For Loops and Comprehensions

Vivek K. S., Deepak G.

Information Systems Decision Sciences (ISDS)
MUMA College of Business
University of South Florida
Tampa, Florida

2017

For Loop and Iterators

Python uses For loop to iterate over sequential data structures.

- Iterators are used to go over sequential data structures, one element at a time.
- The process is incremental in nature.
- They can be used to process data residing in the memory and also data created on the fly. This helps in processing large chunks of data that cannot be fit into the computer's memory at once.
- Lists, Strings, Tuples, Sets, Dictionaries are all iterable objects in Python.

Iterating Over a List

We can iterate over a list as follows:

```
seq = ['lists','tuples','dictionaries','sets']
count = 0
while count < len(seq):
    print(seq[count])
    count += 1

# The more pythonic way of doing it will be.
for s in seq:
    print(s)
```

Iterating Over other Types

Other Iterables can be iterated as follows:

```
# Strings
word = 'Bootcamp'
for letter in word:
        print(letter)

# Dictionaries
information = {'dept': 'isds', 'college': 'Muma',
'university': 'USF'}
for info in information.items():
    print(info)
```

Iterating Multiple Sequences with zip()

Multiple Sequences can be iterated in parallel using zip without the need to write complex nested for loops.

```
days = ['Monday', 'Tuesday', 'Wednesday']
fruits = ['banana', 'orange', 'peach']
drinks = ['coffee', 'tea', 'beer']
desserts = ['tiramisu', 'ice cream', \
'pie', 'pudding']
for day, fruit, drink, dessert in\
 zip(days, fruits, drinks, desserts):
    print(day, ": drink", drink, "- eat",\
     fruit, "- enjoy", dessert)
```

Note - zip() stops when the shortest sequence is done.

Creating Iterables with zip

zip() can also be used to walk through multiple sequences and make tuples from items at the same offsets.

```
english = 'Monday', 'Tuesday', 'Wednesday'
french = 'Lundi', 'Mardi', 'Mercredi'

days = list(zip(english, french))
print(days)
[('Monday', 'Lundi'), ('Tuesday', 'Mardi'),
 ('Wednesday', 'Mercredi')]

dict(zip(english, french))
{'Monday': 'Lundi', 'Tuesday': 'Mardi',
 'Wednesday': 'Mercredi'}
```

The Range() function

The range() function returns a stream of numbers within a specified range. This function helps enormously in saving crucial computer memory.

- Range returns an iterable object.
- The syntax of the method is, range(start,stop,steps)
- If start is not specified, the range starts at 0.
- Stop is absolutely required.
- The default step size is 1. Negative step values are valid as well.

```
Example,
for x in range(0,3):
        print(x)
```

Code exercises at Github.

Comprehensions in Python

Comprehensions is the Pythonic way of creating sequential data and they improve readability.

```
>>> number_list = []
>>> number_list.append(1)
>>> number_list.append(2)
>>> number_list.append(3)
>>> number_list.append(4)
>>> number_list.append(5)
>>> number_list
[1, 2, 3, 4, 5]
```

Comprehensions in Python

The same result could be achived as follows:

```
number_list = []
for num in range(1,6):
    number_list.append(num)
number_list
[1, 2, 3, 4, 5]

# To further shorten it.
number_list = list(range(1, 6))
number_list
[1, 2, 3, 4, 5]
```

Comprehensions in Python

Though all of the approaches stated in the previous slides are valid and will produce the same result, they are not Pythonic. The simplest way to build a list is to use a list comprehension as follows:

```
number_list = [number for number in range(1,6)]

# If that looks big and if you want to
# shorten it.
number_list = [x for x in range(1,6)]
```

In general, a list comprehension takes up the general syntax: [expression for item in iterable]

Examples

Get a list of squares for the numbers 1 through 10.

```
squares = [x**2 for x in range(1,11)]
squares
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Conditional Expressions in Comprehensions

Comprehensions can also include conditional expressions as follows:
[expression for item in iterable if condition]

```
# Lets only get the even squares in the
# previous list

even_squares = [x**2 for x in range(1,11) if (x**
even_squares
[4, 16, 36, 64, 100]
```

Avoiding Nested Loops Using Comprehensions

Comprehensions can get rid of the need for nested loops as seen below:

```
rows = [1,2,3,5,6]
cols = [1,4,6,7,8]
for row in rows:
    for col in cols:
        print(row, col)

# This can be reduced down to
cells = [(row, col) for row in rows for col\
 in cols]
```

Using Comprehensions for Other Sequential Types

- Comprehensions can be used to create other sequential data types as well and not just Lists.
- Dictionary Comprehension take the form:

```
{key_expr: value_expr for expr in iterable}
# expr is short for expression.
```

- Set expressions take a forms as follows:

```
{expression for expression in iterable}
```

- Tuples do not have comprehensions.
- Enclosing a similar syntax inside a pair of parantheses instead creates a generator.

```
>>>compr = (number for number in range(1, 6))
>>>type(compr)
```

Summary

- We understood higher iterating structures such as zip and sequence generators such as range().
- We understood and implemented comprehensions thereby getting rid of the need of complex code and making it more readable.
- We learned Pythonic way of coding.