‹ (/python-novice-inflammation/03-matplotlib/index.html) Programming with Python (/python-novice-inflammation/) › (/pyth novice inflam loop/i

Storing Multiple Values in Lists

Overview

Teaching: 30 min
Exercises: 15 min
Questions

- How can I store many values together?

Objectives

- Explain what a list is.
- Create and index lists of simple values.
- Change the values of individual elements
- Append values to an existing list
- Reorder and slice list elements
- Create and manipulate nested lists

In the previous episode, we analyzed a single file of clinical trial inflammation data. However, after finding some peculiar and potentially suspicious trends in the trial data we ask Dr. Maverick if they have performed any other clinical trials. Surprisingly, they say that they have and provide us with 11 more CSV files for a further 11 clinical trials they have undertaken since the initial trial.

Our goal now is to process all the inflammation data we have, which means that we still have eleven more files to go!

The natural first step is to collect the names of all the files that we have to process. In Python, a list is a way to store multiple values together. In this episode, we will learn how to store multiple values in a list as well as how to work with lists.

Python lists

Unlike NumPy arrays, lists are built into the language so we do not have to load a library to use them. We create a list by putting values inside square brackets and separating the values with commas:

Python

```
odds = [1, 3, 5, 7]
print('odds are:', odds)
```

Output

```
odds are: [1, 3, 5, 7]
```

We can access elements of a list using indices – numbered positions of elements in the list. These positions are numbered starting at 0, so the first element has an index of 0.

Python

```
print('first element:', odds[0])
print('last element:', odds[3])
print('"-1" element:', odds[-1])
```

Output

```
first element: 1
last element: 7
"-1" element: 7
```

Yes, we can use negative numbers as indices in Python. When we do so, the index `-1` gives us the last element in the list, `-2` the second to last, and so on. Because of this, `odds[3]` and `odds[-1]` point to the same element here.

There is one important difference between lists and strings: we can change the values in a list, but we cannot change individual characters in a string. For example:

Python

```
names = ['Curie', 'Darwing', 'Turing']  # typo in Darwin's name
print('names is originally:', names)
names[1] = 'Darwin'  # correct the name
print('final value of names:', names)
```

Output

```
names is originally: ['Curie', 'Darwing', 'Turing']
final value of names: ['Curie', 'Darwin', 'Turing']
```

works, but:

Python

```
name = 'Darwin'
name[0] = 'd'
```

Error

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-220df48aeb2e> in <module>()
      1 name = 'Darwin'
----> 2 name[0] = 'd'

TypeError: 'str' object does not support item assignment
```

does not.

📌 Ch-Ch-Ch-Ch-Changes

Data which can be modified in place is called mutable (../reference.html#mutable), while data which cannot be modified is called immutable (../reference.html#immutable). Strings and numbers are immutable. This does not mean that variables with string or number values are constants, but when we want to change the value of a string or number variable, we can only replace the old value with a completely new value.

Lists and arrays, on the other hand, are mutable: we can modify them after they have been created. We can change individual elements, append new elements, or reorder the whole list. For some operations, like sorting, we can choose whether to use a function that modifies the data in-place or a function that returns a modified copy and leaves the original unchanged.

Be careful when modifying data in-place. If two variables refer to the same list, and you modify the list value, it will change for both variables!

Python

```
salsa = ['peppers', 'onions', 'cilantro', 'tomatoes']
my_salsa = salsa        # <-- my_salsa and salsa point to the *same* list data in memory
salsa[0] = 'hot peppers'
print('Ingredients in my salsa:', my_salsa)
```

Output

```
Ingredients in my salsa: ['hot peppers', 'onions', 'cilantro', 'tomatoes']
```

If you want variables with mutable values to be independent, you must make a copy of the value when you assign it.

Python

```
salsa = ['peppers', 'onions', 'cilantro', 'tomatoes']
my_salsa = list(salsa)        # <-- makes a *copy* of the list
salsa[0] = 'hot peppers'
print('Ingredients in my salsa:', my_salsa)
```

Output

```
Ingredients in my salsa: ['peppers', 'onions', 'cilantro', 'tomatoes']
```

Because of pitfalls like this, code which modifies data in place can be more difficult to understand. However, it is often far more efficient to modify a large data structure in place than to create a modified copy for every small change. You should consider both of these aspects when writing your code.

📌 Nested Lists

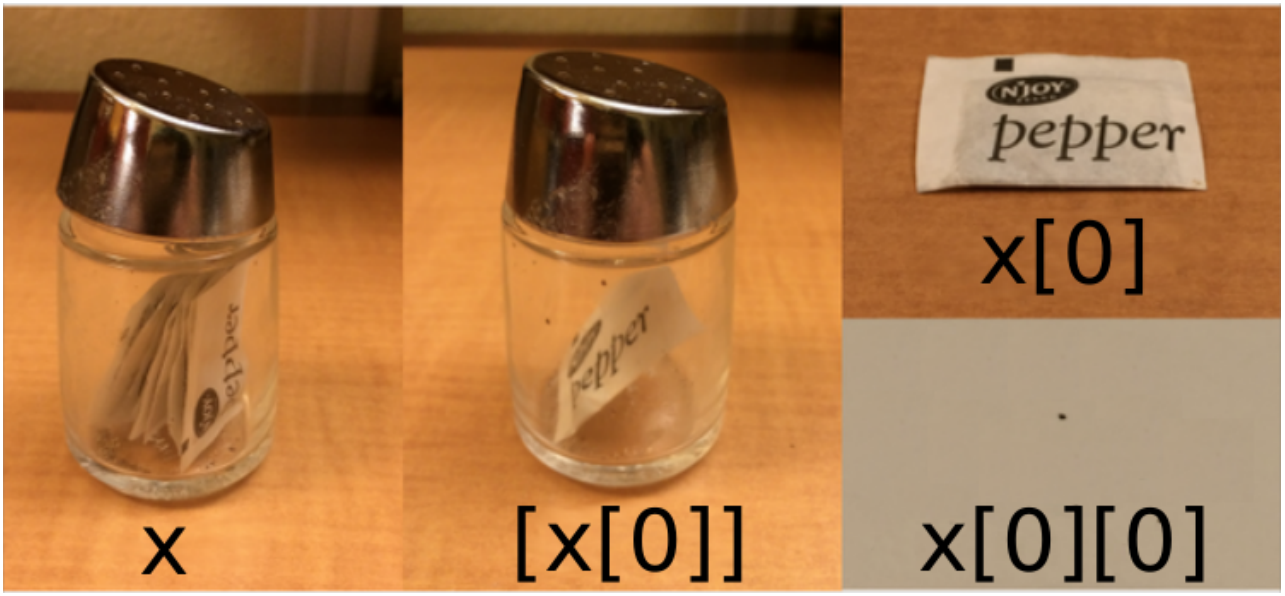
Since a list can contain any Python variables, it can even contain other lists.

For example, we could represent the products in the shelves of a small grocery shop:

Python

```
x = [['pepper', 'zucchini', 'onion'],
     ['cabbage', 'lettuce', 'garlic'],
     ['apple', 'pear', 'banana']]
```

Here is a visual example of how indexing a list of lists `x` works:



(https://twitter.com/hadleywickham/status/643381054758363136)

Using the previously declared list `x`, these would be the results of the index operations shown in the image:

Python

```
print([x[0]])
```

Output

```
[['pepper', 'zucchini', 'onion']]
```

Python

```
print(x[0])
```

Output

```
['pepper', 'zucchini', 'onion']
```

Python

```
print(x[0][0])
```

Output

```
'pepper'
```

Thanks to Hadley Wickham (https://twitter.com/hadleywickham/status/643381054758363136) for the image above.

Heterogeneous Lists

Lists in Python can contain elements of different types. Example:

Python

```
sample_ages = [10, 12.5, 'Unknown']
```

There are many ways to change the contents of lists besides assigning new values to individual elements:

Python

```
odds.append(11)
print('odds after adding a value:', odds)
```

Output

```
odds after adding a value: [1, 3, 5, 7, 11]
```

Python

```
removed_element = odds.pop(0)
print('odds after removing the first element:', odds)
print('removed_element:', removed_element)
```

Output

```
odds after removing the first element: [3, 5, 7, 11]
removed_element: 1
```

Python

```
odds.reverse()
print('odds after reversing:', odds)
```

Output

```
odds after reversing: [11, 7, 5, 3]
```

While modifying in place, it is useful to remember that Python treats lists in a slightly counter-intuitive way.

As we saw earlier, when we modified the `salsa` list item in-place, if we make a list, (attempt to) copy it and then modify this list, we can cause all sorts of trouble. This also applies to modifying the list using the above functions:

Python

```
odds = [1, 3, 5, 7]
primes = odds
primes.append(2)
print('primes:', primes)
print('odds:', odds)
```

Output

```
primes: [1, 3, 5, 7, 2]
odds: [1, 3, 5, 7, 2]
```

This is because Python stores a list in memory, and then can use multiple names to refer to the same list. If all we want to do is copy a (simple) list, we can again use the `list` function, so we do not modify a list we did not mean to:

Python

```
odds = [1, 3, 5, 7]
primes = list(odds)
primes.append(2)
print('primes:', primes)
print('odds:', odds)
```

Output

```
primes: [1, 3, 5, 7, 2]
odds: [1, 3, 5, 7]
```

Subsets of lists and strings can be accessed by specifying ranges of values in brackets, similar to how we accessed ranges of positions in a NumPy array. This is commonly referred to as “slicing” the list/string.

Python

```
binomial_name = 'Drosophila melanogaster'
group = binomial_name[0:10]
print('group:', group)

species = binomial_name[11:23]
print('species:', species)

chromosomes = ['X', 'Y', '2', '3', '4']
autosomes = chromosomes[2:5]
print('autosomes:', autosomes)

last = chromosomes[-1]
print('last:', last)
```

Output

```
group: Drosophila
species: melanogaster
autosomes: ['2', '3', '4']
last: 4
```

Slicing From the End

Use slicing to access only the last four characters of a string or entries of a list.

Python

```
string_for_slicing = 'Observation date: 02-Feb-2013'
list_for_slicing = [['fluorine', 'F'],
                    ['chlorine', 'Cl'],
                    ['bromine', 'Br'],
                    ['iodine', 'I'],
                    ['astatine', 'At']]
```

Output

```
'2013'
[['chlorine', 'Cl'], ['bromine', 'Br'], ['iodine', 'I'], ['astatine', 'At']]
```

Would your solution work regardless of whether you knew beforehand the length of the string or list (e.g. if you wanted to apply the solution to a set of lists of different lengths)? If not, try to change your approach to make it more robust.

Hint: Remember that indices can be negative as well as positive

Solution

Non-Continuous Slices

So far we've seen how to use slicing to take single blocks of successive entries from a sequence. But what if we want to take a subset of entries that aren't next to each other in the sequence?

You can achieve this by providing a third argument to the range within the brackets, called the *step size*. The example below shows how you can take every third entry in a list:

Python

```
primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
subset = primes[0:12:3]
print('subset', subset)
```

Output

```
subset [2, 7, 17, 29]
```

Notice that the slice taken begins with the first entry in the range, followed by entries taken at equally-spaced intervals (the steps) thereafter. If you wanted to begin the subset with the third entry, you would need to specify that as the starting point of the sliced range:

Python

```
primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
subset = primes[2:12:3]
print('subset', subset)
```

Output

```
subset [5, 13, 23, 37]
```

Use the step size argument to create a new string that contains only every other character in the string "In an octopus's garden in the shade". Start with creating a variable to hold the string:

Python

```
beatles = "In an octopus's garden in the shade"
```

What slice of `beatles` will produce the following output (i.e., the first character, third character, and every other character through the end of the string)?

Output

```
I notpssgre ntesae
```

Solution

If you want to take a slice from the beginning of a sequence, you can omit the first index in the range:

Python

```
date = 'Monday 4 January 2016'
day = date[0:6]
print('Using 0 to begin range:', day)
day = date[:6]
print('Omitting beginning index:', day)
```

Output

```
Using 0 to begin range: Monday
Omitting beginning index: Monday
```

And similarly, you can omit the ending index in the range to take a slice to the very end of the sequence:

Python

```
months = ['jan', 'feb', 'mar', 'apr', 'may', 'jun', 'jul', 'aug', 'sep', 'oct', 'nov', 'dec']
sond = months[8:12]
print('With known last position:', sond)
sond = months[8:len(months)]
print('Using len() to get last entry:', sond)
sond = months[8:]
print('Omitting ending index:', sond)
```

Output

```
With known last position: ['sep', 'oct', 'nov', 'dec']
Using len() to get last entry: ['sep', 'oct', 'nov', 'dec']
Omitting ending index: ['sep', 'oct', 'nov', 'dec']
```

Overloading

`+` usually means addition, but when used on strings or lists, it means “concatenate”. Given that, what do you think the multiplication operator `*` does on lists? In particular, what will be the output of the following code?

Python

```
counts = [2, 4, 6, 8, 10]
repeats = counts * 2
print(repeats)
```

1. `[2, 4, 6, 8, 10, 2, 4, 6, 8, 10]`
2. `[4, 8, 12, 16, 20]`
3. `[[2, 4, 6, 8, 10],[2, 4, 6, 8, 10]]`
4. `[2, 4, 6, 8, 10, 4, 8, 12, 16, 20]`

The technical term for this is *operator overloading*: a single operator, like `+` or `*`, can do different things depending on what it’s applied to.

Solution

Key Points

- `[value1, value2, value3, ...]` creates a list.
- Lists can contain any Python object, including lists (i.e., list of lists).
- Lists are indexed and sliced with square brackets (e.g., list[0] and list[2:9]), in the same way as strings and arrays.
- Lists are mutable (i.e., their values can be changed in place).
- Strings are immutable (i.e., the characters in them cannot be changed).

‹ (/python-novice-inflammation/03-matplotlib/index.html)

› (/pyth novice inflam loop/i

Edit on GitHub (https://github.com/swcarpentry/python-novice-inflammation/edit/gh-pages/_episodes/04-lists.md) / Contributing (https://github.com/swcarpentry/python-novice-inflammation/blob/gh-pages/CONTRIBUTING.md) / Source (https://github.com/swcarpentry/python-novice-inflammation/) / Cite (https://github.com/swcarpentry/python-novice-inflammation/blob/gh-pages/CITATION) / Contact (mailto:team@carpentries.org)

Using The Carpentries style (https://github.com/carpentries/styles/) version 9.5.3 (https://github.com/carpentries/styles/releases/tag/v9.5.3).