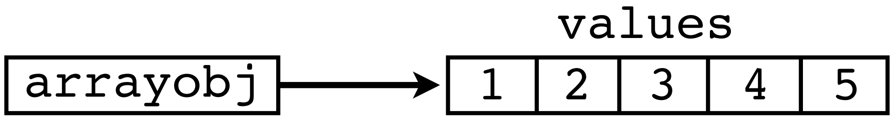
- List

```
a = [1, 2, 3, 4, 5]
```



- Array

```
a = numpy.array([1,2,3,4,5])
```



- Storage is same as arrays in C/C++/Fortran

# Digression

- numpy is a large library (100s of functions)
- This is not meant to be a numpy tutorial
- But, let's discuss the "big picture"

# Vectorized Operations

- arrays prefer operations on the entire array

```
>>> a
array([1, 2, 3, 4])
>>> a + 10
array([11, 12, 13, 14])
>>> numpy.sqrt(a)
array([ 1., 1.41421356,  1.73205081,  2.])
>>>
```

- Operations are implemented in C (very fast)

# Vectorized Conditionals

- Relations produce boolean arrays

```
>>> a
array([1, 2, 3, 4])
>>> a < 3
array([ True,  True, False, False], dtype=bool)
```

- Boolean arrays can filter

```
>>> a[a<3]
array([1, 2])
>>>
```

- Variant: where(cond, x, y)

```
>>> numpy.where(a < 3, -1, 1)
array([-1, -1, 1, 1])
>>>
```

# Array Slicing

- Array slices produce overlays

```
>>> a
array([1, 2, 3, 4])
>>> b = a[0:2]
>>> b
array([1, 2])
>>>
```

- Try changing the data

```
>>> b[0] = 10
>>> b
array([10,  2])
>>> a
array([10,  2,  3,  4])
>>>
```

- This is very different than Python lists (copies)

# Pandas Dataframes

- Dataframe is a collection of named arrays

```
>>> data = pandas.read_csv('portfolio.csv')
>>> data
   name  shares  price
0    AA     100  32.20
1   IBM      50  91.10
2   CAT     150  83.44
3  MSFT     200  51.23
```

- Think columns

```
>>> data['shares']
0     100
1      50
2     150
3     200
Name: shares, dtype: int64
>>>
```

# Pandas Examples

- Creating a new column

```
>>> data['cost'] = data['shares']*data['price']
>>> data
  name  shares  price      cost
0   AA     100  32.20   3220.00
1  IBM      50  91.10   4555.00
2  CAT     150  83.44  12516.00
3 MSFT     200  51.23  10246.00
>>>
```

- Filtering

```
>>> data[data['shares'] < 100]
   name  shares  price     cost
1  IBM       50  91.10  4555.00
>>>
```

# Commentary

- Most "standard" Python code is focused on manipulating objects and records
- Most "scientific" Python code is array focused
- There is a conceptual barrier
- Ideally, you want to understand both worlds

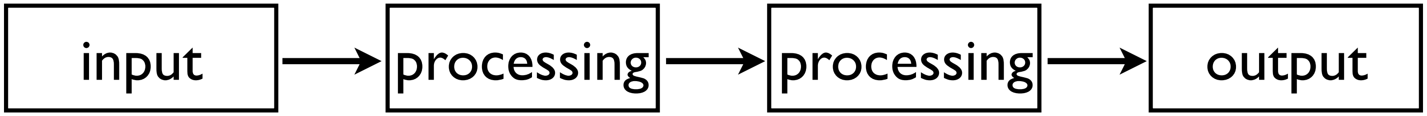
# Challenge

Read the bus data using Pandas. Compare with earlier approaches.

# Part 6: Thinking in Streams

# Stream Processing

- Many problems in data analysis can be broken down into workflows



- Processing stages might transform/filter the data in some way

# Iteration

- Iteration defined: Looping over items

```
a = [2,4,10,37,62]
# Iterate over a
for x in a:
    ...
```

- Most programs do a huge amount of iteration
- One way to view iteration is as a "stream" of elements--the for loop consumes it

# Iteration

- Many parts of Python produce streams

```
zip(a, b)
map(func, s)
filter(func, s)
enumerate(s)
```

- Example:

```
>>> a = [1,2,3]
>>> b = ['a','b','c']
>>> c = zip(a,b)
>>> c
<zip object at 0x108e5f4c8>
>>> for x, y in c:
...     print(f'{x} -> {y}')
...
1 -> a
2 -> b
3 -> c
```

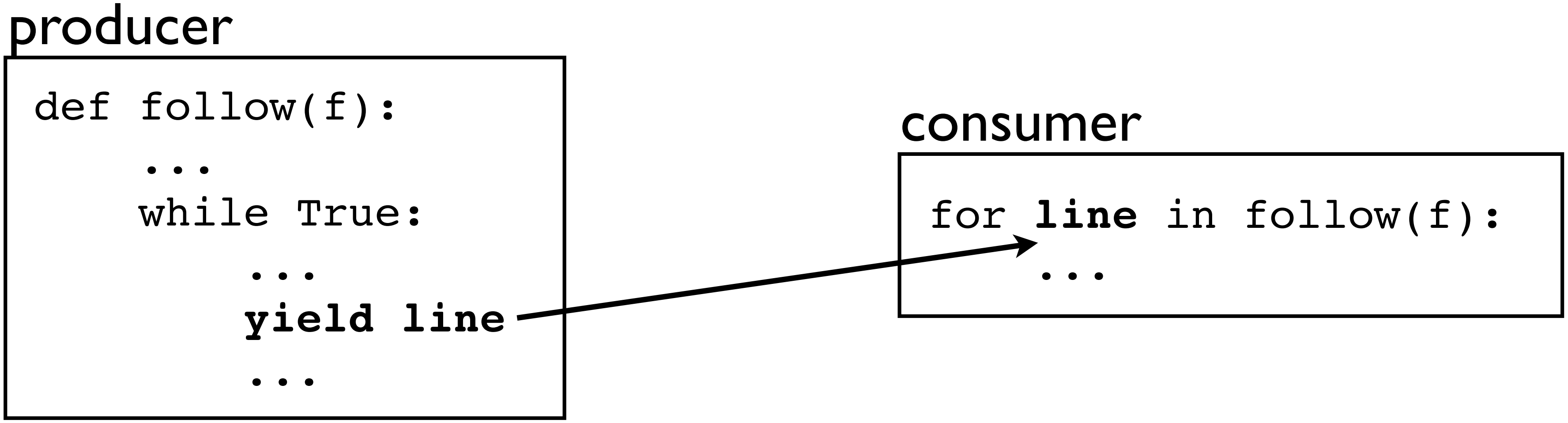
# Generator Functions

- Generators implement customized iteration

```
def countdown(n):
    print('Counting down from', n)
    while n > 0:
        yield n
        n -= 1

>>> for i in countdown(5):
...     print('T-minus', i)
...
Counting down from 5
T-minus 5
T-minus 4
T-minus 3
T-minus 2
T-minus 1
>>>
```

# Producers & Consumers

- Generators are closely related to various forms of "producer-consumer" programming



producer

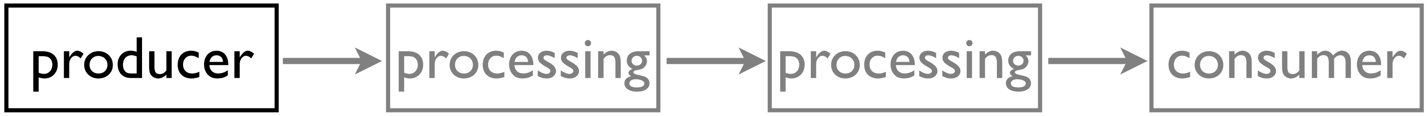
```
def follow(f):
    ...
    while True:
        ...
        yield line
        ...
```

consumer

```
for line in follow(f):
    ...
```

- yield produces values
- for consume values

# Generator Pipelines



```
def producer():
    ...
    yield item
    ...
```

- Producer is typically a generator (although it could also be a list or some other sequence)
- yield feeds data into the pipeline

# Generator Pipelines

producer → processing → processing → **consumer**

```
def producer():
    ...
    yield item
    ...
```

```
def consumer(s):
    for item in s:
        ...
```

- Consumer is just a simple for-loop
- It gets items and does something with them

# Generator Pipelines

producer → processing → processing → consumer

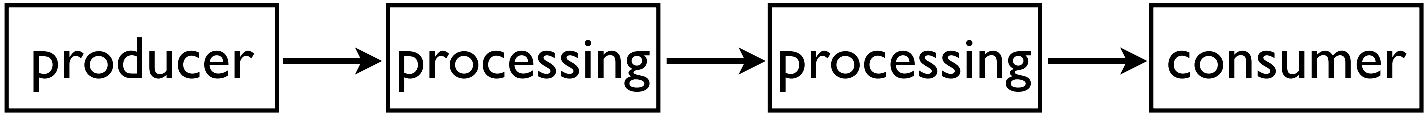
```
def producer():
    ...
    yield item
    ...
```

```
def processing(s):
    for item in s:
        ...
        yield newitem
        ...
```

```
def consumer(s):
    for item in s:
        ...
```

- Intermediate processing stages simultaneously consume and produce items
- They might modify the data stream
- They can also filter (discarding items)

# Generator Pipelines



```
def producer():
    ...
    yield item
    ...
```

```
def processing(s):
    for item in s:
        ...
        yield newitem
        ...
```

```
def consumer(s):
    for item in s:
        ...
```

- Pipeline setup (in your program)

```
a = producer()

b = processing(a)

c = consumer(b)
```

- You will notice that data incrementally flows through the different functions

# Example

- Example: Compute $\sum_{n=1}^{100} \frac{1}{n^2}$

```
def square(nums):
    for x in nums:
        yield x*x

def recip(nums):
    for x in nums:
        yield 1/x

terms = range(1, 101)
result = sum(recip(square(terms)))
```

# Generator Expressions

- A variant of a list comprehension that produces the results incrementally
- Just slightly different syntax (parentheses)

```
nums = [1,2,3,4]
squares = (x*x for x in nums)
```

- To get the results, you use a for-loop

```
for n in squares:
    ...
```

# Example

- Example: Compute $\sum_{n=1}^{100} \frac{1}{n^2}$

```
terms = range(1, 101)
squares = (x*x for x in terms)
recip = (1/x for x in squares)
result = sum(recip)
```

- Thinking in streams often leads to very succinct code (step-by-step)
- Can offer a significant memory savings

# Challenge

Rewrite bus data handling code to use generators and streams. Compare efficiency to earlier approach.

# The End

- Thanks for participating!
- Next step: Looking for commonality with tools and libraries (you will start to see common programming patterns emerge everywhere)
- Twitter: @dabeaz