Overview and namedtuple

Built-in Collections

- “Collections” refers to data structures that contain multiple items
- Reminder: built-in collections types
  - `list`: growable/mutable array
    - For example, `[ 1, 2, 3 ]`
  - `tuple`: fixed size, immutable sequence
    - For example, `( 1, 2, 3 )`
  - `set`: unordered, no duplicates
    - `{ 'a', 'b', 'c', 'd' }`
  - `dict`: key/value pairs, keys are unique
    - `{ 'a': 1, 'b', 2 }`

`collections` Module

- Provides specialized collections that enhance basic data types
- Built-in types are very powerful but do not support some common-use cases

Built-in `tuple` Type

- Sequence of values, fixed size
- Elements cannot be changed
  - Can be mutable objs
- Packed, unpacked, iterated, and referenced by index
- Problem: index meaning is lost

```
## tuple object creation
a = (1, 4.5, 'something')

## packed
b = 2, 9.0, 'anything'

## referenced by position
z = a[1]

## unpacked
x, y, s = a

## iterated
for v in a:
    print(v)
```

namedtuple

```
from collections import namedtuple

## Create class called Point, subclass of tuple
## x and y are the named attributes
Point = namedtuple('Point', 'x y')

## Instantiate a point object
p = Point(1, 2)
assert p.x == 1
assert p.y == 2
```

- `Point` is a class that inherits from `tuple`.
- The attribute specification indicates that there are two attributes; the first is `x` and the second is `y`.

`namedtuple` Factory Variants

```
## all of these are equivalent
Point = namedtuple('Point', 'x y')
Point = namedtuple('Point', 'x, y')
Point = namedtuple('Point', 'x,y')
Point = namedtuple('Point', ['x', 'y'])
```

- The second argument to the `namedtuple` factory method specifies the name and order of the attributes
- List of `str` objects, or comma or space separated `str` objects
- No duplicate names, valid identifiers

`namedtuple` Is a Tuple

```
p = Point(1, 2)

## can be unpacked
x, y = p
copy = Point(*p)

## or referenced by index or range
x, y = p[0], p[1]
b, = p[1:]

## or iterated:
for coord in p:
    print(coord)
```

`namedtuple` objects retain the fundamental capabilities of `tuple`.

`namedtuple` Benefits

```
## attributes can be referred to by name
p = Point(3, 5)
b = p.x
a = p.y

## constructor enforces number of attributes
q = Point(5) ## throws TypeError
q = Point(5, 6, 7) ## throws TypeError

## string output also includes names
print(p)
## outputs: "Point(x=3, y=5)"
```

- Attributes can be referred by name
- Type enforces attribute count
- “Flyweight” pattern

defaultdict and OrderedDict

Built-in `dict` Type

```
d = { 'fname': 'Matt',
    'lname' : 'Rutherford' }

print(d['fname'])
# 'Matt'

print(d['mname'])
# Traceback (most recent call last):
#   File "dict.py", line 5, in <module>
#     print(d['mname'])
# KeyError: 'mname'
```

- Mapping type, relates hashable "key" to a value
- Problem: accessing unknown key throws a `KeyError`

Dealing With `KeyError`

```
anagrams = {}
with open('dictionary.txt') as f:
    for line in f:
        word = line.strip().lower()
        key = ''.join(sorted(word))
        ## check if key already exists in the dict
        if key in anagrams:
            anagrams[key].append(word)
        else:
            anagrams[key] = [word]
```

All code that deals with an unknown set of keys must explicitly check for key existence (**bold** above).

`defaultdict` Type

```
from collections import defaultdict

anagrams = defaultdict(list)
with open('dictionary.txt') as f:
    for line in f:
        word = line.strip().lower()
        key = ''.join(sorted(word))
        anagrams[key].append(word)
```

- The `defaultdict` object handles automatic creation of an entry when an unknown key is referenced.
- The argument to defaultdict is a *factory function* for the empty value (in this case a list).

`OrderedDict` Type

```
from collections import OrderedDict

## establish the order of keys
d = OrderedDict.fromkeys('acebd')
d['a'] = 'A'
d['c'] = 'C'
d['b'] = 'B'
d['e'] = 'E'
d['d'] = 'D'

for k, v in d.items():
    print(k, v)
```

- `OrderedDict` is a subtype of `dict` that iterates based on the insert order of keys.
- Built-in `dict` iterates arbitrarily.

Counter and deque

`Counter` Type

```
from collections import Counter

## counts frequencies of characters in a string:
chars = Counter('This is a string')

## counts frequencies of elements in a list
ints = Counter([4, 2, 4, 5, 6, 3, 4, 2, 1, 3, 2])

## counts frequencies of words in a sentence
words = Counter('This is a list of words.'.split())

## we can update a counter with more values using update(...)
words.update('This is another list of words.'.split())
```

`Counter` is a subclass of `dict` where the keys are the elements of the input sequence and the values are the number of occurrences.

`Counter` Type [2]

```
>>> from collections import Counter
>>> ints = Counter([4, 2, 4, 5, 6, 3, 4, 2, 1, 3, 2])
>>> print(ints)
Counter({4: 3, 2: 3, 3: 2, 5: 1, 6: 1, 1: 1})

>>> Counter("the quick brown fox jumped over the lazy
dog".split())
Counter({'the': 2, 'quick': 1, 'brown': 1, 'fox': 1,
'jumped': 1, 'over': 1, 'lazy': 1, 'dog': 1})
```

- The elements passed into the `Counter` constructor must be compatible with being keys in a dictionary.
- When iterating over the `Counter`, the order is in descending frequency.

`Counter` Type [3]

```
## returns a list tuples of (element, count)
## in descending count order
chars.most_common()

## returns a list of up to 2 tuples of (element, count)
## in descending count order
words.most_common(2)

c = Counter('abcd')
d = Counter('cdef')

## adds counts together
e = c + d

## subtracts totals, leaving only the
## positive counts in the result
f = c - d
```

Utility functions

deque Type

```
from collections import deque

## create an empty deque.grows arbitrarily large
a = deque()

## creates a deque, initially containing elements provided
b = deque([1, 5, 9, 0, 40])
c = deque("str objects are also iterable")

## add element to the right (end)
b.append(100)

## add element to the left (beginning)
b.appendleft(0)
```

- `deque` is pronounced “deck.”
- Sequence type is optimized to efficiently grow from the front and the back.

deque as Stack

```
## use like a stack by adding and removing
## from the same side

## append adds to the right
stack.append(100)
stack.append(200)

## pop removes from the right
v = stack.pop()
assert v == 200
```

- A stack implements LIFO—"last in, first out."
- Appending and popping from the same side provides this.
- `deque.appendleft()` and `deque.popleft()` could also have been used above.

`deque` as Queue

```
## create an empty deque
queue = deque()

## append to the right
queue.append('ADD')
queue.append('SUB')
queue.append('SUB')

## pop from the left to access elements
## in the order they were added
v = queue.popleft()
assert v == 'ADD'
```

- A queue implements FIFO—"first in, first out."
- Appending and popping from different sides provides this.
- `deque.appendleft()` and `deque.pop()` could also have been used above.

Lambdas, Filter, and Map

Lambdas

```
add_two = lambda x: x + 2

print(add_two(1))
print(add_two(2))
print(add_two(3))

## note: this is basically the same as:
# def add_two(x):
#     return x + 2
```

- Lambdas are short, anonymous functions.
  - Short: consist of a single expression that is the return value
  - Anonymous: can be given a name (as above), but are often passed as arguments directly without being named

Lambdas [2]

```
sum_three = lambda x, y, z: x + y + z

print(sum_three(1, 2, 3))
print(sum_three(4, 5, 6))

## note: the previous definition is equivalent to:
# def sum_three(x, y, z):
#     return x + y + z
```

- Lambdas can take as many parameters as needed.
- Lambdas cannot have if statements, loops, or a return statement.

Lambda Usage

- Useful when you need to provide a function object but do not want to bother with formally defining a function
- Code readability **may** be enhanced with lambdas if having the function definition at the call site clarifies your intentions
- Should be used sparingly; often, a named function, list comprehension, or generator expression can be used instead

`filter` Function

```
l = [1, 2, 3, 4, 5, 6]

## usage: filter(function_object, iterable)

## get the even elements from a list
f = filter(lambda x: x%2 == 0, l)

## get the odd elements from a list
f = filter(lambda x: x%2 == 1, l)

## get the non-whitespace strings
l2 = [ "this", " ", "sequence", " ", "has", " ", "spaces" ]
f = filter(lambda x: len(x.strip()) > 0, l2)
```

`filter` returns an iterator over a sequence created out of each element for which the function returns `True`.

`filter` Function [2]

```
## 1st parameter, any function (x) --> bool
## 2nd parameter to filter just has to be iterable
l3 = 'abc123def'
f = filter(str.isalpha, l3)
```

- The first parameter to filter does not have to be a `lambda`.
  - Can be any function that takes a single parameter and returns a `bool`
  - Can be a member function, referenced by the class name (above, "`str`" is the class and "`isalpha`" is defined on that class)
- The second parameter to `filter` just has to be iterable (`str` in example).

`filter` Function [3]

```
l3 = 'abc123def'
f = filter(str.isalpha, l3)
## can be converted into another sequence
l = list(f)
## basically passed into anything taking iterable
s = "".join(filter(str.isalpha, l3))
```

- `filter` returns an iterator.
- This can be used immediately in a loop.
- It can also be passed into anything that takes an iterable.
- *Lazy evaluation* means that the function is only applied one at a time as needed by an iterator.

`map` Function

```
l = [1, 2, 3, 4, 5, 6]

## usage: map(function_object, iterable[, iterable, ...])

## double the elements in a list
m = map(lambda x: 2*x, l)
## convert elements of a list into a string
m = map(str, l)

## methods can also be called by referencing them on the class
strings = ['hello', 'goodbye', 'hello']
m = map(str.upper, strings)
```

`map` returns an iterator over a sequence created out of the result of the function being applied to each element.

`map` With Multiple Arguments

```
## map with multiple
L1 = [1, 2, 3, 4, 5]
L2 = [3, 5, 2, 6, 4]
L3 = [7, 6, 2, 1, 5]

## calculate the element-wise sum of lists
list(map(lambda x, y: x + y, L1, L2))
list(map(lambda x,y,z: x+y+z, L1, L2, L3))

## find the minimum or maximum element for each index
list(map(min, L1, L2, L3))
list(map(max, L1, L2, L3))
```

- With multiple iterable arguments, `map` must be provided a function with the same number of parameters.
- It applies the function element-wise and stops at the end of the shortest iterable.

`map` and `filter` vs. List Comprehensions

```
L = [1, 2, 3, 4, 5, 6]

## double the elements of a list
[ 2*x for x in L ]
list(map(lambda x: 2*x, L))

## convert elements of a list into a string
[ str(x) for x in L ]
list(map(str, L))

## get the even elements of a list
[ x for x in L if x%2 == 0 ]
list(filter(lambda x: x%2 == 0, L))

## get the odd elements of a list
[ x for x in L if x%2 == 1 ]
list(filter(lambda x: x%2 == 1, L))
```

Often, list comprehensions can accomplish the same effect as map/filter.

`map` With Multiple Arguments

```
## map with multiple
L1 = [1, 2, 3, 4, 5]
L2 = [3, 5, 2, 6, 4]
L3 = [7, 6, 2, 1, 5]

## calculate the element-wise sum of lists
list(map(lambda x, y: x + y, L1, L2))
list(map(lambda x,y,z: x+y+z, L1, L2, L3))

## find the minimum or maximum element for each index
list(map(min, L1, L2, L3))
list(map(max, L1, L2, L3))
```

- With multiple iterable arguments, `map` must be provided a function with the same number of parameters.
- It applies the function element-wise and stops at the end of the shortest iterable.

Iterators and Generators

“Iterable”

The concept of objects being “iterable” is fundamental to Python.

- It is used in all kinds of loops.
- All sequence types are iterable.
  - str, list, tuple, dict, set
- `map` and `filter` have iterable parameters.
- Basically, any object that has either the `__iter__` `__getitem__` method defined is iterable.

```
>>> hasattr(set, '__iter__')
True
>>> hasattr(str, '__getitem__')
True
```

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!!!
```

Simple Generator

```
>>> def gen_simple():
...     yield 1
...     yield 2
...     yield 3
...
>>> simple = gen_simple()
>>> next(simple)
1
>>> next(simple)
2
>>> next(simple)
3
>>> next(simple)
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
StopIteration
```

The generator lets you get an iterable object that calculates its “next” value upon demand.

Generators Save Memory

```
## list comprehension to generate a list
## entire list is in memory after the first statement
squares = [ x*x for x in range(10000) ]
for s in squares:
    print(s)

## generator function to accomplish the same thing
## yield keyword returns the "next" element and
## picks up where it left off
def gen_squares():
    for x in range(10000):
        yield x*x

## use the generator to get something iterable
##
squares = gen_squares()
for s in squares:
    print(s)
```

`Yield` Keyword

- Generators define their elements with the `yield` keyword.
- `yield` is like `return`, but it can be executed multiple times.
- When `next()` is called the first time, the generator executes up to the first `yield` statement and returns the value.
- Subsequent calls to `next()` go from after the previous yield to the next one.

Generator Execution Trace

```
def gen_example(limit):
    print('started!')
    for i in range(0, limit):
        print(f'yielding {i}...')
        yield I
        print(f'after yielding {i}...')
    print('finishing...')
```

```
>>> g = gen_example(2)
>>> next(g)
started!
yielding 0...
0
>>> next(g)
after yielding 0...
yielding 1...
1
>>> next(g)
after yielding 1...
finishing...
Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
StopIteration
```

Generator Notes

- Creating the generator doesn't start its execution. Execution begins with the first call to *next(...)*.
- Each time *next(...)* is called, it resumes execution at the last *yield*.
- Once the end of the function is reached, a *StopIteration* exception is automatically raised.
- Just like a normal iterator, continuing to call *next(...)* raises another *StopIteration* exception.