Sets and Dictionaries

Storage
Let's try an experiment
Let's try an experiment

```python
>>> things = set()
>>> things.add('a string')
>>> print things
set(['a string'])
```
Let's try an experiment

```python
>>> things = set()
>>> things.add('a string')
>>> print things
set(['a string'])

>>> things.add([1, 2, 3])
TypeError: unhashable type: 'list'
```
Let's try an experiment

```python
>>> things = set()
>>> things.add('a string')
>>> print things
set(['a string'])

>>> things.add([1, 2, 3])
TypeError: unhashable type: 'list'
```

What's wrong?
Let's try an experiment

```python
>>> things = set()
>>> things.add('a string')
>>> print things
set(['a string'])

>>> things.add([1, 2, 3])
TypeError: unhashable type: 'list'
```

What's wrong?

And what does the error message mean?
How are sets stored in a computer's memory?
How are sets stored in a computer's memory?

Could use a list
How are sets stored in a computer's memory?
Could use a list

```python
def set_create():
    return []
```
How are sets stored in a computer's memory?

Could use a list

def set_create():
    return []

def set_in(set_list, item):
    for thing in set_list:
        if thing == item:
            return True
    return False
def set_add(set_list, item):
    for thing in set_list:
        if thing == item:
            return
    return
    set.append(item)
def set_add(set_list, item):
    for thing in set_list:
        if thing == item:
            return
    set.append(item)

How efficient is this?
def set_add(set_list, item):
    for thing in set_list:
        if thing == item:
            return
    set.append(item)

How efficient is this?

With N items in the set, \texttt{in} and \texttt{add} take 1 to N steps
def set_add(set_list, item):
    for thing in set_list:
        if thing == item:
            return
    set.append(item)

How efficient is this?
With N items in the set, in and add take 1 to N steps
"Average" is N/2
def set_add(set_list, item):
    for thing in set_list:
        if thing == item:
            return
    return set.append(item)

How efficient is this?
With N items in the set, \texttt{in} and \texttt{add} take 1 to N steps
"Average" is N/2
It's possible to do \textit{much} better
def set_add(set_list, item):
    for thing in set_list:
        if thing == item:
            return
    set.append(item)

How efficient is this?
With N items in the set, `in` and `add` take 1 to N steps
"Average" is N/2
It's possible to do much better
But the solution puts some constraints on programs
Start simple: how do we store a set of integers?
Start simple: how do we store a set of integers?

If the range of possible values is small and fixed, use a list of Boolean flags ("present" or "absent")
Start simple: how do we store a set of integers? If the range of possible values is small and fixed, use a list of Boolean flags ("present" or "absent")

\[
\begin{align*}
0 & \rightarrow \text{True} \\
1 & \rightarrow \text{False} \\
2 & \rightarrow \text{True}
\end{align*}
\]

\[\{0, 2\}\]
Start simple: how do we store a set of integers?
If the range of possible values is small and fixed, use a list of Boolean flags ("present" or "absent")

```
<table>
<thead>
<tr>
<th>Value</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>True</td>
</tr>
<tr>
<td>1</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
</tr>
</tbody>
</table>
```

But what if the range of values is large, or can change over time?
Use a fixed-size *hash table* of length L
Use a fixed-size *hash table* of length L

Store the integer I at location I \( \% \) L
Use a fixed-size hash table of length $L$

Store the integer $I$ at location $I \% L$

'%' is the remainder operator
Use a fixed-size hash table of length L
Store the integer I at location I \% L
'\%' is the remainder operator

\{3378, 1625, 101\} \Rightarrow 1625, 101, 3378
Time to insert or look up is constant (!)
Time to insert or look up is constant (!)

But what do we do when there's a collision?
Time to insert or look up is constant(!)  
But what do we do when there's a collision?

```
0    1625
1    101
2
3    3378
4
```

+ 206
Option #1: store it in the next empty slot

0 → 1625
1 → 101
2 → 206
3 → 3378

Sets and Dictionaries

Introduction
Option #2: chain values together

```
0  →  1625
1  →  101  →  206
2
3  →  3378
4
```
Either works well until the table is about 3/4 full
Either works well until the table is about 3/4 full
Then average time to look up/insert rises rapidly
Either works well until the table is about 3/4 full
Then average time to look up/insert rises rapidly
So enlarge the table
Either works well until the table is about 3/4 full
Then average time to look up/insert rises rapidly
So enlarge the table

<table>
<thead>
<tr>
<th>0</th>
<th>1625</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3378</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Either works well until the table is about 3/4 full
Then average time to look up/insert rises rapidly
So enlarge the table

Sets and Dictionaries

Introduction
Either works well until the table is about 3/4 full
Then average time to look up/insert rises rapidly
So enlarge the table
Either works well until the table is about 3/4 full
Then average time to look up/insert rises rapidly
So enlarge the table

Sets and Dictionaries
Introduction
How do we store strings?
How do we store strings?

Use a *hash function* to generate an integer index based on the characters in the string.
"zebra"
"zebra" == zebras
Sets and Dictionaries

Introduction

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>e</td>
<td>b</td>
</tr>
<tr>
<td>122</td>
<td>101</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r</td>
</tr>
<tr>
<td></td>
<td></td>
<td>114</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>97</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>122</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>114</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>97</td>
<td></td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>532</td>
</tr>
</tbody>
</table>
Sets and Dictionaries

Introduction
If we can define a hash function for something, we can store it in a set.
If we can define a hash function for something, we can store it in a set.

So long as nothing changes behind our back.
This is what the previous example really looks like in memory
This is what the previous example really looks like in memory

Let's take a look at what happens if we use a list
['z', 'e', 'b', 'r', 'a']
\[
[\text{'z'}, \text{'e'}, \text{'b'}, \text{'r'}, \text{'a'}] \quad \Rightarrow \quad [\text{'z'}, \text{'e'}, \text{'b'}, \text{'r'}, \text{'a'}]
\]
Sets and Dictionaries

Introduction

\[
\left[ 'z', 'e', 'b', 'r', 'a' \right] = \begin{cases}
0 & \rightarrow 'z' \\
1 & \rightarrow 'e' \\
2 & \rightarrow 'b' \\
3 & \rightarrow 'r' \\
4 & \rightarrow 'a'
\end{cases}
\]

\[ 523 \mod 5 \]
Sets and Dictionaries

Introduction
This is what's actually in memory
This is what's actually in memory

What happens if we change the values in the list?
Sets and Dictionaries

Introduction

0 → 'z' → 0
1 → 'e' → 1
2 → 'b' → 2
3 → 'r' → 3
4 → 'a' → 4
Sets and Dictionaries

Introduction
Sets and Dictionaries

Introduction
The list is stored in the wrong place!
The list is stored in the wrong place!

\[ ['s', 'e', 'b', 'r', 'a'] \text{ in } S \]
The list is stored in the wrong place!

\[
['s', 'e', 'b', 'r', 'a'] \text{ in } S
\]
looks at index 0 and says False
The list is stored in the wrong place!

\[
[\text{'s'},\text{'e'},\text{'b'},\text{'r'},\text{'a'}] \text{ in } S
\]

looks at index 0 and says \text{False}

\[
[\text{'z'},\text{'e'},\text{'b'},\text{'r'},\text{'a'}] \text{ in } S
\]
The list is stored in the wrong place!

`['s', 'e', 'b', 'r', 'a']` in $S$
looks at index 0 and says False

`['z', 'e', 'b', 'r', 'a']` in $S$
looks at index 2 and says True
The list is stored in the wrong place!

\[ ['s', 'e', 'b', 'r', 'a'] \] in \( S \)
looks at index 0 and says \texttt{False}

\[ ['s', 'e', 'b', 'r', 'a'] \] in \( S \)
looks at index 2 and says \texttt{True} (or blows up)
This problem arises with any *mutable* structure
This problem arises with any mutable structure.

Option #1: keep track of the sets an object is in, and update pointers every time the object changes.
This problem arises with any \textit{mutable} structure.

Option #1: keep track of the sets an object is in, and update pointers every time the object changes.

Very expensive.
This problem arises with any *mutable* structure.

Option #1: keep track of the sets an object is in, and update pointers every time the object changes.

Very expensive.

Option #2: allow it, and blame the programmer.
This problem arises with any mutable structure.

Option #1: keep track of the sets an object is in, and update pointers every time the object changes.

Very expensive.

Option #2: allow it, and blame the programmer.

Very expensive when it goes wrong.
This problem arises with any *mutable* structure

Option #1: keep track of the sets an object is in, and update pointers every time the object changes

Very expensive

Option #2: allow it, and blame the programmer

Very expensive when it goes wrong

Option #3: only permit *immutable* objects in sets
This problem arises with any *mutable* structure

Option #1: keep track of the sets an object is in, and update pointers every time the object changes

Very expensive

Option #2: allow it, and blame the programmer

Very expensive when it goes wrong

Option #3: only permit *immutable* objects in sets

(If an object can't change, neither can its hash value)
This problem arises with any mutable structure
Option #1: keep track of the sets an object is in, and update pointers every time the object changes
Very expensive
Option #2: allow it, and blame the programmer
Very expensive when it goes wrong
Option #3: only permit immutable objects in sets
(If an object can't change, neither can its hash value)
Slightly restrictive, but never disastrous
So how do we store values that naturally have several parts, like first name and last name?
So how do we store values that naturally have several parts, like first name and last name?

Option #1: concatenate them
So how do we store values that naturally have several parts, like first name and last name?

Option #1: concatenate them

'Charles' and 'Darwin' stored as 'Charles | Darwin'
So how do we store values that naturally have several parts, like first name and last name?

Option #1: concatenate them

'Charles' and 'Darwin' stored as 'Charles | Darwin'

(Can't use space to join 'Paul Antoine' and 'St. Cyr')
So how do we store values that naturally have several parts, like first name and last name?

Option #1: concatenate them

'Charles' and 'Darwin' stored as 'Charles | Darwin'

(Can't use space to join 'Paul Antoine' and 'St. Cyr')

But data *always* changes...
So how do we store values that naturally have several parts, like first name and last name?

Option #1: concatenate them

'Charles' and 'Darwin' stored as 'Charles|Darwin'

(Can't use space to join 'Paul Antoine' and 'St. Cyr')

But data *always* changes...

Code has to be littered with joins and splits
Option #2 (in Python): use a *tuple*
Option #2 (in Python): use a *tuple*

An immutable list
Option #2 (in Python): use a *tuple*

An immutable list

*Contents cannot be changed after tuple is created*
>>> full_name = ('Charles', 'Darwin')
>>> full_name = ('Charles', 'Darwin')

Use '(' instead of '[]'
>>> full_name = ('Charles', 'Darwin')
>>> full_name[0]

Charles
>>> full_name = ('Charles', 'Darwin')
>>> full_name[0]
Charles

>>> full_name[0] = 'Erasmus'
TypeError: 'tuple' object does not support item assignment

Sets and Dictionaries

Introduction
>>> full_name = ('Charles', 'Darwin')
>>> full_name[0]
Charles

>>> full_name[0] = 'Erasmus'
TypeError: 'tuple' object does not support item assignment

>>> names = set()
>>> names.add(full_name)
>>> names
set([(('Charles', 'Darwin'))])

Sets and Dictionaries

Introduction
This episode has been about the science of computer science
This episode has been about the science of computer science

- Designs for hash tables
This episode has been about the science of computer science
- Designs for hash tables
- Mutability, usability, and performance
This episode has been about the science of computer science
- Designs for hash tables
- Mutability, usability, and performance

It's a lot to digest in one go...
This episode has been about the science of computer science

- Designs for hash tables
- Mutability, usability, and performance

It's a lot to digest in one go...

...but sometimes you need a little theory to make sense of practice