Sets and Dictionaries

Tuples
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?
To understand, need to know how sets are stored
To understand, need to know how sets are stored

Allocate a blob of memory
to store references to
set elements
To understand, need to know how sets are stored

Allocate a blob of memory
to store references to
set elements

Use a hash function to
calculate where to store
each element's reference
How a string is stored
How a string is stored

"zebra"
How a string is stored

"zebra" ==

```
  z
  e
  b
  e
  r
  a
```
How a string is stored

"zebra"  ==  ||  122
   e       ==  101
   b       98
   r       114
   a       97
How a string is stored

```
"zebra" ==
z 122
e 101
b 98
r 114
a 97
== 532
```
How a string is stored

"zebra" ==
  z
  e
  b
  r
  a
==

  122
  101
  98
  114
  97

532
How a list would be stored (if it was allowed)
How a list would be stored (if it was allowed)

['z',
 'e',
 'b',
 'r',
 'a']
How a list would be stored (if it was allowed)

```
[ 'z', 'e', 'b', 'r', 'a' ]
```

Sets and Dictionaries

Tuples
How a list would be stored (if it was allowed)

```python
['z', 'e', 'b', 'r', 'a']
```

Sets and Dictionaries
How a list would be stored (if it was allowed)

```
['z',
 'e',
 'b',
 'r',
 'a']
```

Sets and Dictionaries

Tuples
What happens if we change the list?
What happens if we change the list?

0 \rightarrow 's'
1 \rightarrow 'e'
2 \rightarrow 'b'
3 \rightarrow 'r'
4 \rightarrow 'a'
What happens if we change the list?

```
0: 's'
1: 'e'
2: 'b'
3: 'r'
4: 'a'
```
What happens if we change the list?
What happens if we change the list?

['s', 'e', 'b', 'r', 'a'] in S will give a false negative!
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 *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
This is fine for basic types like integers and strings
This is fine for basic types like integers and strings.

But how do we store values that naturally have several parts, like first name and last name?
This is fine for basic types like integers and strings. But how do we store values that naturally have several parts, like first name and last name?

Option #1: concatenate them
This is fine for basic types like integers and strings
But 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'
This is fine for basic types like integers and strings
But 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')
This is fine for basic types like integers and strings
But 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...
This is fine for basic types like integers and strings. But 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
Option #2 (in Python): use a *tuple*

An immutable list

Contents cannot be changed after tuple is created

```python
>>> full_name = ('Charles', 'Darwin')
```
Option #2 (in Python): use a tuple

An immutable list

Contents cannot be changed after tuple is created

```python
>>> full_name = ('Charles', 'Darwin')
Use '()' instead of '[]'
```
Option #2 (in Python): use a *tuple*

An immutable list

Contents cannot be changed after tuple is created

```python
>>> full_name = ('Charles', 'Darwin')
>>> full_name[0]
Charles
```
Option #2 (in Python): use a *tuple*

An immutable list

Contents cannot be changed after tuple is created

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

>>> full_name[0] = 'Erasmus'
TypeError: 'tuple' object does not support item assignment
```
>>> names = set()
>>> names.add(('Charles', 'Darwin'))
>>> names
set([('Charles', 'Darwin')])
>>> names = set()
>>> names.add(('Charles', 'Darwin'))
>>> names
set([(‘Charles’, ‘Darwin’)]]

Cannot look up partial entries
```python
>>> names = set()
>>> names.add(('Charles', 'Darwin'))
>>> names
set([('Charles', 'Darwin')])
```

Cannot look up partial entries

E.g., cannot look for "any tuple ending in 'Darwin'"
>>> names = set()
>>> names.add(('Charles', 'Darwin'))
>>> names
set([('Charles', 'Darwin')])

Cannot look up partial entries
E.g., cannot look for "any tuple ending in 'Darwin"

Next episode will introduce a data structure that
(sort of) allows this