Announcements

- HW5 due tonight

- Trends project due on Tuesday
  - Partners are required; find one in lab or on Piazza
  - Will not work in IDLE
  - New bug submission policy; see Piazza
Iterables

Iterables provide access to some elements in order but do not provide length or element selection.

Python-specific construct; more general than a sequence.

Many built-in functions take iterables as argument.

- **tuple**: Construct a tuple containing the elements.
- **map**: Construct a map that results from applying the given function to each element.
- **filter**: Construct a filter with elements that satisfy the given condition.
- **sum**: Return the sum of the elements.
- **min**: Return the minimum of the elements.
- **max**: Return the maximum of the elements.

For statements also operate on iterable values.
Generator Expressions
One large expression that combines mapping and filtering to produce an iterable
Generator Expressions

One large expression that combines mapping and filtering to produce an iterable

\[(\langle\text{map exp}\rangle\ \text{for}\ \langle\text{name}\rangle\ \text{in}\ \langle\text{iter exp}\rangle\ \text{if}\ \langle\text{filter exp}\rangle)\]
Generator Expressions

One large expression that combines mapping and filtering to produce an iterable

\[(\text{<map exp>} \ for \ \text{<name>} \ in \ \text{<iter exp>} \ if \ \text{<filter exp>})\]

- Evaluates to an iterable.
Generator Expressions

One large expression that combines mapping and filtering to produce an iterable

\[(\text{<map exp>} \text{ for } \text{name} \text{ in } \text{<iter exp>} \text{ if } \text{<filter exp>})\]

- Evaluates to an iterable.
- \<iter exp> is evaluated when the generator expression is evaluated.
Generator Expressions

One large expression that combines mapping and filtering to produce an iterable

$(\texttt{<map exp> for <name> in <iter exp> if <filter exp>})$

- Evaluates to an iterable.
- $\texttt{<iter exp>}$ is evaluated when the generator expression is evaluated.
- Remaining expressions are evaluated when elements are accessed.
Generator Expressions

One large expression that combines mapping and filtering to produce an iterable

\[(\text{<map exp> for <name> in <iter exp> if <filter exp>})\]

- Evaluates to an iterable.
- \(<\text{iter exp}>\) is evaluated when the generator expression is evaluated.
- Remaining expressions are evaluated when elements are accessed.

No-filter version: \((\text{<map exp> for <name> in <iter exp>})\)
Generator Expressions

One large expression that combines mapping and filtering to produce an iterable

\[(<\text{map exp}> \text{ for } <\text{name}> \text{ in } <\text{iter exp}> \text{ if } <\text{filter exp}>)]\]

- Evaluates to an iterable.
- \(<\text{iter exp}>\) is evaluated when the generator expression is evaluated.
- Remaining expressions are evaluated when elements are accessed.

No-filter version: \[(<\text{map exp}> \text{ for } <\text{name}> \text{ in } <\text{iter exp}>)]\]

Precise evaluation rule introduced in Chapter 4.
Reducing a Sequence
Reducing a Sequence

Reduce is a higher-order generalization of max, min, and sum.
Reducing a Sequence

Reduce is a higher-order generalization of max, min, and sum.

```python
>>> from operator import mul
>>> from functools import reduce
>>> reduce(mul, (1, 2, 3, 4, 5), 1)
120
```
Reducing a Sequence

Reduce is a higher-order generalization of max, min, and sum.

```python
>>> from operator import mul
>>> from functools import reduce
>>> reduce(mul, (1, 2, 3, 4, 5), 1)
120
```

First argument:
A two-argument function
Reducing a Sequence

Reduce is a higher-order generalization of max, min, and sum.

```python
>>> from operator import mul
>>> from functools import reduce
>>> reduce(mul, (1, 2, 3, 4, 5), 1)
120
```

First argument:
A two-argument function

Second argument:
an iterable object
Reducing a Sequence

Reduce is a higher-order generalization of max, min, and sum.

```python
>>> from operator import mul
>>> from functools import reduce
>>> reduce(mul, (1, 2, 3, 4, 5), 1)
120
```

First argument: A two-argument function

Second argument: an iterable object

Optional initial value as third argument
Reducing a Sequence

Reduce is a higher-order generalization of max, min, and sum.

```python
>>> from operator import mul
>>> from functools import reduce
>>> reduce(mul, (1, 2, 3, 4, 5), 1)
120
```

Like accumulate from Homework 2, but with iterables.

First argument:
A two-argument function

Second argument:
an iterable object

Optional initial value as third argument
Reducing a Sequence

Reduce is a higher-order generalization of max, min, and sum.

```python
>>> from operator import mul
>>> from functools import reduce
>>> reduce(mul, (1, 2, 3, 4, 5), 1)
120
```

Like accumulate from Homework 2, but with iterables

```python
def accumulate(combiner, start, n, term):
    return reduce(combiner,
                   map(term, range(1, n + 1)),
                   start)
```
More Functions on Iterables (Bonus)

Create an iterable of fixed-length sequences

```python
>>> a, b = (1, 2, 3), (4, 5, 6, 7)
>>> for x, y in zip(a, b):
...     print(x + y)
...5
...7
...9
```

The `itertools` module contains many useful functions for working with iterables

```python
>>> from itertools import product, combinations
>>> tuple(product(a, b[:2]))
((1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5))
>>> tuple(combinations(a, 2))
((1, 2), (1, 3), (2, 3))
```

Produces tuples with one element from each argument, up to length of smallest argument.
Lists
Lists

```python
>>> a = [3, 1, 2]
>>> a
[3, 1, 2]
```
Lists

```python
>>> a = [3, 1, 2]
>>> a
[3, 1, 2]
```
Create a list using square brackets
Lists

```python
>>> a = [3, 1, 2]
>>> a
[3, 1, 2]
>>> len(a)
3
>>> a[1]
1
```

Create a list using square brackets
Lists

>>> a = [3, 1, 2]
>>> a
[3, 1, 2]
>>> len(a)
3
>>> a[1]
1

Create a list using square brackets
Lists are sequences
>>> a = [3, 1, 2]
>>> a
[3, 1, 2]
>>> len(a)
3
>>> a[1]
1
>>> c, d = a, a[:]
>>> a, c, d
([3, 1, 2], [3, 1, 2], [3, 1, 2])

Create a list using square brackets

Lists are sequences
Create a list using square brackets

Lists are sequences

Bind another name to a list or a slice of a list
Lists

```python
>>> a = [3, 1, 2]
>>> a
[3, 1, 2]
>>> len(a)
3
>>> a[1]
1
>>> c, d = a, a[:]
>>> a, c, d
([3, 1, 2], [3, 1, 2], [3, 1, 2])
>>> c[0] = 4
```

Create a list using square brackets
Lists are sequences
Bind another name to a list or a slice of a list
Lists

>>> a = [3, 1, 2]
>>> a
[3, 1, 2]
>>> len(a)
3
>>> a[1]
1
>>> c, d = a, a[:]
>>> a, c, d
([3, 1, 2], [3, 1, 2], [3, 1, 2])
>>> c[0] = 4

Create a list using square brackets

Lists are sequences

Bind another name to a list or a slice of a list

Modify contents of a list
Lists

```python
>>> a = [3, 1, 2]
>>> a
[3, 1, 2]
>>> len(a)
3
>>> a[1]
1
>>> c, d = a, a[:]
>>> a, c, d
([3, 1, 2], [3, 1, 2], [3, 1, 2])
>>> c[0] = 4
>>> a, c, d
([4, 1, 2], [4, 1, 2], [3, 1, 2])
```

Create a list using square brackets

Lists are sequences

Bind another name to a list or a slice of a list

Modify contents of a list
Lists

```python
>>> a = [3, 1, 2]
>>> a
[3, 1, 2]
>>> len(a)
3
>>> a[1]
1
>>> c, d = a, a[:]
>>> a, c, d
([3, 1, 2], [3, 1, 2], [3, 1, 2])
>>> c[0] = 4
>>> a, c, d
([4, 1, 2], [4, 1, 2], [3, 1, 2])
>>> d[0] = 5
>>> a, c, d
([4, 1, 2], [4, 1, 2], [5, 1, 2])
```

Create a list using square brackets

Lists are sequences

Bind another name to a list or a slice of a list

Modify contents of a list
Lists

>>> a = [3, 1, 2]
>>> a
[3, 1, 2]
>>> len(a)
3
>>> a[1]
1
>>> c, d = a, a[:]
>>> a, c, d
([3, 1, 2], [3, 1, 2], [3, 1, 2])
>>> c[0] = 4
>>> a, c, d
([4, 1, 2], [4, 1, 2], [3, 1, 2])
>>> d[0] = 5
>>> a, c, d
([4, 1, 2], [4, 1, 2], [5, 1, 2])
>>> a[1:2] = [7, 8, 9]
>>> a, c, d
([4, 7, 8, 9, 2], [4, 7, 8, 9, 2], [5, 1, 2])
Lists

>>> a = [3, 1, 2]
>>> a
[3, 1, 2]
>>> len(a)
3
>>> a[1]
1
>>> c, d = a, a[:]
>>> a, c, d
([3, 1, 2], [3, 1, 2], [3, 1, 2])
>>> c[0] = 4
>>> a, c, d
([4, 1, 2], [4, 1, 2], [3, 1, 2])
>>> d[0] = 5
>>> a, c, d
([4, 1, 2], [4, 1, 2], [5, 1, 2])
>>> a[1:2] = [7, 8, 9]
>>> a, c, d
([4, 7, 8, 9, 2], [4, 7, 8, 9, 2], [5, 1, 2])

Create a list using square brackets

Lists are sequences

Bind another name to a list or a slice of a list

Modify contents of a list

wut()?
Objects
An *object* is a representation of information.
Objects

An *object* is a representation of information.

All data in Python are objects.
An object is a representation of information

All data in Python are objects

But an object is not just data; it also bundles behavior together with that data
An object is a representation of information.

All data in Python are objects.

But an object is not just data; it also bundles behavior together with that data.

An object’s type determines what data it stores and what behavior it provides.
An object is a representation of information.

All data in Python are objects.

But an object is not just data; it also bundles behavior together with that data.

An object’s type determines what data it stores and what behavior it provides.

```python
>>> type(4)
<class 'int'>
```

```python
>>> type([4])
<class 'list'>
```
Object Attributes
Object Attributes

All objects have attributes
All objects have attributes
We use dot notation to access an attribute
Object Attributes

All objects have attributes
We use dot notation to access an attribute

```python
>>> (4).real, (4).imag
(4, 0)
```
Object Attributes

All objects have attributes
We use dot notation to access an attribute

```python
>>> (4).real, (4).imag
(4, 0)
```

An attribute may be a method, which is a type of function, so it may be called
Object Attributes

All objects have attributes
We use dot notation to access an attribute

```python
>>> (4).real, (4).imag
(4, 0)
```

An attribute may be a method, which is a type of function, so it may be called

```python
>>> [1, 2, 1, 4].count(1)
2
```
Object Attributes

All objects have attributes
We use dot notation to access an attribute

```
>>> (4).real, (4).imag
(4, 0)
```

An attribute may be a method, which is a type of function, so it may be called

```
>>> [1, 2, 1, 4].count(1)
2
```

Notice that we did not have to pass in the list as an argument; the method already knows the object on which it is operating
Creating and Distinguishing Objects
Calling the constructor of a built-in type creates a new object of that type.
Creating and Distinguishing Objects

Calling the constructor of a built-in type creates a new object of that type

Objects can be distinct even if they hold the same data
Creating and Distinguishing Objects

Calling the constructor of a built-in type creates a new object of that type.

Objects can be distinct even if they hold the same data.

The is and not is operators check if two objects are the same.
Creating and Distinguishing Objects

Calling the constructor of a built-in type creates a new object of that type.

Objects can be distinct even if they hold the same data.

The `is` and `not is` operators check if two objects are the same.

```python
>>> [1, 2, 1, 4] is [1, 2, 1, 4]
False
```
Creating and Distinguishing Objects

Calling the constructor of a built-in type creates a new object of that type.

Objects can be distinct even if they hold the same data.

The is and not is operators check if two objects are the same.

```python
>>> [1, 2, 1, 4] is [1, 2, 1, 4]
False
```

Compare to ==, which checks for equality, not sameness.
Creating and Distinguishing Objects

Calling the constructor of a built-in type creates a new object of that type.

Objects can be distinct even if they hold the same data.

The is and not is operators check if two objects are the same.

```python
>>> [1, 2, 1, 4] is [1, 2, 1, 4]
False

Compare to ==, which checks for equality, not sameness.

```python
>>> [1, 2, 1, 4] == [1, 2, 1, 4]
True
```
Objects and Assignment

Example: http://goo.gl/Xrm4k
Objects and Assignment

Assignment does not create a new object

Example: http://goo.gl/Xrm4k
Objects and Assignment

Assignment does not create a new object

```
1 a = [3, 1, 2]
2 c, d = a, a[:]
```

Example: [http://goo.gl/Xrm4k](http://goo.gl/Xrm4k)
Objects and Assignment

Assignment does not create a new object

Example: [Link to example](http://goo.gl/Xrm4k)
Objects and Assignment

Assignment does not create a new object

```
1  a = [3, 1, 2]
2  c, d = a, a[:]
```

But slicing does!

In our environment diagrams, assignment copies the arrow

Example: [http://goo.gl/Xrm4k](http://goo.gl/Xrm4k)
Objects and Assignment

Assignment does not create a new object

```
1  a = [3, 1, 2]
2  c, d = a, a[:]
```

But slicing does!

In our environment diagrams, assignment copies the arrow

The “arrow” is called a pointer or reference

Example: [http://goo.gl/Xrm4k](http://goo.gl/Xrm4k)
Assignment does not create a new object

In our environment diagrams, assignment copies the arrow.

The “arrow” is called a pointer or reference.

Multiple names can point to or reference the same object.

Example: [http://goo.gl/Xrm4k](http://goo.gl/Xrm4k)
Immutable Types
Immutable Types

An object may be *immutable*, which means that its data cannot be changed.
Immutable Types

An object may be *immutable*, which means that its data cannot be changed.

Most of the types we have seen so far are immutable:

- ints, floats, booleans, tuples, ranges, strings
Immutable Types

An object may be *immutable*, which means that its data cannot be changed.

Most of the types we have seen so far are immutable:

- ints, floats, booleans, tuples, ranges, strings

For an immutable type, it doesn’t matter whether or not two equal objects are the same.
An object may be *immutable*, which means that its data cannot be changed.

Most of the types we have seen so far are immutable:

- ints, floats, booleans, tuples, ranges, strings

For an immutable type, it doesn’t matter whether or not two equal objects are the same.

Neither can change, so one is as good as the other.
An object may be *immutable*, which means that its data cannot be changed.

Most of the types we have seen so far are immutable:

- ints, floats, booleans, tuples, ranges, strings

For an immutable type, it doesn’t matter whether or not two equal objects are the same.

Neither can change, so one is as good as the other:

```python
>>> e, f = 1e12, 1e12
>>> e is f
True
>>> e = 1e12
>>> f = 1e12
>>> e is f
False
```
Mutable Types

Example: http://goo.gl/ornZ8
Mutable Types

Mutable objects, on the other hand, can change, and any change affects all references to that object.

Example: http://goo.gl/ornZ8
Mutable objects, on the other hand, can change, and any change affects all references to that object.

Example: [http://goo.gl/ornZ8](http://goo.gl/ornZ8)
Mutable objects, on the other hand, can change, and any change affects all references to that object.

```
1  a = [3, 1, 2]
2  c, d = a, a[:]
3  c[0] = 4
4  d[0] = 5
```

So we need to be careful with mutation.

Example: [http://goo.gl/ornZ8](http://goo.gl/ornZ8)
List Methods
List Methods

Lists have many useful methods
List Methods

Lists have many useful methods

- **append**: add an element to the end of a list
- **extend**: add all elements from an iterable to the end of the list
- **count**: count the number of occurrences of a value
- **pop**: remove an element from the end of a list
- **sort**: sort the elements of a list
List Methods

Lists have many useful methods

- **append**: add an element to the end of a list
- **extend**: add all elements from an iterable to the end of the list
- **count**: count the number of occurrences of a value
- **pop**: remove an element from the end of a list
- **sort**: sort the elements of a list

These methods (except **count**) mutate the list
List Methods

Lists have many useful methods

- **append**: add an element to the end of a list
- **extend**: add all elements from an iterable to the end of the list
- **count**: count the number of occurrences of a value
- **pop**: remove an element from the end of a list
- **sort**: sort the elements of a list

These methods (except **count**) mutate the list

Compare to **sorted(x)**, which returns a new list
List Methods

Lists have many useful methods

- **append**: add an element to the end of a list
- **extend**: add all elements from an iterable to the end of the list
- **count**: count the number of occurrences of a value
- **pop**: remove an element from the end of a list
- **sort**: sort the elements of a list

These methods (except **count**) mutate the list

Compare to **sorted(x)**, which returns a new list

Call **dir(list)** to see a full list of attributes
List Comprehensions
List Comprehensions

We can construct a list using a *list comprehension*, which is similar to a generator expression.
List Comprehensions

We can construct a list using a list comprehension, which is similar to a generator expression

\[
[\text{map exp} \ for \ <\text{name}> \ in \ <\text{iter exp}> \ if \ <\text{filter exp}>]
\]
We can construct a list using a *list comprehension*, which is similar to a generator expression

\[
[\text{map exp} \ for \ <\text{name}> \ in \ <\text{iter exp}> \ if \ <\text{filter exp}>]
\]

```python
generate_list = [3 / x for x in range(4) if x != 0]
generate_list
# [3.0, 1.5, 1.0]
```
List Comprehensions

We can construct a list using a *list comprehension*, which is similar to a generator expression

\[
[\text{map } \exp \text{ for } \text{name} \text{ in } \text{iter } \exp \text{ if } \text{filter } \exp]
\]

- Evaluates to a list.

```python
>>> [3 / x for x in range(4) if x != 0]
[3.0, 1.5, 1.0]
```
List Comprehensions

We can construct a list using a *list comprehension*, which is similar to a generator expression

\[
<\text{map exp}> \text{ for } <\text{name}> \text{ in } <\text{iter exp}> \text{ if } <\text{filter exp}>
\]

- Evaluates to a list.
- <iter exp> is evaluated once.

```python
>>> [3 / x for x in range(4) if x != 0]
[3.0, 1.5, 1.0]
```
List Comprehensions

We can construct a list using a *list comprehension*, which is similar to a generator expression

```
[<map exp> for <name> in <iter exp> if <filter exp>]
```

- Evaluates to a list.
- `<iter exp>` is evaluated once.
- `<name>` is bound to an element, and `<filter exp>` is evaluated. If it evaluates to a true value, then `<map exp>` is evaluated, and its value is added to the resulting list.

```python
>>> [3 / x for x in range(4) if x != 0]
[3.0, 1.5, 1.0]
```
Dictionaries
Dictionaries

Sequences map integers to values
Dictionaries

Sequences map integers to values

```python
>>> a = [3, 1, 2]
```
Dictionaries

Sequences map integers to values

```python
>>> a = [3, 1, 2]
```

<table>
<thead>
<tr>
<th>Integer</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>3</td>
</tr>
<tr>
<td>-2</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Dictionaries

Sequences map integers to values

```python
>>> a = [3, 1, 2]
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>-2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

What if we wanted arbitrary values in the domain?
Dictionaries

Sequences map integers to values

```python
>>> a = [3, 1, 2]
```

<table>
<thead>
<tr>
<th>Value</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>3</td>
</tr>
<tr>
<td>-2</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

What if we wanted arbitrary values in the domain?

```python
'cain'    -> 2.79
'bumgarner' -> 3.37
'vogelsong' -> 3.37
'lincecum'  -> 5.18
'zito'     -> 4.15
```
Dictionaries

Sequences map integers to values

```python
>>> a = [3, 1, 2]
```

What if we wanted arbitrary values in the domain?

We use a dictionary

```python
'cain'    -> 2.79
'bumgarner' -> 3.37
'vogelsong' -> 3.37
'lincecum'  -> 5.18
'zito'      -> 4.15
```
Sequences map integers to values

```python
>>> a = [3, 1, 2]
-3 -> 3  0 -> 3
-2 -> 1  1 -> 1
-1 -> 2  2 -> 2
```

What if we wanted arbitrary values in the domain?

We use a dictionary

```python
>>> eras = {'cain': 2.79,
          'bumgarner': 3.37,
          'vogelsong': 3.37,
          'lincecum': 5.18,
          'zito': 4.15}

>>> eras['cain']
2.79
```
Dictionary Features
Dictionary Features

Dictionaries are not sequences, but they do have a length and are iterable.
Dictionaries are not sequences, but they do have a length and are iterable
- Iterating provides each of the keys in some arbitrary order
Dictionaries are not sequences, but they do have a length and are iterable

- Iterating provides each of the keys in some arbitrary order

```python
>>> total_era = 0
>>> for pitcher in eras:
...     total_era += eras[pitcher]
...
>>> total_era / len(eras)
3.772
```
Dictionaries are not sequences, but they do have a length and are iterable

- Iterating provides each of the keys in some arbitrary order

```python
>>> total_era = 0
>>> for pitcher in eras:
...   total_era += eras[pitcher]
... 
>>> total_era / len(eras)
3.772
```

Dictionaries are mutable
Dictionaries are not sequences, but they do have a length and are iterable

- Iterating provides each of the keys in some arbitrary order

```python
>>> total_era = 0
>>> for pitcher in eras:
...    total_era += eras[pitcher]
... total_era / len(eras)
3.772
```

Dictionaries are mutable

```python
>>> eras['lincecum'] = 3.0
```
Dictionaries are not sequences, but they do have a length and are iterable

- Iterating provides each of the keys in some arbitrary order

```python
>>> total_era = 0
>>> for pitcher in eras:
...     total_era += eras[pitcher]
...
>>> total_era / len(eras)
3.772
```

Dictionaries are mutable

```python
>>> eras['lincecum'] = 3.0
```

There are dictionary comprehensions, which are similar to list comprehensions
Dictionaries are not sequences, but they do have a length and are iterable

- Iterating provides each of the keys in some arbitrary order

```python
>>> total_era = 0
>>> for pitcher in eras:
...    total_era += eras[pitcher]
...
>>> total_era / len(eras)
3.772
```

Dictionaries are mutable

```python
>>> eras['lincecum'] = 3.0
```

There are dictionary comprehensions, which are similar to list comprehensions

```python
>>> {p: round(eras[p]-1, 3) for p in eras}
{'zito': 3.15, 'cain': 1.79, 'bumgarner': 2.37, 'lincecum': 2.0, 'vogelsong': 2.37}
```
Limitations on Dictionaries
Limitations on Dictionaries

Dictionaries are unordered collections of key-value pairs.
Limitations on Dictionaries

Dictionaries are unordered collections of key-value pairs.

Dictionary keys do have two restrictions:
Limitations on Dictionaries

Dictionaries are unordered collections of key-value pairs.

Dictionary keys do have two restrictions:

- A key of a dictionary cannot be an object of a mutable built-in type.
Limitations on Dictionaries

Dictionaries are unordered collections of key-value pairs.

Dictionary keys do have two restrictions:

- A key of a dictionary cannot be an object of a mutable built-in type.
- Two keys cannot be equal. There can be at most one value for a given key.
Dictionaries are unordered collections of key-value pairs.

Dictionary keys do have two restrictions:

- A key of a dictionary cannot be an object of a mutable built-in type.
- Two keys cannot be equal. There can be at most one value for a given key.

This first restriction is tied to Python's underlying implementation of dictionaries.
Dictionaries are unordered collections of key-value pairs.

Dictionary keys do have two restrictions:

- A key of a dictionary cannot be an object of a mutable built-in type.
- Two keys cannot be equal. There can be at most one value for a given key.

This first restriction is tied to Python's underlying implementation of dictionaries.

The second restriction is an intentional consequence of the dictionary abstraction.