Python Data Handling: A Deeper Dive

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

Pearson

Python Fluency

- Mastery of data handling idioms are a fundamental part of Python literacy
- This course is about understanding the Python data model at a deeper level

Materials and Setup

- Supporting code and data for this course:

  http://www.dabeaz.com/datadeepdive

- Python 3.7+ is assumed
- Any operating system is fine
- Slides are merely a guide. Presentation will also rely heavily on live-demos, examples.

Overview

- Course is divided into three acts
    - Act 1: Foundations/Basics
    - Act 2: Data Representation
    - Act 3: Data Abstraction
- Let's begin...

Act 1: Foundations

The Primitives

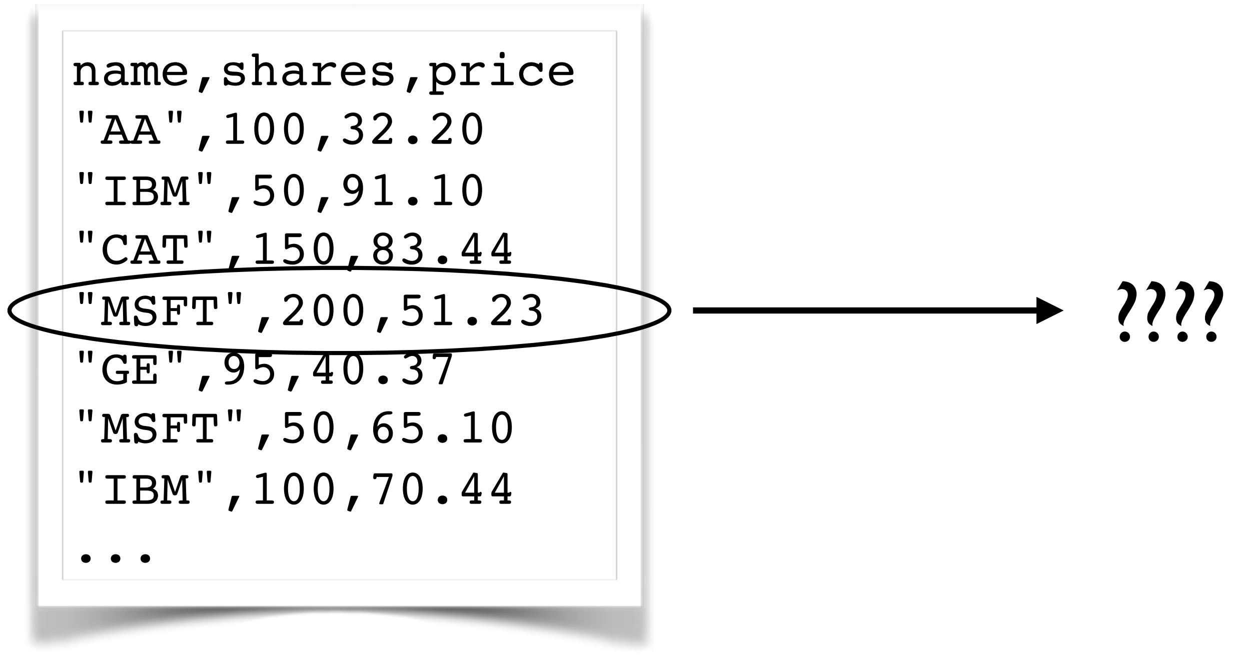
- There's a core set of primitive data types

```
None          # None
True          # Boolean
42            # Integer
4.2           # Float
4+2j          # Complex
"fortytwo"    # String
b'\x42'       # Bytes
```

- The building blocks for everything else

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 represent records/ structures in Python?

Tuples

- A collection of values packed together

```
s = ('AA', 100, 32.2)
```

- 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'   : 'AA',
   'shares' : 100,
   'price'  : 32.2
  }
```

- 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:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

- This gives you the nice "dot" syntax...

```
>>> s = Stock('AA', 100, 32.2)
>>> s.name
'AA'
>>> s.shares * s.price
3220.0
>>>
```

Variation: Dataclasses

```
from dataclasses import dataclass

@dataclass
class Stock:
    name : str
    shares : int
    price : float
```

- Same as a normal class, but adds some nice features like better printing, equality, etc.
- Note: Does not enforce the types (hints)

Variation: Named Tuple

```
from typing import NamedTuple

class Stock(NamedTuple):
    name : str
    shares : int
    price : float
```

- Same as a tuple, but adds "dot" access

```
>>> s = Stock('AA', 100, 32.2)
>>> s.shares * s.price
3220.0
>>> s[1] * s[2]
3220.0
>>>
```

- Note: Does not enforce the types (hints)

Exercise

The file "portfolio.csv" is a CSV file containing some information about stocks purchased

```
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
...
```

Read the data so that you can work with it.

Collecting Things

- Programs often have to work many objects
- And build relationships between objects
- There are some basic building blocks
    - Lists, tuples, sets, dicts
- Use depends on the 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
   ...
```

Element Processing

- Applying a function to elements of a container

```
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 container 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]
>>>
```

Set/Dict Comprehensions

- 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 }
>>>
```

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

Iterables vs. Lists

- The concept of "iteration" is more general than iterating over elements of a list/container
- Items might be produced lazily or "on demand"
- Example: reading lines from a file

```
with open('portfolio.csv') as file:
    for line in file:
        ...
```

- It doesn't read the whole file at once. It's processed piecemeal as iteration proceeds.

Iteration

- Many parts of Python produce iterables

```
zip(a, b)          # (a[0],b[0]), (a[1], b[1]), ...
map(func, s)       # func(s[0]), func(s[1]), ...
enumerate(s)       # (0, s[0]), (1, s[1]), ...
```

- 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 produce a stream of data

```
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
>>>
```

Generator Pipelines

$$\begin{tikzcd}
\boxed{\text{producer}} \arrow[r] & \boxed{\text{processing}} \arrow[r] & \boxed{\text{processing}} \arrow[r] & \boxed{\text{consumer}}
\end{tikzcd}$$

```
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

Generator Expressions

- A variant of a list comprehension that produces the results incrementally
- 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

Part 2: Data Representation

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 memory efficient way to read it into a Python list so that you can work with it?

About Efficiency

- Python is not the most efficient language
- Reading a modest file results in large overhead
- You might start to look for "hacks"

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

Better Than Hacks

- Understand how it works at a deeper level
- Know how to take advantage of that knowledge

Everything is an Object

- Everything you use in Python is an "object"

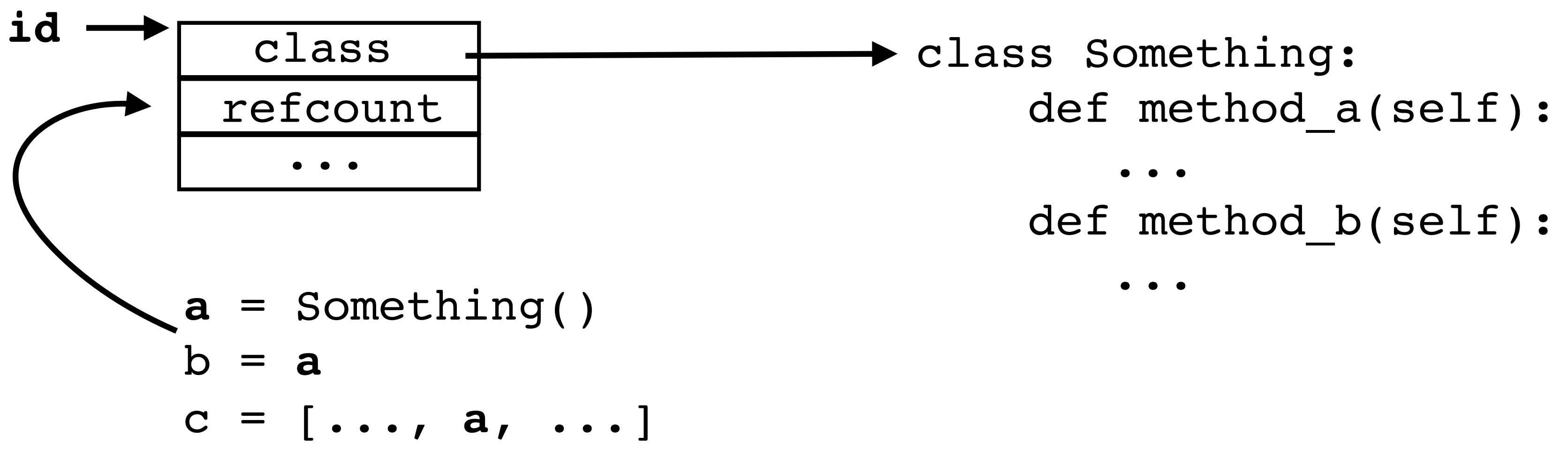
```
a = None
b = 42
c = 4.2
d = "forty two"
e = [1,2,3]
f = ('ACME', 50, 91.5)

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

- Programs are based on manipulating objects

Under the Hood

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



```
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 Hood

- 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
- But, it is extra overhead attached to each object

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
```

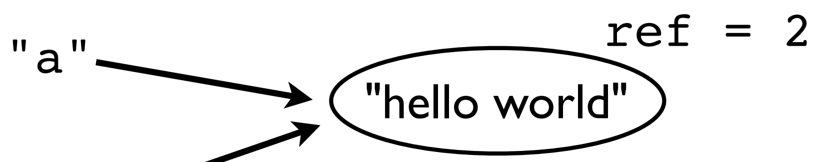
- A caution : assignment operations never make a copy of the value being assigned

- 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
>>>
```



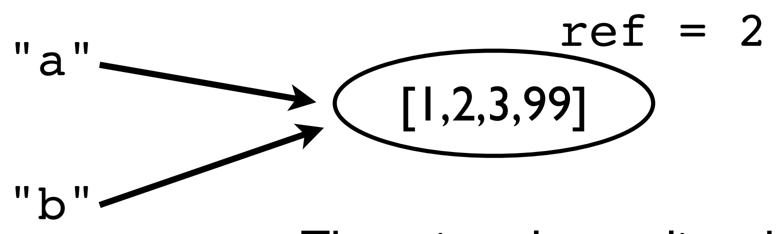
There is only one string. Two different names refer to it.

- 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]
>>>
```

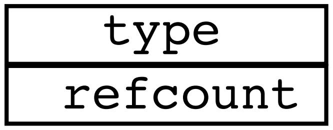


There is only one list object, but there are two references to it

- Both values change!
- Use copy.deepcopy() to make a safe copy

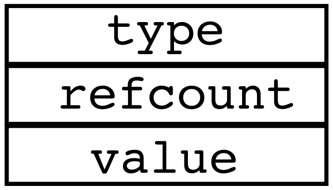
Builtin Representation

- None (a singleton)



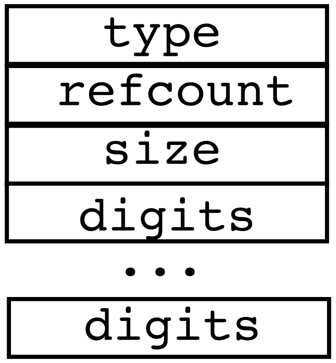
(16 bytes)

- float (64-bit double precision)



(24 bytes)

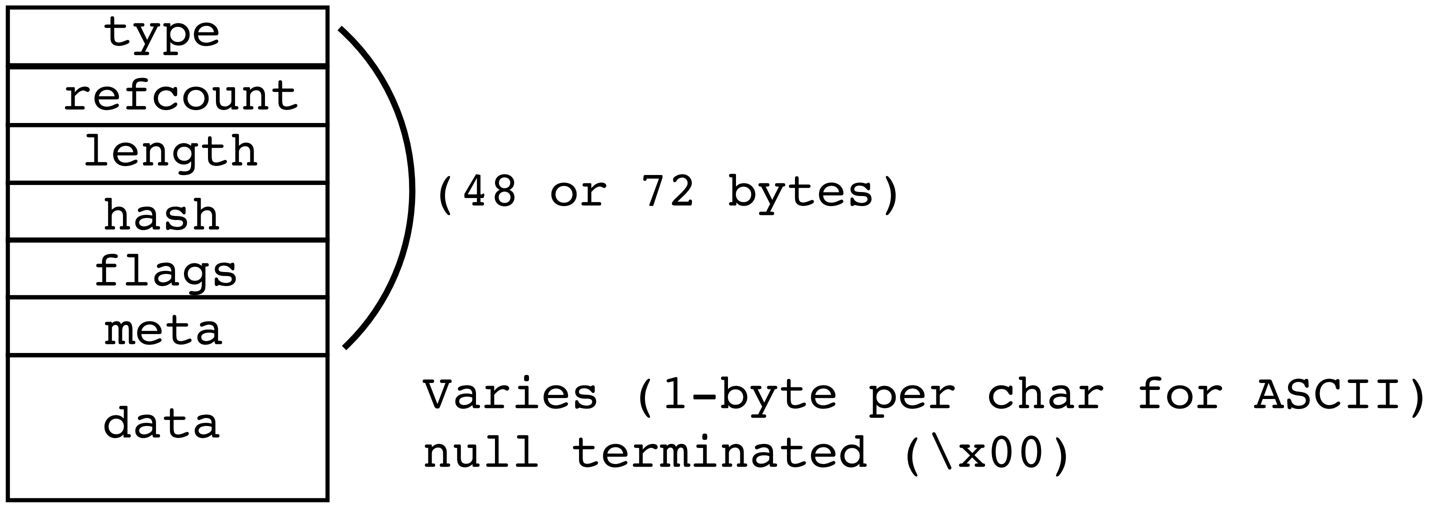
- int (arbitrary precision)



(28-??? bytes)

digits stored in
30-bit chunks

String Representation



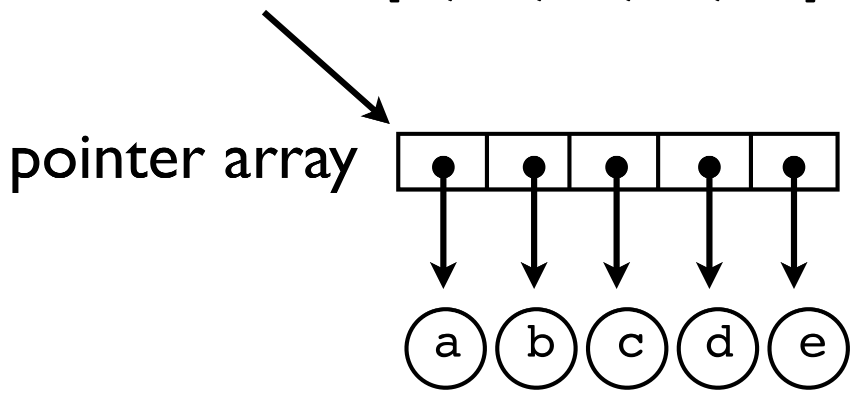
- Strings adapt to Unicode (size may vary)

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

Container Representation

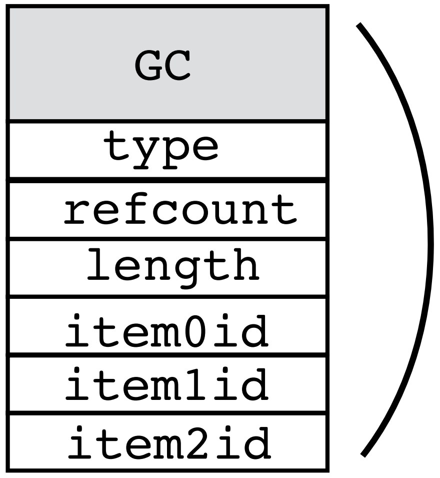
- Container objects only hold references (object ids) to their stored values

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



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

Tuple Representation



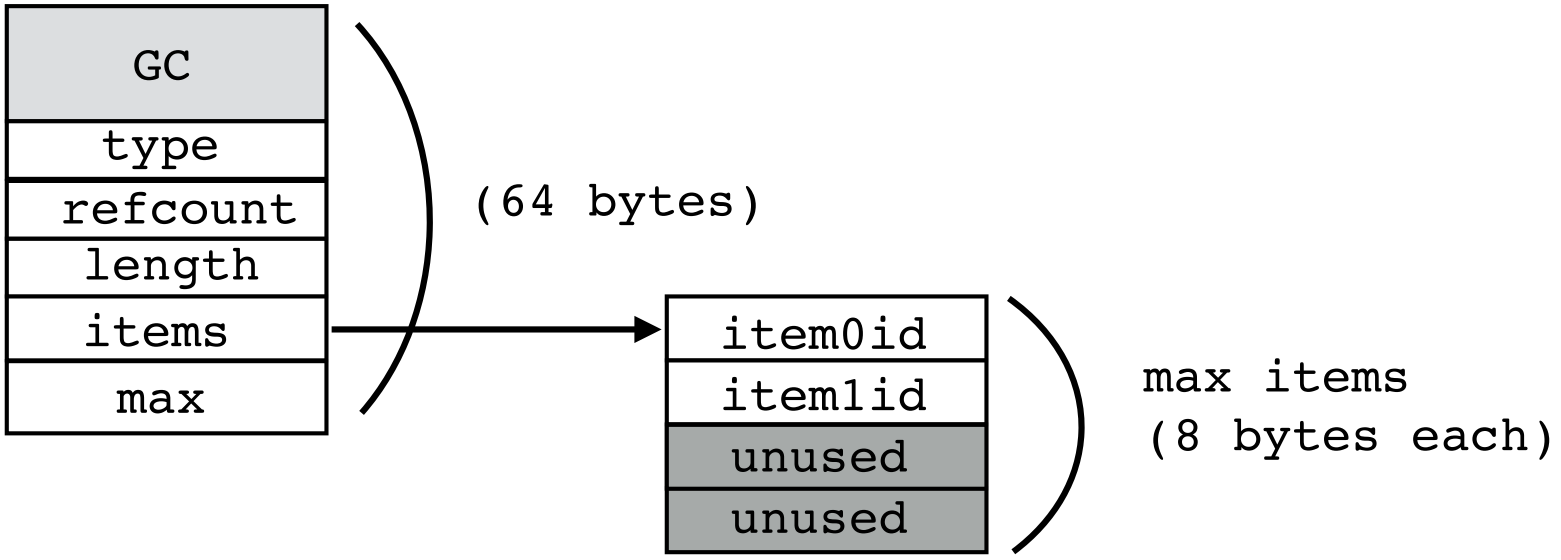
- Examples:

```
>>> 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.

Also: tuples only contain the ids (pointers) of the items.

List Representation

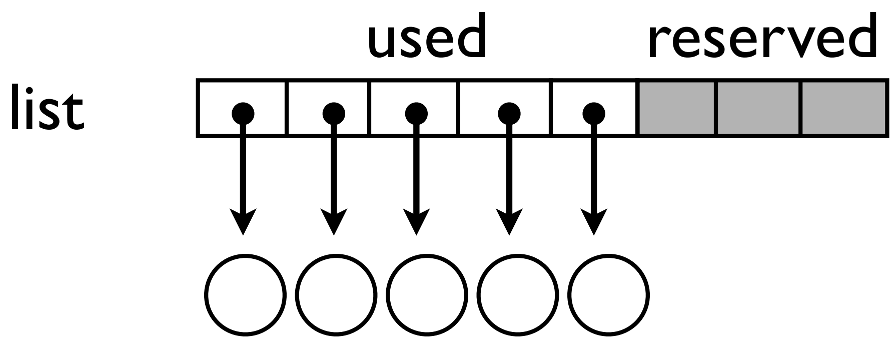


- Lists are resizable (storage space will vary)

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

Over-allocation

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



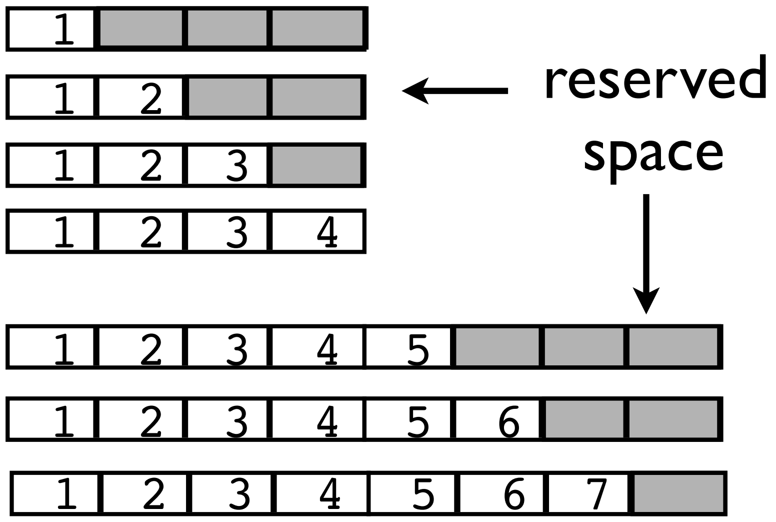
- This is a performance optimization
- Goal is to make appends, insertions fast

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

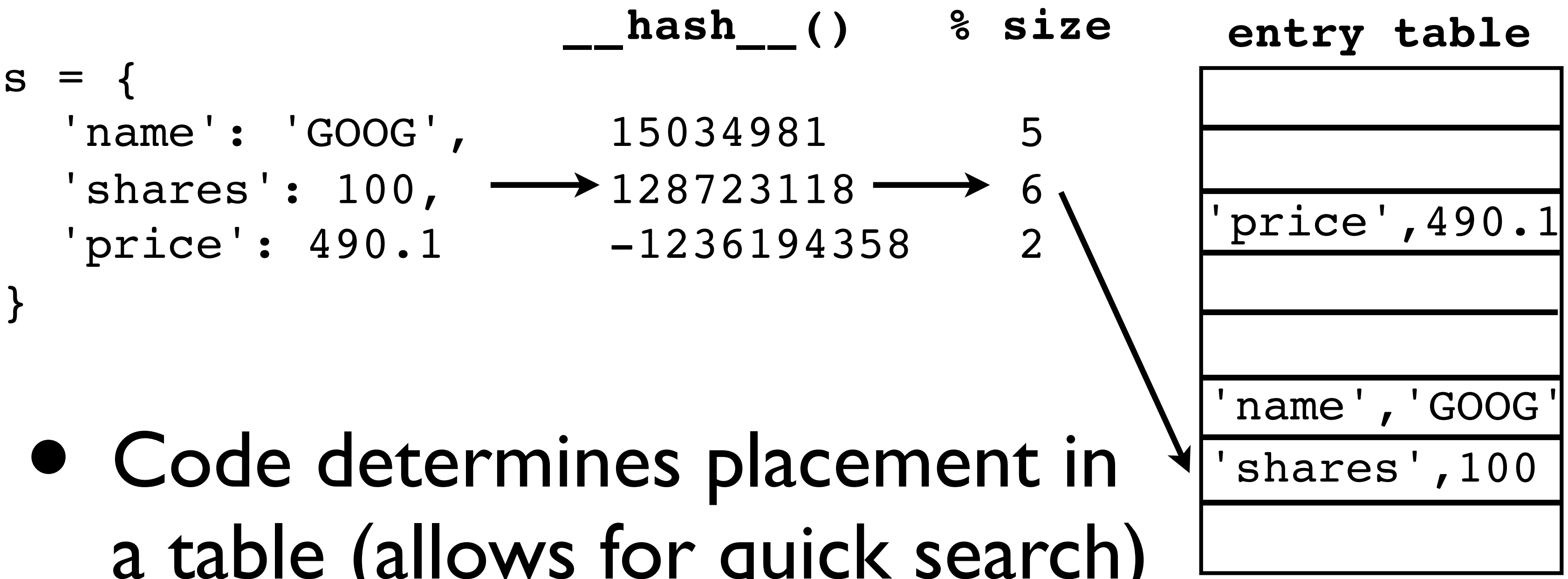
- Sets/dict keys restricted to "hashable" objects

```
>>> 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
}
```

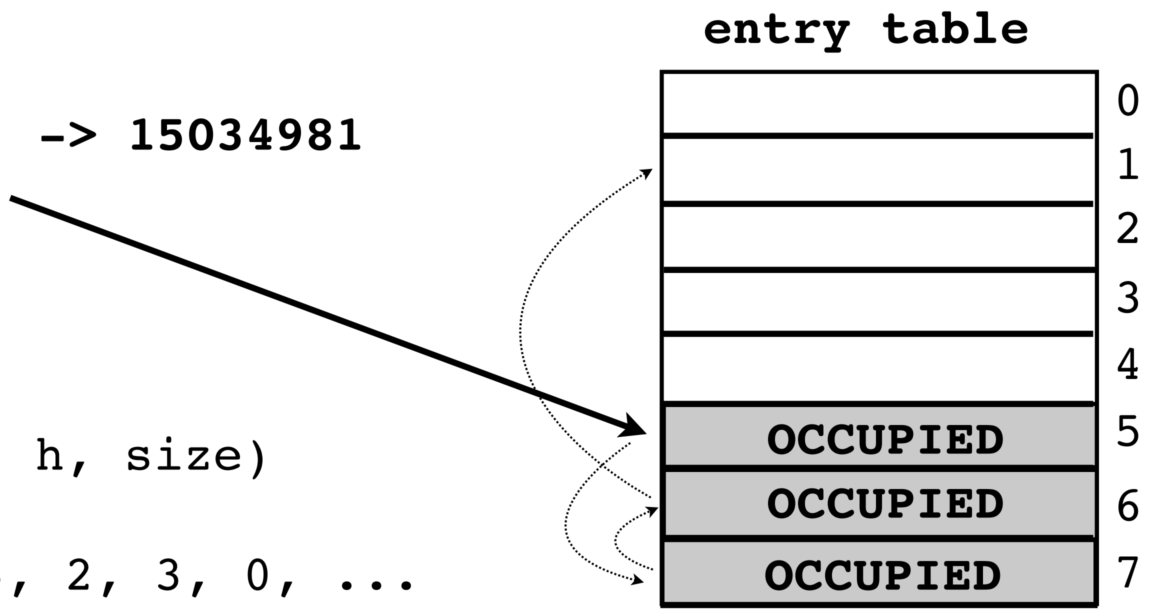
- Code determines placement in a table (allows for quick search)

- 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 Overallocation

- Sets/dicts never fill up completely
- Increase their internal 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
>>>
```

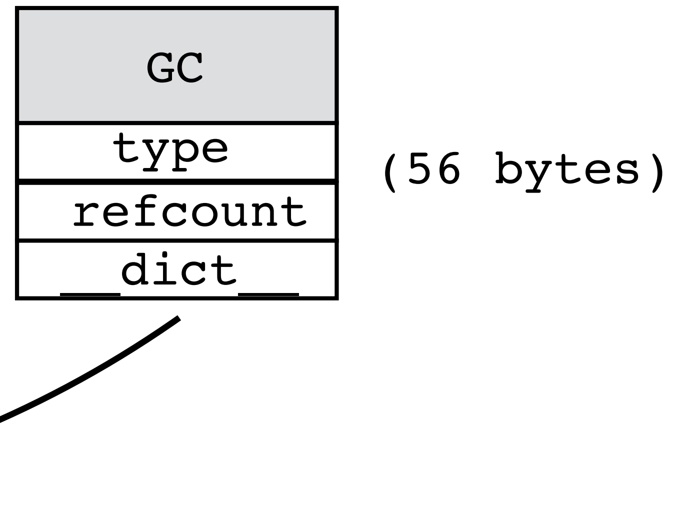
- Empty space required to avoid hash collisions

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 }
>>>
```



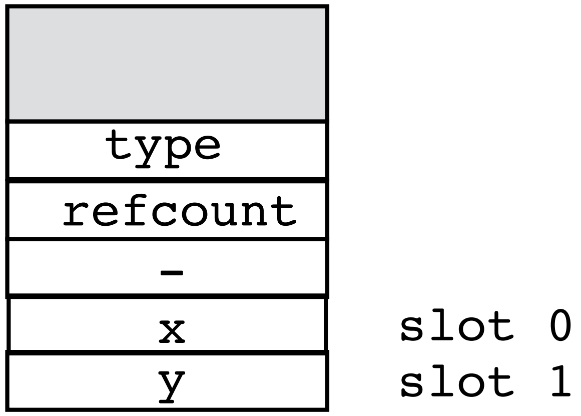
- Note: It's a slightly modified dict that tries to share the key-set across all instances.

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__'
>>>
```



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, ...]
)
```

- Which is more memory efficient?

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)

- Each list item is a simple pointer (64-bits)

Arrays

- Array libraries can take the efficiency further
- A collection of uniformly typed objects

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

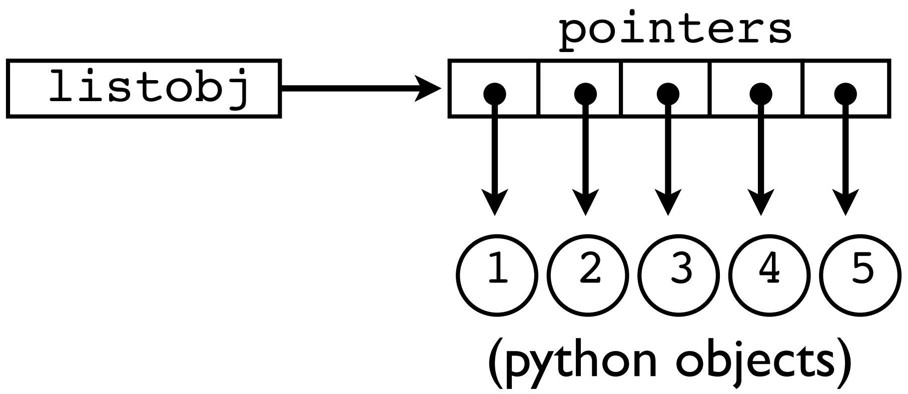
- Differs from a list (heterogenous items)

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

Arrays vs. Lists

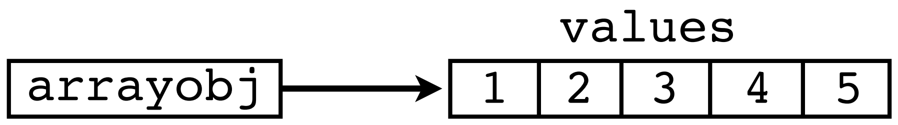
- List

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



- Array

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



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

Challenge

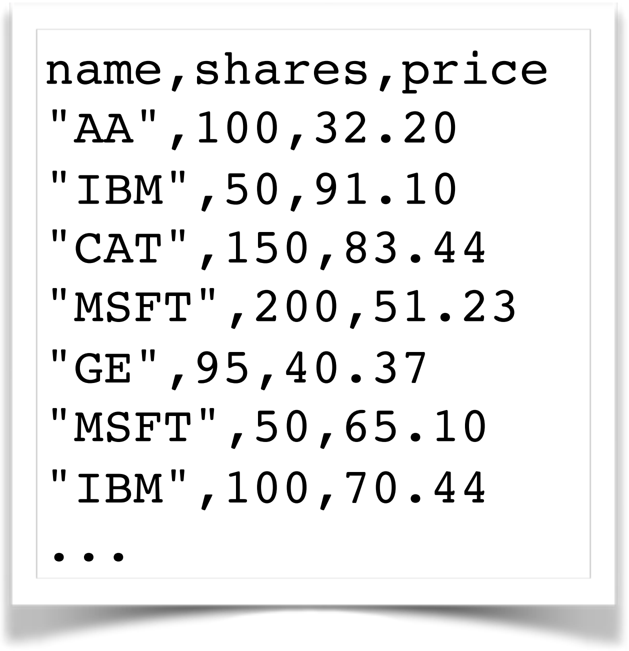
Read the bus data into a Python data structure (of any kind) that uses as little memory as possible. Exploit any available "hack" that you can think to use.

Part 3: Data Abstraction

Question

- Is low-level representation something you want to worry about when writing applications?
- As a general rule: No!
- Instead: You're thinking about higher level abstractions that you use to work with the data

Data: Your Way



```
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 want to work with it????

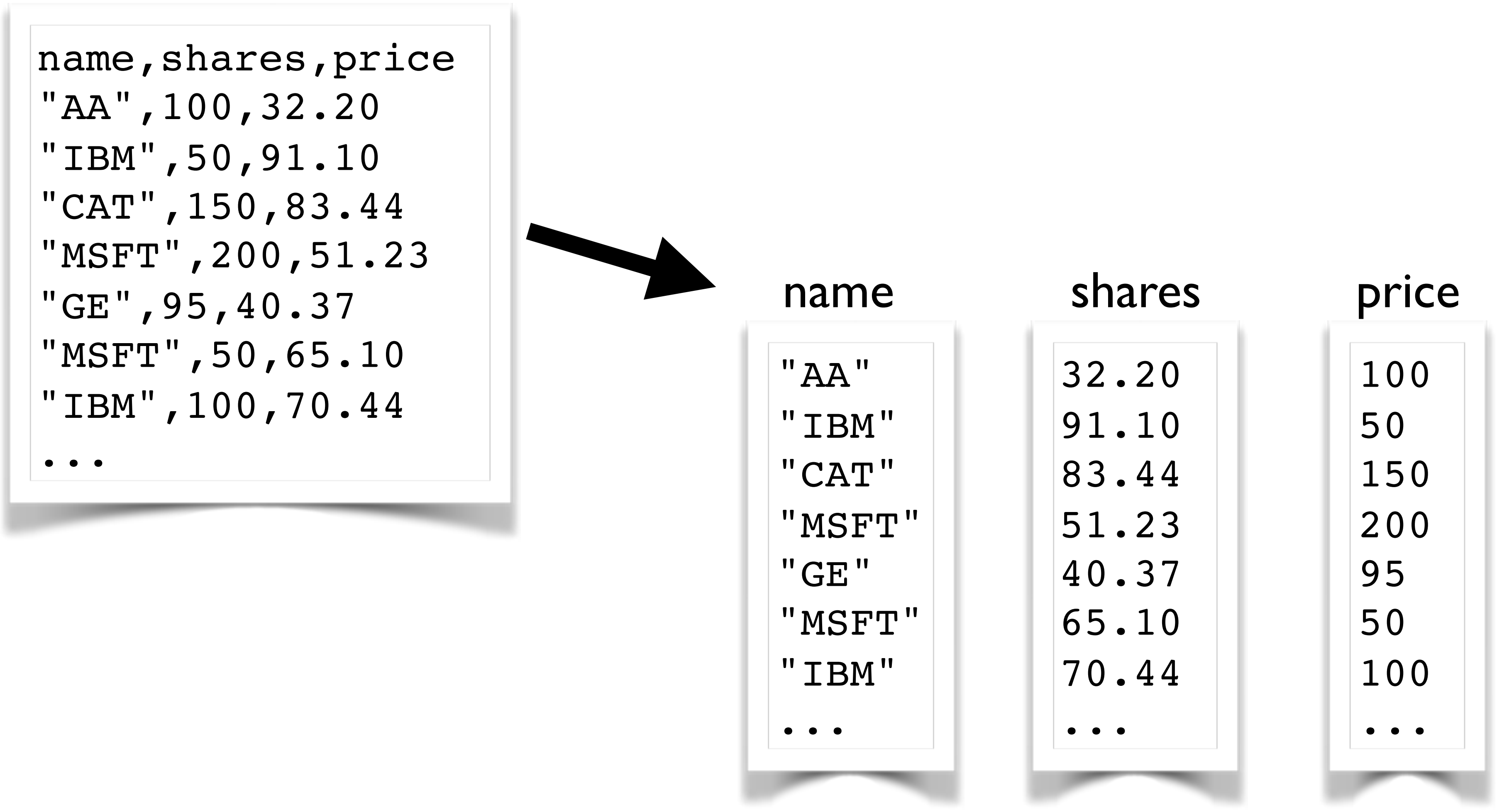
Rows and Objects

```
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
...
```

```
[
  Stock("AA", 100, 32.2),
  Stock("IBM", 50, 91.1),
  Stock("CAT", 150, 83.44),
  Stock("MSFT", 200, 51.23),
  Stock("GE", 95, 40.37),
  Stock("MSFT", 50, 65.10),
  Stock("IBM", 100, 70.44)
]
```

Think databases,
CSV files, etc.

Columns and Arrays



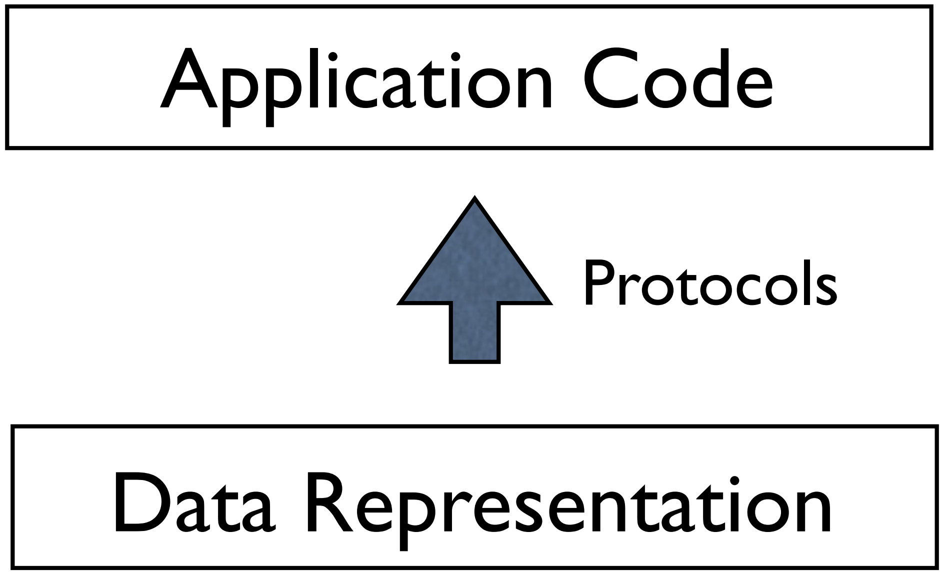
```
name,shares,price
"AA",100,32.20
"IBM",50,91.10
"CAT",150,83.44
"MSFT",200,51.23
"GE",95,40.37
"MSFT",50,65.10
"IBM",100,70.44
...
```

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... (or pandas)

Abstraction Layers

- Python is based on "protocols"



- Protocols allow data to be presented in almost any way that you want

Protocols

- Protocols are the "magic methods"

```
>>> a = 42
>>> a + 10
52
>>> a.__add__(10)
52
>>>

>>> b = ["x","y","z"]
>>> b[1]
"y"
>>> b.__getitem__(1)
"y"
>>>
```

- Almost everything can be customized

Numeric Protocol

- Numeric Methods

```
a + b           # a.__add__(b)
a - b           # a.__sub__(b)
a * b           # a.__mul__(b)
a / b           # a.__truediv__(b)
a // b          # a.__floordiv__(b)
a < b           # a.__lt__(b)
a <= b          # a.__le__(b)
a > b           # a.__gt__(b)
a >= b          # a.__ge__(b)
a == b          # a.__eq__(b)
...
```

- Useful if you want to make a new "primitive type" (e.g., decimals, fractions, dates, etc.)

Sequence Protocol

- Sequence methods

```
len(a)                # a.__len__()
a[index]              # a.__getitem__(index)
a[index] = value # a.__setitem__(index, val)
del a[index]          # a.__delitem__(index)
item in a             # a.__contains__(item)
for x in a:           # a.__iter__()
    ...
```

- Slice objects

```
slice(start, stop, step)  # [start:stop:step]

a[x:y:z]  # -> a.__getitem__(slice(x, y, z))
```

Mapping Protocol

- Mappings are dict-like objects

```
len(a)               # a.__len__()
a[key]               # a.__getitem__(key)
a[key] = value       # a.__setitem__(key, val)
del a[key]           # a.__delitem__(key)
key in a             # a.__contains__(key)
for key in a:        # a.__iter__()
    ...
```

- Same core methods as for sequences, but key is not assumed to be numeric

Object Protocol

- Access to attributes to objects

```
a.name                 # a.__getattribute__(name)
a.name (not found)     # a.__getattr(name)
a.name = value         # a.__setattr(name, value)
del a.name             # a.__delattr(name)
```

- Related: built-in functions

```
getattr(a, "name")          # a.name
setattr(a, "name", value)   # a.name = value
delattr(a, "name")          # del a.name
```

Challenge

Create a Python class that presents your memory efficient data storage to the user as if it were a mutable list of objects. This is not data conversion. The original data should be kept intact to save memory. You're using various protocols to make that data look like something else.

```
>>> representation = read_data(...)
>>> data = AsObjects(representation)
>>> row = data[0]
>>> row
Record(name='AA', shares=100, price=32.2)
>>> row.shares
100
>>> row.shares = 40
>>>
```

Final Words

- Knowing how to utilize Python built-in datatypes is a useful skill
- Knowing how Python objects work can help you write more efficient code
- Knowing how to build your own data abstractions can hide details and keep your code manageable.

The End

- Thanks for participating!
- Twitter: @dabeaz