Iterable Classes

Iterable Classes

Iteration and iterable objects are fundamental to programming in Python.

- `list, set, str, dict,` and `tuple` all support iteration.
- User-defined classes can easily make themselves iterable by defining:
  - `__iter__(self)`
  - `__next__(self)`

Reminder: Iterator Protocol

```
## get a list containing odd numbers less than 9
>>> l = list(range(1, 9, 2))
## get the iterator object
>>> l_iter = iter(l)

## calling next on the iterator returns the
## "next" underlying element
>>> next(l_iter)
1
>>> next(l_iter)
3
>>> next(l_iter)
5
>>> next(l_iter)
7
## when there are no more elements StopIteration is thrown
>>> next(l_iter)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
StopIteration
## note: an iterator only lets you traverse its values once!!!
```

Iterable Class

```python
class CountToTen:
    def __init__(self):
        self.value=0

    ## must return an object that defines __next__
    def __iter__(self):
        return self

    ## returns "next" value and throws StopIteration
    ## when done
    def __next__(self):
        if self.value > 10:
            raise StopIteration
        else:
            ret = self.value
            self.value+= 1
            return ret


for i in CountToTen():
    print(i)
```

Iterable Class [2]

Each method serves a purpose.

- The `__init__` method starts the value of the iterator at 0.
- The `__iter__` method states that this class is its own iterator.
- The `__next__` method does several things.
  - Checks if the iteration is complete, raising a StopIteration error if so
  - Advances to the next item
  - Returns the current item

Hashable Classes

Hashability

Classes must be hashable to be stored in a set or used as a key in a dictionary.

- Hashable objects can be passed to the `hash()` built-in function, which returns an `int`.
- Of built-in types, only immutable types are hashable.
- Immutable collections are not hashable if they contain nonhashable elements.

Nonhashable Built-in Types

```
## values that are not hashable:
a = [1, 2, 3]           ## lists
b = {1, 2, 3}           ## sets
c = {'hello':'world'}   ## dicts
d = (1, 2, [1, 2])      ## tuples containing non-hashable
e = (1, 2, (3, [4]))    ## contains a non-hashable tuple
```

- Making custom classes hashable ensures that they are versatile.
- Custom classes are hashable by default, unless they define the `__eq__` method.

Equality and Hash

```
class A:
    def __init__(self, val):
        self.val = val

class B:
    def __init__(self, val):
        self.val = val
    def __eq__(self, other):
        return self.val == other.val

s = { A(1) }    ## this works
s = { B(2) }    ## this raises a TypeError
```

- Hashable objects must satisfy:
  - if *x* == *y* then *hash(x)* == *hash(y)*
- Changing == test requires changing hash.

Making Hashable

```
class B:
    def __init__(self, val):
        self.val = val
    def __eq__(self, other):
        return self.val == other.val
    def __hash__(self):
        return hash((self.val))

## now, B is hashable again
s = { B(1), B(2), B(3) }
```

Documentation recommends placing attributes in a tuple and hashing the tuple.

Comparison Operators

Comparison of list, tuple, str

```
a = [1, 1, 1, 1, 1]
b = [1, 0, 1000, 1000, 1000]
c = [1]

assert a > b  ## because a[0] == b[0] and a[1] > b[1]
assert a > c  ## because a[0] == c[0] and len(c) == 1
assert b > c  ## because b[0] == c[0] and len(c) == 1
```

Lists, tuples, and strings compare in lexicographic order.

- They compare one element at a time.
- The one that reaches a larger element first is larger.
- If the end of the list is reached, the shorter list is considered less.

Comparison of Sets

```
a = {1, 2, 3}
b = {1, 2}
c = {2, 3}

assert a > b  ## 'a' contains 'b' as a subset
assert a > c  ## 'a' contains 'c' as a subset
```

- For sets, comparisons are subset operations.
- *A* <= *B* if *A* is a subset of *B*.
- *A* < *B* if *A* is a proper subset of *B.*
  - *A* <= *B* and *A* != *B*.
- *A* >= *B* if *A* is a superset of *B*.
- *A* > *B* if *A* is a proper superset of *B.*
  - *A* >= *B* and *A* != *B.*

Defining Custom Comparisons

Comparison operators can be overridden by defining the corresponding magic methods.

`==`	`!=`	`<`	`>`	`<=`	`>=`
`__eq__`	`__ne__`	`__lt__`	`__gt__`	`__le__`	`__ge__`

Default Equality

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

p1 = Point(1, 2)
p2 = Point(1, 2)

## each point is the same as itself (reference equality)
assert p1 == p1
assert p2 == p2

## no __eq__, so p1 and p2 are not equal
assert p1 != p2
```

Custom Equality

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        if type(self) == type(other):
            return self.x == other.x and self.y == other.y
        else:
            return NotImplemented

p1 = Point(1, 2)
p2 = Point(1, 2)

assert p1 == p2
## not equal is now defined as well
## simply as the negation of __eq__
assert not (p1 != p2)
```

`NotImplemented`

```python
p = Point(1, 2)
t = (1, 2)
assert t != p
```

- Python tries a series of different comparisons at runtime.
  1. Tries to call *t.__ne__(p)*
  2. Tries to call *p.__ne__(t)*
  3. Tries to call *not p.__eq__(t)*
  4. Tries reference equality
- Raising `NotImplemented` keeps it trying.

Order Relations

```
class Time:
    def __init__(self, hours, minutes, seconds):
        self.hours = hours
        self.minutes = minutes
        self.seconds = seconds

    def __lt__(self, other):
        if type(self) == type(other):
            return (self.hours, self.minutes, self.seconds)
            < (other.hours, other.minutes, other.seconds)
        else:
            return NotImplemented
```

- Order relations work the same way
- Convenient to use tuples to pack values and do the actual comparison

Numeric Operators

Mathematical Terminology

- With numeric operators, two properties come up frequently: *commutativity* and *associativity*
- An operator ? is:
    - Commutative if for all *x,y* we have *x*y == y*x*
    - Associative if for all *x,y,z* we have *(x*y)*z == x*(y*z)*
- For example
    - + and * on integers are both commutative and associative
    - – and / on integers are neither commutative nor associative

Built-in Set Example

```
s1 = {1, 2, 3}
s2 = {2, 3, 4}

## the union | is the elements in s1 OR s2
assert (s1 | s2) == {1, 2, 3, 4}

## the intersection is the elements in s1 AND s2
assert (s1 & s2) == {2, 3}

## the symmetric difference is the elements
## in s1 OR s2 but not both
assert (s1 ^ s2) == {1, 4}
```

- `set` defines |, &, and ^ operators.
- These implement union, intersection, and difference.

Common Numeric Operators

Custom classes can support these numeric operators by implementing the corresponding magic method.

+	-	*	/	//
`__add__`	`__sub__`	`__mul__`	`__truediv__`	`__floordiv__`

Custom + Operator

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    ## must return a new object, not change self or other
    def __add__(self, oth):
        if type(self) == type(oth):
            return Point(self.x + oth.x, self.y + oth.y)
        else:
            return NotImplemented

p1 = Point(10, 100)
p2 = Point(20, 200)

p3 = p1 + p2
assert p3.x == 30 and p3.y == 300
```

Custom * Operator

```
class Point:
    ...

    def __mul__(self,oth):
        if type(oth) == int:
            return Point(self.x*oth, self.y*oth)
        else:
            return NotImplemented

p1 = Point(10, 100)
p2 = Point(20, 200)

p4 = p1 * 4
assert p4.x == 40 and p4.y == 400
```

- Allows Point to be multiplied by an integer
- Works when int is on the right side

Reflected * Operator

```
class Point:
    . . .
    ## mul operator with int on the l.h.s
    def __rmul__(self,oth):
        ## Point considers '*' commutative
        return self.__mul__(oth)

p1 = Point(10, 100)
p2 = Point(20, 200)

p5 = 5 * p1
assert p5.x == 50 and p5.y == 500
```

- Works when int is on the left side