Collections and generator

Python Programming

Byeongjoon Noh

powernoh@sch.ac.kr

Most of the slides are available on Senseable AI Lab homepage: https://sailab.space/courses/

Contents

1. Tuples
2. Lists
3. Sets
4. Dictionary
5. Generator

Textbook: Chapter 30, Chapter 31, Chapter 32, Chapter 33, Chapter 36

1. Tuples

Python collection types

- Four classes in Python that provide container
    - that is data types for holding collections of other objects
    - Tuples
    - Lists
    - Sets
    - Dictionary

What is a tuple?

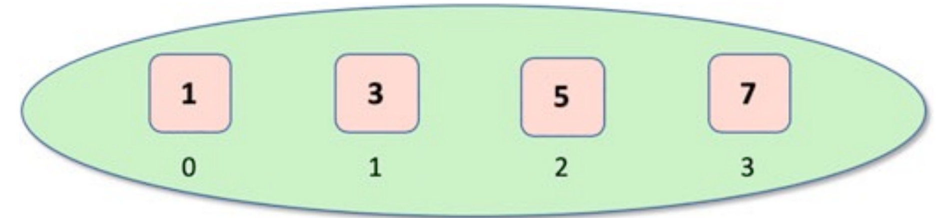
- An ordered collections of heterogeneous data, immutable and allows duplicates
  - ordered
    - part of sequence data types holding order of the data
    - maintaining the index for each item
  - immutable
    - cannot add or delete data to the tuple after creation
  - heterogeneous: allow to store variable of all types
    - a sequence of data of different data types (integer, float, list, string, etc.)
    - can be accessed through indexing and slicing
  - duplicates
    - can have items with the same values

Creating tuples

- Defined using parentheses (i.e. round brackets '( )') around the elements that make up the tuple

```
tup1 = (1, 3, 5, 7)
```

  - a new *Tuple* referenced by `tup1`
  - contains exactly 4 elements (in this case integers)
  - zero-based index




```
print('tup1[0]:\t', tup1[0])
print('tup1[1]:\t', tup1[1])
print('tup1[2]:\t', tup1[2])
print('tup1[3]:\t', tup1[3])
```

```
tup1[0]:	1
tup1[1]:	3
tup1[2]:	5
tup1[3]:	7
```

Creating tuples

- Various tuple declaration and initialization

```
tup1 = (1, 3, 5, 7) # General initialization

empty_tuple = () # An empty tuple

mixed_tuple = (1, "Hello", 3.14) # A tuple with mixed data types

single_element_tuple = (5,) # A single element tuple (note the comma)
```

Accessing elements of a tuple

- Can be accessed using an index in square brackets '[ ]'
  - the index returns the object at that position
- Slice from tuple; comprised of a subset of the original tuple
  - syntax

```
my_tuple[start:stop:step]
```

- `start`: the index where the slice start (inclusive); default to the beginning of tuple if omitted
- `stop`: the index where the slice end (exclusive); default to the end of tuple if omitted
- `stop`: the step size or the interval between each element in the slice; default to 1

  - basic slicing

```
tup1 = (1, 3, 5, 7, 9, 11)
print(tup1[1:3])
```

```
(3, 5)
```

Accessing elements of a tuple

- Slice from tuple
  - omitting start and stop

```
tup1 = (1, 3, 5, 7, 9, 11)
print(tup1[:3])
print(tup1[1:])
```

```
(1, 3, 5)
(3, 5, 7, 9, 11)
```

  - using negative indices

```
tup1 = (1, 3, 5, 7, 9, 11)
print(tup1[-4:-1])
print(tup1[-3::-1])
print(tup1[:-3:-1])
```

```
(5, 7, 9)
(7, 5, 3, 1)
(11, 9)
```

Accessing elements of a tuple

- Slice from tuple
  - specifying steps

```
tup1 = (1, 3, 5, 7, 9, 11)
print(tup1[0:5:2])
print(tup1[6:2:-2])
```

```
(1, 5, 9)
(11, 7)
```

- Reverse trick using `::-1`

```
tup1 = (1, 3, 5, 7, 9, 11)
print(tup1[::-1])
```

```
(11, 9, 7, 5, 3, 1)
```

Characteristics

- Immutable

```
tup1 = (1, 3, 5, 7, 9, 11)
tup1[1] = 9
```

```
Traceback (most recent call last):
  File "C:\Users\#2 파이썬프로그래밍\src\hello_world.py", line 2, in <module>
    tup1[1] = 9
    ~~~~^^^
TypeError: 'tuple' object does not support item assignment
```

- Holding different types
  - not restricted to holding elements all of the same type

```
tup2 = (1, 'John', Person('Phoebe', 21), True, -23.45)
print(tup2)
```

```
(1, 'John', <__main__.Person object at 0x000002425434FD90>, True, -23.45)
```

The tuple() constructor function

- To create a new tuple from an iterable
  - * iterable: an object capable of returning its members one at a time
    - allows it to be used in a loop
    - e.g. set, list, dictionary, also tuple

```
list1 = [1, 2, 3]
t1 = tuple(list1)
print(t1)
```

```
(1, 2, 3)
```

Iterating over tuples

- Can iterate over the contents of a tuple
  - used for the value to which the loop variable will be applied:

```
tup3 = ('apple', 'pear', 'orange', 'plum', 'apple')
for x in tup3:
    print(x)
```

```
apple
pear
orange
plum
apple
```

  - exactly same result as follows:

```
for i in range(0, len(tup3)):
    print(tup3[i])
```

Tuple-related functions and methods

- There are several built-in functions and methods as simple but versatile data structure
  - function: tuple is used as parameter of function
    - `len(Tuple)`, `max(Tuple)`, `min(Tuple)`, `sum(Tuple)`, `sorted(Tuple)`, `Tuple(iterable)`, etc.
  - methods: tuple can call the pre-defined method
    - `Tuple.count(value)`, `Tuple.index(value)`, etc.

Tuple-related functions and methods

- Built-in tuple functions
  - `len(Tuple)`: returns the number of elements in the tuple

```
my_tuple = (1, 2, 3)
print(len(my_tuple))  # Output: 3
```

  - `max(Tuple)` and `min(Tuple)`: returns the largest or smallest element in the tuple

```
my_tuple = (1, 3, 5, 1, 2, 9)
print(max(my_tuple))  # Output: 9
print(min(my_tuple))  # Output: 1
```

  - `sum(Tuple)`: returns the sum of the elements in the tuples; elements must be numbers

```
my_tuple = (1, 3, 5, 1, 2, 9)
print(sum(my_tuple))  # Output: 21
```

Tuple-related functions and methods

- Built-in tuple functions
  - `sorted(Tuple)`: returns a new **list** containing all elements of the tuple in ascending order

```
my_tuple = (1, 3, 5, 1, 2, 9)
print(sorted(my_tuple))  # Output: [1, 1, 2, 3, 5, 9]
```

    - all elements should be the same type (numbers or string)

```
my_tuple = ('a', 1, 'b', 'c')
print(sorted(my_tuple))  # Output: TypeError
```

```
my_tuple = (1, 2, 3, 4.5, -4.7, -3, 0)
print(sorted(my_tuple))  # Output: [-4.7, -3, 0, 1, 2, 3, 4.5]
```

```
my_tuple = ('hi', 'my', 'is', 'an', 'apple')
print(sorted(my_tuple))  # Output: ['an', 'apple', 'hi', 'is', 'my']
```

Tuple-related functions and methods

- Tuple methods
  - `Tuple.count(value)`: returns the number of occurrences of the specified value

```
my_tuple = (1, 2, 2, 3)
print(my_tuple.count(2))  # Output: 2
```

```
my_tuple = ('hi', 'my', 'is', 'an', 'apple')
print(my_tuple.count('a'))  # Output: 0
```

  - `Tuple.index(value)`: returns the index of the **first** occurrence of the specified value
    - raise a value error if the value is not present

```
my_tuple = (1, 2, 3)
print(my_tuple.index(2)) # Output: 1
```

- Tuples, being immutable, do not have methods for adding or removing elements
  - e.g. append, extend, insert, remove, or pop

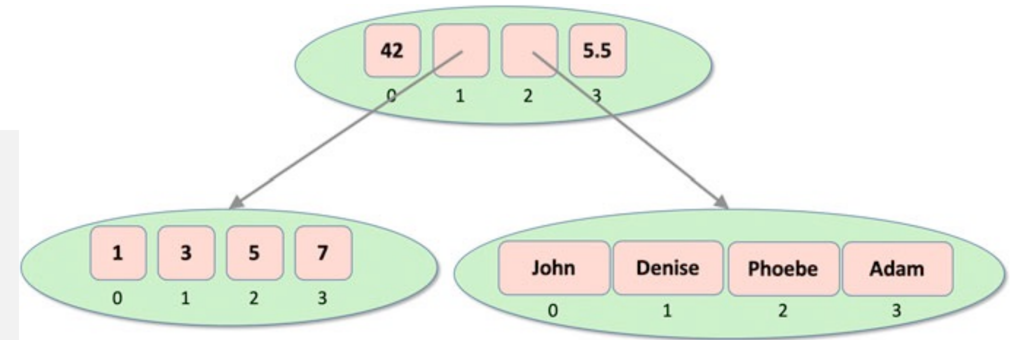
Tips

- Checking if an element exists in the tuple
  - use if statement with in operator

```
tup3 = ('apple', 'banana', 'orange')
if 'orange' in tup3:
    print('orange is in the Tuple')
```

- Nested tuples
  - tuple can contain tuples as elements

```
tuple1 = (1, 3, 5, 7)
tuple2 = ('John', 'Denise', 'Phoebe', 'Adam')
tuple3 = (42, tuple1, tuple2, 5.5)
print(tuple3)
```




```
(42, (1, 3, 5, 7), ('John', 'Denise', 'Phoebe', 'Adam'), 5.5)
```

Quiz

- What is the output of the following code?

```
tup1 = (1, "Hello", 3.14)
print(tup1[3])
```

Quiz

- What is the output of the following code?

```
tup = (10, 25, 4, 12, 3, 8)
sorted(tup)
print(tup)
```

  - a) (10, 25, 4, 12, 3, 8)
  - b) (3, 4, 8, 10, 12, 25)
  - c) (25, 12, 10, 8, 4, 3)
  - d) (3, 4, 8, 10, 12, 25)

    (3, 4, 8, 10, 12, 25)

Quiz

- What is the output of the following code?

```
my_tuple = (5, 12, 19, 3, 25)
tup = my_tuple[-2::-2]
print(tup)
```

  - a) (3, 12)
  - b) (25, 5)
  - c) (12, 3)
  - d) Error

2. Lists

What is lists?

- Mutable, ordered collection of data
  - these items can be of any data types, and a single list can contain items of different types, including other lists
  - mutable
    - list can be altered; items can be added, removed, or changed unlike the tuple
  - ordered
    - order of elements in a list is maintained, meaning items have a defined order
  - dynamic size
    - can grow and shrink in size as needed
  - heterogeneous elements
    - can contain elements of different types, including mixed types in the same list

Creating lists

- General syntax

```
my_list = [item1, item2, item3, ...]
```

  - created using square brackets positioned around the elements that make up list

- List creation

```
list1 = ['John', 'Paul', 'George', 'Ringo']
```

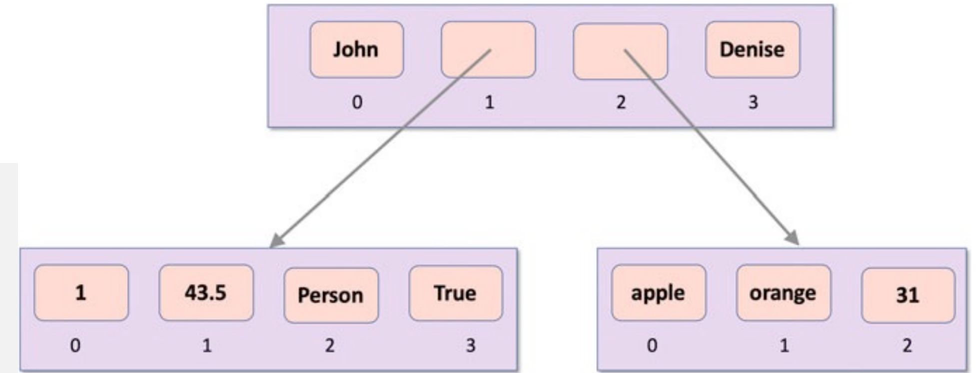
  - zero-based indexing

Creating lists

- Nested lists

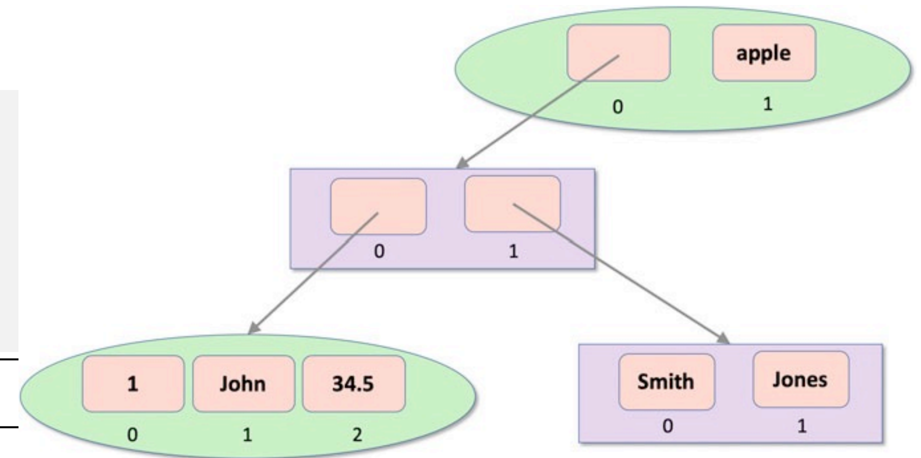
```
l1 = [1, 43.5, Person('Phoebe', 21), True]
l2 = ['apple', 'orange', 31]
root_list = ['John', l1, l2, 'Denise']
print(root_list)
```

```
['John', [1, 43.5, <__main__.Person object at 0x000002B0AC1FF250>, True], ['apple', 'orange', 31], 'Denise']
```




```
t1 = (1, 'John', 34.5)
l1 = ['Smith', 'Jones']
l2 = [t1, l1]
t2 = (l2, 'apple')
print(t2)
```

```
([(1, 'John', 34.5), ['Smith', 'Jones']], 'apple')
```



Creating lists

- List comprehension
  - syntax

```
[expression for item in iterable if condition]
```

    - ‘expression’: an expression to evaluate and add to the list
    - ‘iterator’: the ‘for’ statement for iterating over each element in the iterable
    - ‘condition’: the ‘if’ statement is a filter that will include the item in the output list if the condition evaluates to ‘True’

Creating lists

- Example of the list comprehension
  - generating a list of squares of numbers from 0 to 9:

```
squares = [x**2 for x in range(10)]
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

  - generating a list of squares of even numbers from 0 to 9:

```
even_squares = [x ** 2 for x in range(10) if x % 2 == 0]
print(even_squares)
```

```
[0, 4, 16, 36, 64]
```

Creating lists

- Example of the list comprehension
  - filtering a list to exclude negative numbers:

```
numbers = [1, -2, 3, -4, 5]
non_negative_numbers = [x for x in numbers if x >= 0]
print(non_negative_numbers )
```

```
[1, 3, 5]
```

  - creating a list of uppercase characters from a string:

```
s = "hello"
uppercase_chars = [char.upper() for char in s]
print(uppercase_chars)
```

```
['H', 'E', 'L', 'L', 'O']
```

Creating lists

- Example of the list comprehension
  - using nested list comprehensions

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = [num for row in matrix for num in row]
print(flattened)
```

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

Accessing elements from a list

- Can access elements from a list using an index

```
my_list = ['John', 'Paul', 'George', 'Ringo']
print(my_list[1]) # Output: Paul
```

- Slicing
  - basic slicing

```
my_list = ['John', 'Paul', 'George', 'Ringo', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(my_list[1:5])  # Output: ['Paul', 'George', 'Ringo', 0]
```

  - slicing with negative indices

```
print(my_list[-4:]) # Output: [6, 7, 8, 9]
```

  - slicing with step

```
print(my_list[::2])  # Output: ['John', 'George', 0, 2, 4, 6, 8]
print(my_list[1:8:3]) # Output: ['Paul', 0, 3]
```

List-related functions and methods

- There are several built-in functions and methods as simple but versatile data structure
  - function: set is used as parameter of function
    - `len(List)`, `max(List)`, `min(List)`, `sum(List)`, `List(iterable)`, etc.
  - methods: set can call the pre-defined method
    - `List.append(element)`, `List.extend(element)`, `List.insert(index, element)`, `List.remove(element)`, `List.pop(element)`, `List.clear()`, `List.index(value)`, `List.count(element)`, `List.sort(element)`, `List.reserve()`, `List.copy()`, etc.

List-related functions and methods

- Built-in list functions
  - `len(List)`: returns the number of elements in the list

```
my_list = [1, 2, 3]
print(len(my_list))  # Output: 3
```

  - `max(List)` and `min(List)`: returns the largest or smallest element in the list

```
my_ list = [1, 3, 5, 1, 2, 9]
print(max(my_list))  # Output: 9
print(min(my_list))  # Output: 1
```

  - `sum(List)`: returns the sum of the elements in the lists; elements must be numbers

```
my_list = [1, 3, 5, 1, 2, 9]
print(sum(my_list))  # Output: 21
```

List-related functions and methods

- Adding elements
  - `List.append(element)`: add an element to the end of the list

```
fruits = ["apple", "banana"]
fruits.append("cherry")
print(fruits)  # Output: ["apple", "banana", "cherry"]

fruits = ["apple", "banana"]
fruits.append(["cherry", "orange"])
print(fruits)  # Output: ['apple', 'banana', ['cherry', 'orange']]
```

List-related functions and methods

- Adding elements
  - `List.extend(element)`: adds all elements of an iterable (e.g. another list) to the end

```
numbers = [1, 2, 3]
numbers.extend([4])
print(numbers)  # Output: [1, 2, 3, 4]

numbers = [1, 2, 3]
numbers.extend([4, 5, 6])
print(numbers)  # Output: [1, 2, 3, 4, 5, 6]
```

List-related functions and methods

- Adding elements
  - `` `List.insert(index, element)` ``: insert an element at a specified index

```
numbers = [1, 3, 4]
numbers.insert(1, 2)
print(numbers)  # Output: [1, 2, 3, 4]

numbers = [1, 3, 4]
numbers.insert(1, [5, 6, 7, "orange"])
print(numbers)  # Output: [1, [5, 6, 7, 'orange'], 3, 4]
```

List-related functions and methods

- Removing elements
  - `List.remove(element)`: removes the first occurrence of an element

```
animals = ["dog", "cat", "rabbit", "cat"]
animals.remove("cat")
print(animals)  # Output: ["dog", "rabbit", "cat"]
```

List-related functions and methods

- Removing elements
  - `List.pop(index)`: removes and return an element at index
    - the last item if index is not provided

```
letters = ['a', 'b', 'c', 'd']
popped = letters.pop(2)
print(letters)  # Output: ['a', 'b', 'd']
print(popped)   # Output: 'c'

letters.pop()
print(letters)  # Output: ['a', 'b']
```

List-related functions and methods

- Removing elements
  - `List.clear()`: removes all elements from the list

```
items = [1, 2, 3]
items.clear()
print(items)  # Output: []
```

List-related functions and methods

- Other operations elements
  - `List.index(element)`: returns the index of the first occurrence of an element

```
numbers = [10, 20, 30, 40, 30]
index = numbers.index(30)
print(index)  # Output: 2
```

List-related functions and methods

- Other operations elements
  - `List.count(element)`: counts the number of occurrences of an element in the list

```
letters = ['a', 'b', 'a', 'c', 'a']
count = letters.count('a')
print(count)  # Output: 3
```

List-related functions and methods

- Other operations elements
  - `List.sort()`: sort the list in place
  - `List.reverse()`: reverses the list in place

```
numbers = [3, 1, 4, 1, 5]
numbers.sort()
print(numbers)  # Output: [1, 1, 3, 4, 5]

numbers = [1, 2, 3, 4, 5]
numbers.reverse()
print(numbers)  # Output: [5, 4, 3, 2, 1]
```

List-related functions and methods

- Other operations elements
  - `List.copy()`: returns copy of the list
    - types of copy
      - shallow copy: creates a new list containing references to the original list's items

```
copy = original
original.append(1)
print(copy) # Output: [1, 2, 3, 1]
print(original) # Output: [1, 2, 3, 1]
```

      - deep copy: creates a completely independent copy of the original list

```
original = [1, 2, 3]
copy = original.copy()
copy.append(1)
print(copy)  # Output: [1, 2, 3, 1]
print(original) # Output: [1, 2, 3]
```

List operations

- Concatenation
  - combines two or more lists into one

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
concatenated_list = list1 + list2
print(concatenated_list)  # Output: [1, 2, 3, 4, 5, 6]
```

- Repeating

```
list1 = ["a", "b", "c"]
repeated_list = list1 * 3
print(repeated_list)  # Output: ['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']
```

- Membership testing

```
my_list = [1, 2, 3, 4, 5]
print(3 in my_list)  # Output: True
print(6 in my_list)  # Output: False
```

List operations

- Iterating
  - iterates over each item in the list in loop operation

```
my_list = ['apple', 'banana', 'cherry']
for fruit in my_list:
    print("I like", fruit)
```

```
I like apple
I like banana
I like cherry
```

Quiz

- What is the output of the following code?

```
s = '76543'
print(list(s))
```

  - a) ['7', '6', '5', '4', '3']
  - b) [‘76543’]
  - c) Error
  - d) [‘7’]

Quiz

- What is the output of the following code?

```
lst1 = [1, 2, 4, 3]
lst2 = [1, 2, 3, 4]
print(lst1 != lst2)
```

  - a) False
  - b) True
  - c) Error
  - d) [True, True, False, False]

Quiz

- What is the output of the following code?

```
i=j=[3]
i+=j
print(i, j)
```

  - a) [3]
  - b) Error
  - c) [3, 3], [3, 3]
  - d) [3, 3, 3, 3]

Quiz

- What is the output of the following code?

```
lst = [0, 1]
[lst.append(lst[k - 1] + lst[k - 2]) for k in range(2, 5)]
print(lst)
```

  - a) Error
  - b) [1, 2, 3, 4, 5]
  - c) [0, 1, 1, 2, 3]
  - d) [1, 2, 3, 4, 5]

Quiz

- What is the output of the following code?

```
numbers = [12, 5, 8, 13, 4]
print(numbers[::2])
```

  - a) [12, 8, 4]
  - b) [5, 13]
  - c) [12, 8]
  - d) [4, 13, 8, 5, 12]

Quiz

- What is the output of the following code?

```
msg = 'programming'
s = list(msg[:4])[::-1]
print(s)
```

  - a) [‘p’, ‘r’, ‘o’, ‘g’]
  - b) ‘prog’
  - c) [‘p’, ‘r’, ‘o’, ‘g’, ‘r’]
  - d) ‘rgorp’

Quiz

- What is the output of the following code?

```
k = [2, 1, 0, 3, 0, 2, 1, 0]
print(k.count(k.index(0)))
```

  - a) 3
  - b) 2
  - c) 1
  - d) 0

2-dimensional list

- 2-dimensional list
  - list of lists; each element in the main list is another list representing a row of the 2D list

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

  - accessing an entire row: `matrix[row]`
  - accessing a specific element: `matrix[row][col]`

```
second_row = matrix[1]  # Output: [4, 5, 6]
element = matrix[1][2]  # Output: 6
```

2-dimensional list

- Iterating over 2-d list

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

# Iterate over rows
for row in matrix:
    print(row)

# Iterate over each element
for row in matrix:
    for element in row:
        print(element)
```

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

N-dimensional list

- N-dimensional list is a list of lists of lists… (N levels)
  - 3-d list could be a list of matrices, and 4-d list could be a list of 3-d list, and so on
  - VERY common use in data science, machine learning, and scientific computing; a.k.a **tensor** strucuture

```
three_d_list = [
    [[1, 2], [3, 4]],
    [[5, 6], [7, 8]],
    [[9, 10], [11, 12]]
]
```

N-dimensional list

- Accessing n-dimensional list is a list of lists of lists… (N levels)
  - access 2-d list: `three_d_list[index]`
  - access row: `three_d_list[index][row]`
  - access a specific elementt: `three_d_list[index][row][col]`

```
# Access the first 2D list
first_matrix = three_d_list[0]  # Output: [[1, 2], [3, 4]]

# Access the first row of the first 2D list
first_row = three_d_list[0][0]  # Output: [1, 2]

# Access the first element of the first row of the first 2D list
element = three_d_list[0][0][0]  # Output: 1
```

Various form of the nested lists

- List of lists of varying lengths
  - demonstrates that inner lists can have different sizes in Python

```
varying_lengths = [[1], [2, 3], [4, 5, 6]]
```

- 3D list (list of lists of lists)
  - represents a 3-dimensional data structure

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

- Nested lists as a table of information
  - can be used to store tabular data (e.g., spreadsheet)

```
table = [["Name", "Age", "City"], ["Alice", 30, "New York"], ["Bob", 25, "Los Angeles"]]

complex_structure = [
  ["Header", [1, 2, 3]],
  ["Data", [[True, False, True], [False, True, True]]],
  ["Footer", ["End"]]
]
```

Various form of the nested lists

- Nested list comprehension
  - creating matrix

```
matrix = [[row + col for col in range(4)] for row in range(3)]
print(matrix)
```

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

  - flattening a nested list

```
nested_list = [[1, 2, 3], [4, 5], [6, 7]]
flat_list = [item for sublist in nested_list for item in sublist]
print(flat_list)
```

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

  - generating a multiplication table

```
multiplication_table = [[i * j for j in range(1, 6)] for i in range(1, 6)]
print(multiplication_table)
```

```
[[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20], [5, 10, 15, 20, 25]]
```

Various form of the nested lists

- Nested list comprehension
  - creating a 3D list

```
three_d_list = [[[i + j + k for k in range(3)] for j in range(3)] for i in range(3)]
print(three_d_list)
```

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

Quiz

- What is the output of the following code?

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(matrix[1][2])
```

  - a) 5
  - b) 6
  - c) 7
  - d) Error

Quiz

- What is the output of the following code?

```
lst = ["programming", [4, 8, 12, 16]]
print(lst[1][3])
```

  - a) 4
  - b) 8
  - c) ‘o’
  - d) Error

Quiz

- What is the output of the following code?

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(matrix[0][:])
```

  - a) [1, 2, 3]
  - b) 1 2 3
  - c) [1]
  - d) [1, 4, 7]

  - how about this

```
print(matrix[:][0])
```

Quiz

- What is the output of the following code?

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(matrix[1:])
```

  - a) [[4, 5, 6], [7, 8, 9]]
  - b) [4, 5, 6], [7, 8, 9]
  - c) [1, 2, 3]
  - d) [4, 5, 6]

Quiz

- What is the output of the following code?

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(matrix[1:][1:2])
```

  - a) [7, 8, 9]
  - b) 7, 8, 9
  - c) [[7, 8, 9]]
  - d) [[4, 5, 6], [7, 8, 9]]

Quiz

- What is the output of the following code?

```
lst = [[8, 7], [6, 5]]
result = [p +q for p, q in lst]
print(result)
```

  - a) [8, 5]
  - b) [14, 12]
  - c) [15, 11]
  - d) [7, 6]

Quiz

- What is the output of the following code?

```
def f1(a, b=[]):
    b.append(a)
    return b
print(f1(2, [3, 4]))
```

  - a) [2, 3, 4]
  - b) [3, 4, 2]
  - c) [3, 2, 4]
  - d) [4, 3, 2]

In class practice

- P05-01 두 행렬의 곱셈을 수행하는 함수를 작성해여라
  - input: two NxN matrix
  - output: result of matrix multiplication
  - requirement: check the sizes of two matrix if multiplication is possible

```
def matrix_multiplication(matrixA, matrixB):
    ''' CODE HERE '''

    return result

if __name__ == '__main__':
    matrixA = [[1, 2, 3], [3, 4, 5], [5, 6, 7]]
    matrixB = [[5, 6, 7], [7, 8, 9], [9, 10, 11]]

    result = matrix_multiplication(matrixA, matrixB)

    print("\nResult of matrix multiplication:")
    for row in result:
        print(row)
```

```
Result of matrix multiplication:
[46, 52, 58]
[88, 100, 112]
[130, 148, 166]
```

Note

- 행렬을 사용자로부터 입력받기

```
matrixA = []

print("3x3 행렬의 각 행을 입력하세요 (각 숫자는 공백으로 구분):")
for i in range(3):
    row = list(map(int, input().split()))
    matrixA.append(row)

print("입력받은 행렬:")
for row in matrixA:
    print(row)
```

```
input_str = input("전체 3x3 행렬을 입력하세요 (행은 ';'로 구분, 숫자는 공백으로 구분): ")

rows = input_str.split(';’)

matrixA = [list(map(int, row.split())) for row in rows]

print("입력받은 행렬:")
for row in matrixA:
    print(row)
```

3. Sets

What is a set?

- an unordered collection of unique and immutable items
  - the presence of each element is more important than the order or frequency of elements
  - unordered
    - the elements in a set do not maintain any order
  - mutable
    - set themselves are mutable; can add or remove items from a set
  - unique elements
    - each element in a set is distinct; duplicates are not allowed
  - immutable elements
    - elements of a set must be of an immutable type; **numbers, strings, tuples, booleans, None type**
  - Not indexable
    - do not support indexing or slicing

Creating sets

- General syntax

```
my_set = {element1, element2, element3, ...}
```

  - using the `set()` for creating an empty set or converting from other iterables

```
my_set = set([element1, element2, element3, ...])
```

- Set creation

```
my_set = {1, 2, 3}
print(my_set) # Output: {1, 2, 3}
my_set = {1, 2, 3, 4, 5, 5, 5}
print(my_set) # Output: {1, 2, 3, 4, 5}
```

  - Note: elements within a set must be of an immutable data type
    - integers, floats, strings, tuples, booleans, and None type

Iterating over sets

- Can iterate over the contents of a set
  - used for the value to which the loop variable will be applied:

```
my_set = {1, 2, 3, 4, 5, 'a'}
for x in my_set:
    print(x)
```

```
apple
pear
orange
plum
apple
```

Set-related functions and methods

- There are several built-in functions and methods as simple but versatile data structure
  - function: set is used as parameter of function
    - `len(Set)`, `max(Set)`, `min(Set)`, `sum(Set)`, `sorted(Set)`, `Set(iterable)`, etc.
  - methods: set can call the pre-defined method
    - `Set.add(element)`, `Set.remove(element)`, `Set.discard(element)` , `Set.pop()` , `Set.intersection(set1, set2, …)` , `Set.union(set1, set2, …)` , `Set.symmetric_difference(set)`, etc.

Set-related functions and methods

- Adding elements
  - `Set.add(element)`: adds an element to the set

```
my_set = {1, 2, 3}
my_set.add(4)
print(my_set)  # Output: {1, 2, 3, 4}
```

  - `Set.update(iterable)`: adds an elements from an iterable (e.g. list, tuple) to the set

```
my_set = {1, 2, 3}
print(my_set) # Output: {1, 2, 3}
my_list = ['a', 'b', 'c']
my_set.update(my_list) # Output: {1, 2, 3, 4, 'a', 'b', 'c'}
```

Set-related functions and methods

- Removing elements
  - `Set.remove(element)`: removes an element from the set
    - raises KeyError if the element is note present

```
my_set = {1, 2, 3}
my_set.remove(2)
print(my_set) # Output {1, 3}
```

Set-related functions and methods

- Removing elements
  - `` `Set.discard(element)` ``: removes an element from the set if it is a member
    - if the element is not a member, do nothing

```
my_set = {1, 2, 3, 4}
my_set.discard(2)
my_set.discard(99)  # This will not raise an error
print(my_set)  # Output: {1, 3, 4}
```

Set-related functions and methods

- Removing elements
  - `Set.pop()`: removes and returns and arbitrary set element
    - raises KeyError if the set is empty

```
my_set = {2, 1, 3, 4, 'a'}
print(my_set) # Output: {1, 2, 3, 4}
my_set.pop()
print(my_set) # Output: {2, 3, 4, 'a'}
```

  - `Set.clear()`: removes all elements from the set

Set-related functions and methods

- Set operations
  - `Set.union(set1, set2, …)`: returns a new set with elements from the set and all others

```
set_a = {1, 2, 3}
set_b = {4, 5, 6}
set_union = set_a.union(set_b)
print(set_union)  # Output: {1, 2, 3, 4, 5, 6}
print(set_a) # Output: {1, 2, 3}

set_a = {1, 2, 3}
set_b = {4, 5, 6}
set_c = {-1, 0, 3, 5}
set_union = set_a.union(set_b, set_c)
print(set_union)  # Output: {0, 1, 2, 3, 4, 5, 6, -1}
```

Set-related functions and methods

- Set operations
  - `Set.intersection(set1, set2, …)`: returns a new set with elements common to the set and all others

```
set_a = {1, 2, 3}
set_b = {2, 3, 5}
set_inter = set_a.intersection(set_b)
print(set_inter) # Output: {2, 3}
print(set_a) # Output: {1, 2, 3}
```

Set-related functions and methods

- Set operations
  - `` `Set.difference(set1, set2, …)` ``: returns a new set with elements in the set that are not in the others

```
set_a = {1, 2, 3}
set_b = {2, 3, 5}
set_diff = set_a.difference(set_b)
print(set_diff) # Output: {1}
print(set_a) # Output: {1, 2, 3}
```

Set-related functions and methods

- Set operations
  - `Set.symmetric_difference(set1)`: returns a new set with elements in either the set or the other but not both

```
set_a = {1, 2, 3}
set_b = {2, 3, 5}
set_sym_diff = set_a.symmetric_difference(set_b)
print(set_sym_diff) # Output: {1, 5}
```

Tips

- Set membership testing

```
print(3 in set_a)  # Output: True
print(7 in set_a)  # Output: False
```

- Set comprehensions

```
squared_set = {x**2 for x in range(10)}
print(squared_set)  # Output: {0, 1, 4, 9, 16, 25, 36, 49, 64, 81}
```

Quiz

- What is the output of the following code?

```
lst = [2, 7, 8, 6, 5, 5, 4, 4, 8]
s = {True if n % 2 == 0 else False for n in lst}
print(s)
```

  - a) Error
  - b) {False, True}
  - c) {True, True}
  - d) {False}

Quiz

- What is the output of the following code?

```
s = {1, 3, 7, 6, 5}
s.discard(4)
print(s)
```

  - a) {1, 3, 7, 5}
  - b) {1, 3, 5, 6, 7}
  - c) {1, 3, 7, 6, 5, 4}
  - d) {1, 3, 7, None, 5}

Quiz

- What is the output of the following code?

```
s1 = {1, 2, 3}
s2 = {2, 3, 4}
print(s2 - s1)
```

  - a) {2, 3}
  - b) {4}
  - c) {2, 3, 4}
  - d) {1}

4. Dictionary

What is a dictionary?

- a collection of key-value pairs
  - each key value pair maps the key to its associated vaule
  - key-value storage
    - `'{key1: value1, key2: value2, ...}’`
  - unordered, dynamic, nested
  - mutable
    - key-value pairs can be added removed, and changed
  - keys must be immutable
    - e.g. strings, numbers, tuples with immutable elements
  - value can be of any type
  - indexed by **key**

Creating dictionaries

- General syntax

```
my_dict = {'key1': 'value1', 'key2': 'value2'}
```

  - using the `dict()` for creating an empty dictionary

- Dictionary creation
  - general creation

```
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
```

  - using `zip()` function for key and value lists

```
keys = ['name', 'age', 'city']
values = ['John', 30, 'New York']
my_dict = dict(zip(keys, values))
```

Accessing values

- Use `key` as index to obtain `value`

```
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
print(my_dict['name']) # Output: John
```

  - Adding or changing values

```
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
my_dict['age'] = 31
my_dict['height'] = 185.3
print(my_dict)
```

```
{'name': 'John', 'age': 31, 'city': 'New York', 'height': 185.3}
```

- `Dictionary.clear()`: removes all items from the dictionary

```
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
my_dict.clear()
print(my_dict)
```

```
{}
```

Key methods

- `Dictionary.get(key, default=None)`: returns the value for `key` if `key` is in the dictionary, else `default`
- `Dictionary.keys()`: returns a view object displaying a list of all the keys
- `Dictionary.values()`: returns a view object displaying a list of all the values
- `Dictionary.items()`: returns a view object displaying a list of the dictionary's key-value tuple pairs

```
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}

print(my_dict.get('name'))  # Output: 'John'
print(my_dict.get('location', 'Not Available'))  # Output: 'Not Available'

# Getting all keys, values, and items
print(list(my_dict.keys()))  # Output: ['name', 'age', 'city']
print(list(my_dict.values()))  # Output: ['John', 30, 'New York']
print(list(my_dict.items()))  # Output: [('name', 'John'), ('age', 30), ('city', 'New York')]
```

Key methods

- Useful in the loop statement
  - Usage of `keys()`

```
grades = {
    'Alice': 85,
    'Bob': 93,
    'Charlie': 87,
    'Diana': 78
}

for student in grades.keys():
    print(f"{student} scored {grades[student]}")
```

```
Alice scored 85
Bob scored 93
Charlie scored 87
Diana scored 78
```

Key methods

- `` `Dictionary.update(others)` ``: updates the dictionary with the key-value pairs from `` `other` `` overwriting existing keys

```
student = {
    'name': 'Alex',
    'age': 21,
    'major': 'Computer Science'
}

student.update({'graduation_year': 2024})
student.update({'age': 22, 'grade': 'A'})
print(student)
```

```
{'name': 'Alex', 'age': 21, 'major': 'Computer Science'}
{'name': 'Alex', 'age': 22, 'major': 'Computer Science', 'graduation_year': 2024, 'grade': 'A'}
```

Key methods

- `` `Dictionary.pop(key)` ``: remove the specific key and returns the corresponding value

  if key is not found, `` `default` `` is returned if given, otherwise KeyError is raised

- `` `Dictionary.popitem()` ``: remove and returns a (key, value) pair as 2-tuple; returned in LIFO order
  - LIFO: Last-In-First-Out

```
employee = {
    'name': 'John Doe',
    'position': 'Software Engineer',
    'email': 'john.doe@example.com'
}

removed_item = employee.popitem()

print(removed_item) # Output: ('email', 'john.doe@example.com')
print(employee) # Output: {'name': 'John Doe', 'position': 'Software Engineer'}

position = employee.pop('position')
print(position)  # Output: 31
print(employee)  # Output: {'name': 'John', 'email': 'john@example.com'}
```

Tips

- List of dictionaries

```
students = [
    {'name': 'Alice', 'grade': 'A', 'major': 'Engineering'},
    {'name': 'Bob', 'grade': 'B', 'major': 'History'},
    {'name': 'Charlie', 'grade': 'C', 'major': 'Biology'}
]

for student in students:
    print(f"{student['name']} is a {student['major']} major and grade is {student['grade']}.")
```

```
Alice is a Engineering major and grade is A.
Bob is a History major and grade is B.
Charlie is a Biology major and grade is C.
```

Tips

- Dictionary of lists

```
courses = {
    'Engineering': ['Alice', 'Dan', 'Eve'],
    'History': ['Bob', 'Frank'],
    'Biology': ['Charlie', 'Grace']
}

for course, students in courses.items():
    print(f"The following students are enrolled in {course}: {', '.join(students)}")
```

```
The following students are enrolled in Engineering: Alice, Dan, Eve
The following students are enrolled in History: Bob, Frank
The following students are enrolled in Biology: Charlie, Grace
```

Tips

- List within a dictionary within a list

```
conference = [
    {
        'name': 'TechCon',
        'topics': ['AI', 'Big Data', 'Cloud Computing'],
        'attendees': 1200
    },
    {
        'name': 'HealthSymposium',
        'topics': ['Biotech', 'Genetics', 'Bioinformatics'],
        'attendees': 800
    }
]

for event in conference:
    print(f"{event['name']} covers topics such as {', '.join(event['topics'])} with
{event['attendees']} attendees.")
```

```
TechCon covers topics such as AI, Big Data, Cloud Computing with 1200 attendees.
HealthSymposium covers topics such as Biotech, Genetics, Bioinformatics with 800 attendees.
```

Practical usage

- Storing data
  - dictionaries are ideal for storing data that can be logically connected with a unique key
- JSON
  - closely resemble the JSON format and can be directly converted to and from JSON
- Hash table
  - internally, dictionaries are implemented as has tables

Quiz

- What is the output of the following code?

```
my_dict = {'a':1, 'b':2, 'c':3}
print(my_dict['a'] + my_dict['b'])
```

  - a) 12
  - b) 3
  - c) ‘ab’
  - d) Error

Quiz

- What is the output of the following code?

```
d = {1:1, 2:'2', '2':3}
d['1'] = 2
print(d[d[d[str(d[1])]]])
```

- a) 2
- b) 3
- c) ‘2’
- d) KeyError

Quiz

- What is the output of the following code?

```
x = {0:4, 1:8, 2:16, 3:32}
print(x)
print(list(x.values())[2])
```

  - a) [4, 8]
  - b) [4, 8, 16]
  - c) 16
  - d) 8

Quiz

- What is the output of the following code?

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

result = {
    "even": [num for num in numbers if num % 2 == 0],
    "odd": [num for num in numbers if num % 2 != 0]
}

print(result)
```

Summary

LIST	DICTIONARY	SET
pop()	get()	add()
sort()	pop()	pop()
copy()	copy()	copy()
append()	clear()	clear()
insert()	items()	union()
reverse()	values()	issubset()
remove()	update()	isuperset()
extend()	setdefault()	difference()
index()	popitem()	intersection()
count()	keys	isdisjoint()
clear()	fromkeys()	setdiscard()

In class practice

- P05-02 Frequency count: Write a function to count the frequency of each element in a given list and stores the results in a dictionary
  - input: a list of integers
  - output: a dictionary where key-value; (unique elements from the list)-(count of those elements)

```
def count_frequency(lst):
    frequency = {}

    ''' CODE HERE '''

    return frequency

if __name__ == '__main__':
    input_list = [1, 2, 2, 3, 3, 3, 4]
    result = count_frequency(input_list)
    print(result)
```

```
{1: 1, 2: 2, 3: 3, 4: 1}
```

5. Generator

What is a generator?

- Concept and definition
  - Python에서 sequence를 생성하기 위해 사용되는 반복가능한 (iterable) 타입의 object (iterable object)
  - 반복될 때 마다 한 번에 하나씩 항목을 “생성” (index 활용할 수 없음)
    - 그러나, for loop와 같은 반복문에서 적절히 사용 가능

  - generator 표현식 (list comprehension 표현과 유사하지만 [ ] 대신 ()를 사용함)

```
gen = (x**2 for x in range(3))
print(gen) # Output: <generator object <genexpr> at 0x00000218DF0D6740>
```

  - use “for” loop statement for generator (iterable)

```
gen = (x**2 for x in range(3))
for value in gen:
    print(value, end=' ') # Output: 0 1 4
```

Generator vs iterable

- Iterable (object)
  - can loop over with `for` loop
  - lists, strings, tuples, dictionaries, sets, and more

```
my_list = [1, 2, 3]
for num in my_list:
    print(num)
```

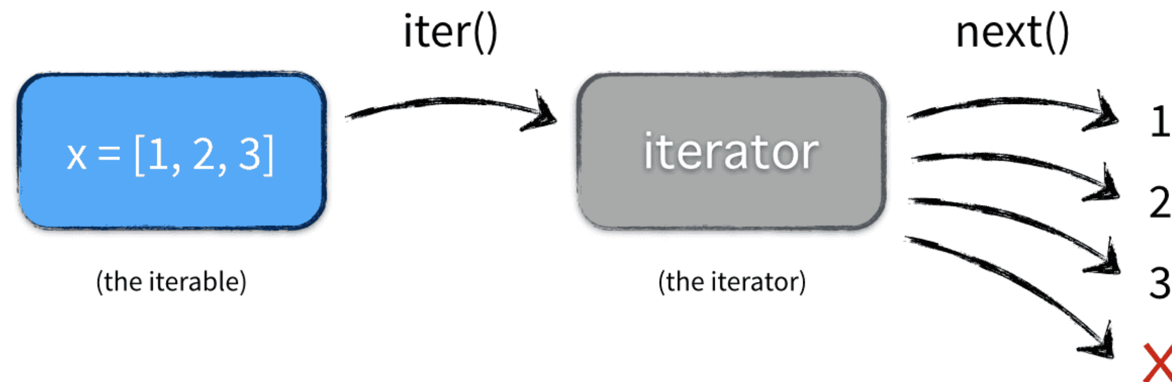
```
1
2
3
```

Generator vs iterable

- Iterable (object) ➔ iterator
  - 'iter()' function: to convert iterable object into iterator
  - 'next()' function: can access each item without loop statement

```
my_list = [1, 2, 3]
my_iterator = iter(my_list)
print(next(my_iterator))
print(next(my_iterator))
print(next(my_iterator))
```

```
1
2
3
```



Generator vs iterable

- Iterable (object) ➔ iterator
  - can also use loop statement
  - iterator generates values on the fly and can be iterated over **only once**

```
my_list = [1, 2, 3]
my_iterator = iter(my_list)
print(next(my_iterator)) # Output: 1

for value in my_iterator:
    print(value)
```

```
1
2
3
```

Generator vs iterable

- Generator
  - a special type of iterator
  - generates values on the fly and can be iterated over **only once**

```
gen = (x for x in range(3))
for value in gen:
    print(value)

for value in gen:
    print(value)
```

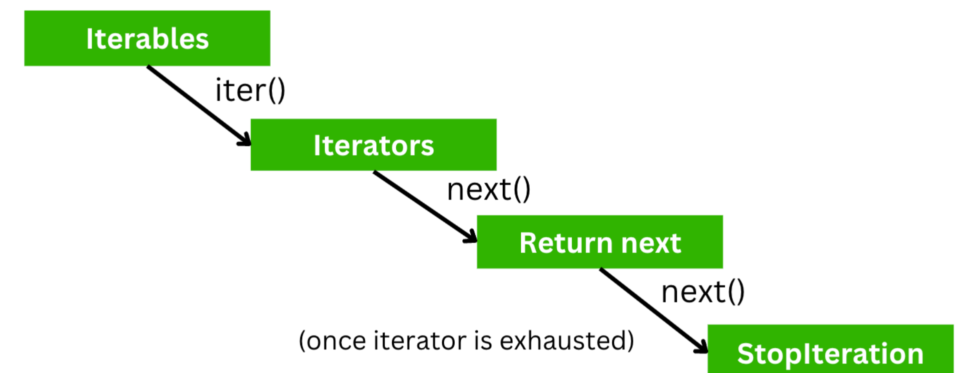
```
0
1
2
```

Accessing the value in generator

- Use `next` keyword to access the value in generator without loop statement

```python
gen = (x for x in range(3))
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
```

```
0
1
2
Traceback (most recent call last):
  File "C:\Users\#2 파이썬프로그래밍\src\generator.py", line 5, in <module>
    print(next(gen))
          ^^^^^^^^^
StopIteration
```



Defining a generator function

- Generators are created using functions and `yield` statement
  - to return data to make generator in regular function
  - can access over the generator using `next()` function

```
def my_generator():
    yield 1
    yield 2
    yield 3

gen = my_generator()
print(next(gen))
print(next(gen))
print(next(gen))
```

```
1
2
3
```

Defining a generator function

- Generators are created using functions and `yield` statement
  - accessing values in loop statement
    - Note: generator is an iterator

```
def my_generator():
    yield 1
    yield 2
    yield 3

for num in my_generator():
    print(num)
```

```
1
2
3
```

Defining a generator function

- Note: generator generates the value **only once**

```
def my_generator():
    yield 1
    yield 2
    yield None

gen = my_generator()
print(next(gen)) # Output: 1

for value in gen:
    print(value)
```

```
1
2
None
```

Defining a generator function

- Iterated yield in generator

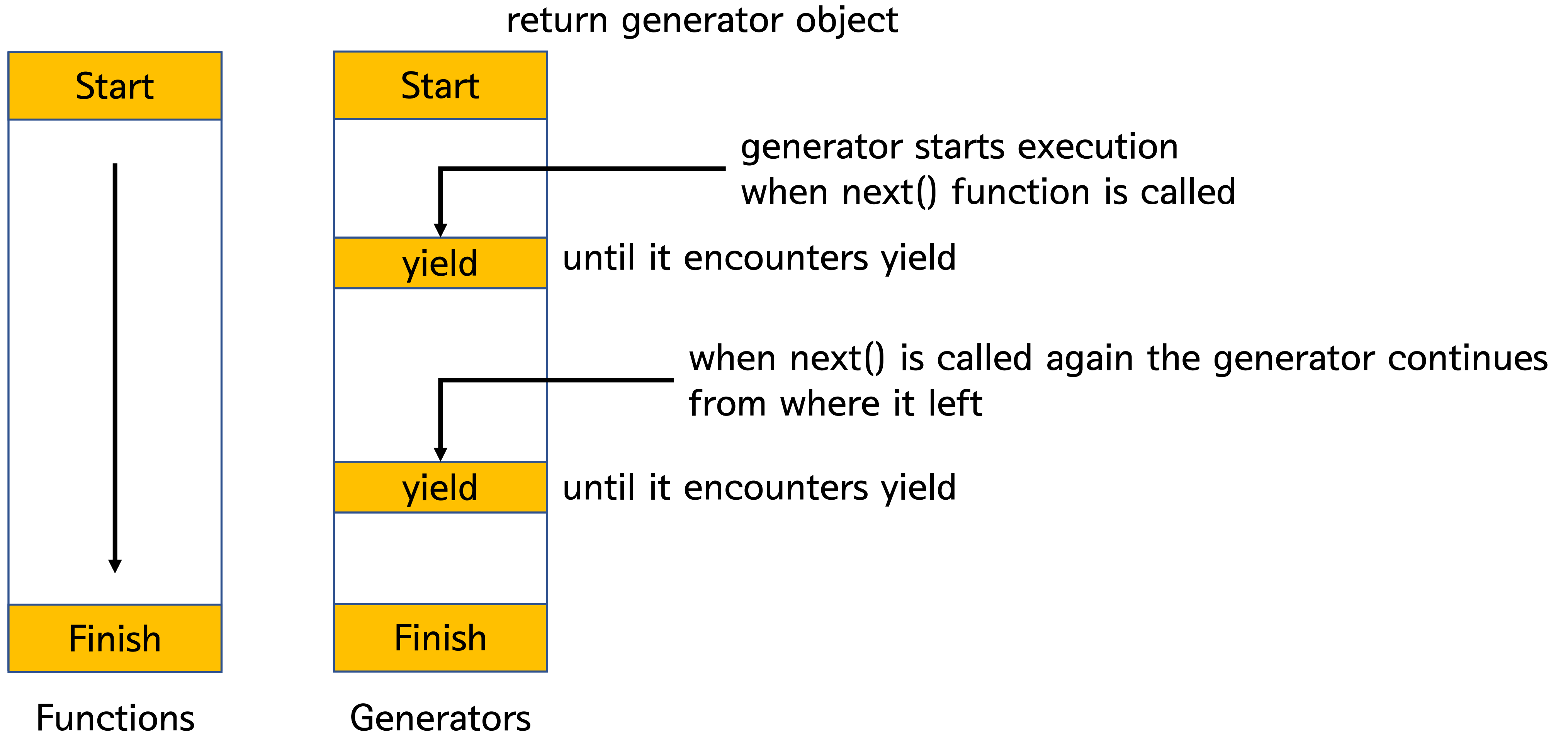
```python
def generate_even_numbers(limit):
    for i in range(limit):
        if i % 2 == 0:
            yield i

for number in generate_even_numbers(10):
    print(number)
```

```
0
2
4
6
8
```

Defining a generator function

- When do the yield statement execute?



Defining a generator function

- When do the yield statement execute?

```python
def gen_numbers():
    print('Start')
    yield 1
    print('Continue')
    yield 2
    print('Final')
    yield 3
    print('End')

print(next(gen_numbers()))
```

```
Start
1
```

Generator usage

- Example
  - to generate integer numbers in dice between 1 and 6

```
import random

def broken_dice(n):
    while True:
        number = random.randint(1, 6)
        if number != n:
            yield number
```

Map

- Built-in function used to apply a given function to each item of an iterable and return a list of the results

- General syntax

```
map(function, iterable, ...)
```

  - 동작 방식: 두 번째 인자로 들어온 iterable (반복가능한 자료형, list or tuple)을 첫 번째 인자로 들어온 함수에 하나씩 집어넣어서 함수를 수행
    - ex) map(적용시킬 함수, 적용할 값들)
  - 주의사항: map 함수의 반환 값은 기본적으로 map 객체이므로, 이를 실질적으로 활용하기 위해서는 list 또는 tuple로 형변환 시켜주어야 함

Map

- Examples

```
def to_upper_case(s):
    return s.upper()

strings = ['hello', 'world', 'python', 'programming']
upper_strings = map(to_upper_case, strings)
print(list(upper_strings))
```

  - 각 리스트의 값을 대문자로 변환

  - Recall – matrix input

```
print("3x3 행렬의 각 행을 입력하세요 (각 숫자는 공백으로 구분):")
for i in range(3):
    row = list(map(int, input().split()))
    matrixA.append(row)
```

Map

- Examples
  - often used with lambda function, and
  - might use list() to convert the map object into a list

```
numbers = [1, 2, 3, 4, 5]
squared = map(lambda x: x**2, numbers)
print(squared)
print(list(squared)) # Output: [1, 4, 9, 16, 25]
```

Filter

- Definition
  - filter() function: constructs a iterator from those element of iterable for which function returns true
    - if the function is None, it simply returns the item of iterable that are true

- General syntax

```
filter(function, iterable)
```

Filter

- Usage: (example) to find even numbers between 1 to 10
  - regular function

```
target, result = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], []
def is_even(n):
      return True if n % 2 == 0 else False

for value in target:
     if is_even(value):
        result.append(value)

print(result)  # Output : [2, 4, 6, 8, 10]
```

  - filter

```
target = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def is_even(n):
    return True if n % 2 == 0 else False

result = filter(is_even, target)
print(list(result)) # Output : [2, 4, 6, 8, 10]
```

Filter

- Usage
  - also used with lambda function
  - might use list() to convert the filter object into a list

```
target = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
result = filter(lambda x: x % 2 == 0, numbers)
print(result) # Output: <filter object at 0x00000218DF0C1B10>
print(list(result)) # Output : [2, 4, 6, 8, 10]
```

Tips

- `all`
  - takes an iterable object; returns `True` if all elements are true, and `False` if at least one element is false

```
print(all([1, 2, 3])) # Output: True
print(all([0, 1, 2, 3])) # Output: False
```

- `any`
  - also takes an iterable objects; returns `True` if at least one element is true, and `False` only if all elements are false

```
print(any([False, 3])) # Output: True
print(any([False, 0, []])) # Output: False
```

Tips

- `zip`
  - combines elements from input iterables into an iterator of tuples
  - particularly useful when you need to loop over two or more iterable in parallel

```
a = zip([1,2,3], (4,5,6))
print(next(a)) # Output: (1, 4)
print(next(a)) # Output: (2, 5)
print(next(a)) # Output: (3, 6)
```

```
country = ['Republic of Korea', 'United States', 'China']
capital = ['Seoul','Washington', 'Beijing']
for coun, cap in zip(country, capital):
    print('Nation : {}, Captial : {}'.format(coun,cap))
```

```
Nation : Republic of Korea, Captial : Seoul
Nation : United States, Captial : Washington
Nation : China, Captial : Beijing
```

Tips

- `enumerate`
  - adds a counter to an iterable and returns it as an enumerate object
  - typically used in a loop to retrieve both the index and the value of each item in the iterable

```
t = [1, 5, 7, 33, 39, 52]
for p in enumerate(t):
    print(p)
```

```
(0, 1)
(1, 5)
(2, 7)
(3, 33)
(4, 39)
(5, 52)
```

Tips

- `enumerate`
  - adds a counter to an iterable and returns it as an enumerate object
  - typically used in a loop to retrieve both the index and the value of each item in the iterable

```
t = [1, 5, 7, 33, 39, 52]
for i, v in enumerate(t):
    print("index : {}, value: {}".format(i,v))
```

```
index : 0, value: 1
index : 1, value: 5
index : 2, value: 7
index : 3, value: 33
index : 4, value: 39
index : 5, value: 52
```

Quiz

- What is the output of the following code?

```
l = [1, 0, 2, 0, 'hello', '', []]
print(list(filter(bool, l)))
```

  - a) [1, 2, ‘hello’, ‘’, []]
  - b) [1, 2, ‘hello’]
  - c) [1, 0, 2, 0, ‘hello’, ‘’, []]
  - d) Error

Quiz

- Using a generator
  - can iterate over the generator using `next()` function
  - Note: generator generates the value **only once**

```
def my_generator():
    yield 1
    yield 2
    yield None

gen = my_generator()
print(next(gen)) # Output: 1

for value in gen:
    print(value)
```

```
1
2
None
```

End of slide