Python Workshop

4. List operations

Quick jump: List slicing | List copying | Advanced topics

List operations

- Consider the following example:

```
myList = [1, 2, 3, 4, 5, 6]
myList2 = [myList[0:3], myList[3:]]
myList3 = list(myList2)
print(myList2)
```

- Notation `myList[0:3]` is getting a **slice** of the list, in this case, the slice is `[1, 2, 3]`.
- `myList2` is a list that consists of two lists.
- `myList3` is a copy of `myList2`. We need to know how `myList3` is affected if we modify `myList2`.

Self-learning topics (~40min)

- List slicing
- List copying
- Nested list
- Advanced topics:
    - List manipulation
- You are encouraged to copy-and-paste the code to Spyder and test them.

List slicing

Slicing

- Instead of accessing one item, we can access a **slice** of list using `[ : ]`:

```
myList = [0, 1, 2, 3, 4, 5]
print('Sub-list from 1 up to 4:', myList[1:4])
```

Output:

```
Sub-list from 1 up to 4: [1, 2 ,3]
```

- Note that index starts from 0.
- The use of values is similar to `range()`, the second number is exclusive.

Slicing defaults

- The two values can be omitted, in that case they defaults to the beginning and the end of the list.

```
myList = [0, 1, 2, 3, 4, 5]
print(myList[:3])  # output [0, 1, 2]
print(myList[3:])  # output [3, 4, 5]
```

- First value defaults to `0`.
- Second value defaults to the size of list.
- As the ending value is exclusive, the notation ensure that `[:n]` and `[n:]` together gives the original list.

Negative values

- Values could be negative, that means we count from the back instead.

```
myList = [0, 1, 2, 3, 4, 5]
print(myList[:-2], myList[-2:])
```

Output:

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

- A value of `-2` is identical to the length of list - 2.

Using variables

- It is also possible to use a variable instead.

```
myList = [0, 1, 2, 3, 4, 5]
for i in range(3):
    print(myList[:i + 1])
```

Output:

```
[0]
[0, 1]
[0, 1, 2]
```

The third value

- Very similar to `range()`, we can specify the third value for steps between values.

```
myList = [0, 1, 2, 3, 4, 5]
print(myList[0:4:2])
```

Output:

```
[0, 2]
```

Exercises (1)

Exercise 1

- Write a program that reads a number of integers until a zero is received (you can modify from your code in the previous exercises).
- After reading all integers, repeatedly remove the first and last item from the list and print it. For example, if the input is 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, the output will be:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 3, 4, 5, 6, 7, 8]
[3, 4, 5, 6, 7]
[4, 5, 6]
[5]
```

Exercise 2

- Modify your code in exercise 1, instead of print the list directly, print the list with only the items at odd positions.
- For example, if the input is 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, the output will be:

```
[1, 3, 5, 7, 9]
[2, 4, 6, 8]
[3, 5, 7]
[4, 6]
[5]
```

List references

List copying

- Variable holding a list is actually keeping a **reference** to the list only. Same goes for all other data structures.
- A simple assignment using `=` operator will not create a new list. For example:

```
myList = [0, 1, 2, 3]
myList2 = myList
```

- `myList` and `myList2` will be the same list, modifying items in `myList` will also affect `myList2` (and vice versa).

List assignment (2)

```
myList = [0, 1, 2, 3]
myList2 = myList
myList[1] = 100
print(myList2)
```

Output:

```
[0, 100, 2, 3]
```

- Modification to `myList` will affect `myList2`.

List copying (1)

To create a copy of a list instead, we can do either of the following.

- Create a slice of the whole list:

```
t = s[:]
```

- Use the `copy()` method of a list:

```
t = s.copy()
```

- Use the `list()` function to create a new list:

```
t = list(s)
```

List copying (2)

- For example, the following code create a copy of `myList` as `myList2`.

```
myList = [0, 1, 2, 3]
myList2 = myList[:]
myList[1] = 100
print(myList)
print(myList2)
```

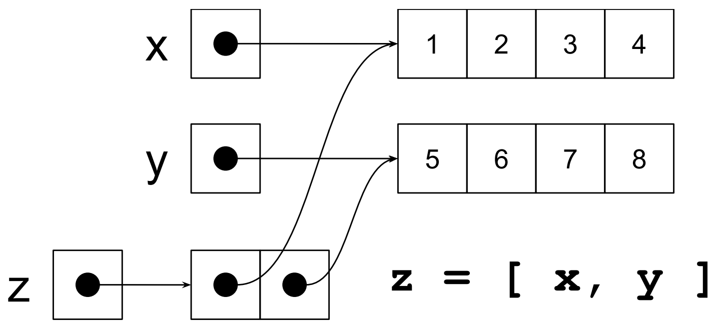
Output:

```
[0, 100, 2, 3]
[0, 1, 2, 3]
```

Nested list

Nested list

- We can put a list in a list. The following example create a 2D list:



```
x = [1, 2, 3, 4]
y = [5, 6, 7, 8]
z = [x, y]
print('x:', x)
print('y:', y)
print('z:', z)
```

Output:

```
x: [1, 2, 3, 4]
y: [5, 6, 7, 8]
z: [[1, 2, 3, 4], [5, 6, 7, 8]]
```

Accessing nested list items

- We can use multiple `[]` operators to access items in a multi-dimensional list.

```
z = [[1, 2, 3, 4], [5, 6, 7, 8]]
print('z[0][0]:', z[0][0])
print('z[1][2]:', z[1][2])
```

Output:

```
z[0][0]: 1
z[1][2]: 7
```

Nested list assignment (1)

- Note that when we put a list in a list, only a reference is assigned. For example:

```
x = [1, 2, 3, 4]
y = [5, 6, 7, 8]
z = [x, y]
print('x:', x)
print('y:', y)
print('z:', z)
```

- The part `z = [x, y]` will assign a reference of `x` and `y` into the list `z`.
- In this case, modifying `x` or `y` will also affect `z`.

Nested list assignment (2)

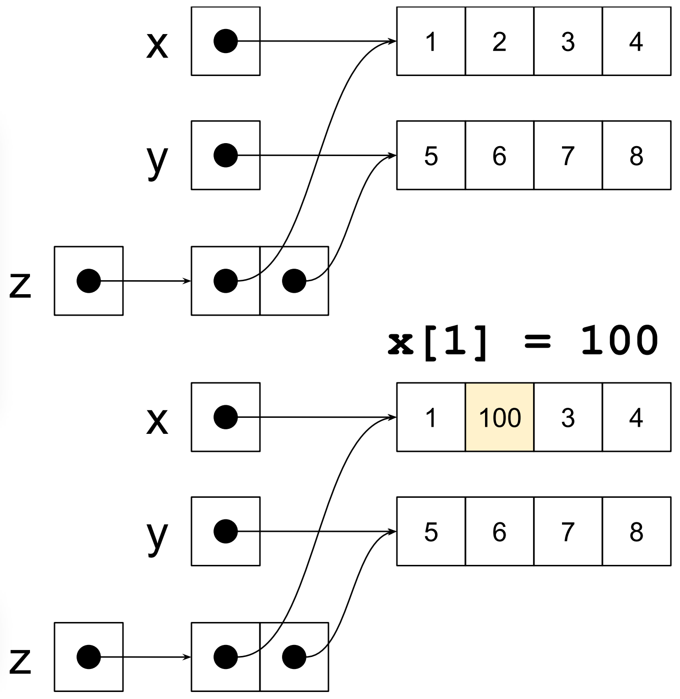
- Now try this:

```
x = [1, 2, 3, 4]
y = [5, 6, 7, 8]
z = [x, y]
x[1] = 100
print('z:', z)
```

Output:

```
[[1, 100, 3, 4], [5, 6, 7, 8]]
```

- Note that z is affected by the change of value in list x.



Copying list again

- To avoid the previous problem, we can create a copy instead.

```
x = [1, 2, 3, 4]
y = [5, 6, 7, 8]
z = [list(x), list(y)]
x[1] = 100
print('z:', z)
```

Output:

```
[[1, 2, 3, 4], [5, 6, 7, 8]]
```

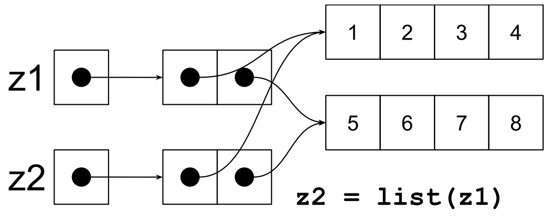
Copying 2D list

- What happens if we copy a 2D list?

```
z1 = [[1, 2, 3, 4], [5, 6, 7, 8]]
z2 = list(z1)
z1[0][1] = 100
z1[1] = [200, 300]
print(z2)
```

shallow copy

- A **shallow copy** is performed when copying lists, i.e., if the value is a list, only the reference is copied.
- So consider the example again:



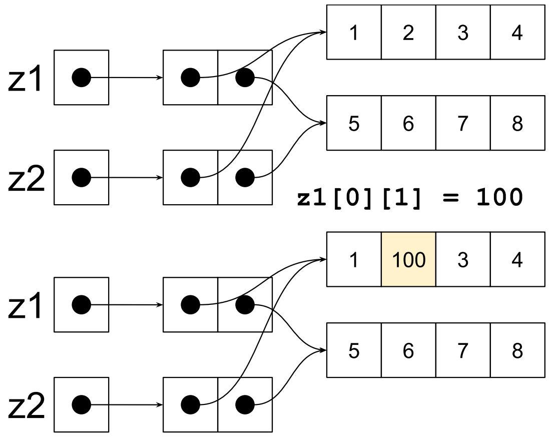
```
z1 = [[1, 2, 3, 4], [5, 6, 7, 8]]
z2 = list(z1)
```

- `z1` and `z2` are two different list with the same set of sublist.

shallow copy (2)

- What happens to the two assignments afterwards?

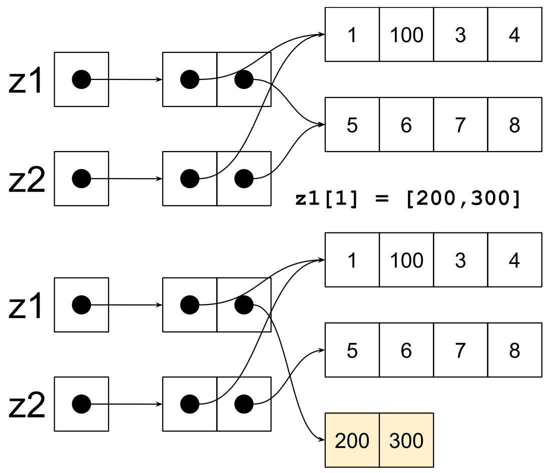
```
z1 = [[1, 2, 3, 4], [5, 6, 7, 8]]
z2 = list(z1)
z1[0][1] = 100
z1[1] = [200, 300]
```



- `z1[0][1] = 100` will modify the list `[1, 2, 3, 4]`, and thus affect `z2` also.

Shallow copy (3)

- On the other hand, `z1[1] = [200, 300]` will replace the reference in `z1` by another reference, the list `[5, 6, 7, 8]` is not modified at all. Therefore, `z2` is not affected at all.



```
z1 = [[1, 2, 3, 4], [5, 6, 7, 8]]
z2 = list(z1)
z1[0][1] = 100
z1[1] = [200, 300]
```

shallow copy (4)

- Now you should be able to predict the result of this example.

```
z1 = [[1, 2, 3, 4], [5, 6, 7, 8]]
z2 = list(z1)
z1[0][1] = 100
z1[1] = [200, 300]
print(z2)
```

Output:

```
[[1, 100, 3, 4], [5, 6, 7, 8]]
```

Deep copy (1)

- In contrast to shallow copy, a **deep copy** will create a copy of multi-dimensional list completely.
- This is an example of deep-copying a 2D list:

```
z1 = [[1, 2, 3, 4], [5, 6, 7, 8]]
z2 = []
for i in z1:
  tmp = []
  for j in i:
    tmp.append(j)
  z2.append(tmp)
```

Deep copy (2)

- We may also use a nested generator expression:

```
z1 = [[1, 2, 3, 4], [5, 6, 7, 8]]
z2 = [ [ j for j in i ] for i in z1 ]
z1[0][1] = 100
z1[1] = [200, 300]
print(z2)
```

Deep copy (3)

- Previous methods only applies to 2D list. A recursive function or library function will be more appropriate for higher dimension lists.
    - Both concept which will be covered in a later section.
- Here is an example that uses a library function:

```python
from copy import deepcopy
z1 = [[1, 2, 3, 4], [5, 6, 7, 8]]
z2 = deepcopy(z1)
z1[0][1] = 100
z1[1] = [200, 300]
print(z2)
```

Exercises (2)

Exercise 3

- Write a program that reads **9 integers** and keep them in a **3 x 3 matrix**. (a 2D list)
- After reading all 9 integers, print the matrix, as well as the same matrix rotated 180 degree. For example:

```
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]
```

```
[8, 7, 6]
[5, 4, 3]
[2, 1, 0]
```

Advanced topics

- Remember: you may skip this section and come back later.

List manipulation

List operations (1)

- Operators `+` and `*` can be used on list.
- `+` is used for concatenation:
- `*` is used to generate a repeating pattern:

```
myList = [1, 2] + [3, 4]
myList2 = [1, 2] * 3
print(myList, myList2)
```

Output:

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

List operations (2)

- Operators `+=` and `*=` can also be used.

```
myList = [1, 2]
myList *= 3
myList += [3, 4]
print(myList)
```

Output:

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

More list operations

- There are many other list operations that could be useful, you are advised to try them yourselves:
    - `s.append(x)`: Append value `x` to list `s`.
    - `s.insert(i, x)`: Insert value `x` to list `s` at index `i`.
    - `i = s.index(x)`: Find the index of the first item with value `x`, store the index in variable `i`.
    - `c = s.count(x)`: Count the numbe of items with value `x`, store the count in variable `c`.

Much more list operations

- `s.clear()`: Empty list `s`.
- `s.remove(x)`: remove first item with value `x` from list `s`.
- `x = s.pop()`: remove first item from list `s` and assign the removed item to variable `x`.
- `x = s.pop(i)`: remove i-th item from list `s` and assign the removed item to variable `x`.
- `s.sort()`: Sort list `s`.
- `s.reverse()`: Reverse the order of values in list `s`.

Using del

- We can use `del` to remove one item or a slice of list:

```
myList = [0, 1, 2, 3, 4, 5, 6, 7, 8]
del myList[7]
del myList[1:4]
print(myList)
```

Output:

```
[0, 4, 5, 6, 8]
```

Slice replacement (2)

- It is also possible to replace a slice by another list.

```
myList = [0, 1, 2, 3, 4, 5]
myList[1 : 4] = [100, 200]
print(myList)
```

Output:

```
[0, 100, 200, 4, 5]
```

- Size of list will be different after such assignment.

Slice replacement (2)

- Slice replacement can be used to support many of the list operations.

Operation	**Equivalent**
`s.append(x)`	`s[len(s):] = [x]`
`s += t`	`s[len(s):] = t`
`s.clear()`	`s[:] = []`
`s.insert(i, x)`	`s[i:i] = [x]`