



List Comprehension and Generators

List Comprehensions

Populate a list with a for loop

```
nums = [12, 8, 21, 3, 16]
new_nums = []
for num in nums:
    new_nums.append(num + 1)
print(new_nums)
->[13, 9, 22, 4, 17]
```

A list comprehension

```
nums = [12, 8, 21, 3, 16]
new_nums = [num + 1 for num in nums]
print(new_nums)
->[13, 9, 22, 4, 17]
```

For loop and list comprehension syntax

```
#list comprehension Syntax
new_nums = [num + 1 for num in nums]

#for loop
for num in nums:
    new_nums.append(num + 1)
print(new_nums)
->[13, 9, 22, 4, 17] #same list output
```

List comprehension with range()

```
result = [num for num in range(11)]
print(result)
->[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

List comprehensions

- Collapse for loops for building lists into a single line
- Components
 - 1) Iterable
 - 2) Iterator variable (represent members of iterable)
 - 3) Output expression

Nested loops

```
pairs_1 = []
for num1 in range(0, 2):
    for num2 in range(6, 8):
        pairs_1.append(num1, num2)
print(pairs_1)
->[(0, 6), (0, 7), (1, 6), (1, 7)]

#list comprehension
pairs_2 = [(num1, num2) for num1 in range(0, 2) for num2 in range(6, 8)]
print(pairs_2)
->[(0, 6), (0, 7), (1, 6), (1, 7)]

#tradeoff is readability
```

Advanced Comprehensions

Conditionals in comprehensions

```
#Conditionals on the iterable
[num ** 2 for num in range(10) if num % 2 == 0]
->[0, 4, 16, 36, 64]
'''Python documentation on the % operator: The % (modulo) operator yields the remainder
from the division of the first argument by the second.'''
5 % 2
->1
6 % 2
->0
#Conditionals on the output expression
[num ** 2 if num % 2 == 0 else 0 for num in range(10)]
->[0, 0, 4, 0, 16, 0, 36, 0, 64, 0]
```

Dict comprehensions

- Create dictionaries
- Use curly braces {} instead of brackets []

```
pos_neg = {num: -num for num in range(9)}
print(pos_neg)
->{0: 0, 1: -1, 2: -2, 3: -3, 4: -4, 5: -5, 6: -6, 7: -7, 8: -8}
print(type(pos_neg))
-><class 'dict'>
```

Introduction to generator expressions

Generator expressions

```
#Recall list comprehension
[2 * num for num in range(10)]
->[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
#Use ( ) instead of [ ] -> generator
(2 * num for num in range(10))
-><generator object <genexpr> at 0x1046bf888>
''' a list is not created but an "iterable" object is "generated" '''
```

List comprehensions vs. generators

- List comprehension - returns a list

- Generators - returns a generator object
- Both can be iterated over

Printing values from generators

```

result = (num for num in range(6))
for num in result:
    print(num)
->0
1
2
3
4
5
result = (num for num in range(6))
print(list(result))
[0, 1, 2, 3, 4, 5]

result = (num for num in range(6))
#Lazy evaluation -> value is delayed until required, doesn't store in memory
print(next(result))
->0
print(next(result))
->1
print(next(result))
->2
print(next(result))
->3
print(next(result))
->4

```

Conditionals in generator expressions

```

even_nums = (num for num in range(10) if num % 2 == 0)
print(list(even_nums))
->[0, 2, 4, 6, 8]

```

Generator functions

- Produces generator objects when called
- Defined like a regular function - def
- Yields a sequence of values instead of returning a single value
- Generates a value with yield keyword

Build a generator function

```
def num_sequence(n):
    """Generate values from 0 to n."""
    i = 0
    while i < n:
        yield i # yield is used instead of return
        i += 1
```

Use a generator function

```
result = num_sequence(5)
print(type(result))
-><class 'generator'>
for item in result:
    print(item)
->0
1
2
3
4
```

Wrapping up comprehensions and generators

Re-cap: list comprehensions

- Basic
[output expression for iterator variable in iterable]
- Advanced
[output expression + conditional on output for iterator variable in iterable + conditional on iterable]