Introduction to Python

Iterables, Iterators and Generators

13 November 2015

Examples 1

- We have already seen that loops in Python are pretty flexible:

Examples 1

- We have already seen that loops in Python are pretty flexible:

```
for i in [1,2,3]:
    print i # prints elements
for i in "123":
    print i # prints characterwise
```

Examples 2

- We have already seen that loops in Python are pretty flexible (excuse lack of indentation):

```
for i in xrange(4):
    print i #prints the range without storing
for i in enumerate([1,2,3]):
    print i # gives me indices as well as elements
```

Examples 2

- We have already seen that loops in Python are pretty flexible (excuse lack of indentation):

```
for i in xrange(4):
    print i #prints the range without storing
for i in enumerate([1,2,3]):
    print i # gives me indices as well as elements
```

- Hopefully you see this and think how do I do that in my code!

Definition

- An **iterable** is any object that responds to the function iter(obj)

Definition

- An **iterable** is any object that responds to the function iter(obj)
- Python likes to hide this!

```
for i in x:
    # Do loop stuff
```

- Call iter(x) and assign to some name that is never used in the code.

Definition

- An **iterable** is any object that responds to the function iter(obj)
- Python likes to hide this!

```
for i in x:
    # Do loop stuff
```

- Call iter(x) and assign to some name that is never used in the code.
- iter(x) returns an object that can do one thing and one thing alone - respond to __next__()

Definition

- An **iterable** is any object that responds to the function iter(obj)
- Python likes to hide this!

```
for i in x:
    # Do loop stuff
```

- Call iter(x) and assign to some name that is never used in the code.
- iter(x) returns an object that can do one thing and one thing alone - respond to __next__()
- Assign i to the response to __next__() each time round until it throws StopIteration exception (sort of an error) and then move on.

Definition

```
for i in x:
    # Do loop stuff
```

- This is equivalent to:

```
randomName = iter(x)
while(True):
    try:
        i = randomName.__next__()
        # Do loop stuff
    except StopIteration:
         break
```

Uses for Iterables

- Iterables can be used all over the place!

```
new_list = list(iterable)
total = sum(iterable)
smallest = min(iterable)
largest = max(iterable)
combined = ''.join(iterable)
result = [f(x) for x in iterable]
```

Make your own iterables

- To make an object and iterable you only need to define the __iter__() hook.
- **Generators** are a great way to create your own iterables that are extremely flexible

Make your own iterables

- To make an object and iterable you only need to define the __iter__() hook.
- **Generators** are a great way to create your own iterables that are extremely flexible
- You define generators in a similar way to function but using the keyword **yield**

```
def myXRange(n):
    i = 0
    while( i < n):
        yield i
        i += 1
```

Make your own iterables

- To make an object and iterable you only need to define the __iter__() hook.
- **Generators** are a great way to create your own iterables that are extremely flexible
- You define generators in a similar way to function but using the keyword **yield**

```
def myXRange(n):
    i = 0
    while( i < n):
        yield i
        i += 1
```

- This responds to the *next* call by running until it hits yield and then stopping and waiting.

Make your own iterables

- To make an object and iterable you only need to define the __iter__() hook.
- **Generators** are a great way to create your own iterables that are extremely flexible
- You define generators in a similar way to function but using the keyword **yield**

```
def myXRange(n):
    i = 0
    while( i < n):
        yield i
        i += 1
```

- This responds to the *next* call by running until it hits yield and then stopping and waiting.
- Demo Time!

Make your own iterables 2

- No reason to make our iterable only finite!
- Lets have all of the positive even numbers

```
def evens():
    i = 2
    while True:
        yield i
        i += 2
```

Make your own iterables 2

- No reason to make our iterable only finite!
- Lets have all of the positive even numbers

```
def evens():
    i = 2
    while True:
        yield i
        i += 2
```

- Be careful when using these! Don't try to make a list out of this one...

Make your own iterables 3

- Perhaps most importantly, we can make iterables that depend on other iterables.

Make your own iterables 3

- Perhaps most importantly, we can make iterables that depend on other iterables.

```
def myEnumerate(iterable):
    index = 0
    for i in iterable:
        yield (index,i)
        index += 1
```

- Demo Time!

Fin

- Suggested watching: Loop Like A Native, Ned Batchelder
- Suggested Library: itertools (This probably has what you need!)