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` constructs a tuple containing the elements.
- `map` constructs a map that results from applying the given function to each element.
- `filter` constructs a filter with elements that satisfy the given condition.
- `sum` returns the sum of the elements.
- `min` returns the minimum of the elements.
- `max` returns 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:

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

- Evaluates to an iterable.
- `<iter exp>` is evaluated when the generator expression is evaluated.
- Remaining expressions are evaluated when elements are accessed.

No-filter version: `<map exp> for <name> in <iter exp>`

Precise evaluation rule introduced in Chapter 4.

Reducing a Sequence

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

```
>>> 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.

```
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.

```
>>> 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.

```
>>> 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))
```
**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’s type determines what data it stores and what behavior it provides

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

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

**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` operators check if two objects are the same

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

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

```
>>> [1, 2, 1, 4] == [1, 2, 1, 4]
True
```

**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

- `int`, `float`, `bool`, `str`, `tuple`, `range`

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

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

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

So we need to be careful with mutation.

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

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.

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

Dictionaries

Sequences map integers to values

```
>>> a = [3, 1, 2]
>>> a[3] = 3.15
>>> a[1] = 2.79
>>> a[2] = 5.18
```

What if we wanted arbitrary values in the domain?

We use a dictionary:

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

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

Dictionary Features

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

- Iterating provides each of the keys in some arbitrary order

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

Dictionaries are mutable

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

There are dictionary comprehensions, which are similar to list comprehensions:

```
>>> (p: round(eras[p] / 1, 3) for p in eras)
{'zito': 3.15, 'cain': 2.79, 'bumgarner': 2.37, 'lincecum': 2.8, 'vogelsong': 2.37}
```

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.

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

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