Iterators:

This paper is written by Brady Auen, Paul Kirilchuk, and Kevin Vo for CSCI 3155.

Why Propose:

This proposal proposes an iteration interface that objects can offer to control the behavior of 'for' loops.

Reasons for Propose:

1. Gives an extensible iterator interface.
    - Previously there were pseudo iterators, but it's better to have a built in one so that we have an actual interface to work with rather than an abstract concept.
2. Performance improvements to list iteration.
    - With the new change, we can manipulate lists quicker and more directly.
3. Massive performance improvements to dictionary iteration.
    - Same general concept as virtue two.
4. Provides an interface for iterating without pretending to offer random access to elements.
    - We don't have to cite obj[1], we actually have a literal iterator that's going through the contents of our object.
5. Backward compatible with all existing user-defined classes and extension objects.
    - Bringing iterators in won't cause any problems among objects and classes that already exist, the iterators will be compatible with them.
6. Code iterating over non-sequence collections more brief and clear.
    - This is referring to iterating things that aren't sequences (think Lists), iterators make this process easier.

How it Works:

The iterator gives a **'get next value'** operation that makes the next item in the sequence each time it is called. The operation gives an exception when no more things are accessible. There is only one required method, next(), which takes no arguments and returns the next value. When no values are left to be returned, calling next() should give the StopIteration exception.

How it Works Examples:

Before this update there wasn't a clear way for the user to iterate through the contents of objects in Python. If the user wanted to create a "for item in object" sort of function, the method would look sort of like this:

```
def __getitem__(self, index):
    return <next item>
```

Even then, getitem methods were most commonly used to randomly access via indexing, so we could write **object[x]** to get the **x+1'th item**. Of course there are certain objects that can't be randomly accessed like this, so in 2.2 Python really advanced itself by allowing these getitem methods to be limited solely to classes that support it. We have two built in functions that we can use to get iterators, the first is:

```
iter(obj)
```

and the second is:

```
iter(C, sentinel)
```

The first method is rather simple, obj is just the name of some object and thus we're provided an iterator for that object. **iter(C, sentinel)** returns an iterator that will invoke C until "sentinel" is returned, in which case the iterator is finished, so sentinel is essentially just a stop case. Python classes can also define an **iter()** method which creates and returns an iterator for an object, if that object is an iterator then it will return itself. The purpose of iterators is in the name, they iterate through things. They have one required method called **next()**. Next() takes no arguments and only returns the next value, when there is no next value a **"StopIteration"** exception is called. A StopIteration except will continue to be called if next() is used again, it's an exception that signals the end of an iteration. If we were to make an iterator for a list type object, the code might look something this:

```
List = [1,2,3,4,5]
iterator = iter(List)
```

If we wrote "print iterator" at this point, we'd get the location of the iterator object in memory, but if we used next() like so: iterator.next()

The iterator would then hold 1, then 2, and so on.

Iterating in Dictionaries:

Iterators can also with in dictionaries with their keys. For those not familiar with python, dictionaries are similar to structs in C, so in coding a dictionary could look like this: dictionary = {'Age': 21, 'Weight': 200, 'Name': 'Brady'}

Inside a dictionary are pairs of "keys" and values that they correspond to, in this case our keys would be Age, Weight, and Name. So we're allowed to write lines of code like:

```
for k in dict:
for key in dict.iterkeys():
for value in dict.itervalues():
```

So we could use our iterator to traverse the keys of a dictionary and do whatever we want from there. Other mappings that support iterators should also be able to iterate over keys, but this isn't an absolute rule as specific applications may have different requirements. Iterator support has been added to some of Python's basic types. Calling iter() on a dictionary will return an iterator which loops over its keys:

CODE EXAMPLE: http://structure.usc.edu/python/whatsnew/node4.html

```
>>> m = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'Jun': 6,
     'Jul': 7, 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec':
      12}
>>> for key in m: print key, m[key]

    Mar 3
    Feb 2
    Aug 8
    Sep 9
    May 5
    Jun 6
    Jul 7
    Jan 1
    Apr 4
    Nov 11
    Dec 12
    Oct 10
```

To iterate over keys or values, you can call the **iterkeys(), itervalues(),** or **iteritems()** methods to get the right iterator. So key in dict is equivalent to **dict.has_key(key)**. Dictionaries implement a **tp_iter** slot that returns an effective iterator that iterates over the keys of the dictionary. During the iteration, the dictionary's data is not changed in any way.

CODE EXAMPLE: https://www.python.org/dev/peps/pep-0234/

This means that we can write

```
for k in dict: ...
```

which is equivalent to, but much faster than

```
for k in dict.keys(): ...
```

Iterating in Files: Files also provide an iterator, which calls the readline() method until there are no more lines in the file. This means that you can now read each line of a file using code similar below:

CODE EXAMPLE: http://structure.usc.edu/python/whatsnew/node4.html for line in file: # do something for each line...

The following proposal is useful because it offers us with a good answer to the problem of iterate over the lines in a slow and nasty fashion. Ultimately, using an iterator is faster and more clear.

CODE EXAMPLE: https://www.python.org/dev/peps/pep-0234/ Files implement a tp_iter slot that is equivalent to iter(f.readline, ""). This means that we can write: for line in file:...

which is equivalent to, but faster than

```
while 1:
        line = file.readline()
        if not line:
            break
```

Some iterators are destructive: they devour the values and a second iterator can't simply be created that iterates individually over the same data.

CODE EXAMPLE:

Because the file iterator uses an internal buffer, combining it with other file operations like, file.readline() won't work right. Also, the following code:

```
for line in file:
        if line == "\n":
            break
        for line in file:
        print line,
```

This code doesn't work as you might imagine. This is because the iterator made by the second for-loop doesn't take the first for loop into consideration. The correct way to write it:

CODE EXAMPLE: https://www.python.org/dev/peps/pep-0234

```
it = iter(file)
        for line in it:
            if line == "\n":
                break
            for line in it:
            print line,
```

The iterator version is significantly faster than calling readline() because of the interior buffer in the iterator.

This change also allows us to iterate through files more effectively, so if you ever have a python application that requires input from a text file you could use the line:

```
for line in file:
```

This allows us to iterate through each line of the txt file much faster than we ever had before. This simply iterates through a text file line by line in for loop style fashion.

Apply the terminology and concepts we have used throughout the course:

-“syntax that makes certain common tasks easier or less error prone in the language, perhaps describe the syntax in the context of allowed grammar productions.” -pdf

What Does the Iterator Feature Add?:

- A new exception is defined, StopIteration, which signals the end of an iteration.
- A new slot called tp_iter, that adds an iterator to the type object structure.
- Another new slot is added to the type structure called tp_iternext. It’s for getting the next value in the iteration.
- When return value is NULL, 3 possibru cases then:
  1. No exception is set; which means the end of the iteration.
  2. The StopIteration exception is set; this signals the end of the iteration.
  3. Some other exception is set.

The Python byte code created for 'for' loops is transformed to use new codes, FOR_ITER and GET_ITER that use the iterator procedure rather than the sequence procedure to get the next value for the loop value. Its then possible to use a 'for' loop to loop over non sequential objects that help the tp_iter slot.

Iterators should implement the tp_iter slot as recurring a reference to themselves. This makes it possible to use an iterator in a for loop, as opposed to use a sequence.

Python API Specification:

A new built-in function is defined, iter(). It can be called in two ways: 1. iter(obj) which can also call PyObject_GetIter(obj). 2. iter(obj) = returns an iterator for the object obj 3. iter(C, sentinel) returns an iterator that will call the callable object C until it returns sentinel to indicate that the iterator is complete.

Iterator objects that are returned by iter() have a next() method. This method returns the next value in the iteration or calls StopIteration.

Why the Community Passed this Proposal:

It is a cleaner, faster and more user friendly method to traverse a list, dictionary or file.

Works Cited

1. "Welcome to Python.org." Python.org. N.p., n.d. Web. 11 Dec. 2014.
    - https://www.python.org/dev/peps/pep-0234
2. "3 PEP 234: Iterators." 3 PEP 234: Iterators. N.p., n.d. Web. 11 Dec. 2014.
    - http://structure.usc.edu/python/whatsnew/node4.html