Python Part IV - Storing Multiple Values in Lists

Jean-Yves Sgro

February 28, 2017

Contents

1 Software Carpentry: Storing Multiple Values in Lists **1**
1.1 Overview: 1
1.2 Key points summary 2

2 Lists **2**
2.1 Creating a list: 2
2.2 List index 2
2.3 Changing values of list item 2

3 Mutable or immutable **3**

4 Nested lists **4**
4.1 Modifying a list: `append`, `del`, `reverse` 5
4.2 Copy a simple list safely 6

5 Test your understanding: **6**
5.1 Turn a String Into a List 6

6 Slicing (subsetting) a list **7**
6.1 Exercise: Slicing From the End 8
6.2 Exercise: Non-continuous slices 8
6.3 Exercise: Exchanges 9
6.4 Exercise: Overloading 10

References and/or Footnotes **10**

```
## Warning: package 'knitr' was built under R version 3.5.2
```

1 Software Carpentry: Storing Multiple Values in Lists

This lesson *"Storing Multiple Values in Lists"* is lesson 03 from Software Carpentry ("Programming with Python" 2016).

1.1 Overview:

Questions

- How can I store many values together?

Objectives

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

1.2 Key points summary

- `[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).

2 Lists

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).

2.1 Creating a list:

We create a list by putting values inside square brackets:

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

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

2.2 List index

By now we know that Python likes to start counting with zero `0` as the "first" item as we have seen previously with `range()` for example.

In the same way, items in the list are indexed starting with zero `0`; the last item is referenced as `-1`.

```
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:

2.3 Changing values of list item

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: Within the list we can change one of the elements with a new value. In this case we will substitute the second element on the list (therefore indexed as `1` if `0` is the first one) with a new value: `Darwing` will be replaced by `Darwin`:

```python
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'])
```

The replacement of element`names[1]` of ***list*** `names` was successful.

Therefore a list is said to be ***mutable***.

On the other hand, if we have a ***variable*** called `name` we cannot change any of the individual elements of the variable as in this example where we try to substitute a `d` for the `D` in `Darwin`; this will cause an error:

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-cb94b05a418a> in <module>()
      1 name = 'Darwin'
----> 2 name[0] = 'd'

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

Therefore a variable is said to be ***immutable***.

3 Mutable or immutable

Ch-Ch-Ch-Changes

Data which can be modified in place is called ***mutable***, while data which cannot be modified is called ***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 are mutable.

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

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.

4 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, and we could then use an indexing method (starting with `0` as usual in Python) to extract any sub-list in various ways. Note the "double indexing" `x[0][0]` to first extract the first list from `x` and then extract the first item from that extracted list:

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

# print the first sub-list as a list of 3 items:)
print("x[0] = ", x[0])

# print the first item of the first sublist:
print("x[0][0] = ", x[0][0])

# print the first sublist as an item within a list:
print("[x[0]] = ", [x[0]])
```

```
('x[0] = ', ['pepper', 'zucchini', 'onion'])
('x[0][0] = ', 'pepper')
('[x[0]] = ', [['pepper', 'zucchini', 'onion']])
```

This image provided by Hadley Wickham gives a visual example of the ideas of list and sublists we just explored:

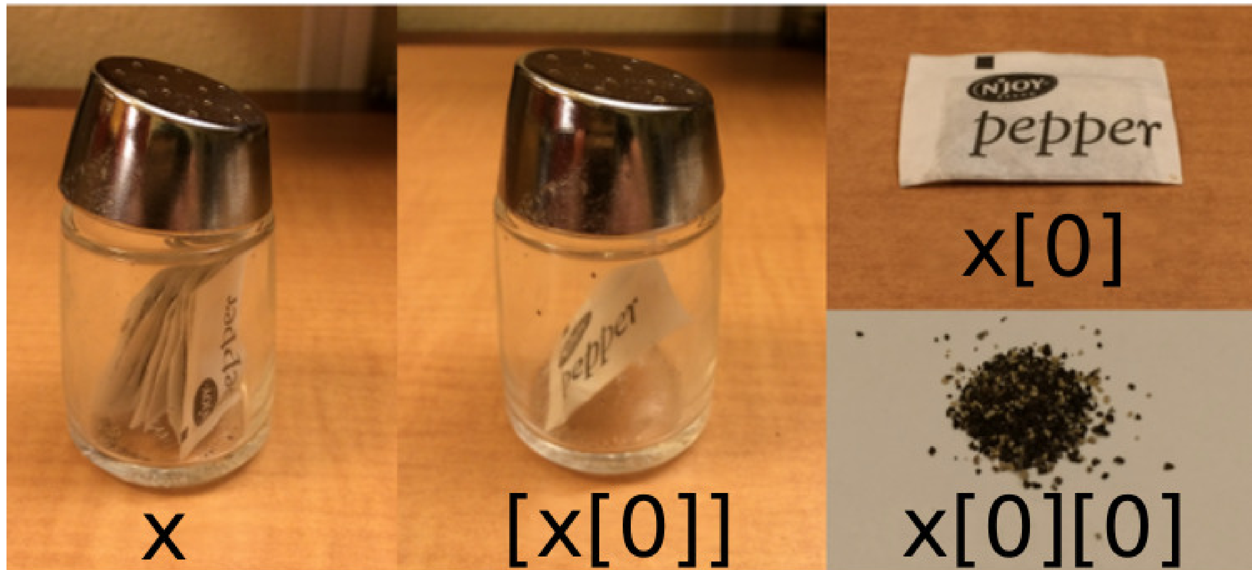

Figure 1.

Illustration of lists, and idexed items.

4.1 Modifying a list: `append`, `del`, `reverse`

4.1.1 `append()` (add) to an existing list:

Let's recreate the `odds` list and then add one more item once the list has been created.

We encounter again the "dot notation" that we have seen previously with library commands properties.

The command structure is **based on properties of lists predefined** wihtin Python. In other words, a Python list *by nature* has inherent properties that you can list with the `dir()` command we saw before.

`append` would be one of the properties, and therefore the command `odds.append()` would be a "built-in" method to append (add) to the existing list `odds`.

```
odds = [1, 3, 5, 7]
# append (add) one more item
odds.append(11)
print('odds after adding a value:', odds)
```

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

Note: Another method to add an item at the end of the list is by using the `+=` operand. Therefore the code could also be written as:

```
odds = [1, 3, 5, 7]
# add item [11] at the end
odds += [11]
print('odds after addition with +=:', odds)
```

```
('odds after addition with +=:', [1, 3, 5, 7, 11])
```

4.1.2 `del` delete an item:

The method to remove an item uses command `del` and specifying an indexed item. It is in that sense a different mode of operation than we just saw with `odds.append()`.

Let's redeclare the longer `odds` list and then remove the first item:

```
odds = [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])
```

The item can also be in the middle of the list, for example the 3rd element (therefore index `2`.) We reconstruct the `odds` list to start anew:

```
odds = [1, 3, 5, 7, 11]
del odds[2]
print('odds after removing the 3rd element:', odds)
```

```
('odds after removing the 3rd element:', [1, 3, 7, 11])
```

4.1.3 Reversing a list with `reverse()`

Here again the method uses a "dot notation" method as part of an *inherent property of a Python list* (use `dir odds` to see the list again if you wish.)

We start again with a renewed `odds` list and then reverse it with the built-in list property `odds.reverse()`:

```
odds = [1, 3, 5, 7, 11]
odds.reverse()
print('odds after reversing:', odds)
```

```
('odds after reversing:', [11, 7, 5, 3, 1])
```

4.1.4 Causing trouble

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. In the code below we create the list `odds` and then make a copy named `primes` which we later modify by adding element `2` with `+= [2]` (which of course is not and odd number but is a prime number.)

In doing so, the original `odds` list will be modified also:

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

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

Therefore we inadvertantly added `2` to the `odds` list....

This is because Python stores a list in memory, and then can use multiple names to refer to the same list. See the next paragraph for the safe way to copy a list.

4.2 Copy a simple list safely

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. The difference is that rather than writing `primes = odds` we create a new list with `primes = list(odds)`:

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

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

In this case the original list `odds` was safe from any modification.

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

5 Test your understanding:

5.1 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 = []
```

The provided solution does the following:

- create an empty list called `my_list`
- open a `for` loop with the declared variable `"hello"` in quotes.
- use the keyword `char` as the loop variable. `char` represent characters (letters) one at a time
- use the `append` property built in all Python list to add `char` at the end of the list
- when the loop is finished the final list is printed.

```
my_list = []
for char in "hello":
    my_list.append(char)
print(my_list)
```

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

What could be improved or changed?

1. the string "hello" is "built-in" the code but could be made a variable instead. This would make the method more flexible and changeable by simply changing the value of the variable, here `my_string`.

```
my_list = []
my_string = "hello"
for char in my_string:
    my_list.append(char)
print(my_list)
```

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

We could go even further in abstraction and use one more variable `my_char` to temporary store the value of `char` and then use the `+=` method rather than the `append` method as we have seen previously:

```
my_list = []
my_string = "hello"
for char in my_string:
    my_char = char
    my_list += my_char
print(my_list)
```

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

Note: using `my_` before as a prefix to a variable name is a good idea to clarify that fact that you "made up" that variable name and it is not built-in the Python program.

6 Slicing (subsetting) a list

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.

The code below shows slicing methods for a variable `binomial_name = "Drosophila melanogaster"` by taking the first 10 letters (from positions `0` to `9`.)

The next segment takes the other portion of the variable, from positions `11` to `23`.

The next segment makes a subset of a list, starting at position 3 (but indexed as 2 because 0 is the first.)

The last segment prints the last item from a list with position -1.

```
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')
```

6.1 Exercise: 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"], ["astati

len_str = len(string_for_slicing)
len_list = len(list_for_slicing )

print(len_str, len_list)

my_substring = string_for_slicing[len_str -4:len_str]
print(my_substring)

my_sublist = list_for_slicing[len_list -4:len_list]
print(my_sublist)
```

```
(29, 5)
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.

6.2 Exercise: 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"
subset = beatles[0::2]
print("subset:", subset)
```

```
('subset:', 'I notpssgre ntesae')
```

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:]
print("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'])
```

6.3 Exercise: Exchanges

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

```
temp = left
left = right
right = temp
#
print("left = ", left, "right = ", right)
```

```
('left = ', 'R', 'right = ', 'L')
```

Compare to:

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

left, right = right, left
print("left = ", left, "right = ", right)
```

```
('left = ', 'R', 'right = ', 'L')
```

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

6.4 Exercise: 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)
```

Choose one output from the list:

```
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.

References and/or Footnotes

"Programming with Python." 2016. (Archived May 14, 2016 http://go.wisc.edu/db1733). http://swcarpentry.github.io/python-novice-inflammation.