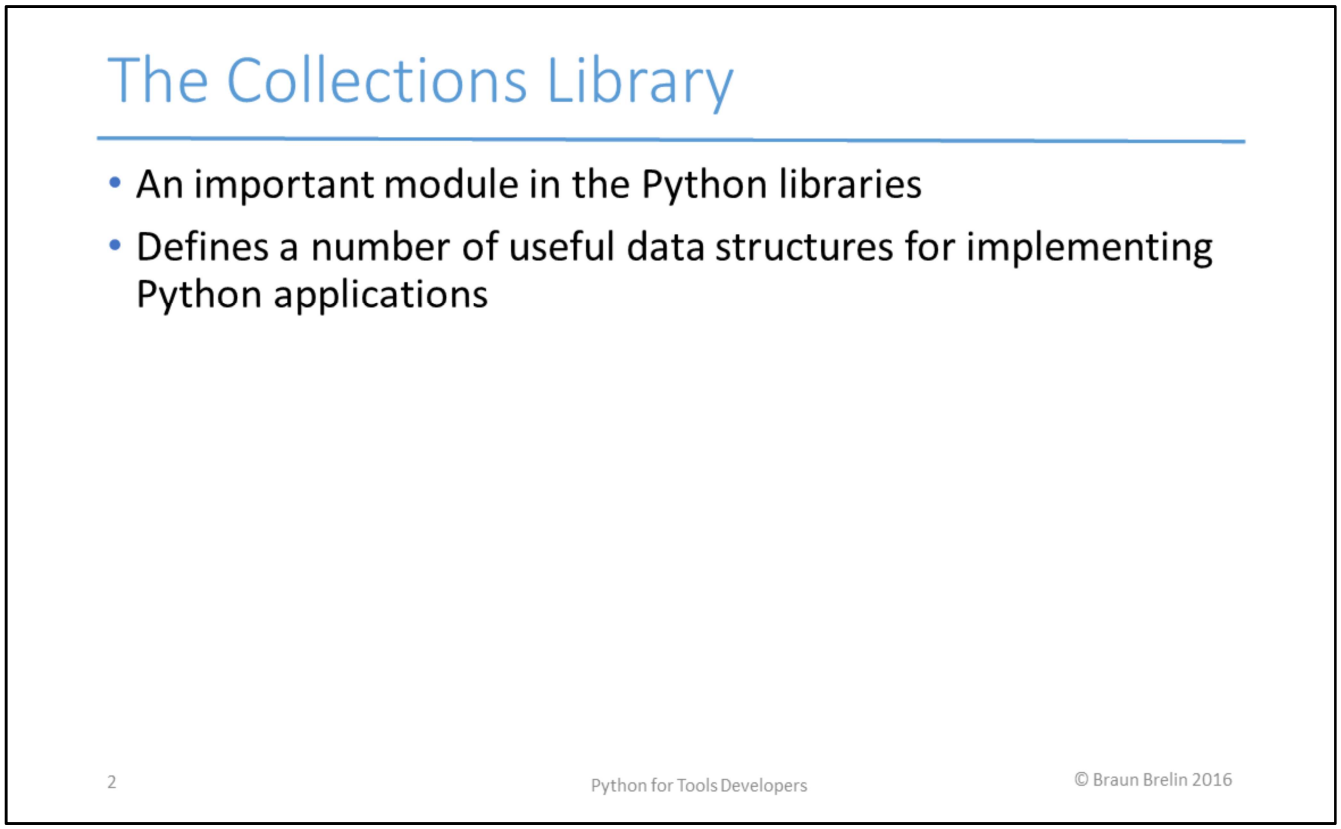
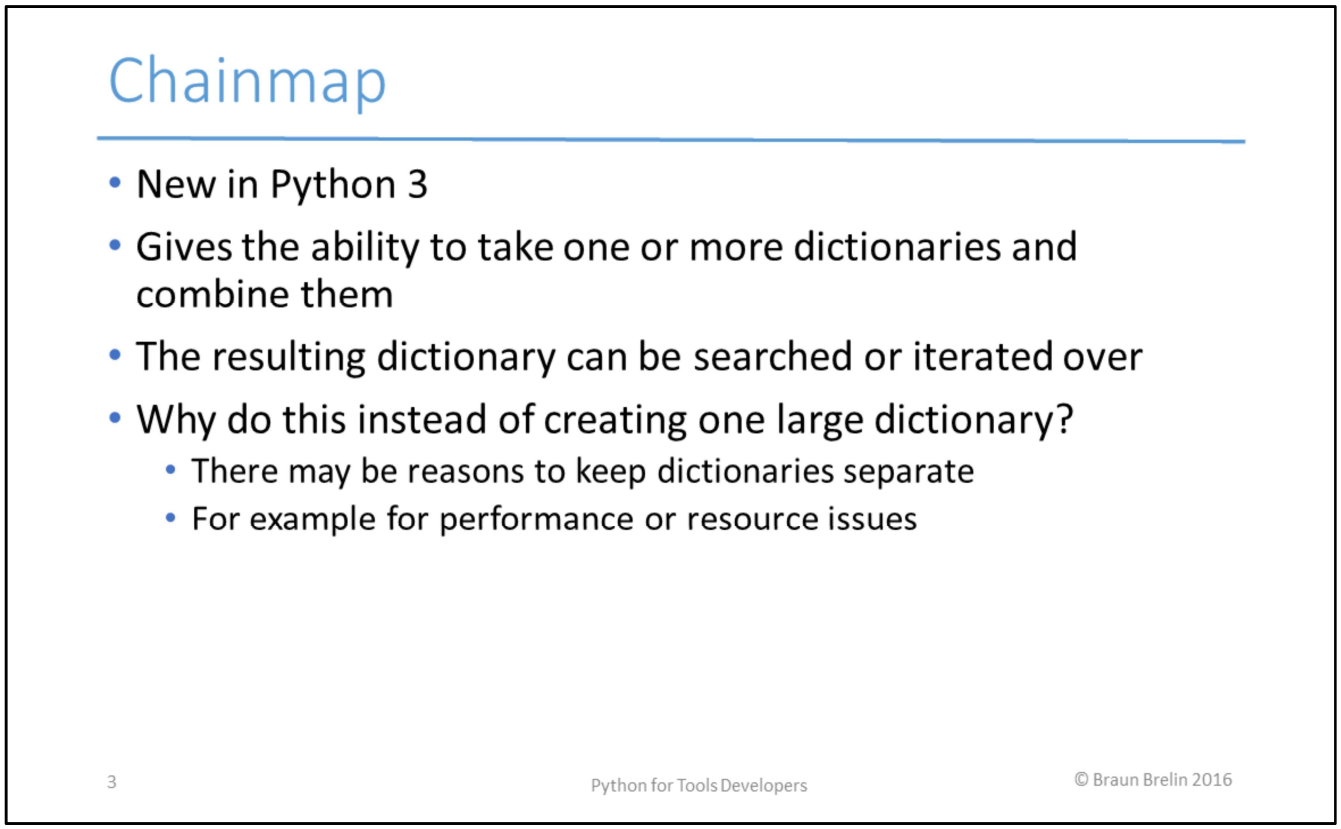
Module 4

The Python 3 Collections Library

Python for Tools Developers



An important module in the Python libraries is the Collections library. This library defines a number of useful data structures for implementing Python applications. Let's take a tour through this module.



The Chainmap

Chainmaps are new to Python 3. A chainmap provides the ability to take one or more (usually more) dictionaries and combine then into one large dictionary that can be searched on or iterated over.

Let's see an example of this:

```
#!/usr/bin/python3

from collections import ChainMap

# Here are three  separate dictionaries that contain a car
part name
# and a part number.
car_parts
={'hood':'1P994','engine':'2X2001','front_door':'1Y8884'}
car_options={'air_conditioning':'9B0003','Turbo':'1D9110',
'rollbar':'5Z0123'}
car_accessories =
{'Cover':'4T1413','hood_ornament':'5N0512','seat_cover':'7
C0316'}
```

Continued ….

```
# Here we use the chainmap to combine the three
dictionaries into one.
car_manifest =
ChainMap(car_parts,car_options,car_accessories)

print (car_manifest['engine'])
car_manifest['Cover'] = '4T1414'
print (car_manifest['Cover'])

print (car_manifest.maps[0])
```

The output of this program is:

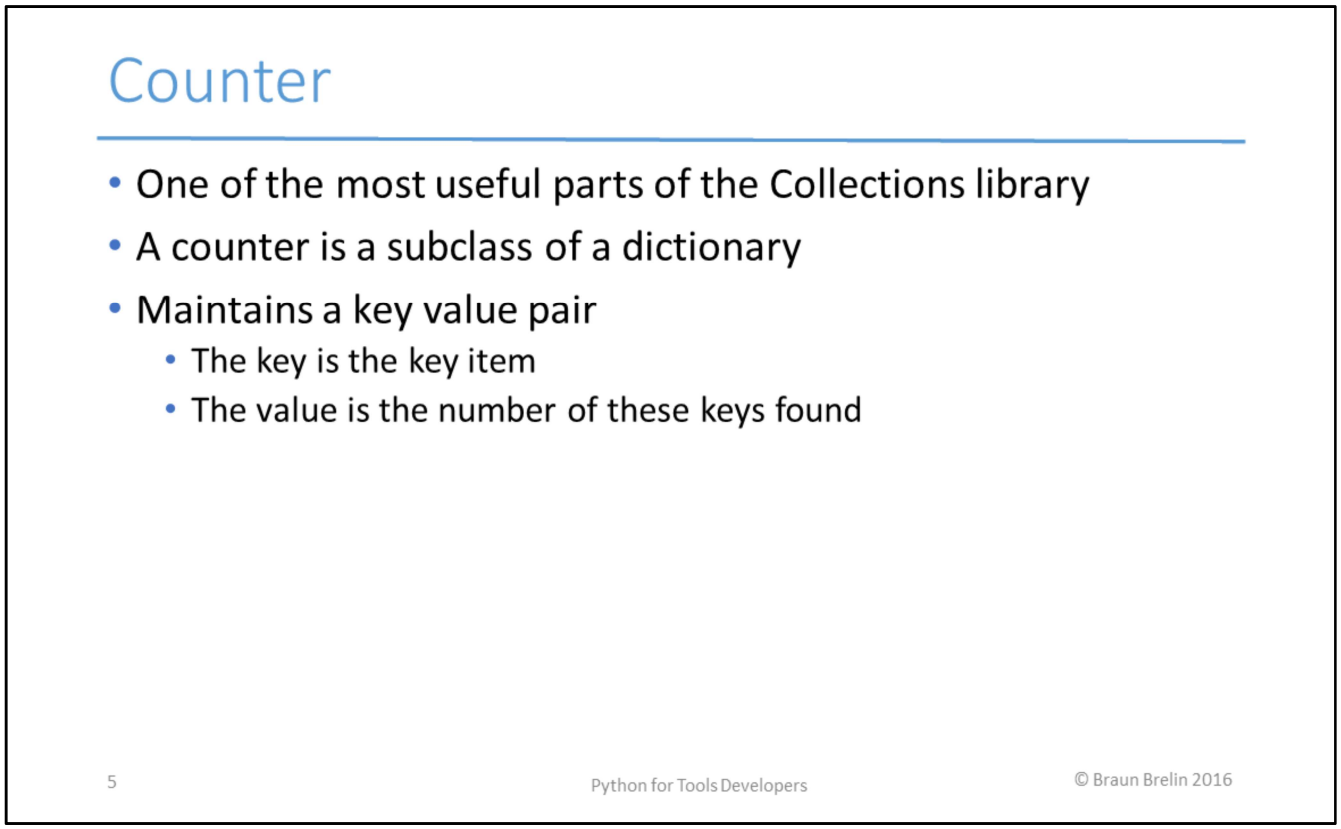
2X2001

4T1414

{'hood': '1P994', 'Cover': '4T1414', 'engine': '2X2001', 'front_door': '1Y8884'}

Why do this instead of creating one large dictionary? There may be reasons in your application to keep the dictionaries separate for other purposes. For example, if you often do iterations over the car parts dictionary, it makes sense to keep it as small as possible for performance and resource reasons.

A couple of notes on `ChainMap`. The `ChainMap` map attribute is a list of the maps in the chain, i.e. in this case `car_manifest.maps[0]` references the `car_parts` dictionary. The `new_child` method allows the creation of a new `ChainMap` containing a new map at the front of the list, followed by all of the other maps in the list. It defaults to an empty map at the start of the maps list if no map is given.



Counter

This is one of the most useful components of the Collections library. A counter is a subclass of a dictionary. Counters maintain a key value pair where the key is the key item and the value is the number of these keys found. For example:

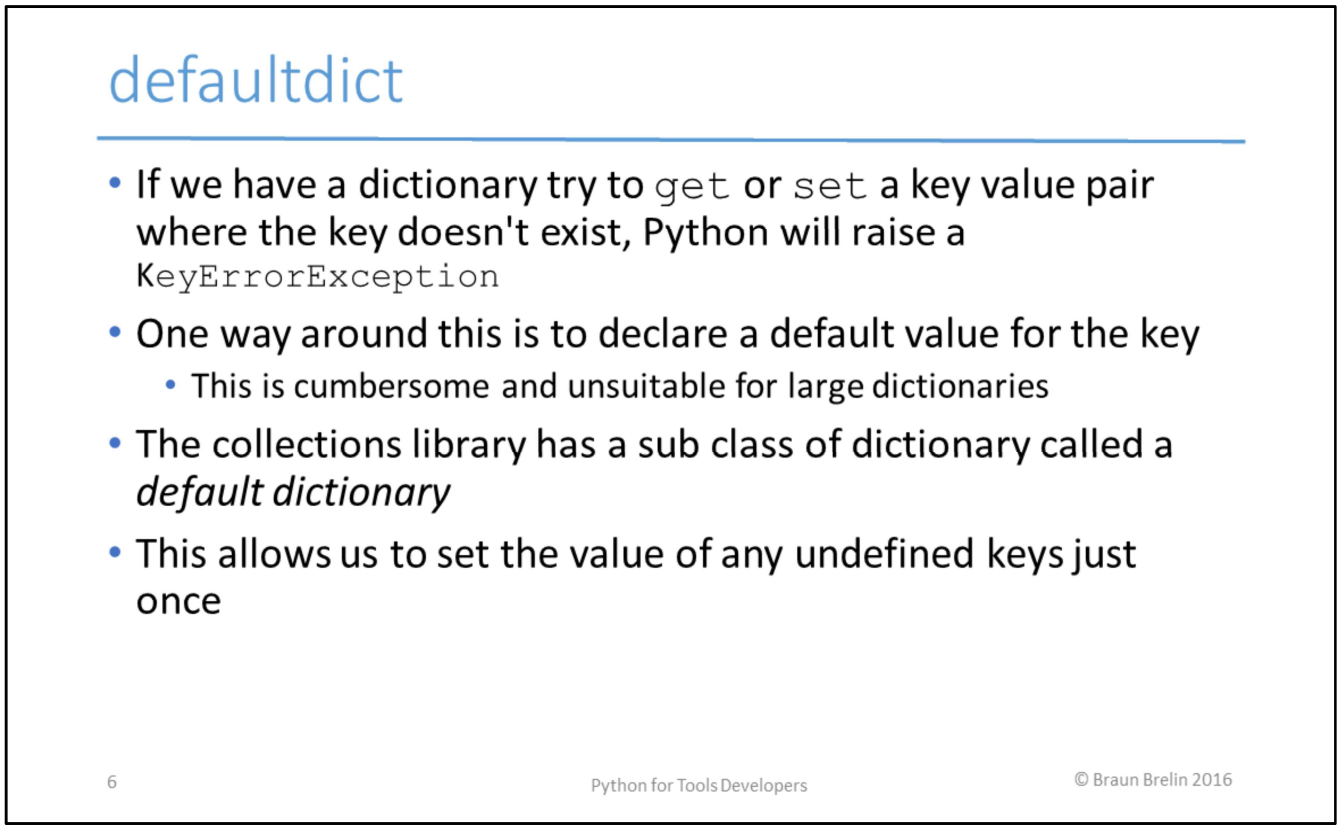
```
#!/usr/bin/python3

from collections import Counter

wordList =
['foo','foo','bar','bar','baz','blech','foo','foo','bar','meh','feh','feh'
]

wordCount = Counter(wordList)
# Print the count of key values
print (wordCount)
# Print the top two keys
print (wordCount.most_common(2))
```

The output of this program is:

Counter({'foo': 4, 'bar': 3, 'feh': 2, 'baz': 1, 'blech': 1, 'meh': 1})

[('foo', 4), ('bar', 3)]

As we can see, the `wordCount` variable contains a dictionary which has the elements of the `wordList` list as the keys and the number of times it finds each key in the list as its value. We can also call methods such as `most_common(n)` which gives back the `n` most common elements in the list. There are many use cases for using the `Counter` dictionary such as finding the most common word or words in a set (such as a book).



The defaultdict

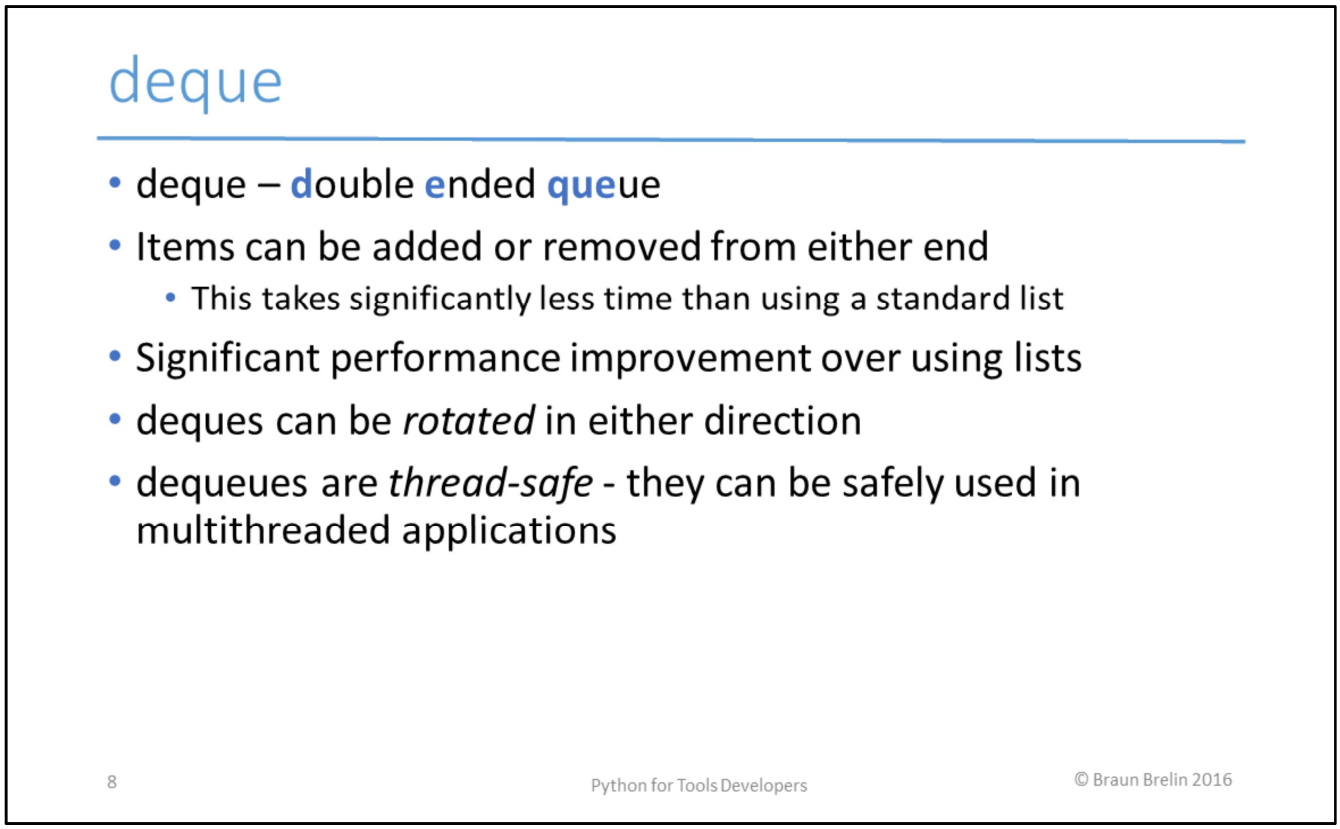
If we have a dictionary `d` and I try to `get` or `set` a key value pair where the key doesn't exist, Python will raise a `KeyErrorException`. This may not be the behaviour you desire.

One way around this is to declare a default value for the key. For example, if we have an empty dictionary `d` and I want to print a default value for the key 'foo' so that it doesn't throw an exception, we can do the following : `d.setdefault['foo',0].`

Now, every time we try to print `d['foo'],` if the `'foo'` key hasn't been set for `d`, then it will print a `0`. However, this method is somewhat cumbersome, for a number of reasons that we will see in the next section. For one thing, we'd have to call the `setdefault()` method for every key in the dictionary. For large dictionaries this becomes unsuitable.

However, the collections library allows us to have a sub class of dictionary called a default dictionary. This allows us to set the value of any undefined keys just once. For example:

```
#!/usr/bin/python3
 from collections import defaultdict

d = {}

# Here we use the setdefault method from the python dictionary
to
# set the default value for the key 'foo' in dictionary d to 0.
# However, the problem here is that we need to do this for every
key
# that we expect to be able to insert if we don't want to get a
# KeyErrorException thrown when accessing d['foo'] if it hasn't
# already got a value.

d.setdefault('foo',0)
print (d['foo'])

# Better way.  Make d a defaultdict.  Note that we have to pass
into
# the constructor a callable object.  You can look in the Python
# documentation for built in functions to find the callable
built in
# objects (or create and pass in your own callable object).
Passing
# an int object automatically means that the defaultdict will
set a value
# of 0 for any undefined key/value pairs.

d = defaultdict(int)

# Now, everytime we try to print out a key that doesn't exist in
the
# dictionary, we get a 0 printed instead of an exception thrown.

print (d['foo'])
```



The deque

A `deque` (pronounced 'deck') is a short hand term for a double ended queue.

In a double ended queue, items can be added to or removed from either end of the queue. The main advantage of a `deque` is that the time it takes to insert or delete values from either end of the queue is significantly less than using a standard Python list. Here's an example of this:

```
In [1]: from collections import deque

In [2]: s=range(10000)  <-- Setup a list with 10,000 elements

In [3]: d=deque(s)  <-- Create a deque using the list as input

In [4]: s_append,s_pop= s.append, s.pop

In [5]: d_append, d_pop = d.append, d.pop

In [6]: %timeit s_pop(); s_append(None)
10000000 loops, best of 3: 113 ns per loop  <-- Timings using a
standard python list

In [7]: %timeit d_pop();d_append(None)
10000000 loops, best of 3: 88.3 ns per loop  <-- Timings using a
deque
```

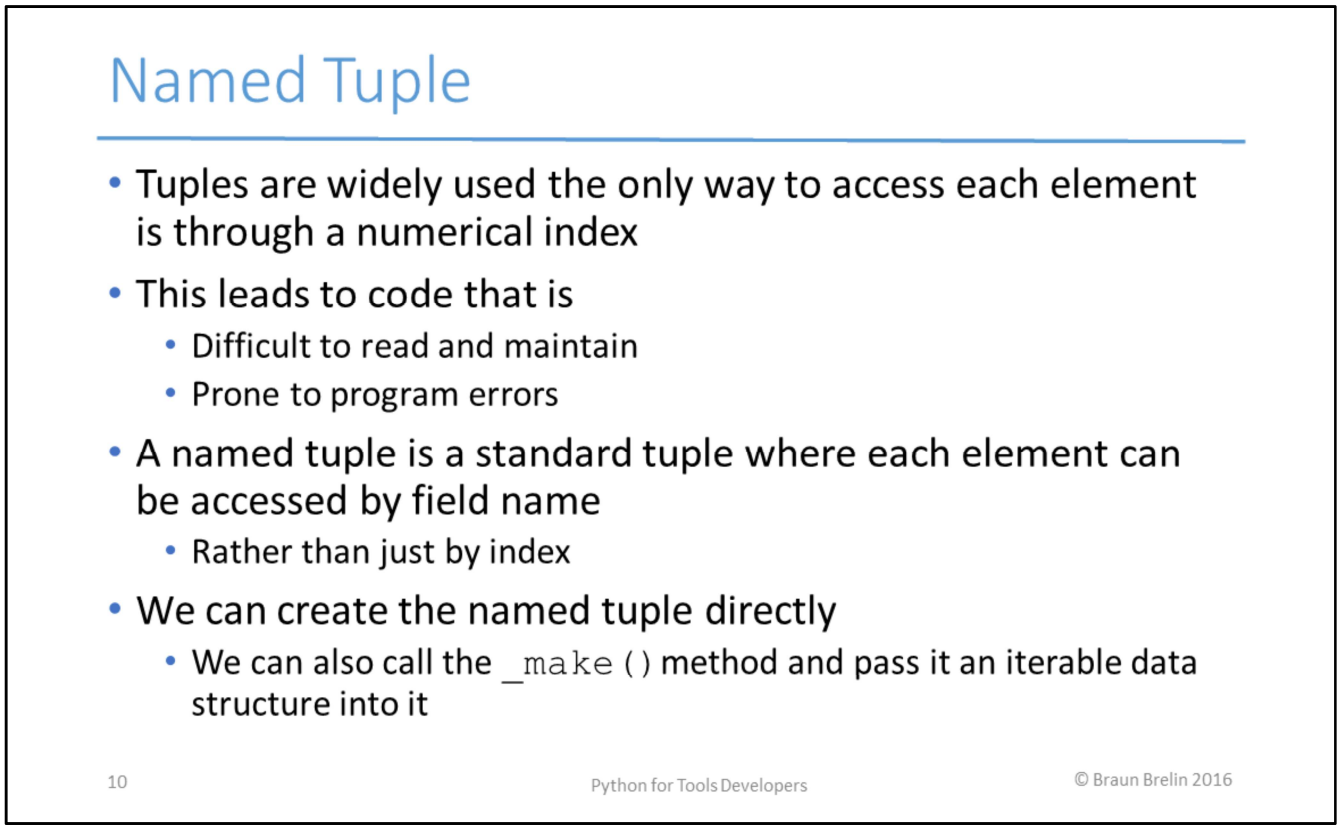
Note that there is a performance improvement of greater than 20% for using deques rather than lists when creating data structures where you consistently add or delete elements from either end. Another singular advantage of deques over lists is the ability to *rotate* them in either direction.

For example:

```
In [1]: from collections import deque

In [2]: dq = deque([1,2,3,4,5])

In [3]: dq.rotate(2)

In [4]: print (dq)
deque([4, 5, 1, 2, 3])

In [5]: dq.rotate(-3)

In [6]: print (dq)
deque([2, 3, 4, 5, 1])
```

Note that providing a positive number to the `rotate()` method rotates the deque to the right whereas a negative number rotates the deque to the left.

Additionally dequeues are *thread-safe* which means that they can be safely used in multithreaded applications.



The named tuple

Tuples are one of the most widely used data structures available in Python. While tuples are useful for many applications, they do have one drawback. In the case of tuples with many elements, the only way to access each element is through a numerical index. This can lead to code that is difficult to read and maintain and can lead to programmatic errors in extreme cases. The collections library in Python offers a class called a *named tuple*. Simply put, this is a standard tuple in which each element can now be accessed by field name rather than just by index. Let's see an example of this.

```
#!/usr/bin/python3

from collections import namedtuple
# Here we create a standard tuple of personnel information that
contains
# the following fields, first name, last name, Employee ID and
Address

standardtuple = ('Braun','Brelin','12345','1234 Main Street')

# With a standard tuple, if I want to access the first and last
name, I need to
# specify it with the index value of the tuple, i.e. first name
is index
# position 0, last name is index position 1.

print (standardtuple[0] + " "+ standardtuple[1])
```

Let's see how to do the same thing with a named tuple.

```
# We use the named tuple factory to create a named tuple type
called 'Person'
# The first argument is the named tuple's name, the second
argument is the
# list of fields
Person = namedtuple('Person','firstname lastname ID Address')

# Now we can use this named tuple type to create Persons.  Such
as:

Braun = Person('Braun','Brelin','12345','1234 Main Street')

# Note here that we can now access elements by field name,
rather than
# index position
print (Braun.firstname + " " + Braun.lastname)

# We can create a person object by using the _make method and
passing some
# sort of iterable data structure such as a list.

Bob=Person('Bob','Bird','12346','2345 Main Street')
print (Bob.firstname + " " + Bob.lastname)
```

Note that not only can we create the named tuple directly, but we can also call the `_make()` method and pass an iterable data structure such as a list into it and create the tuple that way. There are a number of other methods contained in the Python documentation which lists some of the other methods available for a named tuple.

There are a number of other containers available in the collections library, however, the ones discussed above are the most useful and commonly used data structures in the Python environment.