

[< \(../02-loop/\)](#)

Programming with Python (../)

[> \(../04-files/\)](#)

Storing Multiple Values in Lists

Overview

Teaching: 30 min**Exercises:** 0 min**Questions**

- How can I store many values together?

Objectives

- Explain what a list is.
- Create and index lists of simple values.

Just as a `for` loop is a way to do operations many times, a list is a way to store many values. Unlike NumPy arrays, lists are built into the language (so we don't have to load a library to use them). We create a list by putting values inside square brackets and separating the values with commas:

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

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

We select individual elements from lists by indexing them:

```
print('first and last:', odds[0], odds[-1])
```

```
first and last: 1 7
```

and if we loop over a list, the loop variable is assigned elements one at a time:

```
for number in odds:
    print(number)
```

```
1
3
5
7
```

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:

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

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

works, but:

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

```
-----
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-Changes

Data which can be modified in place is called mutable (`./reference/#mutable`), while data which cannot be modified is called immutable (`./reference/#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!

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

```
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.

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

```
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.

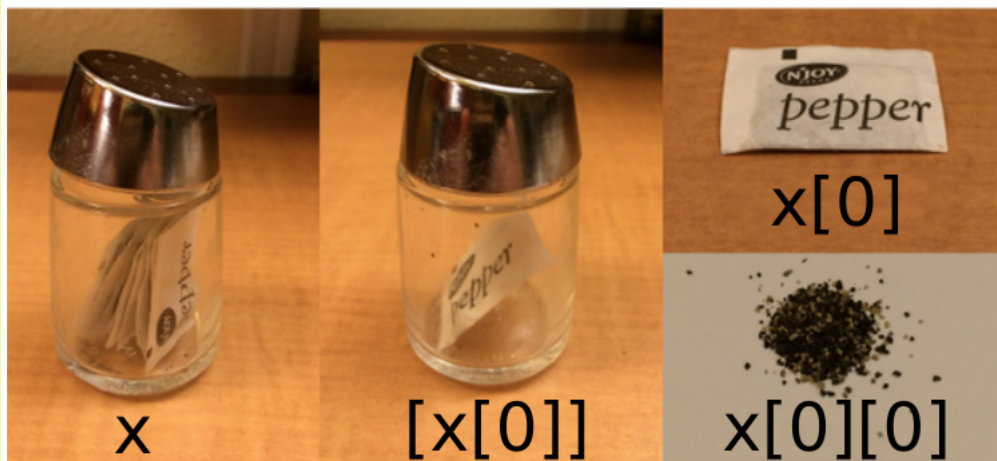
✦ Nested Lists

Since lists can contain any Python variable, it can even contain other lists.

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

```
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:

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

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

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

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

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

```
'pepper'
```

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

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

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

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

```
del odds[0]
print('odds after removing the first element:', odds)
```

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

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

```
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.

If we make a list and (attempt to) copy it then modify in place, we can cause all sorts of trouble:

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

```
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 use the `list` function, so we do not modify a list we did not mean to:

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

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

This is different from how variables worked in lesson 1, and more similar to how a spreadsheet works.

Turn a String Into a List

Use a for-loop to convert the string "hello" into a list of letters:

```
["h", "e", "l", "l", "o"]
```

Hint: You can create an empty list like this:

```
my_list = []
```

 **Solution** 

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.

```

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

species = binomial_name[11:24]
print("species:", species)

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

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

```

```

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.

```

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

```

```

"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.

 **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:

```

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

```

```
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:

```

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

```

```
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"

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

```
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:

```

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

```

```
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:

```
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:]
("Omitting ending index:", sond)
```

```
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"]
```

Swapping the contents of variables

Explain what the overall effect of this code is:

```
left = 'L'
right = 'R'

temp = left
left = right
right = temp
```

Compare it to:

```
left, right = [right, left]
```

Do they always do the same thing? Which do you find easier to read?

 **Solution** 

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?

```
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 are indexed and sliced 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).

 (../02-loop/)

 (../04-files/)

Copyright © 2016–2017 Software Carpentry Foundation (<https://software-carpentry.org>)

Edit on GitHub (https://github.com/swcarpentry/python-novice-inflammation/edit/gh-pages/_episodes/03-lists.md) / Contributing (<https://github.com/swcarpentry/python-novice-inflammation/blob/gh-pages/CONTRIBUTING.md>) / Source

