ch.2_crash_course_in_python

공백 서식 지정

많은 언어들이 코드 블록들의 경계를 정하기 위해 곱슬곱슬한 괄호를 사용합니다. 파이썬은 **indentation**:

```
for i in [1, 2, 3, 4, 5]:
    print(i)
    for j in [1, 2, 3, 4, 5]:
        print(j)
        print(i + j)
    print(i)
print("done looping")
```

```
1
1
2
2
3
3
4
4
5
5
6
1
2
1
3
2
4
3
5
4
6
5
7
2
3
1
4
2
5
3
6
4
7
5
8
3
4
1
5
2
```

```
6
3
7
4
8
5
9
4
5
1
6
2
7
3
8
4
9
5
10
5
done looping
```

- 괄호 및 괄호 안에 공백 **무시**

```
long_winded_computation = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 +
                           13 + 14 + 15 + 16 + 17 + 18 + 19 + 20)
```

- 코드를 읽기 쉽게 만들기 위해

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

- **backslash**를 사용하여 문장이 다음 행으로 계속됨을 나타냅니다

```
two_plus_three = 2 + \
                 3
```

Modules

기능이 포함된 모듈 가져오기

- 정규 표현식 모듈 가져오기: 정규 표현식을 사용하기 위한 함수 및 상수를 포함하는 모듈이 있습니다.

```
import re
my_regex = re.compile("[0-9]+", re.I)
```

- **alias**를 사용할 수 있습니다

```
import re as regex
my_regex = regex.compile("[0-9]+", regex.I)
```

```
import matplotlib.pyplot as plt
```

- 명시적으로 가져와 사용할 수 있습니다 **자격없이 사용할 수 있습니다.**

```
match = 10
from re import *     # uh oh, re has a match function
print(match)          # "<function re.match>"

# Reslut: <function match at 0x7cce84681a20>
```

Arithmetic

- 기억하기 **분수-잔류 정리**

$n=d\cdot q+r$

- d는 약수, q 는 몫, r는 나머지,
- q가 양수일 때 , $0\leq r<q$, q가 음수일 때 $q<r\leq 0$

```
print(2 ** 10)       # 1024
print(2 ** 0.5)      # 1.414...
print(2 ** -0.5)     #
print(5 / 2)         # 2.5
print(5 % 3)         # 2
print(5 // 3)        # 1
print((-5) % 3)      # 1
print((-5) // 3)     # -2
print(5 % (-3))      # -1
print((-5) // (-3))  # 1
print((-5) % (-3))   # -2
print(7.2 // 3.5)    # 2.0
print(7.2 % 3.5)     # 0.2
```

Function

- 함수는 0 이상의 입력을 받고 해당 출력을 반환하는 규칙입니다.

```
# This
# is
# a
# comment
# CTRL + / toggles comment/uncomment


# for PEP on docstring, refer to https://www.python.org/dev/peps/pep-0257/#abstract
def double(x):
    """this is where you put an optional docstring
    that explains what the function does.
    for example, this function multiplies its input by 2"""
    return x * 2

double(2)
```

```
help(double)

Help on function double in module __main__:
```

```
double(x)
    this is where you put an optional docstring
    that explains what the function does.
    for example, this function multiplies its input by 2
```

- Python 함수는 **first-class**이므로 다른 인수와 마찬가지로 변수에 할당하고 함수에 전달할 수 있습니다.
  - Function(함수)를 변수처럼 Assign

```
def apply_to_one(f):
    """calls the function f with 1 as its argument(인자로 전달)"""
    return f(1)

my_double = double
x = apply_to_one(my_double)

print(x)
```

- Lambda 함수: 짧은 익명 함수
  - 이름 없는 Function, 짧은 Function

```
y = apply_to_one(lambda x: x + 4)
print(y)

# Result: 5
```

```
another_double = lambda x: 2 * x
def another_double(x): return 2 * x   # more readable
```

```
add = lambda x, y : x + y
add(1,2)

# Result: 3
```

- 함수 매개변수는 **default 인수**도 지정할 수 있습니다

```
def my_print(message="my default message"): # message 값이 아닌, 그냥 그대로 print
    print(message)

my_print("hello")  # prints 'hello'
my_print()         # prints 'my default message'
```

```
def subtract(a=0, b=0):
    return a - b

subtract(10, 5) # returns 5
subtract(0, 5)  # returns -5
subtract(b=5)   # same as previous
subtract(b=5, a=20)
```

```
# Result: 15
```

String

- 문자열은 단일 또는 이중 따옴표로 구분할 수 있습니다.

```
single_quoted_string = 'data science'
double_quoted_string = "data science"
```

```
tab_string = "\t"   # represents the tab character
len(tab_string)     # is 1

# Result: 1
```

- 삼중 이중 quotes을 사용한 다중 줄 문자열

```
multi_line_string = """This is the first line.
and this is the second line
and this is the third line"""

multi_line_string

# Result: 'This is the first line. \nand this is the second line \nand this is the third line'
```

Exceptions

- 뭔가 잘못되면 Python은 예외를 제기합니다.

```
try:
    print(0 / 0) # ZeroDivisionError로 바로 이동
except ZeroDivisionError:
    print("cannot divide by zero")

# Result: cannot divide by zero
```

Lists

- 파이썬에서 가장 기본적인 데이터 구조

```
integer_list = [1, 2, 3]
heterogeneous_list = ["string", 0.1, True]
list_of_lists = [ integer_list, heterogeneous_list, [] ]

list_length = len(integer_list)    # eqauals 3, 리스트의 길이 반환 (3개)
list_sum    = sum(integer_list)    # equals 6, 리스트에 있는 모든 요소의 합을 계산 (1+2+3 = 6)
```

- **integer_list = [1, 2, 3] - 정수형 List**
- **heterogeneous_list = ["string", 0.1, True] - 서로 다른 type의 List**
- **list_of_lists = [integer_list, heterogeneous_list, []] - 리스트를 요소로 가지는 리스트를 정의**

- 괄호가 있는 목록의 n번째 요소를 가져오거나 설정할 수 있습니다

```
x = list(range(10))      # is the list [0, 1, ..., 9]
zero = x[0]          # equals 0, lists are 0-indexed
one = x[1]           # equals 1
nine = x[-1]         # equals 9, 'Pythonic' for last   element
eight = x[-2]        # equals 8, 'Pythonic' for next-to-last element
x[0] = -1            # now x is [-1, 1, 2, 3, ..., 9]
```

- 대괄호를 사용하여 목록을 "Slice"할 수도 있습니다.

```
first_three = x[:3]                  # [-1, 1, 2], -1 ~ 2
three_to_end = x[3:]                 # [3, 4, ..., 9], 3부터 시작
one_to_four = x[1:5]                 # [1, 2, 3, 4], 1부터 4까지 있는 값 출력
last_three = x[-3:]                  # [7, 8, 9], -3부터 (반대, 역순)
without_first_and_last = x[1:-1]     # [1, 2, ..., 8] 앞에서 1번, 뒤에서 1번 사이 출력
copy_of_x = x[:]                     # [-1, 1, 2, ..., 9] 복사해서 출력
```

```
print(x[::2]) # 리스트 위치 숫자 2번 앞에 값 출력, 반복
print(x[::-1]) # 반대로 출력
```

```
# Result

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

- **in** 연산자를 통해 목록 멤버십을 확인할 수 있습니다

```
1 in [1, 2, 3]  # True
0 in [1, 2, 3]  # False

# Result: False
```

- List를 함께 연결 하려면?

```
x = [1, 2, 3]
x.extend([4, 5, 6])  # x is now [1,2,3,4,5,6] 리스트 확장
```

```
x = [1, 2, 3]
y = x + [4, 5, 6]  # y is [1, 2, 3, 4, 5, 6]; x is unchanged. x는 변함이 없다.
```

- 한번에 하나의 List를 추가하려면?

```
x = [1, 2, 3]
x.append(0)  # x is now [1, 2, 3, 0]
y = x[-1]    # equals 0 (역순 출력)
z = len(x)   # equals 4 (List 길이 출력)
```

- List를 풀려면?

```
x, y = [1, 2]  # now x is 1, y is 2
_, y = [1, 2] # now y == 2, didn't care about the first element.

# '_'는 실제로 사용하지 않는 값을 의미합니다.
```

Tuples

- 튜플은 List의 불변의 사촌입니다.
- ‘,‘이 있으면 튜플
- 만약 상대에게 data를 줄때 변형을 원하지 않으면? → 튜플로 List화 하면 안됨

```
my_list = [1, 2] # 리스트, 값 변경 가능
my_tuple = (1, 2) # 튜플, 값 변경 불가
other_tuple = 3, 4 # 튜플을 생성하는 다른 방법. 값들을 쉼표로 구분하여 튜플 생성 가능
my_list[1] = 3   # my_list is now [1, 3]

try:
    my_tuple[1] = 3 # 튜플의 2번째 요소를 3으로 변경하려고 함. 튜플은 변경 불가능 해서 연산불가
except TypeError:
    print("cannot modify a tuple")

# Result: cannot modify a tuple
```

- 튜플은 함수에서 여러 값을 반환하는 편리한 방법입니다.

```
def sum_and_product(x, y):
    return (x + y),(x * y)

sp = sum_and_product(2, 3)    # equals (5, 6)
# 함수에서 반환된 튜플은 (5, 6)이므로 sp는 (5, 6)

s, p = sum_and_product(5, 10) # s is 15, p is 50, s = x + y, p = x * y
# 함수를 호출하여 반환된 튜플을 s와 p 변수에 각각 저장
# 함수에서 반환된 튜플은 (15, 50)이므로 s는 15이고 p는 50이 됩니다.
```

- Tuple(및 목록)은 **multiple assignment**:에도 사용할 수 있습니다.

```
x, y = 1, 2   # now x is 1, y is 2
x, y = y, x   # Pythonic way to swap variables; now x is 2, y is 1
```

- Python Tuple

Leaving Google Colab

https://colab.research.google.com/corgiredirector?site=https://wiki.python.org/moin/TupleSyntax

- Python 튜플은 괄호가 아닌 후행 쉼표로 정의됩니다 1, 1,2, 1,2,3, 괄호는 선택사항입니다

Dictionaries

- **값과 키를 연결하는 또 다른 기본 데이터 구조**
- 주어진 키에 해당하는 값을 빠르게 검색할 수 있습니다:

```
empty_dict = {}                       # Pythonic
empty_dict2 = dict()                  # less Pythonic
grades = { "Joel" : 80, "Tim" : 95 } # dictionary literal. Joel: Key, Tim: Value
```

- 대괄호를 사용하여 키 값을 검색할 수 있습니다:

```
joels_grade = grades["Joel"]         # equals 80
```

```
grades = { "Joel" : 80, "Tim" : 95, "Tim" : 94 } # dictionary literal
grades

# Result: {'Joel': 80, 'Tim': 94}, Key 값이 중복이 되면 마지막에 지정된 값이 사용됩니다.
```

```
len(grades) # 2
grades.keys() # dict_keys(['Joel', 'Tim'])
grades.values() # dict_values([80, 94])
grades["Tim"] # 94
```

- KeyError 사전에 없는 키를 요청하는 경우.

```
try:
    kates_grade = grades["Kate"]
except KeyError:
    print("no grade for Kate!")

# no grade for Kate!
```

- 키의 존재 여부를 확인할 수 있습니다

```
joel_has_grade = "Joel" in grades    # True
kate_has_grade = "Kate" in grades    # False
```

- 사전에는 사전에 없는 키를 검색할 때 예외를 높이는 대신 기본값을 반환하는 get 메서드가 있습니다:
  - 만약 기본값을 지정하지 않으면 기본값은 None

```
joels_grade = grades.get("Joel", 0)    # equals 80, 만약 Key(Joel)이 없으면 기본값으로 0 사용
kates_grade = grades.get("Kate", 0)    # equals 0, Kate 키는 딕셔너리에 없으므로 기본값인 0을 반환
no_ones_grade = grades.get("No One")   # default default is None, 키가 존재하지 않으면 기본값으로 None
no_ones_grade == None # no_ones_grade에 할당된 값이 None인지를 확인.

# Result: True
```

- 동일한 대괄호를 사용하여 Key-Value 쌍을 할당합니다.

```
grades["Tim"] = 99          # replaces the old value
grades["Kate"] = 100        # adds a third entry
num_students = len(grades)  # equals 3
```

- 우리는 정형 데이터를 나타내는 간단한 방법으로 Dictionary를 자주 사용할 것입니다.

```
tweet = {
    "user" : "joelgrus",
    "text" : "Data Science is Awesome",
    "retweet_count" : 100,
    "hashtags" : ["#data", "#science", "#datascience", "#awesome", "#yolo"]
}
```

- Iteration: 우리는 그것들을 모두 볼 수 있습니다.

```
tweet_keys   = tweet.keys()     # 딕셔너리의 모든 key를 가져와서 리스트로 변환 (key-이름)
tweet_values = tweet.values()   # 딕셔너리의 모든 value들을 가져와서 리스트로 변환 (text)
tweet_items  = tweet.items()    # 딕셔너리의 모든 (키, 값) 쌍을 가져와서 리스트로 변환

"user" in tweet_keys            # True, but uses a slow list in
# tweet_keys라는 변수에 저장된 리스트에서 "user"를 찾는 작업을 수행

"user" in tweet                 # more Pythonic, uses faster dict in
# 딕셔너리의 키를 더 효율적으로 확인

"joelgrus" in tweet_values      # True
# tweet_values에는 딕셔너리의 값들이 들어 있으므로, 이 코드는 "joelgrus"라는 값이 tweet 딕셔너리의 값 중에 있는

# Result: True
```

```
tweet_values

# Result: dict_values(['joelgrus', 'Data Science is Awesome', 100, ['#data', '#science', '#datas
```

- WordCount Example: Key가 단어이고 Value가 카운트인 사전을 만듭니다.

```
document = ['I', 'am', 'a', 'boy', 'I', 'love', 'you']
```

- First Approach

```
word_counts = {}  # 단어의 등장 횟수를 저장할 빈 딕셔너리를 생성합니다.
for word in document:  # 문서(document)에서 각 단어(word)를 반복합니다.
    if word in word_counts:  # 현재 가져온 단어가 이미 word_counts 딕셔너리에 있는지 확인합니다.
        word_counts[word] += 1  # 만약 단어가 이미 딕셔너리에 있다면 해당 단어의 등장 횟수를 1 증가시킵니다.
    else:
        word_counts[word] = 1  # 만약 단어가 딕셔너리에 없다면 해당 단어를 추가하고 등장 횟수를 1로 설정합니다
```

- Second Approach

```
word_counts = {}  # 단어의 등장 횟수를 저장할 빈 딕셔너리를 생성합니다.
for word in document:  # 문서(document)에서 각 단어(word)를 반복합니다.
    try:
```

```
        word_counts[word] += 1  # 단어가 이미 딕셔너리에 있는 경우 등장 횟수를 1 증가시킵니다.
    except KeyError:  # KeyError가 발생한 경우 (단어가 딕셔너리에 없는 경우)
        word_counts[word] = 1  # 새로운 단어로 추가하고 등장 횟수를 1로 설정합니다.
```

- Third Approach

```
word_counts = {}  # 단어의 등장 횟수를 저장할 빈 딕셔너리를 생성합니다.
for word in document:  # 문서(document)에서 각 단어(word)를 반복합니다.
    previous_count = word_counts.get(word, 0)  # 해당 단어의 이전 등장 횟수를 가져옵니다. 없으면 기본값으로
    word_counts[word] = previous_count + 1  # 이전 등장 횟수에 1을 더하여 현재 단어의 등장 횟수를 업데이트

word_counts

# Result: {'I': 2, 'am': 1, 'a': 1, 'boy': 1, 'love': 1, 'you': 1}
```

Defaultdict

- 기본 딕션은 포함되지 않은 키를 검색하려고 할 때 생성할 때 제공한 **zero-argument 함수**를 사용하여 먼저 해당 키에 대한 값을 추가한다는 점을 제외하고 일반 사전과 같습니다.

```
from collections import defaultdict

word_counts = defaultdict(int)    # int()은 0을 생성합니다. 즉, 기본값으로 0을 사용합니다.
# word_counts = defaultdict(lambda: 100)    # lambda 함수를 사용하여 기본값으로 100을 반환할 수도 있습니

for word in document:  # 문서(document)에서 각 단어(word)를 반복합니다.
    word_counts[word] += 1  # 해당 단어의 등장 횟수를 1 증가시킵니다.

print(word_counts)  # 딕셔너리를 출력합니다.

# defaultdict(<class 'int'>, {'I': 2, 'am': 1, 'a': 1, 'boy': 1, 'love': 1, 'you': 1})
```

```
int() # 0
```

```
# defaultdict를 사용하여 기본값으로 빈 리스트를 가지는 딕셔너리를 초기화합니다.
dd_list = defaultdict(list)

# 키 2에 해당하는 값인 리스트에 1을 추가합니다.
# 이전에 존재하지 않는 키였으므로 빈 리스트가 생성된 후에 1이 추가됩니다.
dd_list[2].append(1)

# defaultdict를 사용하여 기본값으로 빈 딕셔너리를 가지는 딕셔너리를 초기화합니다.
dd_dict = defaultdict(dict)

# 키 "Joel"에 해당하는 값인 딕셔너리의 "City" 키에 "Seattle"을 할당합니다.
# 이전에 존재하지 않는 키였으므로 빈 딕셔너리가 생성된 후에 "City" 키가 추가되고 "Seattle"이 할당됩니다.
dd_dict["Joel"]["City"] = "Seattle"

# defaultdict를 사용하여 기본값으로 [0, 0] 리스트를 가지는 딕셔너리를 초기화합니다.
dd_pair = defaultdict(lambda: [0, 0])

# 키 2에 해당하는 값인 리스트의 인덱스 1에 1을 할당합니다.
# 이전에 존재하지 않는 키였으므로 [0, 0] 리스트가 생성된 후에 해당 위치에 1이 할당됩니다.
dd_pair[2][1] = 1
```

Counter

- 카운터는 일련의 값을 기본 dict(int)와 같은 개체 매핑 키로 변환하여 카운트합니다.
- **histograms**를 만드는 데 주로 사용할 것입니다

```
from collections import Counter

# Counter를 사용하여 리스트 내의 각 요소의 등장 횟수를 세어 c에 저장합니다.
c = Counter([0, 1, 2, 0])  # c는 (기본적으로) { 0 : 2, 1 : 1, 2 : 1 }
```

```
# Counter를 사용하여 문서(document)에서 각 단어의 등장 횟수를 세고 word_counts에 저장합니다.
word_counts = Counter(document)

# 결과를 출력합니다.
print(word_counts)

# Counter({'I': 2, 'am': 1, 'a': 1, 'boy': 1, 'love': 1, 'you': 1})
```

- 문서(document)에서 각 단어의 등장 횟수를 세어 딕셔너리 형태로 반환
- `word_counts` 변수에는 문서 내의 각 단어의 등장 횟수가 저장

- **help** 기능을 사용하여 main 페이지 보기

```
help(word_counts)
```

```
# Counter의 most_common() 메서드를 사용하여 빈번하게 등장하는 단어 상위 10개, 등장 횟수를 출력.
for word, count in word_counts.most_common(10):
    print(word, count)
```

```
I 2
am 1
a 1
boy 1
love 1
you 1
```

Sets

- 또 다른 데이터 구조는 **set**이며, 이는 **distinct** 요소의 집합을 나타냅니다.

```
s = set()
s.add(1)      # s is now { 1 }
s.add(2)      # s is now { 1, 2 }
s.add(2)      # s is still { 1, 2 }
x = len(s)    # equals 2
y = 2 in s    # equals True
z = 3 in s    # equals False
```

- 구성원 자격 테스트의 경우 목록보다 set가 더 적합합니다
- in은 set에서 매우 빠른 작업입니다.

```
hundreds_of_other_words = []
stopwords_list = ["a", "an", "at"] + hundreds_of_other_words + ["yet", "you"]

# 리스트를 이용하여 "zip"이 stopwords에 포함되어 있는지 확인합니다.
# 리스트를 이용한 멤버십 검사는 모든 요소를 확인해야 하므로 느립니다.
"zip" in stopwords_list  # False

# 집합을 이용하여 "zip"이 stopwords에 포함되어 있는지 확인합니다.
# 집합을 이용한 멤버십 검사는 매우 빠르며 해싱을 사용하여 멤버를 찾습니다.
stopwords_set = set(stopwords_list)
"zip" in stopwords_set  # False
```

- Collection에서 distinct 항목을 찾으려면?

```
item_list = [1, 2, 3, 1, 2, 3]
num_items = len(item_list)          # 6
item_set = set(item_list)           # {1, 2, 3}
num_distinct_items = len(item_set)  # 3
distinct_item_list = list(item_set) # [1, 2, 3]
```

Control Flow

```
if 1 > 2:
    message = "if only 1 were greater than two..."
elif 1 > 3:
    message = "elif stands for 'else if'"
else:
    message = "when all else fails use else (if you want to)"
```

- **일직선상의 삼원형 else**

```
parity = "even" if x % 2 == 0 else "odd"
```

- **while** statement:

```
x = 0
while x < 10:
    print(x, "is less than 10")
    x += 1
```

```
0 is less than 10
1 is less than 10
2 is less than 10
3 is less than 10
4 is less than 10
5 is less than 10
6 is less than 10
7 is less than 10
8 is less than 10
9 is less than 10
```

- for statement.

```
for x in range(10):
    print(x, "is less than 10")
```

```
0 is less than 10
1 is less than 10
2 is less than 10
3 is less than 10
4 is less than 10
5 is less than 10
6 is less than 10
7 is less than 10
8 is less than 10
9 is less than 10
```

- continue & break statement

```
for x in range(10):
    if x == 3:
        continue   # go immediately to the next iteration
    if x == 5:
        break      # quit the loop entirely
    print(x)
```

```
0
1
2
4
```

Truthiness

```
one_is_less_than_two = 1 < 2       # equals True, 참 - True 할당
true_equals_false = True == False  # equals False, True & False가 같은지 비교. 다르니까 False
```

- **python은 None 값을 사용하여 존재하지 않는 값을 나타냅니다.**

```
x = None
print(x == None)   # prints True, but is not Pythonic
print(x is None)   # prints True, and is Pythonic

# True
# True
```

- **다음은 모두 "Falsy"입니다.**

```
False
None
[] : (an empty list)
{} : (an empty dict)
""
set()
```

```
0
0.0
```

```
s = 'abc'
if s: # if s:는 문자열 s가 비어있지 않으면 True를 반환
    first_char = s[0]
else: # 만약 문자열이 비어있으면 빈 문자열("")이 first_char 변수에 할당
    first_char = ""
```

```
# and 연산자는 첫 번째 피연산자인 s가 참인지 확인,
# 그 결과에 따라 두 번째 피연산자인 s[0]을 반환.
# 만약 s가 비어있지 않으면 s[0]이 반환되고, 그렇지 않으면 s 자체가 반환

first_char = s and s[0]   # A simpler way of doing the same
```

```
x = None
safe_x = x or 0    # 첫 번째 피연산자인 x를 확인하고, 만약 x가 거짓 값이라면(여기서는 None) 두 번째 피연산자인
safe_x

# safe_x에는 x의 값이 할당되지만, x가 None이면 0이 할당
# Result: 0
```

- Python에는 목록을 가져 모든 요소가 참일 때 True를 정확히 반환하는 모든 함수와 적어도 하나의 요소가 참일 때 True를 반환하는 모든 함수가 있습니다.

```
all([True, 1, { 3 }])    # True
all([True, 1, {}])       # False, {} is falsy
any([True, 1, {}])       # True, True is truthy, 주어진 iterable의 요소 중 하나라도 참이면 True를 반환
all([])                  # True, no falsy elements in the list, 거짓 요소가 없으므로 참
any([])                  # False, no truthy elements in the list, 참인 요소가 없으므로 False
```

```
all([] + [True, True]) == all([]) and all([True, True])

# True
```

- 빈 리스트 `[]`와 `[True, True]`를 결합하면 `[True, True]`이 되고, `all()` 함수는 이 리스트의 모든 요소가 참이어야 하므로 결과는 `True`가 됩니다.
- `all([])`는 빈 리스트이므로 모든 요소가 참입니다. 따라서 첫 번째 부분은 `True`가 됩니다. 그리고 `all([True, True])`도 `True`입니다.

```
any([] + [True, False]) == any([]) or any([True, False])

# True, 빈 리스트와 [True, False]를 연결한 리스트에는 적어도 하나의 True가 있기 때문
```

- 빈 리스트 `[]`와 `[True, False]`를 연결하여 `[True, False]`를 만듭니다.
  - 그런 다음 `any()` 함수는 이 리스트의 요소 중 하나라도 `True`인지 확인하지만, 두 요소 모두 `False`이므로 결과는 `False`입니다.
- `any([])`: 이 식은 빈 리스트에는 요소가 없으므로 `False`로 평가
- `any([True, False])`: 이 식은 최소한 하나의 요소 (`True`)가 `True`임을 확인하기 때문에 `True`로 평가됩니다.

The Not-So-Basics

Sorting

```
x = [4,1,2,3]
y = sorted(x)    # is [1,2,3,4], x is unchanged
x.sort()         # now x is [1,2,3,4]
```

```
# sort the list by absolute value from largest to smallest
x = sorted([-4,1,-2,3], key=abs, reverse=True) # is [-4,3,-2,1]
# sort the words and counts from highest count to lowest
wc = sorted(word_counts.items(),
            key=lambda x: x[1], # x[1] 두번째 값을 기준으로 정렬
            reverse=True)

wc
# [('I', 2), ('am', 1), ('a', 1), ('boy', 1), ('love', 1), ('you', 1)]
```

- x 리스트는 절댓값에 따라 내림차순으로 정렬됩니다.
  - `sorted()` 함수는 입력 리스트를 정렬합니다.
  - `key=abs` 인자는 정렬 키를 절댓값으로 설정합니다.
  - `reverse=True` 인자는 내림차순으로 정렬됨을 나타냅니다.

- `wc`는 단어 및 해당 카운트를 가진 딕셔너리를 가장 많은 카운트부터 가장 적은 카운트로 정렬
  - `sorted()` 함수의 `key` 매개변수는 각 요소에 대해 적용되는 함수를 지정합니다.
  - 여기서는 람다 함수를 사용하여 요소의 두 번째 값(카운트), 단어의 출연 빈도를 기준으로 정렬합니다.
  - `reverse=True` 인자는 내림차순으로 정렬됨을 나타냅니다.

> 💡 `word_counts.items()`는 딕셔너리의 각 항목을 튜플로 반환합니다.
>
> 예를 들어, `{ 'I': 2, 'am': 1, 'a': 1 }`이라는 딕셔너리가 있다면, `.items()` 메소드는 `('I', 2)`, `('am', 1)`, `('a', 1)`과 같은 튜플들의 리스트를 반환
>
> `sorted()` 함수에 의해 두 번째 요소(즉, 튜플의 두 번째 값)를 기준으로 내림차순으로 정렬됩니다.
>
> 따라서 `word_counts.items()`가 튜플들의 리스트를 반환하므로 두 번째 값은 해당 단어의 출현 빈도를 나타냅니다.
>
> x[0]: '단어', x[1]: '숫자'

- 단어와 해당 카운트를 가장 많은 카운트부터 가장 적은 카운트로 정렬

List Comprehensions

- 특정 요소만을 선택하거나 요소를 변환하거나 둘 다를 선택하여 목록을 다른 목록으로 변환하고자 할 것입니다.
- 목록의 포괄성은 파이토닉 방식에 의해 결정됩니다:
- 가능하면 항상 List Comprehensions를 사용합니다.

```
even_numbers = [x for x in range(5) if x % 2 == 0]  # [0, 2, 4], 0 ~ 4 사이중
squares      = [x * x for x in range(5)]            # [0, 1, 4, 9, 16]
even_squares = [x * x for x in even_numbers]        # [0, 4, 16]
```

- **마찬가지로 List을 Dictionary 또는 Set으로 변환할 수 있습니다.**

```
square_dict = { x : x * x for x in range(5) }  # { 0:0, 1:1, 2:4, 3:9, 4:16 }
square_set = { x * x for x in [1, -1] }        # { 1 } # set은 중복 요소 포함 X
```

- **변수로 밑줄을 사용하는 것이 일반적입니다.**
  - '_'은 값이 필요하지 않을때 사용. even_numbers = [0, 2, 4] → Zeros List: [0, 0, 0]

```
zeroes = [0 for _ in even_numbers]    # has the same length as even_numbers
```

- **List Comprehension 다음과 같은 여러 가지 이유가 포함될 수 있습니다:**

```
pairs = [(x, y)
         for x in range(10)
         for y in range(10)] # 100 pairs (0,0) (0,1) ... (9,8), (9,9)
```

- 컴퓨터 사용이 쉬운 거리 행렬, 나중에는 이전의 결과를 사용할 수 있습니다:

```
increasing_pairs = [(x, y)
                    for x in range(10)
                    for y in range(x + 1, 10)]

# [(0,1), (0,2), (0,3) ... (8, 9)]
```

- 0부터 9까지인 모든 숫자 쌍을 생성합니다.
- 이 쌍들은 첫 번째 요소가 두 번째 요소보다 작은 증가하는 순서쌍.

Generators & Iterators

- List의 문제점은 List가 쉽게 매우 커질 수 있다는 것입니다.
- range(1000000)는 실제 100만 개의 원소 목록을 만듭니다.
- 만약 당신이 한 번에 하나씩만 다루면 된다면, 이것은 엄청난 비효율성의 원천(또는 메모리 부족의 원인)이 될 수 있습니다.
- 만약 당신이 잠재적으로 처음 몇 개의 값만 필요하다면, 그것들을 모두 계산하는 것은 낭비입니다.
- Generator는 사용자가 반복할 수 있지만(우리의 경우, 일반적으로 에 사용합니다) 필요에 따라 값이 생성되는 것(lazily)입니다.
- Generator를 만드는 한 가지 방법은 함수와 yield 연산자를 사용하는 것입니다:

```
def lazy_range(n):
    """a lazy version of range"""
    i = 0
    while i < n:
        yield i
        i += 1
```

```
# The following loop will consume the yield ed values one at a time until none are left:
for i in lazy_range(10):
    print(i)
```

```
0
1
2
3
4
5
```

```
6
7
8
9
```

```
# The following loop will consume the yield ed values one at a time until none are left:
for i in lazy_range(10000):
    if i == 3: break
    print(i)
```

```
0
1
2
```

```
t = lazy_range(3)
next(t)
next(t)
next(t)
#next(t)

# 2
```

```
def lazy_inf_range():
    i = 0
    while True:
        yield i
        i += 1

t = lazy_inf_range()
next(t)
next(t)
next(t)

# 2
```

- Generator를 만드는 두 번째 방법은 괄호 안으로 포장된 comprehension에 사용하는 것입니다:

```
lazy_evens_below_20 = (i for i in lazy_range(20) if i % 2 == 0)
lazy_evens_below_20

# <generator object <genexpr> at 0x7cce6849c4a0>
```

Randomness

- 난수를 생성하기 위해, 우리는 난수 모듈을 사용할 수 있습니다
- random.random()는 0에서 1 사이의 숫자를 균일하게 생성합니다

```
import random
```

```
four_uniform_randoms = [random.random() for _ in range(4)]
four_uniform_randoms
```

```
[0.15001730378211198,
 0.047689363188983425,
 0.4438845111618783,
 0.8064273339306516]
```

- **reproducible 가능한 결과를 얻으려는 경우.**

```
random.seed(10)
print(random.random())
random.seed(10)
print(random.random())
```

```


0.5714025946899135
```

- random.randrange는 1 또는 2개의 인수를 사용하고 해당 범위 ()에서 임의로 선택한 요소를 반환합니다.

```
random.randrange(10)     # choose randomly from range(10) = [0, 1, ..., 9]
random.randrange(3, 6)   # choose randomly from range(3, 6) = [3, 4, 5]
```

- **random.shuffle는 목록의 요소를 임의로 재정렬합니다:**

```
up_to_ten = list(range(10))
random.shuffle(up_to_ten)
print(up_to_ten)

# 4, 5, 8, 1, 2, 6, 7, 3, 0, 9]
```

- **목록에서 한 요소를 임의로 선택하는 방법:**

```
my_best_friend = random.choice(["Alice", "Bob", "Charlie"])
```

- 대체하지 않고 원소 표본을 임의로 선택하는 방법(즉, 중복되지 않음)

```
lottery_numbers = range(60)
winning_numbers = random.sample(lottery_numbers, 6)
```

- 대체 요소 표본을 선택하는 방법(즉, 중복 허용)

```
four_with_replacement = [random.choice(range(10)) for _ in range(4)]
four_with_replacement

# [2, 9, 5, 6]
```

- `random.choice(range(10))`는 0부터 9까지의 숫자 중에서 하나를 무작위로 선택하는 함수입니다
- 이를 리스트 컴프리헨션 으로 4번 반복하여 리스트를 생성. 중복 허용, 무작위로 선택된 0부터 9까지의 숫자가 4개 포함.

Regular Expressions

- 정규 표현식은 텍스트를 검색하는 방법을 제공합니다.
- 그것들은 믿을 수 없을 정도로 유용하지만 또한 꽤 복잡해서 그것들에 대한 전체 책이 있습니다.

```
import re

# 정규 표현식을 사용하여 다양한 문자열 작업을 수행하고, 각 작업의 결과를 검사합니다.
# 모든 조건이 참일 경우 True를 출력합니다.

print(all([
    not re.match("a", "cat"),                # "cat" 문자열은 "a"로 시작하지 않음
    re.search("a", "cat"),                    # "cat" 문자열에 "a"가 포함되어 있음
    not re.search("c", "dog"),                # "dog" 문자열에 "c"가 포함되어 있지 않음
    3 == len(re.split("[ab]", "carbs")),    # "carbs" 문자열을 "[ab]"를 기준으로 분할하면 ['c', 'r',
    "R-D-" == re.sub("[0-9]", "-", "R2D2")    # "R2D2" 문자열에서 숫자를 "-"로 대체하면 "R-D-"가 됨
]))    # 출력 결과는 True
```

Object-Oriented Programming

```
# 관례적으로, 클래스는 PascalCase 이름을 사용합니다.
class Set:
    # 이것들은 멤버 함수입니다.
    # 각 함수는 "self"라는 첫 번째 매개변수를 가져야 합니다(또 다른 관례입니다).
    # 이 "self"는 사용되는 특정 Set 객체를 가리킵니다.

    def __init__(self, values=None):
        """이것은 생성자입니다.
        새로운 Set을 만들 때 호출됩니다.
        다음과 같이 사용할 수 있습니다.
        s1 = Set()           # 빈 집합
        s2 = Set([1,2,2,3]) # 값으로 초기화"""

        self.dict = {}  # 각 Set 인스턴스마다 고유한 dict 속성이 있습니다.
                        # 이 속성은 멤버십을 추적하는 데 사용됩니다.
        if values is not None:
            for value in values:
                self.add(value)

    def __repr__(self):
        """이것은 Set 객체의 문자열 표현입니다.
        Python 프롬프트에서 입력하거나 str()에 전달하면 사용됩니다."""
        return "Set: " + str(self.dict.keys())

    # 각 요소의 멤버십은 self.dict의 키로 표시됩니다.
    def add(self, value):
        self.dict[value] = True

    # 값이 집합에 있는지 여부는 사전의 키로 판별됩니다.
    def contains(self, value):
        return value in self.dict

    def remove(self, value):
        del self.dict[value]
```

```
s = Set([1,2,3])
s.add(4)
print(s.contains(4))    # True
s.remove(3)
print(s.contains(3))    # False
```

Function Tools

- 함수를 전달할 때 새 함수를 만들기 위해 부분적으로 (또는 카레) 함수를 적용하고 싶을 때가 있습니다

```
def exp(base, power):
    return base ** power

def two_to_the(power):
    return exp(2, power)

two_to_the(3)

# 8
```

- **functools.partial을 사용하는 것은 다른 접근 방식**

```
from functools import partial

two_to_the = partial(exp, 2)      # is now a function of one variable
print(two_to_the(3))              # 8

square_of = partial(exp, power=2)
print(square_of(3))    # 9
```

- 또한 map, reduce 및 filter를 사용하여 이해를 나열하는 기능적 대안을 제공하기도 합니다:
- 항상 map 사용하고 가능하면 reduce하고 filtering합니다.

Map

```
def double(x):
    return 2 * x

xs = [1, 2, 3, 4]

# 리스트 컴프리헨션을 사용하여 각 요소를 두 배로 만듭니다.
twice_xs = [double(x) for x in xs]

# map 함수를 사용하여 각 요소를 두 배로 만듭니다.
twice_xs = map(double, xs)

# partial 함수를 사용하여 map 함수에 double 함수를 적용합니다.
list_doubler = partial(map, double)
# list_doubler를 사용하여 xs의 각 요소를 두 배로 만듭니다.
twice_xs = list_doubler(xs)
```

```
def multiply(x, y): return x * y

products = map(multiply, [1, 2], [4, 5])    # [1 * 4, 2 * 5] = [4, 10]
list(products)

# [4,10]
```

```
def multiply(x, y, z): return x * y * z

products = map(multiply, [1, 2], [4, 5], [10, 20])    # [1 * 4 * 10, 2 * 5 * 20]
list(products)

# [40, 200]
```

Filter

```
def is_even(x):
    """x가 짝수이면 True, 홀수이면 False를 반환합니다."""
    return x % 2 == 0

xs = [1, 2, 3, 4]

# 리스트 컴프리헨션을 사용하여 짝수만 필터링합니다.
x_evens = [x for x in xs if is_even(x)]
print(x_evens)

# filter 함수를 사용하여 짝수만 필터링합니다.
x_evens = filter(is_even, xs)
print(list(x_evens))

# partial 함수를 사용하여 filter 함수에 is_even 함수를 적용합니다.
list_evener = partial(filter, is_even)
# list_evener를 사용하여 xs에서 짝수만 필터링합니다.
x_evens = list_evener(xs)
print(list(x_evens))
```

```
[2, 4]
[2, 4]
```

Reduce

```
def multiply(x, y): return x * y

xs = [1,2,3]
x_product = reduce(multiply, xs)
print(x_product)
list_product = partial(reduce, multiply)
x_product = list_product(xs)
print(x_product)
```

```
6
```

```
6
```

Enumerate

- 목록에서 반복하고 해당 요소와 인덱스를 모두 사용하려면 다음과 같이 하십시오:

```
documents = ["I", "am", "a", "boy"]
# not Pythonic
for i in range(len(documents)):
    document = documents[i]
    print(i, document)

# also not Pythonic
i = 0
for document in documents:
    print(i, document)
    i += 1
```

```
0 I
1 am
2 a
3 boy
0 I
1 am
2 a
3 boy
```

- Pythonic solution은 열거형으로 튜플(인덱스, 요소)을 생성합니다:

```
for i, document in enumerate(documents):
    print(i, document)
```

```


0 I
1 am
2 a
3 boy
```

```
for i in range(len(documents)): print(i)    # not Pythonic

for i, _ in enumerate(documents): print(i)  # Pythonic
```

```


0
1
2
3
0
1
2
3
```

Zip & Unzip

- 둘 이상의 목록을 함께 압축합니다.
- zip은 여러 목록을 해당 요소의 튜플 단일 목록으로 변환합니다:

```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]
list(zip(list1, list2))        # is [('a', 1), ('b', 2), ('c', 3)]
```

- **이상한 속임수를 사용하여 List를 "Unzip"할 수도 있습니다:**

```
pairs = [('a', 1), ('b', 2), ('c', 3)]
letters, numbers = zip(*pairs)
print(letters, numbers)

# ('a', 'b', 'c') (1, 2, 3)
```

```
pairs = [('a', 1), ('b', 2), ('c', 3)]
letters, numbers = zip(('a', 1), ('b', 2), ('c', 3))
print(letters, numbers)

# ('a', 'b', 'c') (1, 2, 3)
```