7 Iteration

7.1 Introduction

In this chapter, we will learn strategies for **iteration**, or performing a repeated task over many values.

Consider the following task: Suppose I want to get a computer to print out the lyrics to the song 99 Bottles of Beer on the Wall.

Now, we could certainly simply type up all the lyrics ourselves:

```
print(
  "99 Bottles of beer on the wall, 99 Bottles of beer. Take one down, pass it arc
)
```

```
99 Bottles of beer on the wall, 99 Bottles of beer. Take one down, pass it around, 98
Bottles of beer on the wall. 98 Bottles of beer on the wall, 98 Bottles of beer. Take one
down, pass it around, 97 Bottles of beer on the wall.
```

That sounds like… not much fun to do.

But we notice that in the song, there is a ton of repetition! Every verse is almost identical, except for the number of bottles of beer.

This is a great time for some *iteration*.

Just our opinion…

Fortunately, we aren't going to try to sing this version: https://www.youtube.com/watch?v=R30DnFfVtUw

7.2 For loops

The most basic approach to iteration is a simple `for` loop. At each step of the loop the value of our placeholder, `i`, changes to the next step in the provided list, `range(100,97,-1)`.

```
for i in range(100,97,-1):
  print(str(i) + " bottles of beer on the wall")
  print(str(i) + " bottles of beer")
  print(" take one down, pass it around,")
  print(str(i-1) + " bottles of beer on the wall")
```

```
100 bottles of beer on the wall
100 bottles of beer
```

```
 take one down, pass it around,
99 bottles of beer on the wall
99 bottles of beer on the wall
99 bottles of beer
 take one down, pass it around,
98 bottles of beer on the wall
98 bottles of beer on the wall
98 bottles of beer
 take one down, pass it around,
97 bottles of beer on the wall
```

Check In

After last chapter, hopefully we immediately see that this is a great opportunity to write a function, to make our code clearer and avoid repetition.

Write a function called `sing_verse()` to replace the body of the `for` loop.

Check In

```
def sing_verse(num):
  song = str(num) + " bottles of beer on the wall \n" + str(num) + " bottles of beer \n"

  return song
```

Notice that in this function, instead of `print`-ing out the lines of the song, we return the an object consisting as one long string.

Often, when running a `for` loop, you want to end the loop process with a single object. To do this, we start with an empty object, and then add to it at each loop.

```
song = ""

for i in range(100,97,-1):
  song = song + sing_verse(i)

print(song)
```

```
100 bottles of beer on the wall
100 bottles of beer
 take one down, pass it around,
99 bottles of beer on the wall
99 bottles of beer on the wall
99 bottles of beer
 take one down, pass it around,
98 bottles of beer on the wall
98 bottles of beer on the wall
98 bottles of beer
```

```
 take one down, pass it around,
97 bottles of beer on the wall
```

Check In

Modify the code above so that instead of creating one long string, we create a *list*, where each element is one of the verses of the song.

7.2.1 Vectorized functions

The function that we wrote above can be described as **not vectorized**. What we mean by that it is designed to only take one value, `num`. If we instead try to input a list or vector of numbers, we get an error:

```
sing_verse([100,99,98])
```

```
TypeError: unsupported operand type(s) for -: 'list' and 'int'
```

This is why, in order to get results for a list of number, we needed to iterate.

However, plenty of functions are designed to work well for single numbers *or* lists and vectors. For example:

```
a_num = 5
a_vec = [1,3,5,7]

np.sqrt(a_num)
np.sqrt(a_vec)
```

```
array([1.        , 1.73205081, 2.23606798, 2.64575131])
```

When we want to perform a function over many values - whether it's one we wrote or not - we first need to ask ourselves if the function is vectorized or not. Using a `for` loop over a vectorized function is unnecessarily complicated and computationally slow!

```
result = []
for i in a_vec:
  result = result + [np.sqrt(i)]

result
```

```
[1.0, 1.7320508075688772, 2.23606797749979, 2.6457513110645907]
```

7.2.2 Vectorizing and booleans

A common reason why a custom function is written in an unvectorized way is that it makes use of `if` statements. For example, consider the task of taking the square root of only the *positive* numbers in a list.

Here is an approach that *does not* work:

```
a_vec = np.array([-2, 1, -3, -9, 7])

if a_vec > 0:
  a_vec = np.sqrt(a_vec)

a_vec
```

```
ValueError: The truth value of an array with more than one element is ambiguous. Use
a.any() or a.all()
```

The statement `if a_vec > 0` makes no sense for a vector! The `if` statement needs either a single `True` or a single `False` to determine if the subsequent code will be run - but `a_vec > 0` returns a list of five booleans.

Instead, we would need to iterate over the values:

```
a_vec = np.array([-2, 1, -3, -9, 7])

for val in a_vec:
  if val > 0:
    val = np.sqrt(val)
  print(val)
```

```
-2
1.0
-3
-9
2.6457513110645907
```

However, there is a nicer approach to this variety of problem, which is to use **boolean masking**:

```
is_pos = a_vec > 0

a_vec[is_pos] = np.sqrt(a_vec[is_pos])

a_vec
```

```
array([-2,  1, -3, -9,  2])
```

Check In

Write two functions:

- `sqrt_pos_unvec()` takes in a single value as an argument, and returns the square root if the value is positive. Then, write a `for` loop that uses this function to construct a new vector where the positive values are square rooted.
- `sqrt_pos_vec()` takes in a vector of values, and returns a vector with the positive values square rooted. **Do not** use a for loop inside your function.

7.3 Iterable functions

Although `for` loops are a clear and basic procedure, it can become very tedious to use them frequently. This is especially true if you want to save the results of the iteration as a new object.

However, it will not be possible or convenient to write every function in a vectorized way.

Instead, we can use **iterable functions**, which perform the same iteration as a `for` loop in shorter and more elegant code.

7.3.1 `map()`

The `map()` function requires the same information as a `for` loop: what values we want to iterate over, and what we want to do with each value.

```
song = map(sing_verse, range(100, 97, -1))
song = list(song)
print("".join(song))
```

```
100 bottles of beer on the wall
100 bottles of beer
 take one down, pass it around,
99 bottles of beer on the wall
99 bottles of beer on the wall
99 bottles of beer
 take one down, pass it around,
98 bottles of beer on the wall
98 bottles of beer on the wall
98 bottles of beer
 take one down, pass it around,
97 bottles of beer on the wall
```

Note

Notice that the output of the `map()` function is a special object structure, of the type "map object". We automatically convert this to a list with the `list()` function.

Then, we make use of the `join()` string method to turn the list into one long string. Finally, we `print()` our final string out so that it looks nice.

7.3.1.1 Double mapping

Sometimes, we want to loop through multiple sets of values at once. The `map()` function has the ability to take as many *iterables*, or lists of values, as you want.

Suppose we want to change our `sing_verse()` function so that it has two arguments, the number of bottles and the type of drink.

```
def sing_verse_2(num, drink):
  song = str(num) + " bottles of " + drink + " on the wall \n"
  song = song + str(num) + " bottles of " + drink + "\n"
  song = song + " take one down, pass it around, \n"
  song = song + str(num-1) + " bottles of " + drink + " on the wall \n"

  return song
```

Now, we use `map()` to switch the number *and* the drink at each iteration:

```
nums = range(100, 97, -1)
drinks = ["beer", "milk", "lemonade"]
song = map(sing_verse_2, nums, drinks)
print("".join(list(song)))
```

```
100 bottles of beer on the wall
100 bottles of beer
 take one down, pass it around,
99 bottles of beer on the wall
99 bottles of milk on the wall
99 bottles of milk
 take one down, pass it around,
98 bottles of milk on the wall
98 bottles of lemonade on the wall
98 bottles of lemonade
 take one down, pass it around,
97 bottles of lemonade on the wall
```

Check In

Write a `sing_verse_3()` function that also lets us change the container (e.g. bottle, can, …) at each step of the loop.

Use `map()` to sing a few verses.

What happens if you supply three different drinks, but only two different types of containers?

7.4 Lambda functions

What would you do if you still wanted to count down the number of bottles, but you wanted them all to be lemonade?

In this case, we want *one* of the arguments of our function to be iterated over many values, and the other one to stay consistent.

One rather inelegant way we could accomplish this is with a new function:

```
def sing_verse_lemonade(num):
  return sing_verse_2(num, "lemonade")
```

```
        song = map(sing_verse_lemonade, nums)
        print("".join(list(song)))
```

```
100 bottles of lemonade on the wall
100 bottles of lemonade
 take one down, pass it around,
99 bottles of lemonade on the wall
99 bottles of lemonade on the wall
99 bottles of lemonade
 take one down, pass it around,
98 bottles of lemonade on the wall
98 bottles of lemonade on the wall
98 bottles of lemonade
 take one down, pass it around,
97 bottles of lemonade on the wall
```

This is a lot of extra lines of code, though, for a task that should be straightforward - and we'll probably never use `sing_verse_lemonade()` again, so it's a bit of a waste to create it.

Instead, we will use what is called a **lambda function**, which is like making a new `sing_verse_lemonade` wrapper function for temporary use:

```
        song = map(lambda i: sing_verse_2(i, "lemonade"), nums)
        print("".join(list(song)))
```

```
100 bottles of lemonade on the wall
100 bottles of lemonade
 take one down, pass it around,
99 bottles of lemonade on the wall
99 bottles of lemonade on the wall
99 bottles of lemonade
 take one down, pass it around,
98 bottles of lemonade on the wall
98 bottles of lemonade on the wall
98 bottles of lemonade
 take one down, pass it around,
97 bottles of lemonade on the wall
```

The code `lambda i: sing_verse_2(i, "lemonade")` made a new **anonymous function** - sometimes called a **headless function** - that takes in one argument, `i`, and sends that argument into `sing_verse_2`.

Check In

Use a lambda function with `sing_verse_3()` to sing a few verses about milk in glasses.

7.5 Iterating on datasets

This task of repeating a calculation with many inputs has a natural application area: datasets!

It is extremely common that we want to perform some calculation involving many *variables* of the dataset, and we want to repeat that same calculation over the values in each row.

For this situation, we use an iterable function that is very similar to `map()`: the `.apply()` method from `pandas`.

At its core, the `.apply()` method is meant for repeating a calculation over *columns*:

```
dat = pd.DataFrame({"x": [99, 50, 2], "y": [1, 2, 3]})

dat.apply(np.sqrt)
```

	x	y
0	9.949874	1.000000
1	7.071068	1.414214
2	1.414214	1.732051

In this chapter, though, we are more interested in using it to repeat a function, using each row as input:

```
dat.apply(np.sum, axis=1)
```

```
0    100
1     52
2      5
dtype: int64
```

Suppose we have a `pandas` dataframe consisting of all the numbers, drinks, and containers that we are interested in singing about:

```
dat = pd.DataFrame({"num": [99, 50, 2], "drink": ["beer", "soda", "Capri Sun"], "

dat
```

	num	drink	container
0	99	beer	bottles
1	50	soda	cans
2	2	Capri Sun	pouches

Our goal is to **apply** the `sing_verse_3` function over all these *combinations* of values.

Unfortunately, this doesn't happen automatically:

```
dat.apply(sing_verse_3, axis=1)
```

```
TypeError: sing_verse_3() missing 2 required positional arguments: 'drink' and
'container'
```

This is because `.apply` doesn't "know" which columns below with which arguments of the `sing_verse_3` function. We'll need to use a lambda function to help it out:

```
dat.apply(lambda x: sing_verse_3(x['num'], x['drink'], x['container']), axis=1)
```

```
0    99 bottles of beer on the wall \n99 bottles of...
1    50 cans of soda on the wall \n50 cans of soda\...
2    2 pouches of Capri Sun on the wall \n2 pouches...
dtype: object
```

Check In

Click here to open the practice activity.

Edit this page