항상 기뻐하라 쉬지 말고 기도하라 범사에 감사하라 이것이 그리스도 예수 안에서 너희를 향하신 하나님의 뜻이니라 성령을 소멸하지 말며 예언을 멸시하지 말고 범사에 헤아려 좋은 것을 취하고 악은 어떤 모양이라도 버리라 (살전5:16-22)

NOTE: The following materials have been compiled and adapted from the numerous sources including my own. Please help me to keep this tutorial up-to-date by reporting any issues or questions. Send any comments or criticisms to `idebtor@gmail.com` Your assistances and comments will be appreciated.

Lists

학습 목표

- 리스트에 대해 이해한다.
- 파이썬에서 제공하는 리스트 메소드를 살펴본다.

학습 내용

1. 슬라이싱 문법
2. 다양한 리스트 메소드
3. 리스트 작업 수행

Most of programs work not only with variables, but also use lists of variables. For example, a program can handle students informationin in a class by reading the list of students from the keyboard or from a file.

To store such data, you can use the data structure called **list**.

- Lists are sequences of objects.
- Lists can contain a mix of any type of objects:
- Lists are one of the workhorse data structures.

A list is a sequence of elements numbered from 0, just as characters in the string. You create a list by putting square brackets around a comma-separated list of other Python items:

In [ ]:
```
primes = [2, 3, 5, 7, 11, 13]
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
```

Like strings, you can concatenate lists using the add operation:

In [ ]:
```
# combine two lists
c = None
print(c)
```

and you can multiply a list by an integer to repeat the items in a list multiple times:

In [ ]:
```
[True] * None
```

Lists are mutable

Unlike strings, list indexing is different that you can set the values in the list: strings are **immutable**, but lists are **mutable**.

Example 6: Change the second element into uppercase

```
['red', 'ORANGE', 'yellow', 'green', 'blue', 'indigo', 'violet']
```

In [ ]:
```
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
None
print(rainbow)
```

Making a list

Make a list called `seq` with **odd** numbers between 1 and 20 as shown below:

```
seq = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Solution 1:

- Create an empty list or initialize a list.
- Use a `for` loop and `range()`.
- Use `append()` method to append an item to the list one by one.

In [ ]:
```
None
print(seq)
```

Solution 2 - a challenge problem:

- You may do this now, or you would be able to do this after finishing this whole lesson.
- Produce `seq = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]` in the following code
- Use a list additon instead of `append()`.

In [ ]:
```
seq = []
for x in range(1, 20, 2):
    None                    # use addition or in-place addition
print(seq)
```

Indexing

Indexing on lists works much the same way as strings.

To get the first element of a list, use index 0:

In [ ]:
```
# print the first color
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
print(rainbow[0])
```

Example 1: List elments in three different ways

Three ways of printing each color in the rainbow, one element at a time as shown below:

1. using index
2. using iterable
3. using enumerate

```
1:red 2:orange 3:yellow 4:green 5:blue 6:indigo 7:violet
```

solution 1: Using list indexing

In [ ]:
```
# using indexing
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
None
```

solution 2: Using list iterable

In [ ]:
```
# using list which is iterable
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
None
```

solution 3: Using enumerating list

In [ ]:
```
# enumerating list with indices
None
```

In [ ]:
```
# enumerating list with indices
None
```

Slicing

Slicing is used to extract a subsequence out of your sequence(e.g. list, string):

```
var[lower:upper:step]
```

The element which has index equal to the lower bound is included in the slice, but the element which has index equal to the upper bound is excluded, so mathematically, the ie slice is `[lower, upper)`. If you think of the indices as being between the elements, then this mentally works very nicely, as we'll see.

The `step` argument is optional, and indicates the strides between elements in the subsequence, so a step of 2 takes every second element.

List slicing also has very similar semantics to slicing on strings.

```
- a[start:end]  items start through end-1
- a[start:]     items start through the rest of the array
- a[:end]       items from the beginning through end - 1
```

```
- a[:]          a copy of the whole list
- a[start:end:step]  start through end - 1, by step
```

Example 1: Slice `[11, 12]`

- slice [11, 12]

In [ ]:
```
# slice [11, 12]
a = [10, 11, 12, 13, 14, 15]
None
```

Example 2: Slice the first three elements `[10, 11, 12]`

In [ ]:
```
# slice [10, 11, 12]
a = [10, 11, 12, 13, 14, 15]
None
```

Example 3: Slice all elements except the last one using a negative indices

In [ ]:
```
# slice [10, 11, 12, 13, 14], use negative indices
a = [10, 11, 12, 13, 14, 15]
None
```

Example 4:

- There is a list that consists of pairs of name and age.
- Make a list of names or every other elements as shown in the following list, `data` .
- Compute the average age.

```
data = ['john', 3, 'kim', 7, 'peter', 13, 'paul', 17, 'david', 10]

names: ['john', 'kim', 'peter', 'paul', 'david']
aveage age:10.00
```

Solution 1: Using enumerate() or for loop

In [ ]:
```
data = ['john', 3, 'kim', 7, 'peter', 13, 'paul', 17, 'david', 10]

None

print('names:', names)
print('average age: %.2f' % age)
print('average age: {:.2f}'.format(age))
```

Solution 2: Using slicing - the simplest solution

In [ ]:
```
data = ['john', 3, 'kim', 7, 'peter', 13, 'paul', 17, 'david', 10]
None

print('names:', names)
```

```
print('average age: %.2f' % age)
print('average age: {:.2f}'.format(age))
```

Replace elements with

Unlike strings, because lists are mutable, you can write values into a slice.

Example 6: Replace `11, 12` with `1, 2`

In [ ]:
```
# replace 11, 12 with 1, 2
a = [10, 11, 12, 13, 13, 15]
None
print(a)
```

This overwrites the second and third elements. However you can inject extra elements into the list in the same way: the length of the slice and the length of the injected list don't have to match.

Example 7: Replace `11, 12` with `1, 2, 3, 4`

Insert `[1, 2, 3, 4]` into the slice from 1 to 3 such that it produces `[10, 1, 2, 3, 4, 13, 14, 15]`. This will remove the current contents of the slice and replace it with the new list, increasing the length of the list:

In [ ]:
```
# replace, 1, 2 with 1, 2, 3, 4
a = [10, 1, 2, 13, 14, 15]
None
print(a)
```

Delete elements

It's also possible to use the same idea for deleting elements.

If we take the slice from 1 to 3 and inject an empty list into the slice, it will **delete** the second and third elements:

Example 8: Delete elements `1, 2`

Complete the code to produce `[10, 3, 4, 13, 14, 15]`.

In [ ]:
```
a = [10, 1, 2, 3, 4, 13, 14, 15]
None
print(a)
```

Copying elements

Finally, you can **slice or copy all of the elements** with a slice like this:

Example 9:

For example, we want to make a copy of list `a` and have `red` is replaced with `RED` as shown below:

```
a = ['red', 'green', 'blue']
b = ['RED', 'green', 'blue']
```

What is wrong with the following code?

In [2]:
```
a = ['red', 'green', 'blue']
b = a
b[0] = b[0].upper()

print('b =', b)
print('a =', a)
```

```
b = ['RED', 'green', 'blue']
a = ['RED', 'green', 'blue']
```

Use `[:]` to get a copy of a list which is a mutable object.

In [ ]:
```
a = ['red', 'green', 'blue']
None
```

Tuple is immutable.

Tuple data type is defined by `( )` instead of `[ ]` as shown below:

```
a = ('red', 'green', 'blue')
a = ('RED', 'green', 'blue')
```

Use `tuple()` constructor to convert a list to a tuple type.

If you want to change the first element of `a` into uppercase, which line of code would be a problem?

In [ ]:
```
a = ('red', 'green', 'blue')
a[0] = a[0].upper()
print(a)
```

To obtain `a = ('RED', 'green', 'blue')`, do the following:

- convert `a` to a list type and call it `b`.
- change `b[0]` to `RED`.
- change `b` back to a tuple type `a`.

In [ ]:
```
b = list(a)
None
print(a)
```

List Methods

Lists are objects, so they have a lot of methods available on them.

- `lst.append(item)` adds item to the end of list lst
- `lst.extend(item)` adds a sequence item to the list lst
- `lst.count(item)` returns the number of occurences of item in list lst
- `lst.index(item)` returns the index of the first occurence of item in list lst
- `lst.insert(index, item)` inserts item into list just before index
- `lst.pop()` removes the item in the list
- `lst.pop(i)` removes an item by index and returns the value
- `lst.remove(item)` removes first occurrence of item in the list
- `lst.reverse()` reverses the order of items in the list
- `lst.sort` sorts the list **in-place**

Example 1:

`append()` adds a single element to the end of a list:

In [ ]:
```
# add orange at the end
rainbow = ['red', 'green', 'blue']
None
print(rainbow)
```

Using an addition operator:

In [ ]:
```
rainbow = ['red', 'green', 'blue']
rainbow += ['orange']
print(rainbow)
```

Example 2:

add a sequence such as `[orange, black]`

If you instead want to add a sequence to the end of a list, then you can use `extend()`:

```
rainbow = ['red', 'green', 'blue']
rainbow = ['red', 'green', 'blue', 'orange', 'black']
```

Solution 1:

In [ ]:
```
# add [indigo, violet] at the end
rainbow = ['red', 'green', 'blue']
rainbow = ['red', 'green', 'blue', 'orange', 'black']
None
print(rainbow)
```

Note 1: This is the same as using **inplace addition**:

In [ ]:
```
rainbow = ['red', 'green', 'blue']
None
print(rainbow)
```

Note 2: Using `append()` with a list produces a differnt result than `extend()`:

In [ ]:
```
rainbow = ['red', 'green', 'blue']
rainbow.append(['orange', 'black'])
print(rainblow)
len(rainbow)
```

This adds a single element to the end of the list, and that element contains the thing we appended, in this case the list `['indigo', 'violet']` .

Example 3:

Count how many `red` in the list rainbow.

The `count()` method tells you how many of a certain element are in a list:

In [ ]:
```
rainbow = ['red', 'green', 'blue', 'green', 'blue', 'red']
rainbow.count('red')
```

Example 4:

Remove all elements after `yellow` in the list rainbow.

The `index()` method tells you the index of the first occurence of an element:

It produces the following

```
['red', 'orange', 'yellow']
```

In [ ]:
```
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'yellow', 'indigo', 'violet']
None
print(rainbow)
```

Notice that even though we have two 'yellow's we only get the index of the first one.

The `insert()` method inserts a new element into a list at a particular index.

Example 5:

Insert `pink` at the index = 2 in the list.
The `insert()` method inserts a new element into a list at a particular index.

It produces the following:

```
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo',
'violet']
rainbow = ['red', 'orange', 'pink', 'yellow', 'green', 'blue',
'indigo', 'violet']
```

Solution 1: Using `insert(index, value)`

In [ ]:
```
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
None
print(rainbow)
```

Solution 2: Using slicing

In [ ]:
```
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
None
print(rainbow)
```

Example 6:

Remove `yellow` in the list.
Use the `remove()` method to produce the following:

```
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo',
'violet']
rainbow = ['red', 'orange', 'green', 'blue', 'indigo', 'violet']
```

Solution 1: Using `remove(value)`

In [1]:
```
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
None
print(rainbow)
```

```
['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
```

Solution 2: Using slicing

In [ ]:
```
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
None
print(rainbow)
```

Example 7: Remove third element and print its value

Expected Output:

```
['red', 'orange', 'green', 'blue', 'indigo', 'violet']
yellow
```

The `pop()` method returns the value and removes an element from a list, but **by index** rather than by value.

In [ ]:
```
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
None
print(None)
```

Example 8: `sort()` : in-place sort

Lists have a `sort()` method which is quite powerful. The sort method takes various arguments that allow you to control how the sorting is done, and we'll cover that in another lecture. But if you just call it without any argument it will sort the elements in ascending order, which is fairly useful by itself:

```
a = [10, 1, 11, 13, 11, 2, 15, 14]
a = [1, 2, 10, 11, 11, 13, 14, 15]
```

In [ ]:
```
a = [10, 1, 11, 13, 11, 2, 15, 14]
None
print(a)
```

Note: Using function `sorted()`

```
a = [10, 1, 11, 13, 11, 2, 15, 14]
b = [10, 1, 11, 13, 11, 2, 15, 14]
c = [1, 2, 10, 11, 11, 13, 14, 15]
```

In [ ]:
```
a = [10, 1, 11, 13, 11, 2, 15, 14]
None
print(b)
print(c)
```

This is an **in-place sort**, so it actually modifies the list `a`. If you want to sort a list, but don't want to modify it, but instead create a new, sorted list, then the `sorted()` built-in function will do this for you. Some built-in functions such as sum, len, max can be used.

Example 9: reverse()

Finally, there is the `reverse()` method that reverses the elements of the list **in-place**:

Operations on Lists

We've seen the generic length function in Python and **it works on any kind of sequence**, including lists:

- `len()` - returns the length of an object
- `sorted()` - returns sorted list of an object
- `del` - deletes an item from a list by index
- `in` - tests for membership in a list
- `not in` - tests for membership in a list

Example 1:

Get the length of the list given:

In [ ]:
```
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
len(rainbow)
```

Example 2:

Use `sorted()` function to sort the sequence.

Note: A list also has the `sort()` method which performs the same way as `sorted()`. The only difference being, the `sort()` method doesn't return any value and changes the original list.

In [ ]:
```
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
rsorted = sorted(rainbow)
print(rainbow)
print(rsorted)
```

Example 3:

Use `sorted()` function to reverse the sequence.

In [ ]:
```
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
rsorted = sorted(rainbow, reverse = True)
print(rainbow)
print(rsorted)
```

Example 4:

Use `sorted()` to sort by its length

In [ ]:
```
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
rsorted = sorted(rainbow, key = len)
print(rainbow)
print(rsorted)
```

Example 5:

Test if `purple` is in `rainbow`.

A powerful feature is being able to test for membership in a list using the `in` keyword.

In [ ]:
```
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
'purple' in rainbow
```

You can invert the idea and test to see if something is `not in` the list:

In [ ]:
```
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
'purple' not in rainbow
```

The result is `True` since 'purple' is not in `rainbow`.

In the following code snippet, replace `None` with an logical expression such that it prints `True` since 0 is not in `a`.

In [1]:
```
a = [1, 2, 3, 4, 5, 6]
print(None)
```

```
True
```

Exercises

Swap neighbours

Given a list of numbers, swap adjacent items in pairs (`a[0]` with `a[1]` , `a[2]` with `a[3]` , etc.). Print the resulting list. If a list has an odd number of elements, leave the last element in place.

Sample Runs:

```
a = [1, 2, 3]
# your code here
print(a)

[2, 1, 3]
```

```
a = [0, 9, 4, 5, 2, 3]
# your code here
print(a)

[9, 0, 5, 4, 3, 2]
```

```
a = [0, 9, 4, 5, 2, 3, 1]
# your code here
print(a)

[9, 0, 5, 4, 3, 2, 1]
```

Hint: Believe it or not, mutiple assignments are possible. In the following code, two values of `x` and `y` are swapped.

```
x, y = 1, 2       # x = 1, y = 2
x, y = y, x       # x = 2, y = 1
```

In [ ]:
```
a = [0, 9, 4, 5, 2, 3]
for i in None:
    None
```

swap_two()

Make the functionality of 'swap neighbours' coded above into a function. There are two ways to implement this function.

- swap_two_inplace() in place returns None, but changes the input list itself in the argument
- swap_two() returns a new list without changing the input list in the argument

Sample Run 1:

```
if __name__ == '__main__':
    a = [0, 9, 4, 5, 2, 3, 1]
    b = swap_two_inplace(a)
    print("a:", a)
    print("b:", b)
```

Expected Output:

```
a: [9, 0, 5, 4, 3, 2, 1]
b: None
```

Sample Run 2:

```
if __name__ == '__main__':
    a = [0, 9, 4, 5, 2, 3, 1]
    b = swap_two(a)
    print("a:", a)
    print("b:", b)
```

Expected Output:

```
a: [0, 9, 4, 5, 2, 3, 1]
b: [9, 0, 5, 4, 3, 2, 1]
```

Sample Run 3:

```
if __name__ == '__main__':
    a = [1, 2, 3, 4, 5]
    b = swap_two(a)
    print("a:", a)
    print("b:", b)
```

Expected Output:

```
a: [1, 2, 3, 4, 5]
b: [2, 1, 4, 3, 5]
```

In [ ]:
```
# test 1
def swap_two_inplace(a):
    None
```

In [ ]:
```
# test 2
def swap_two(a):
    None
```

In [ ]:
```
# test 3
def swap_two(a):
    None
```

The largest element and its index

With a given list of integers, the function `largest()` returns the value of the largest element and the index number. If the largest element is not unique, print the index of the first instance.

Hint: You can solve this problem without using a loop if you use a method available for the list object as well as built-in functions in Python.

Sample Run:

```
if __name__ == '__main__':
    value, index = largest( [6, 9, 6, 23, 12, 19, 14, 26, 11] )
    print(value, index)
```

Explected Output:

```
26 7
```

In [ ]:
```
def largest(a):
    None

if __name__ == '__main__':
    a = [6, 9, 6, 23, 12, 19, 14, 26, 11]
    value, index = largest( a )
    print(value, index)
```

Unique elements

Given a list of integers, this function, `only_once(alist)`, returns a list with the elements that appear in the original list only once. The elements must be listed in the order in which they occur in the original list. Don't use `set`.

Sample Runs:

```
a = [4, 3, 5, 2, 5, 1, 3, 5]
print(only_once(a))
[4, 2, 1]

b = [6, 9, 6, 23, 12, 19, 14, 26]
print(only_once(b))
[9, 23, 12, 19, 14, 26]
```

Hint: Use a method available for a list object. Remember that `obejct.<tab>` displays a list of methods available for that `object`.

In [ ]:
```
def only_once(alist):
    b = None

    return b
```

In [ ]:
```
a = [4, 3, 5, 2, 5, 1, 3, 5]
b = [6, 9, 6, 23, 12, 19, 14, 26]
print(only_once(a))
print(only_once(b))
```

new_stairs()*

For a given positive integer N ≤ 9 print a stair of N steps. The k-th step consists of the integers from k to 1 without spaces between them. The result of new_stairs(8) produces the following:

Sample Run:

```
[1]
[1, 21]
[1, 21, 321]
[1, 21, 321, 4321]
[1, 21, 321, 4321, 54321]
[1, 21, 321, 4321, 54321, 654321]
[1, 21, 321, 4321, 54321, 654321, 7654321]
[1, 21, 321, 4321, 54321, 654321, 7654321, 87654321]
```

Hints: if your returns `stairs` at the end, it contains the last line of stairs. For new

- Need two for-loops, one for range(N) and the other for range(?,0, -1)
- At the end of two loops when stairs(8), `stairs[]` contains the follwoing:

```
[1, 21, 321, 4321, 54321, 654321, 7654321, 87654321]
```

In [ ]:

학습 정리

1. 슬라이싱 문법
2. 다양한 리스트 메소드
3. 리스트 작업 수행

참고자료

- Python Lists

슬기로운 자는 재앙을 보면 숨어 피하여도 어리석은 자들은 나가다가 해를 받느니라. 잠언27:12