List comprehensions

PYTHON DATA SCIENCE TOOLBOX (PART 2)

Hugo Bowne-Anderson
Data Scientist at DataCamp

datacamp

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]
```

2. Populate a list with a for loop
You could initialize a new empty list, loop through the old list, add 1 to each entry and append all new values to the new list, but for loops are inefficient, both computationally and in terms of coding time and space, particularly when you could do this in one line of code. "One line of code?" I hear you asking.

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]
```

3. A list comprehension
Welcome to the wonderful world of list comprehensions! The syntax is as follows: within square brackets, you write the values you wish to create, otherwise known as the output expression, followed by the for clause referencing the original list. So in our case, you open the square bracket, followed by num + 1 for num in nums and then you close the square bracket. This is a list comprehension and creates precisely the desired list!

For loop and list comprehension syntax

```
new_nums = [num + 1 for num in nums]
```

```
for num in nums:
    new_nums.append(num + 1)
print(new_nums)
```

```
[13, 9, 22, 4, 17]
```

4. For loop and list comprehension syntax
See here the relationship between the for loop syntax and the list comprehension syntax. The power of list comprehensions is not merely relegated to the world of lists, however, you can write a list comprehension over any iterable.

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
  - Iterable
  - Iterator variable (represent members of iterable)
  - Output expression

6. List comprehensions
list comprehensions collapse for loops for building lists into a single line and the required components are 1) an iterable, 2) an iterator variable that represents the members of the iterable and 3) an output expression. That's it. You can also use list comprehensions in place of nested for loops.

Nested loops (1)

```
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)]
```

- How to do this with a list comprehension?

7. Nested loops (1)
For example, lets say that we wanted to create a list of all pairs of integers where the first integer is between 0 and 1 and the second between 6 and 7. This nested for loop would produce the required result. The question is, can we do the same with a list comprehension? And the answer is, yes, as follows.

Nested loops (2)

```
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: readability

8. Nested loops (2)
Once again, within the square brackets, place the desired output expression followed by the two required for loop clauses. You may observe that while it keeps to a single line of code, we sacrifice some readability of the code as a tradeoff, so you'll have to consider if you'd like to use list comprehensions in cases such as this. The more often you use this, the more you get used to reading list comprehensions, so readability may not be a problem for you later on. But do remember that others may have to read your code as well!

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 2)

Advanced comprehensions

PYTHON DATA SCIENCE TOOLBOX (PART 2)

Hugo Bowne-Anderson
Data Scientist at DataCamp

datacamp

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
```

2. Conditionals in comprehensions advanced comprehension capabilities, such as conditionals! Here we see that we can filter the output of a list comprehension using a conditional on the iterable: in this example, the resulting list is the square of the values in range(10) under the condition that the value itself is even. If you have not seen it before, the percent operation that you see being used in the comprehension is called the modulo operator. We can look at the Python documentation to see how the modulo operator is used and it shows that it produces the remainder from the division of the first argument by the second. Thus an integer modulo two is equal to zero if and only if the integer is even.

Conditionals in comprehensions

- 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]
```

3. Conditionals in comprehensions
We can also condition the list comprehension on the output expression.
Here, for an even integer we output its square. In any other case, signified by the else clause, that is for odd integers, we output 0.

Dict comprehensions

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

4. Dict comprehensions
Now we can also write dictionary comprehensions to create new dictionaries from iterables. The syntax is almost the same as in list comprehensions and there are 2 differences. One, we use curly braces instead of square brackets. Two, the key and value are separated by a colon in the output expression as we can see here. In this example, we are creating a dictionary with keys positive integers and corresponding values the respective negative integers.

```
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'>
```

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 2)

Introduction to generator expressions

PYTHON DATA SCIENCE TOOLBOX (PART 2)

Hugo Bowne-Anderson
Data Scientist at DataCamp

datacamp

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 `[ ]`

```
(2 * num for num in range(10))
```

```
<generator object <genexpr> at 0x1046bf888>
```

List comprehensions vs. generators

- List comprehension - returns a list
- Generators - returns a generator object
- Both can be iterated over

3. List comprehensions vs. generators
Now the question on everybody's lips is, "What is this generator object?"
Well, a generator is like a list comprehension except it does not store the list in memory: it does not construct the list, but is an object we can iterate over to produce elements of the list as required.

Printing values from generators (1)

```
result = (num for num in range(6))
for num in result:
    print(num)
```

```
0
1
2
3
4
5
```

4. Printing values from generators (1)
Here we can see that looping over a generator expression produces the elements of the analogous list. We can also pass a generator to the function list to create the list. Moreover,

```
result = (num for num in range(6))
print(list(result))
```

```
[0, 1, 2, 3, 4, 5]
```

Printing values from generators (2)

```
result = (num for num in range(6))
```

- Lazy evaluation

```
print(next(result))
```

```
0
```

```
print(next(result))
```

```
1
```

```
print(next(result))
```

```
2
```

```
print(next(result))
```

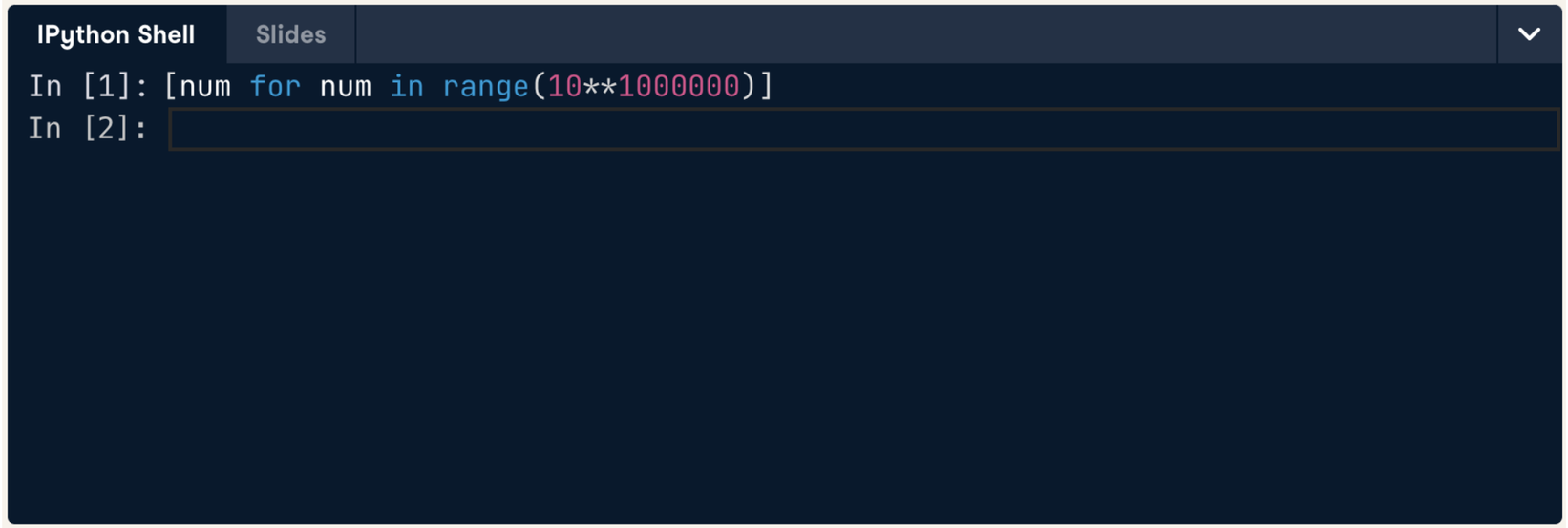
```
3
```

```
print(next(result))
```

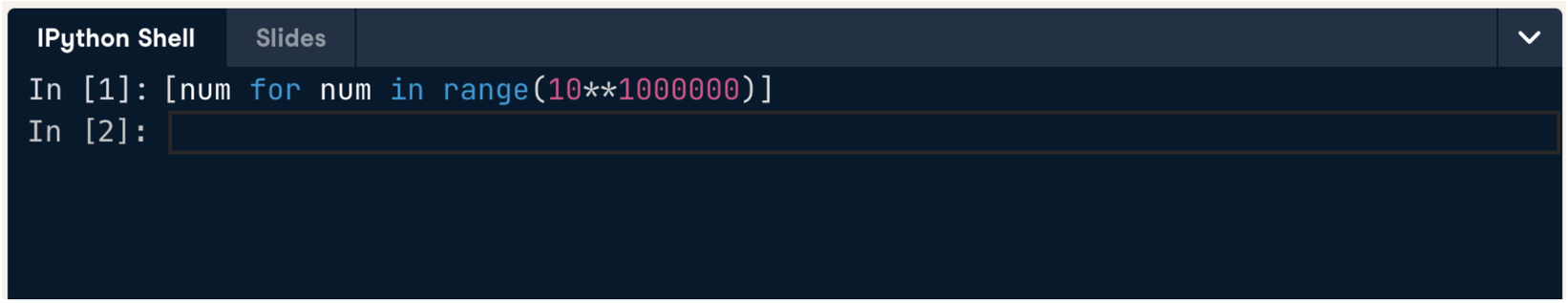
```
4
```

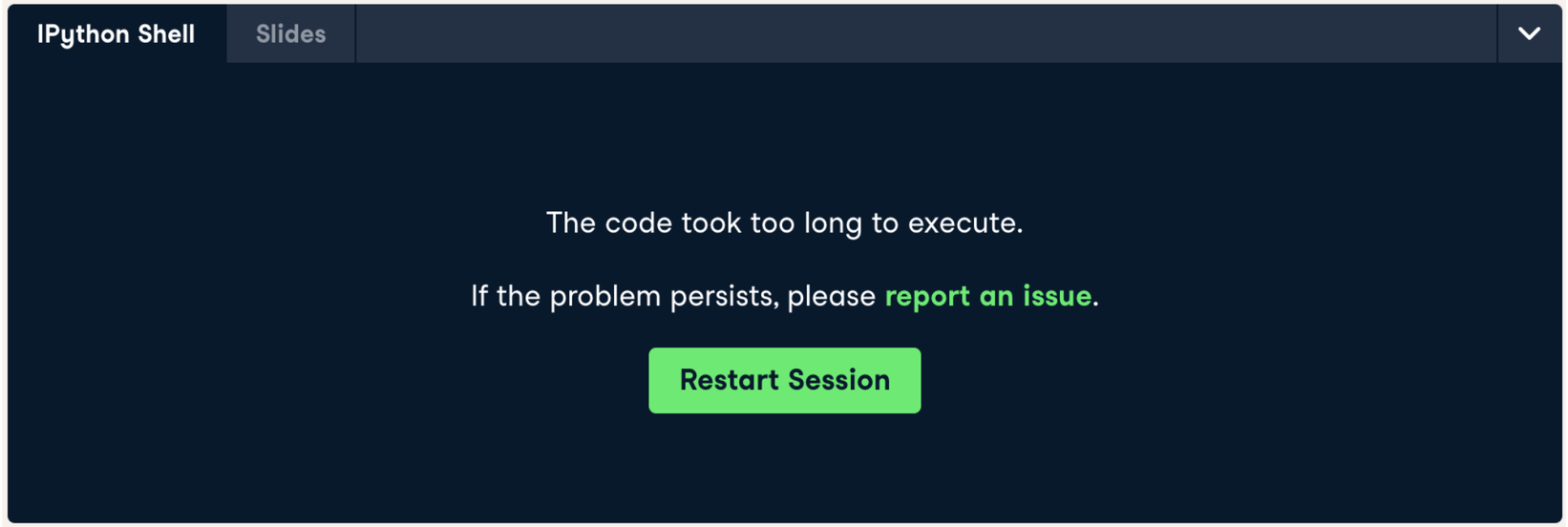
5. Printing values from generators (2)
like any other iterator, we can pass a generator to the function next in order to iterate through its elements. For the geeks like me, this is an example of something called lazy evaluation, whereby the evaluation of the expression is delayed until its value is needed. This can help a great deal when working with extremely large sequences as you don't want to store the entire list in memory, which is what comprehensions would do; you want to generate elements of the sequence on the fly.

Generators vs list comprehensions



Generators vs list comprehensions





6. Generators vs list comprehensions
Let's say that we wanted to iterate over a very large sequence of numbers, such as from 0 up to 10 to the power of a million, or at least wanted to do so until another condition was satisfied. Look what happens when I try to build such an iterable list using a comprehension on DataCamp's servers.

7. Generators vs list comprehensions
My colleagues disconnect me because the list I'm trying to create can't even be stored in memory! Be warned though, don't try this at home, on our servers or yours!

Generators vs list comprehensions

IPython Shell | Slides

```
In [1]: (num for num in range(10**1000000))
Out[1]: <generator object <genexpr> at 0x7f8c2447a7d8>

In [2]:
```

7. Generators vs list comprehensions
My colleagues disconnect me because the list I'm trying to create can't even be stored in memory! Be warned though, don't try this at home, on our servers or yours!

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]
```

9. Conditionals in generator expressions
What's really cool is that anything we can do in a list comprehension such as filtering and applying conditionals, we can also do in a generator expression, such as you see here.

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

10. Generator functions
The last thing to discuss before you get coding is the ability to write generator functions.
Generator functions are functions that, when called, produce generator objects.
Generator functions are written with the syntax of any other user-defined function, however instead of returning values using the keyword return, they yield sequences of values using the keyword yield.

Build a generator function

- sequence.py

```
def num_sequence(n):
    """Generate values from 0 to n."""
    i = 0
    while i < n:
        yield i
        i += 1
```

11. Build a generator function

Here I have defined a generator function that, when called with a number n, produces a generator object that generates integers 0 though n. We can see within the function definition that i is initialized to 0 and that the first time the generator object is called, it yields i equal to 0. It then adds one to i and will then yield one on the next iteration and so on. The while loop is true until i equals equals n and then the generator ceases to yield values.

Use a generator function

```
result = num_sequence(5)
print(type(result))
```

```
<class 'generator'>
```

```
for item in result:
    print(item)
```

```
0
1
2
3
4
```

12. Use a generator function
This generator function can be called as you do any other function. Here I call the generator function with the argument, 5. We see that it produces a generator object and that we can iterate over this generator object with a for loop to print the values it yields. Generator functions are a powerful and customizable way to create generators.

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 2)

Wrapping up comprehensions and generators.

PYTHON DATA SCIENCE TOOLBOX (PART 2)

Hugo Bowne-Anderson
Data Scientist at DataCamp

datacamp

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]
```

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 2)