**Iterations & Looping**

miércoles, 7 de septiembre de 2022

4:37 p. m.

C. Schafer - [Python Tutorial for Beginners 7: Loops and Iterations - For/While Loops](https://www.youtube.com/watch?v=6iF8Xb7Z3wQ)

Check also:

C. Schafer - [Python Tutorial: Iterators and Iterables - What Are They and How Do They Work?](https://www.youtube.com/watch?v=jTYiNjvnHZY&list=PL-osiE80TeTt2d9bfVyTiXJA-UTHn6WwU&index=91)

(8.2 & 8.3) - The for and the while statements: <https://docs.python.org/3/reference/compound_stmts.html#the-for-statement>

(4.4) - break and continue Statements: <https://docs.python.org/3/tutorial/controlflow.html?highlight=loops#break-and-continue-statements-and-else-clauses-on-loops>

(5.6) - Looping Techniques: <https://docs.python.org/3/tutorial/datastructures.html#tut-loopidioms>

Iterations

Any time a iterable must be run through looping is needed and for that there're two kinds of loops: the For and the While loop.

The main difference between the For and the While loop is the duration of the looping, meaning that if the number of loops is know, the best fit would be the For loop, but if the number of loops will depend on a condition to be met independently of how many time the iteration happens, the While loop could be the best fit.

Here are some tips when looping <https://docs.python.org/3/tutorial/datastructures.html#tut-loopidioms>

Break, Continue and pass Statements

Essentially these three simple statement works in service of steering the flow of information within a loop: Break will cut off the entire loop; Continue will cut the current iteration to move over the next one ignoring the rest of the code in the loop and pass will continue to the rest of the code within the current iteration, without jumping to the next item as Continue does.

To better illustrate this, the following code will show how the three statement work.

nums = [x for x in range(6)]

# break

for num in nums:

    print(num)

    if num >= 3:

        break

        print('higher than three')

    print('Next one!')

Console:

0

Next one!

1

Next one!

2

Next one!

3

# continue

for num in nums:

    print(num)

    if num >= 3:

        continue

        print('higher than three')

    print('Next one!')

Console:

0

Next one!

1

Next one!

2

Next one!

3

4

5

# pass

for num in nums:

    print(num)

    if num >= 3:

        pass

        print('higher than three')

    print('Next one!')

Console:

0

Next one!

1

Next one!

2

Next one!

3

higher than three

Next one!

4

higher than three

Next one!

5

higher than three

Next one!

Else statement in loops (meaning without the preceding if)

Source: [Python Tutorial: Else Clauses on Loops](https://www.youtube.com/watch?v=Dh-0lAyc3Bc)

The "Else" statement can be used without the preceding if it is within a loop, and to be more accurate, it serves as a "no break" statement, meaning that the code enclose in the else will be executed at the end of the iteration if it's not broken before that.

nums = [x for x in range(4)]

for num in nums:

    print(num)

else:

    print('hit no-break')

Console:

0

1

2

3

Hit no-break

for num in nums:

    print(num)

    if num == 2: brea

else:

    print('hit no-break')

Console:

0

1

2

* A practical example of the use of "Else" as a no break statement:

# Else as no-break practical example

def index\_finder(seq, target):

    for i,v in enumerate(seq):

        if v == target:

            break

    else:

        return -1

    return i

names = ['Corey', 'Rick', 'John']

index\_location = index\_finder(names, 'John')

print(f'The resulting index is {index\_location}')

Console: The resulting index is 2

…

index\_location = index\_finder(names, 'Steve')

print(f'The resulting index is {index\_location}')

Console: The resulting index is -1

Iterators

To know whether something is an iterable, it has to have an \_\_iter\_\_( ) method

print(dir(list)) = ['\_\_add\_\_', '\_\_class\_\_', '\_\_class\_getitem\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getstate\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iadd\_\_', '\_\_imul\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_reversed\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert',

'pop', 'remove', 'reverse', 'sort']

A clarification, an iterator and an iterable are not the same, and iterable is an object that can be read over, but an iterator *"is an object with a state so that it remembers where it is during iteration*" and it is capable to return the next value on the iteration with the \_\_next\_\_( ) method

The way to obtain an iterator from an iterable (which is what the for loop does in the background) is to call the \_\_iter\_\_( ) method of the iterable as follows, or what could be more correct in this language, to use the iter( ) function, which returns the iterator of an object:

nums = [1, 2, 3, 4]

i\_nums = nums.\_\_iter\_\_()

print(dir(i\_nums)) = ['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getstate\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_length\_hint\_\_', '\_\_lt\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_next\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_setstate\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_']

A iterator is an iterable in itself, and that is why it also has an iter special method in the dir printout. And the next( ) function would return one by one the elements of the iterable until the iterable is exhausted.

print(next(i\_nums)) = 1

print(next(i\_nums)) = 2

print(next(i\_nums)) = 3

print(next(i\_nums)) = 4

print(next(i\_nums)) = StopIterarion Exception!

This is basically the same that a for loop does in the background, but to represent this, the following code illustrates the case:

while True:

    try:

        item = next(i\_nums)

        print(item)

    except StopIteration:

        break

**Note:** It is not possible to pause or going backwards while iterating, even with the next() method, if necessary, a new iterable must be created.

Now, to create an object that is an iterator could be done by adding the \_\_iter\_\_( ) and \_\_next\_\_( ) methods to the class definition as follows.

class MyRange:

    def \_\_init\_\_(self, start, end):

        self.value = start

        self.end = end

    def \_\_iter\_\_(self):

        return self

    def \_\_next\_\_(self):

        if self.value >= self.end:

            raise StopIteration

        current = self.value

        self.value += 1

        return current

nums = MyRange(1,10)

for num in nums:

    print(num) = 1

2

…

9

Generators - Iterators

Generators, in this case, works pretty similar to functions but instead of returning values, they yield values.

The yield reserved word in the documentation: *"Yield expressions and statements are only used when defining a generator function, and are only used in the body of the generator function. Using yield in a function definition is sufficient to cause that definition to create a generator function instead of a normal function."*

Generators are basically iterators with \_\_iter\_\_( ) and \_\_next\_\_( ) methods created implicitly.

Now, the actual syntax of a generator is basically the same of a normal function but with yield instead of return, and since it is not a Class what's being created, the first positional parameter *self* is not required.

The syntax to create a Generator that does the same of the Class built in the prior example would be the following:

def my\_range(start, end):

    current = start

    while current < end:

        yield current

        current += 1

print(list(my\_range(1,10))) = [1, 2, 3, 4, 5, 6, 7, 8, 9]

One of the perks of working with generators is that they does not need to have a limiter or an end, and different from functions, as they yield (produce or generate) values as they run, the does not have to store and return data, therefore are memory efficient solutions.