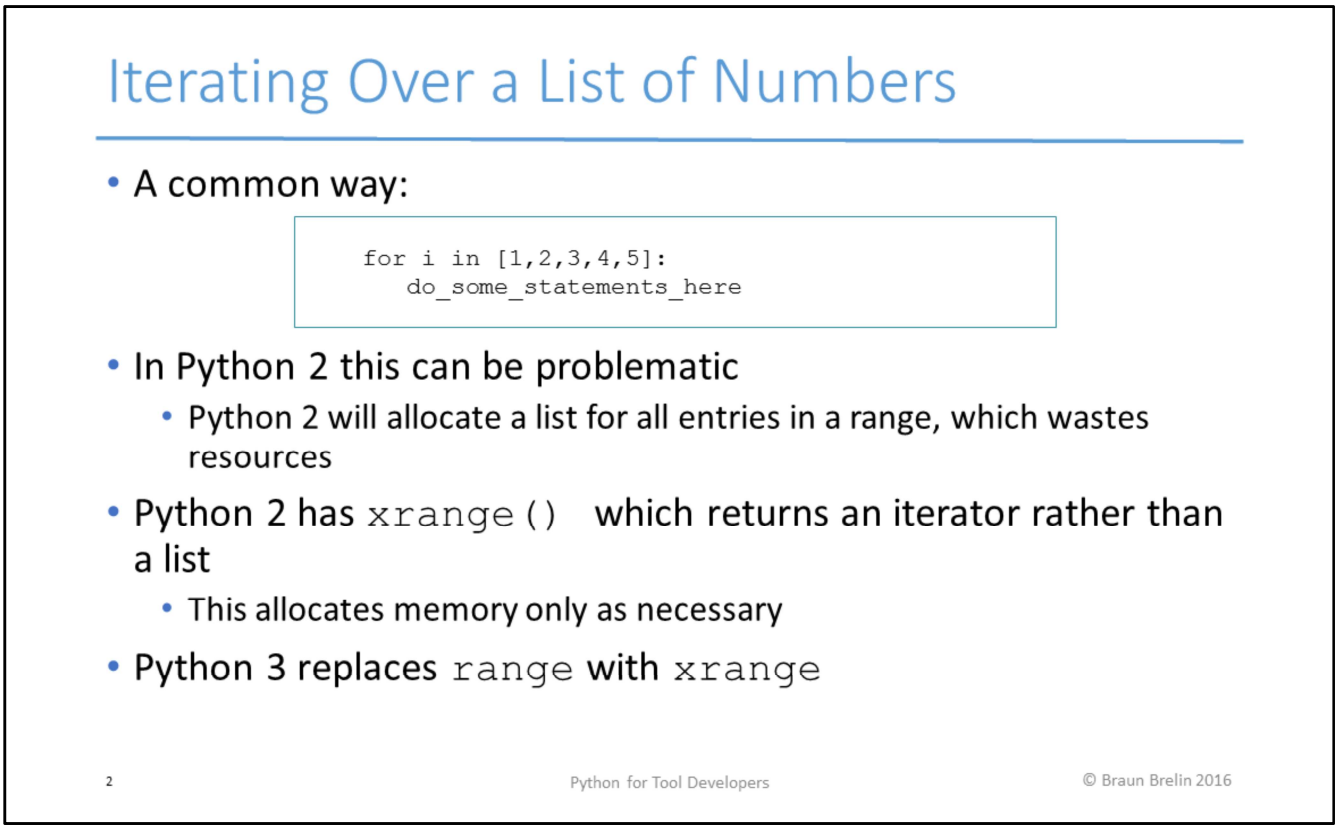
Module 5

Writing Idiomatic Python

Note. This chapter was inspired by a talk given by Raymond Hettinger, a Python core developer, at the 2013 Pycon conference. The talk can be found here on YouTube. It is highly recommended that students watch this video in order to increase their knowledge of Python.

This purpose of this chapter is to give students some guidance on the 'proper' way to write Python code. Unlike languages such as Perl, whose design philosophy is 'there is more than one way to do things', Python assumes that there is a proper 'Pythonistic' way to write your code. Herein are some tips students should know when writing Python applications.



Iterating over a list of numbers

A common way to iterate over a list is as follows:

```
for i in [1,2,3,4,5]:
   do_some_statements_here
```

Alternatively, in Python 2, you could do the following:

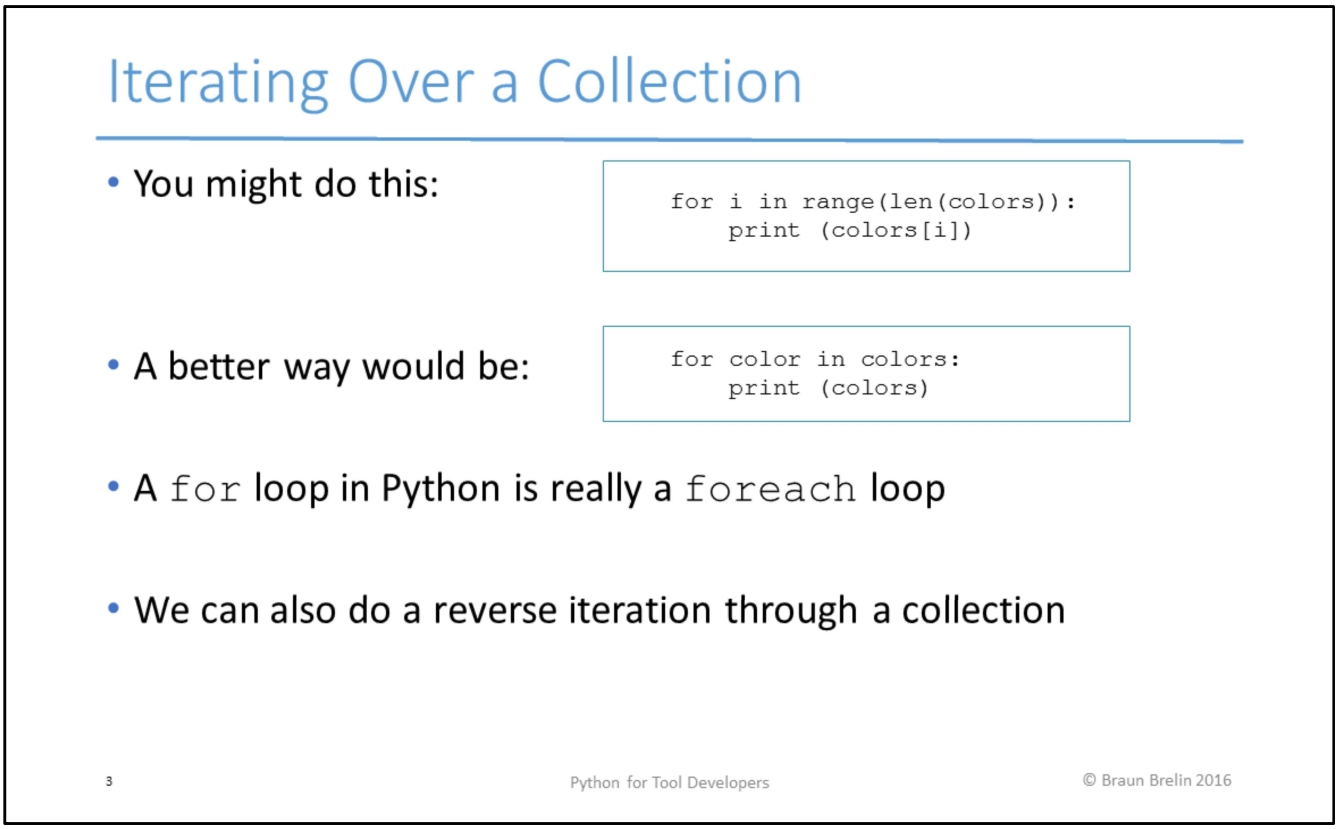
```
For i in range(1,6):

  do_some_statements_here
```

However, in Python 2 the second statement can be problematic.

If I `do` a `range` between 1 and one million, Python 2 will allocate a list for a million entries. This can take up to 32 Megabytes on a 64 bit machine, with a corresponding waste of resources. Python 2 offers the `xrange()` function, which returns an iterator rather the actual list. In this instance, `xrange` will only allocate memory as necessary rather than all at once.

Python 3 replaces the `range` function with the `xrange` function (but still calls it `range`).



Iterating over a collection

Given a collection such as the following:

```
colors = ['red','green','blue','yellow','purple']
```

How would we iterate over this? People who come from the C/C++ world might do something like this:

```
for i in range(len(colors)):
    print (colors[i])
```

The better way to iterate over a collection would be to do this:

```
for color in colors:
    print (colors)
```

Note that the `for` loop in Python is really a `foreach` loop. Although you can emulate a numeric `for` in the style of C or Java, it's not the 'Pythonic' way to do things. Also, using the `timeit` option in the Python REPL shows that iterating over a list using the second method is significantly faster.

Iterating Over a Collection

- A reverse iteration through a collection – a Pythonic approach

```
for color in reversed(colors):
    print (color)
```

- A Pythonic way to loop over the index and collection

```
for i,color in enumerate(colors):
    print (i , " = ",color)
```

- The enumerate function returns an enumerate object
  - It can be converted into an iterable

Similarly we can also do a reverse iteration through a collection. The non-Pythonic way to do this might be as follows:

```
colors = ["red","green","blue","yellow","purple"]
    for i in range(len(colors)-1, -1, -1):
        print (colors[i])
```

A better approach would be to do the following:

```
for color in reversed(colors):
    print (color)
```

Again, this is not only more readable, it executes faster as well.

Another common technique is to loop over the index and the collection like so:

```
colors = ['red','green','blue','yellow','purple']
for i in range(len(colors)):
    print (i, " = " , colors[i])
```

A better, more Pythonic way to do this:

```
for i,color in enumerate(colors):
    print (i , " = ",color)
```

The `enumerate builtin` function returns an `enumerate` object, which can be converted into an iterable such as a list of tuples of which the first element in the tuple is the iterable index position and the second element is the actual iterable element.

Looping over Multiple Collections

- There are a number of ways to do this, of varying efficiency

1. Emulating a numeric `for` loop
   - Least efficient and non-Pythonic
2. Using the `zip()` `builtin` function
   - Less inefficient but still creates problems
   - This only applies to Python 2 – Python 3 uses `izip`
3. Using `izip` instead of `zip`
   - Most efficient
   - This applies to Python 2 – in Python 3 the `zip` function now returns an iterator instead of a list

5 Python for Tool Developers

Looping over multiple collections

When looping over multiple conditions, there are a number of ways to do this in Python. Here we present three options ranging from the least to the most efficient.

```
colors = ['red', 'green','blue','yellow','purple']
names = ['Braun','Guido','Bob','Sue']


# Option 1 -  Least efficient

n = min(len(names),len(colors)) # Find the smaller length of
the two lists.
for i in range(n):
    print (names[i],” = “, colors[i])

# Option 2 - Less inefficient

for name,color in zip(colors,names):
    print (name, “ = “ , color)

# Option 3 - Most efficient

for name,color in izip(colors,names):
    print (name, “ = “, color)
```

Let's examine the three options.

Option 1 is the non-Pythonic way to do it. It's again emulating a numeric `for` loop and looping over the smallest list and mapping it to the larger list.

Option 2 uses the `zip() builtin` function. `zip` takes the first element of the first list and the first element of the second list and builds a third list containing, as each element, a `tuple` of the first and second list elements. It then repeats this for every element until it reaches the end of the shorter of the two lists.

While this is better than option 1, `zip` still has a problem. It creates, in memory, a third collection which is larger than the two source collections combined. For large collections, this can mean that you suffer CPU cache misses which can cause significant performance slowdowns. Note that this only applies to Python 2; the Python 3 version replaces `zip` with `izip`.

Option 3 is the optimal pattern. Instead of using `zip`, it uses `izip`, which creates an iterator object rather than pre-allocating memory to the target container itself. Again, in Python 3, the `zip` function now returns an iterator instead of a list.

The advantage here is that only the memory actually needed in order to hold the container information is allocated at run-time.

Sorting Container Elements with Python

- The easiest way to loop over an unordered container in sorted order is to use a `for` loop
- Sometimes you want to create a custom comparator
  - Traditionally - create a comparator function and pass this to the sorted function
  - This is idiomatic and slow
- The better way to do this is to use a `key` function:

```
colors = ['red','blue','green','yellow','purple']

print sorted(colors,key=len)
```

- A key function is one that takes one parameter (usually the container element) and returns a value to base the sorting on

7 Python for Tool Developers

Sorting container elements with Python

The easiest way to loop over an unordered container in sorted order is the following:

```
colors = ['red','green','blue','yellow','purple']


for color in sorted(colors):
    print (color)

# Sorts in reverse order.

for color in sorted(colors, reverse = True):
    print (color)
```

There are times, however, when you want to create a custom comparator in order to perform container sorts. The traditional way to do this was to create a comparator function and pass this to the sorted function like this:

```
colors = ['red','blue','green','yellow','purple']

def compare_length(c1,c2):
    if len(c1) < len(c2):
        return -1
    if len(c1) > len(c2):
        return 1
    return 0

print sorted(colors,cmp=compare_length)
```

This is not only not idiomatic, it is slow. For any container, the number of calls to the `compare_length` function is (n * logn). In other words, for a container with a million elements, the log base 2 of 1000000 is 20. Therefore 20 million calls are made to the `compare_length()` function.

The better way to do this is to use a `key` function like so:

```
colors = ['red','blue','green','yellow','purple']

print sorted(colors,key=len)
```

A `key` function is one that takes one parameter (usually the container element) and returns a value to base the sorting on.

In this scenario, the key function `len()` only gets called once for each element so it is significantly faster than a custom comparator.

Remember that `len` is a `builtin` function in Python that returns the length of an item such as a `string`, a `tuple` or a `list`. In Python 3 the custom comparator component has been removed from the sorted `builtin` function.

Using the `iter builtin` Function

- A common pattern in Python is to iterate over something until a *sentinel* value is reached
- Here's a better way

```
from functools import partial

buffer= []
for block in iter(partial(f.read,32), '')
    buffer.append(block)
```

- The `iter()` `builtin` function behaves differently depending on how many parameters are passed to it

9 Python for Tool Developers

Using the `iter builtin` function to iterate over non-iterable objects

A common pattern in Python is to iterate over something until a *sentinel* value is reached. Here is an example of this:

```
buffer = []
while True:
    block = f.read(32)  <-- f is defined earlier as the return
value from an open()
    if block = ' ':
        break
    buffer.append(block)
```

In this code snippet, we iterate over a file object and read data from it until we come to the end of the file.

Here is a better way to do it :

```
from functools import partial

buffer= []
for block in iter(partial(f.read,32), '')
    buffer.append(block)
```

Let's analyse this code.

The `iter() builtin` function behaves differently depending on how many parameters are passed to it.

If only one parameter is passed, then that parameter must be an iterable object. That is, it must have a `__next__()` and a `__hasNext__()` method defined for that container.

However, if two parameters are passed to it, the first parameter must be callable. In other words, the passed object must implement the `__call()` method. The second parameter is the sentinel value, so that when this value is returned from the call, the `iter` method will stop.

The partial function is imported from the `functools` library - it creates a callable object that, when called, will call the `f.read()` method. Note that this code is substantially shorter than the first method. It is also faster.

Defining multiple exits from loops using the `else` clause

- A common pattern with two possible exits from the loop
- We would then like to do some operation or operations
  - Depending on whether we exited the loop normally
  - Or via the `break` statement
- The old way of doing this is to set some sort of a flag value, in this case `foundRed`
- Python has a much cleaner way

```
colors =
['red','green','blue','yellow','purple']
foundRed == False
for color in colors:
    if color == 'red':
        foundRed = True
        break
if not foundRed:
    print ("There is no color red in the list")
```

```
colors =
['red','green','blue','yellow','purple']
for color in colors:
    if color == 'red':
        break
else:
    print ("There is no color red in the list")
```

11 Python for Tool Developers

Defining multiple exits from loops using the `else` clause

Let's take a look at a common pattern in programming:

```
colors = ['red','green','blue','yellow','purple']
foundRed == False
for color in colors:
    if color == 'red':
        foundRed = True
        break
if not foundRed:
    print ("There is no color red in the list")
```

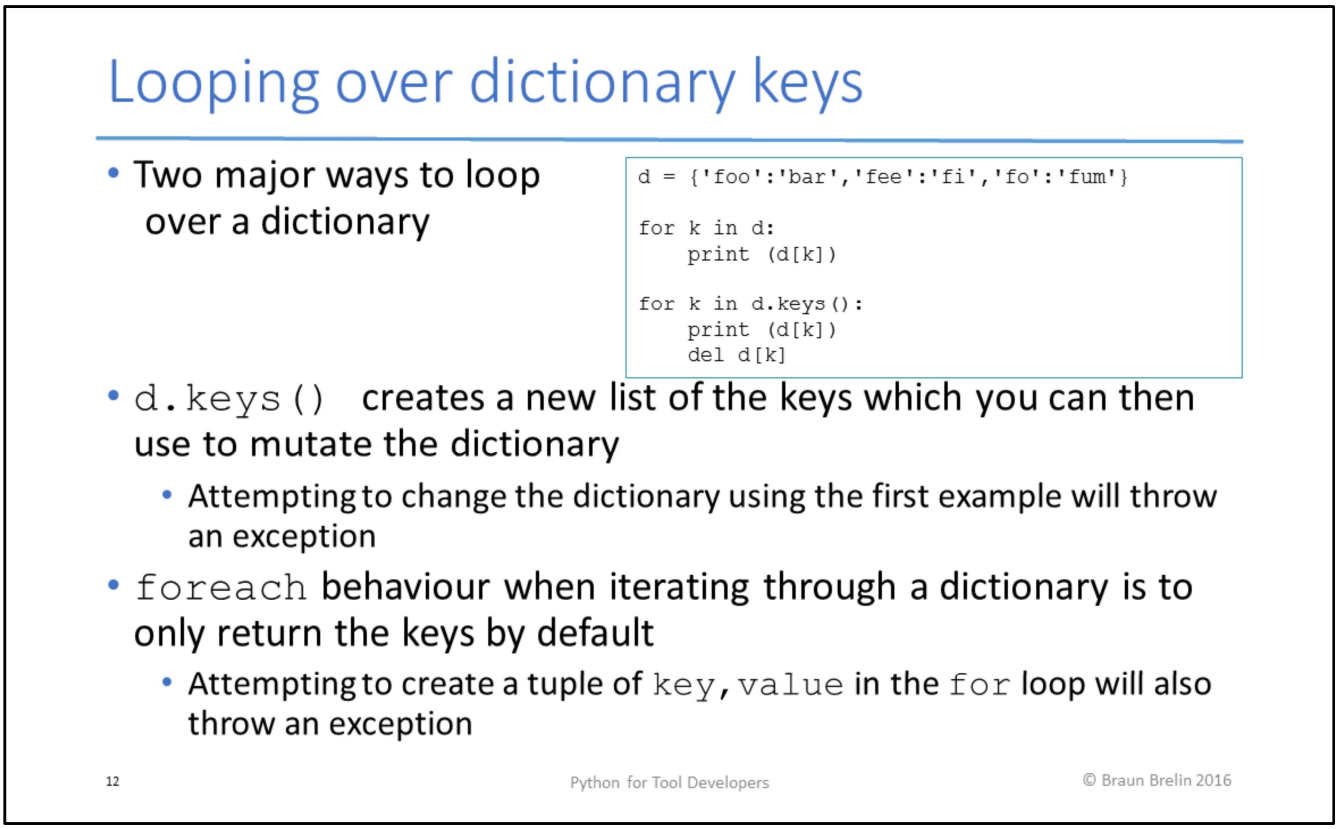
In this case, we notice that there are two possible exits from the loop.

- In case one, we find an element called 'red' and immediately exit from the loop
- In case two, we don't find a 'red' element and exit from the loop normally

We would then like to do some operation or operations depending on whether we exited the loop normally or via the `break` statement. The old way of doing this is to set some sort of a flag value, in this case `foundRed`. We set the `foundRed` flag to `true` if we found a 'red' element and then break. Python, however, gives us a much cleaner way to do this:

```
colors = ['red','green','blue','yellow','purple']
for color in colors:
    if color == 'red':
        break
else:
    print ("There is no color red in the list")
```

Notice the `else` clause attached to the `for` loop. The `else` clause will only run if the loop has exited normally. It will not run if we broke out of the loop for any reason.



Looping over dictionary keys

There are two major ways to loop over a dictionary:

```
d = {'foo':'bar','fee':'fi','fo':'fum'}

for k in d:
    print (d[k])

for k in d.keys():
    print (d[k])
    del d[k]
```

What's the difference between the two? In the second example, `d.keys()` creates a new list of the keys which you can then use to mutate (i.e. change) the dictionary. Attempting to change the dictionary using the first example will throw an exception.

Note also that the `foreach` behaviour when iterating through a dictionary is to only return the keys by default. Attempting to create a tuple of `key,value` in the `for` loop will also throw an exception.

Let's see how to get both the keys and values from a dictionary when looping over it.

```
# First way
for k in d:
    print (k, "=",d[k])

# Second way
for k,v in d.items():
    print (k, "=",v)
```

Which way is better? Clearly the second way is faster - in the first example the dictionary hash has to be recalculated for every item in the dictionary (the `d[k]` operation).

In the second way, `d.items()` creates an iterator which is only created once. We can then iterate over the object.

Note that in Python2, the `items()` method created a new list, which, although better than the first way, still created a list in memory. To emulate the iterator in Python 2, use `d.iteritems()`. Python 3 creates iterators by default.

Constructing a dictionary from a pair of lists

- How to construct a dictionary from a pair of lists
  - One list contains the keys and the other contains the values

```
keys = ['fee','fi','fo','fum']
values = ['blood','of','an','Englishman']

d = dict(zip(keys,values))
```

- We can create a dictionary using the `zip` function which returns an iterator of tuples
- This is highly efficient
  - The iterator only has to allocate space for one tuple which is re-used for each operation
  - In Python 2, you would use the `izip builtin` function

14 Python for Tool Developers

Constructing a dictionary from a pair of lists

What is the best way to construct a dictionary from a pair of lists where one list contains the keys and the other one contains the values?

```
keys = ['fee','fi','fo','fum']
values = ['blood','of','an','Englishman']

d = dict(zip(keys,values))
```

Here, we see that we can create a dictionary using the `zip` function which returns an iterator of tuples.

Note that this is highly efficient because the iterator only has to allocate space for one tuple which is re-used for each operation rather than allocating space for a new tuple each time. In Python 2, you would use the `izip builtin` function.

Using a dictionary to count the number of elements

- Given a list of potentially repeatable values, we need to count the number of occurrences of any value
- Let's look at three ways to do this:

```
# First way
d = {}
for number in mynumbers:
    if number not in d:
        d[number] = 0
    d[number] +=1

# Second way
d={}
for number in numbers:
    d[number]= d.get(number,0)
+1

# Third way
d = defaultdict(int)
for number in mynumbers:
    d[number] +=1
```

15 Python for Tool Developers

Using a dictionary to count the number of elements

Let's look at a number of ways to perform this function. Often times we will be given a list of potentially repeatable values and we will need to count the number of times that we receive any arbitrary value. For example if we have a list of values:

```
mynumbers = [1,2,4,2,3,6,4,3,3,3,7,8,6,9,9,0,0,1]
```

How do we count the number of "1's", "2's", etc.. Let's look at three ways to do this:

```
# First way
d = {}
for number in mynumbers:
    if number not in d:
        d[number] = 0
    d[number] +=1

# Second way
d={}
for number in numbers:
    d[number]= d.get(number,0) +1

# Third way
d = defaultdict(int)
for number in mynumbers:
    d[number] +=1
```

The first method for counting items in a dictionary is the standard, default way of doing it.

Very simply, the program checks to see if the key exists. If it doesn't it creates a key with a value of `0` and then increments it, the whole point being that we want to avoid raising a `KeyErrorException` and halting program execution.

The second way effectively does the same thing as the first method; however, here we use the `get()` method to either retrieve the value if it exists or a zero if it doesn't. `get` will then create the key with a value of zero in the dictionary.

The last method is to use the `defaultdict` class which is described in the previous module. Note that there may be some use cases where we will have to convert the `defaultdict` back into a regular `dict` with the `dict()` built in function.

Grouping with dictionaries

- A common programming pattern:
  - Take a list of values
  - Put them into a dictionary by some characteristic of the value
  - e.g. the first character of the string value or its length
- Here is the naive, first implementation of this pattern

```
names = ['Braun','Bob','Sue','Dave','Ben','Mark','Rory']

d = {}
for name in names:
    key = len(name)
    if key not in d:
        d[key] = []
    d[key].append(name)

{3: ['Bob', 'Sue', 'Ben'], 4: ['Dave', 'Mark', 'Rory'], 5: ['Braun']}
```

17 Python for Tool Developers

Grouping with dictionaries

A common programming pattern is to take a list of values and put them into a dictionary by some characteristic of the value, e.g. the first character of the string value or its length.

Here is the naive, first implementation of this pattern.

```
names = ['Braun','Bob','Sue','Dave','Ben','Mark','Rory']

d = {}
for name in names:
    key = len(name)
    if key not in d:
        d[key] = []
    d[key].append(name)

{3: ['Bob', 'Sue', 'Ben'], 4: ['Dave', 'Mark', 'Rory'], 5:
['Braun']}
```

In this first implementation, we create an empty dictionary `d`. We iterate through the names list and make the length of each element of the list a key for the dictionary. If the key/value pair doesn't exist, create the key and initialize the value as an empty list. Once that's done, we append the list element as a new element of the value list. Note that in order to change the grouping criteria here, we only need to change one line of code, in this case the key assignment.

Grouping with dictionaries

- A nicer way of implementing this pattern

```
names = ['Braun','Bob','Sue','Dave','Ben','Mark','Rory']
d = {}
for name in names:
    key = len(name)
    d.setdefault(key,[]).append(name)
```

- The up-to-date modern way using `defaultdict` to implement grouping by dictionary

```
names = ['Braun','Bob','Sue','Dave','Ben','Mark','Rory']
d=defaultdict(list)
for name in names:
    key = len(name)
    d[key].append(name)
```

18 Python for Tool Developers

Let's see a nicer way of implementing this pattern.

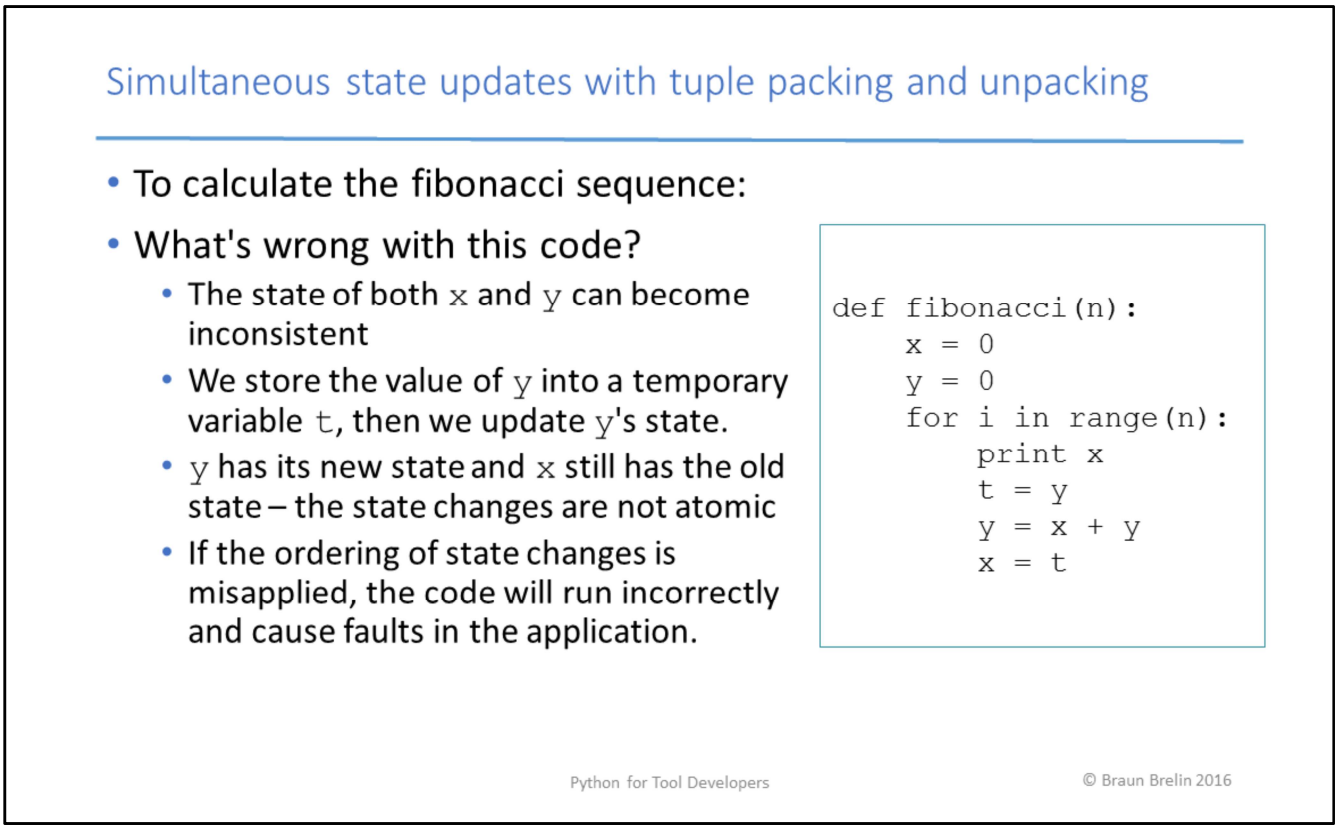
```
names = ['Braun','Bob','Sue','Dave','Ben','Mark','Rory']
d = {}
for name in names:
    key = len(name)
    d.setdefault(key,[]).append(name)
```

Here we use the `setdefault()` method to add the key value pair. If the key doesn't exist, the `setdefault` method will create an empty list as the value and add the key/value pair to the dictionary. Once that's done, we then append the new name element to the value list.

Let's see the up-to-date modern way using `defaultdict` to implement grouping by dictionary.

```
names = ['Braun','Bob','Sue','Dave','Ben','Mark','Rory']
d=defaultdict(list)
for name in names:
    key = len(name)
    d[key].append(name)
```

As we saw in our last example, we can use a `defaultdict` from the collections library to implement these patterns. We pass in a list parameter to the `defaultdict` factory to tell it that for this `defaultdict`, if the key/value pair doesn't exist create an empty list as the value and insert the key/value pair into the `defaultdict`.



Simultaneous state updates with `tuple` packing and unpacking

Consider the following code to calculate the fibonacci sequence:

```
def fibonacci(n):
    x = 0
    y = 0
    for i in range(n):
        print x
        t = y
        y = x + y
        x = t
```

What's wrong with this code fragment? For one thing, the state of both `x` and `y` can become inconsistent. Here we store the value of `y` into a temporary variable `t`, then we update `y`'s state. Now, however, `y` has its new state and `x` still has the old state. In other words, the state changes are not atomic.

Additionally, if the ordering of state changes is misapplied, then the code will run incorrectly and cause faults in the application.

Simultaneous state updates with tuple packing and unpacking

- What's a better way to write this?

```
def fibonacci(n):
x,y=0,1
for i in range(n):
   x,y = y, x + y
```

- Instead of having to break down the transformation into details, we simply say that we want to update `x` and `y` according to the relevant equations
- Additionally, state changes are now atomic

20 Python for Tool Developers

What's a better way to write this?

```
def fibonacci(n):
x,y=0,1
for i in range(n):
   x,y = y, x + y
```

Now we note that instead of having to break down the transformation into details, we simply say that we want to update `x` and `y` according to the relevant equations. Additionally, state changes are now atomic.

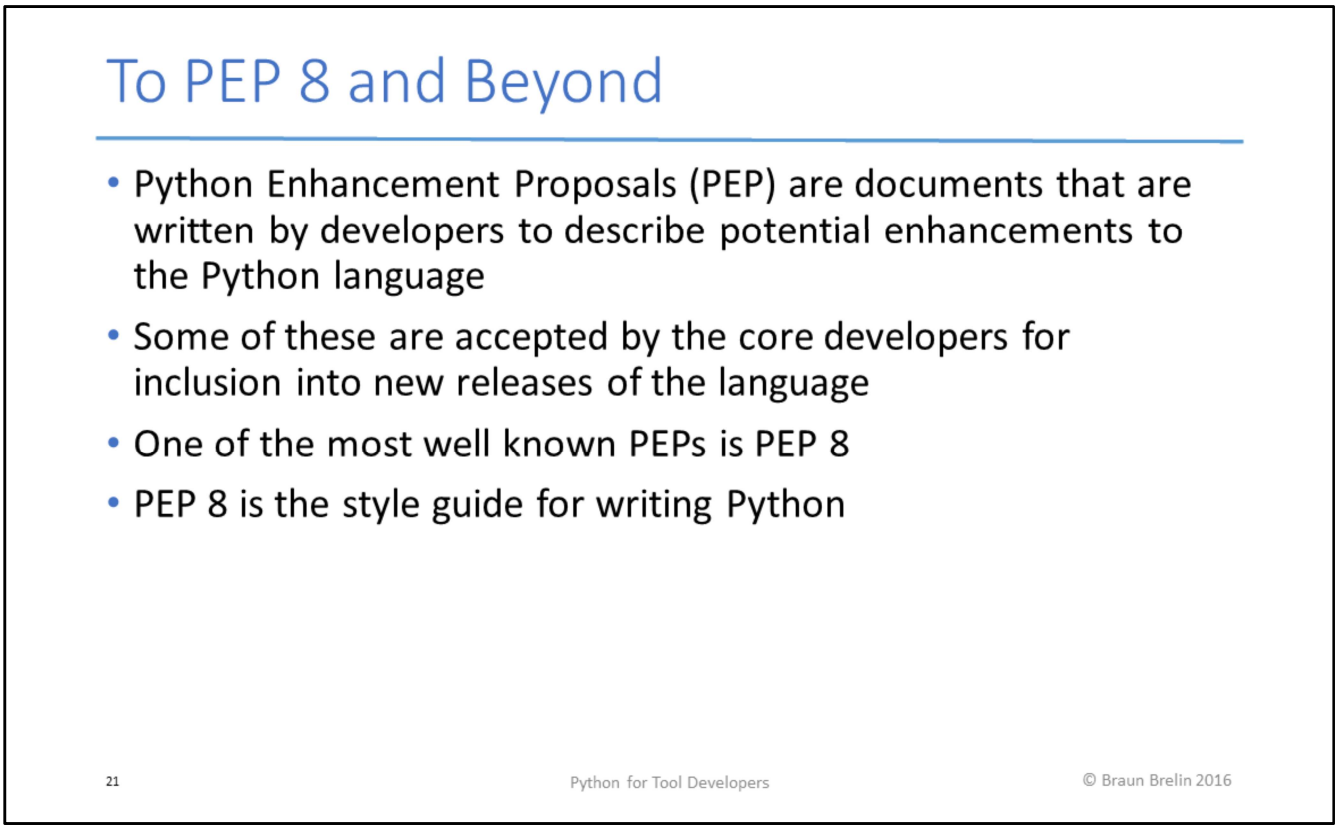
Finally, some tips on writing better Python code.

1. Don't put too much code on one line
2. Don't break atoms of thought into subatomic particles

While these rules sound conflicting, they're really not.

Write your Python code logic as if you're writing it in English. As we saw with the fibonacci example, don't break down the code into individual steps; this becomes difficult to code and maintain. Additionally, you are no longer thinking in “higher order” terms but rather are bogged down in details.

Additionally, don't try and pack multiple thoughts into one line of code, as that also causes maintainability problems.



To PEP 8 and beyond

Python Enhancement Proposals (PEP) are documents that are written by developers to describe potential enhancements to the Python language. Some of these are accepted by the core developers for inclusion into new releases of the language.

One of the most well known PEPs is PEP 8. PEP 8 is the style guide for writing Python. While detailed analysis of this document is best left to the student, let's see how we can use PEP 8 to clean up some code and even go beyond the style guide.

To PEP 8 and Beyond

- This program connects to a network routing device
- Iterates over its routing table
- Prints out the route name and IP address as a report

```
import jnettool.tools.elements.NetworkElement, \
       jnettool.tools.Routing, \
       jnettool.tools.RouteInspector

ne=jnettool.tools.elements.NetworkElement( '171.0.2.45' )
try:
    routing_table=ne.getRoutingTable() # fetch routing table

except jnettool.tools.elements.MissingVar:
# Record table fault
    logging.exception('''No routing table found ''')
# Undo partial changes
    ne.cleanup('''rollback''')
else:
    num_routes-routing_table.getSize() # determine table size
    for RToffset in range( num_routes ) :
        route=routing_table.getRouteByIndex(RToffset)
        name=route.getName()  # route name
        ipaddr = route.getIPAddr() # ip address
        print ("%15s -> %s" % *name,ipaddr) # format nicely

finally:
    ns.cleanup( '''commit''' ) #lock in changes
    ns.disconnect()
```


Let's consider the following program:

```
import jnettool.tools.elements.NetworkElement, \
       jnettool.tools.Routing, \
       jnettool.tools.RouteInspector

ne=jnettool.tools.elements.NetworkElement( '171.0.2.45' )
try:
    routing_table=ne.getRoutingTable() # fetch routing table

except jnettool.tools.elements.MissingVar:
# Record table fault
    logging.exception('''No routing table found ''')
# Undo partial changes
    ne.cleanup('''rollback''')

else:
    num_routes-routing_table.getSize() # determine table size
    for RToffset in range( num_routes ) :
        route=routing_table.getRouteByIndex(RToffset)
        name=route.getName()  # route name
        ipaddr = route.getIPAddr() # ip address
        print ("%15s -> %s" % *name,ipaddr) # format nicely

finally:
    ns.cleanup( '''commit''' ) #lock in changes
    ns.disconnect()
```

This program connects to a network routing device, iterates over its routing table and prints out the route name and IP address as a report. Note that the library was written in Java and we're using Python to connect to it.

To PEP 8 and Beyond

- Let's look at the next version of this with PEP 8 applied

```
#!/usr/bin/python3

# Put the import statements on separate lines for
readability
# Also, it's nice to alphabetize them if you have
a lot
# of imports.
import jnettool.tools.elements.NetworkElement
import jnettool.tools.Routing
import jnettool.tools.RouteInspector

# Put spaces around the '=' sign.
# Also, delete the space between the parameter and
the parenthesis.
ne =
jnettool.tools.elements.NetworkElement('171.0.2.45
')

# Tighten up the code by removing the spacing
between the try/except/else
# /finally blocks.  Also, get rid of useless
comments in the code.
# Get rid of the ''' double quotes.  Just use the
single quotes.
```

```
try:
    routing_table = ne.getRoutingTable()
except jnettool.tools.elements.MissingVar:
    logging.exception('No routing table found')
    ne.cleanup('''rollback''')
else:
    num_routes-routing_table.getSize()
    for RToffset in range(num_routes) :

route=routing_table.getRouteByIndex(RToffset)
        name=route.getName()
        ipaddr = route.getIPAddr()
# Put a space between function arguments.
        print ("%15s -> %s" % *name, ipaddr)
finally:
    ns.cleanup('commit')
    ns.disconnect()
```

23 Python for Tool Developers

How can we use PEP 8 to make this code better? Let's look at the next version of this with PEP 8 applied.

```
#!/usr/bin/python3

# Put the import statements on separate lines for readability
# Also, it's nice to alphabetize them if you have a lot
# of imports.
import jnettool.tools.elements.NetworkElement
import jnettool.tools.Routing
import jnettool.tools.RouteInspector

# Put spaces around the '=' sign.
# Also, delete the space between the parameter and the
parenthesis.
ne = jnettool.tools.elements.NetworkElement('171.0.2.45')

# Tighten up the code by removing the spacing between the
try/except/else
# /finally blocks.  Also, get rid of useless comments in the
code.
# Get rid of the ''' double quotes.  Just use the single
quotes.
```

Continued ....

```
try:
    routing_table = ne.getRoutingTable()
except jnettool.tools.elements.MissingVar:
    logging.exception('No routing table found')
    ne.cleanup('''rollback''')
else:
    num_routes-routing_table.getSize()
    for RToffset in range(num_routes) :
        route=routing_table.getRouteByIndex(RToffset)
        name=route.getName()
        ipaddr = route.getIPAddr()
# Put a space between function arguments.
        print ("%15s -> %s" % *name, ipaddr)
finally:
    ns.cleanup('commit')
    ns.disconnect()
```

This code now conforms to PEP 8 standards. It's much clearer and easier to read. However, this code, while PEP8 conformant, isn't really Pythonic. Let's now take a look at the next program.

```
#!/usr/bin/python3

from nettools import NetworkElement
with NetworkElement('171.0.2.45') as ne:
    for route in ne.routing_table:
        print ("%15S -> %s" % (route.name, route.ipaddr)
```

This code does exactly the same thing as the first two programs, however note how much shorter it is.

It does have one drawback, however. This code as it stands doesn't work. It doesn't work because the API library is written in Java and the Python bindings as written doesn't allow for Pythonic code. For example, it requires that the imports use dotted notation for packages. While this may be the correct way to do things in Java, in Python its use is contra-indicated. Packages are used to make sure that `namespace` collisions don't happen. If you don't have a very large code base, then it is pointless to create packages and sub-packages.

Also note that we're now abstracting away the setup and teardown code from the business logic of the program. The business logic is to print out a routing table with two columns, a route name and a route IP address. The setup and teardown code should be moved aside into its own module.

How do we then make this application more Pythonic?

Let's examine the options.

1. Any time we have reoccurring set up and tear down code, create a context manager so that we can use a `with` statement in Python to abstract away this code from the business logic.
2. Take multiple packages and combine them into a single module for ease of readability and maintenance.
3. Where we see code that uses a method to get the size of a list, loops over a range of elements and calls a function to get something by index, we do the following:
    a. Instead of calling a `getSize()` method, use the `builtin len()` function in Python.
    b. Instead of calling a `getValueByIndex()` method, use the '`[]`' operator in Python.
    c. In Python, anything where you can access an element by the `[]` operator and get its length via the `len()` method is known as a *sequence*. One of the properties of a sequence is that it is *iterable*. This means that we can loop directly over the sequence in Python.
    d. Anytime you see getter and setter methods in Python, replace them with properties or descriptors.
    e. Create custom exceptions to give clear and understandable error messages back to the clients.
    f. To generalize on point *C*, use the `builtin` methods in Python, such as `__len__` and `__getitem__` rather than custom methods to perform the same work.
    g. Create your own `__repr__` methods for debugging purposes.

Let's now look at an example of code that implements these options.

```
#!/usr/bin/python3

class NetworkElementError(Exeception):
    pass

class NetworkElement(object):

    def __init__(self,ipaddr):
        self.ipaddr = ipaddr
        self.oldne =
jnettool.tools.elements.NetworkElement(ipaddr)

    @property
    def routing_table(self):
        try:
            return RoutingTable(self.oldne.getRoutingTable())
        except jnettool.tools.element.MissingVar:
            raise NetworkElementError('No routing table found')

    def __enter__(self):
        return (self)

    def __exit__(self,exctype, excinst, exctb):
        if exctype == NetworkElementError:
            logging.exception('No routing table found')
            self.oldne.cleanup('rollback')
        else:
            self.oldne.cleanup('commit')
        self.oldne.disconnect()

    def __repr__(self):
        return ('%s(%r)' %  (self.__class__.__name__,
self.ipaddr)

class RoutingTable(object):

    def__init__(self,oldrt):
        self.oldrt = oldrt

    def __len__(self):
        return self.oldrt.getSize()

    def __getitem__(self, index):
        if index >= len(self):
            raise IndexError
        return Route(self,oldrt.getRouteByIndex(index))
```

Continued ....

```
class Route(object):

    def __init__(self, old_route):
        self.old_route = old_route

    @property
    def name(self):
        return self.old_route.getName()

    @property
    def ipaddr(self):
        return (self.old_route.getIPAddr()
```

This code is an implementation of the *Adapter* design pattern. The idea here is that we wrap the API in a new class that allows us to write the re-usable, Pythonic code shown in example 3 above.

We start with the creation of a `NetworkElementError` class which derives from the `BaseException` class. We do this mainly because we can now call our exception something useful.

We next create our `NetworkElement` class. The `__enter__()` and `__exit__()` methods are used to create a context manager. This is the set up and tear down code that is run when the `with` statement is executed. Note that the `__exit__` routine now defines the tear down logic. If an exception is passed then we do the logging and rollback, otherwise we do the commit. Finally, we disconnect from the router. Note that this code is now reusable.

Anytime we write Python code that opens the router, we merely have to use the `with` statement. We no longer have to re-write the setup/teardown routines in our business logic. We then define the Routing Table. Note that since we've defined the `__getitem__` and `__len__` methods ourselves, we can now directly iterate over the `ne.routingtable` iterable. Also note that each call to `__getitem__` returns a route object. Finally for each route object, instead of using getter methods for the name and the IP address, we can define properties for the attributes so that we can simply instead call `route.name` and `route.ipaddr` directly, even though under the covers we're actually calling the getter methods.

We've now created an adapter class that can be re-used everywhere it's needed. We've separated the business logic from the infrastructure logic so that the code is far cleaner, shorter, more readable and more maintainable.