Python 특징

- 인터프리터 기반의 객체지향 언어
- 플랫폼에 구애받지 않는 언어
- 동적 타이핑 방식의 언어
- 리플렉션이 지원하는 언어
- 확장성이 뛰어난 언어

Number(숫자 자료형)

```
# 기본적인 사칙연산

print(5 + 6)        # 11

print(5 - 2)        # 3

print(3 * 8)        # 24

print(3 ** 3)       # 27 제곱

print(8 / 2)        # 4.0 float형

print(8 // 2)       # 4 int형

print(8 % 3)        # 2 나머지
```

String(문자열 자료형)

```
test = “Hello World!”
print(test)          # Hello World!

test = ‘Hello’
print(test)          # Hello

test = ‘I don\’t need Coke!’
print(test)          # I don’t need Coke!

test = “I don’t need Coke!”
print(test)          # I don’t need Coke!
```

“, ‘ 로 감싸진 문자열을 string으로 인식합니다.
싱글쿼터 혹은 더블쿼터로 문자열을 사용하려면 앞에 \(역슬래시)가 들어가야 합니다.
다른 방법으로는 더블쿼터로 문자열을 감싸고 문자열 내에서 싱글쿼터를 사용하는 것입니다.

String(문자열 자료형)

```
test = r'C: \Nature'
print(test)              # C: |Nature
```

r' '로 문자열을 감싸주게 되면 raw라는 뜻으로 아무 의미없는 문자열이라는 것을 나타내줍니다.

```
first = 'Myungseo'
last = 'Kang'

print(first + last)              # Myungseo Kang
print(last * 5)                  # KangKangKangKangKang
```

+ 기호를 이용해서 문자열을 합치는 것이 가능합니다.
또한 * 기호를 이용해서 문자열을 반복하는 것이 가능합니다.

Slicing String(문자열 슬라이싱)

```
test_str = 'Leopold'

print(test_str[0])          # L
print(test_str[1])          # e
print(test_str[-1])         # d
print(test_str[-2])         # l
```

List의 인덱스 부분에 음수를 넣어서 오른쪽으로 가져올 수 있습니다.
주의할 점은 음수로 인덱싱할 경우에는 0부터 시작이 아니라 1부터 시작합니다.

```
print(test_str[2:5])        # opo
print(test_str[3:6])        # pol
print(test_str[ :5])        # Leopo
print(test_str[3: ])        # pold
```

이렇게 범위를 인덱스로 지정해서 호출하는 것도 가능합니다.
주의할 점은 콜론 앞의 숫자는 포함되지만 뒤의 숫자는 포함되지 않습니다.
시작지점을 지정하지 않으면 처음부터 콜론 뒷 부분 숫자의 인덱스까지 출력하고,
끝지점을 지정하지 않으면 콜론 앞 부분 숫자부터 끝까지 출력합니다.

if, elif, else(조건문)

```
test = 'Leopold'

if name is 'Myungseo':
        print('Hello Myungseo')
elif name is 'Kang':
        print('Hello Kang!')
else:
        print('Hello Everyone!')
```

```
if 조건문:
        코드
elif 조건문2:
        코드
else:
        코드
```

특이한 점은 C언어처럼 else if를 쓰는 것이 아니라, elif를 쓴다는 것입니다.

List(리스트 자료형)

```
a = [ ]                  # a= list( )와 동일
b = [1, 3, 5]
c = [ 'Leopold' , 'Myungseo' , 'Kang' , 'L3opold7' ]
d = [7 , 9 , [ 'Myungseo' , 'L3pold7'] ]
```

List는 배열이라고 생각하면 됩니다.
List 안에는 여러가지 자료형을 담을 수 있습니다.
List에도 Slicing String에서 말한 것들을 적용할 수 있습니다.

```
print(b[-1])                 # 5
print(c[-2])                 # Kang
print(d[-1] [0])             # Myungseo
```

이중 List에서 인덱싱은 다음과 같이 할 수 있습니다.

List(리스트 자료형)

```
# List 값 수정
test = [1, 2, 3, 4, 5]
test[3] = 6

print(test)          # [1, 2, 3, 6, 5]
```

이렇게 인덱스를 지정해서 직접 값을 바꿔줄 수 있습니다.

```
# List  연속된 값으로 변경
test = [1, 2, 3, 4, 5]
test[2 : 3] = ['a', 'b', 'c']

print(test)          # [1, 2, 'a', 'b', 'c', 4, 5]
```

2이상 3미만의 인덱스 부분에 a,b,c List를 변경해주는 것입니다.

List(리스트 자료형)

```
# List 요소 삭제
test = ['a', 'b', 'c', 'd', 'e']
test[2:4] = [ ]

print(test)          # ['a', 'b', 'c']

# del 함수 사용
test = ['a', 'b', 'c', 'd', 'e']
del test[2]

print(test)          # ['a', 'b', 'd', 'e']
```

del 함수를 사용해서 삭제할 수도 있습니다.

```
test = ['a', 'b', 'c', 'd', 'e']
del test[2:4]

print(test)          # ['a', 'b', 'e']
```

마찬가지로 인덱스를 범위로 지정하는 것 또한 가능합니다.

List 내장 함수들!

```
test = [1, 2]
test.append(3)      # 맨 뒤에 값 추가

print(test)         # [1, 2, 3]
```

append(x) 함수는 인자를 1개밖에 받지 않기 때문에 여러 개의 인자를 넘겨줄 경우 에러가 발생합니다.

```
test = [3, 1 , 2 , 5 , 4]
test.sort( )
print(test)         # [1, 2, 3, 4, 5]

test.sort(reverse = True)
print(test)         # [5, 4, 3, 2, 1]
```

sort() 함수는 List를 자동으로 정렬해줍니다.
역순으로 정렬하기 위해서는 sort 함수에 reverse 옵션을 True로 설정해주면 됩니다.

List 내장 함수들!

```
test = [3, 1 , 2]
test.reverse( )

print(test)            # [2, 1, 3]
```

reverse() 함수는 현재의 List를 역순으로 뒤집어 줍니다.
정렬은 하지 않고 현재의 List를 역순으로 뒤집어 줍니다.

```
test = [1, 2 , 3 , 4 , 5]
print(test.index(3))          # 2
print(test.index(5))          # 4
```

index(x) 함수는 x라는 값이 있는 경우, x의 인덱스를 반환해주는 함수입니다.

List 내장 함수들!

```
test = [1, 2 , 3 , 4 , 5]
test.insert(0, 6)

print(test)            # [6, 1, 2, 3, 4, 5]
```

insert(x, y) 함수는 x 위치에 y라는 값을 삽입해주는 함수입니다.

```
test = [1, 2 , 3 , 4 , 5]
test.remove(3)

print(test)            # [1, 2, 4, 3]
```

remove(x) 함수는 첫 번째로 나오는 x라는 값을 List에서 삭제해주는 함수입니다.

List 내장 함수들!

```
test = [1, 2 , 3]

print(test.pop( ))   # 3
print(test)          # [1, 2]
```

pop( ) 함수는 List의 가장 마지막 인덱스의 값을 반환해주고 그 값을 삭제해주는 함수입니다.
위의 예제에서 굳이 3이라는 값이 필요없을 경우에는 print( ) 함수를 빼도 상관없습니다.

```
test = [1, 2 , 3 , 1 , 1]

print(test.count(1))          # 3
```

count(x) 함수는 x라는 값이 List 안에 몇 개나 있는지 반환해주는 함수입니다.

List 내장 함수들!

```
test = [1, 2 , 3]
test.extend([4, 5, 6])

print(test)             # [1, 2, 3, 4, 5, 6]
```

extend(x) 함수는 x 부분에 List를 받아서 원래의 List와 병합시켜주는 함수입니다.

List에서는 이제까지 봤던 내장 함수들을 사용할 수 있습니다.
여기에 더해서 len( ) 함수로 List값들의 개수를 얻을 수 있습니다.

Tuple(튜플 자료형)

```
tp1 = ( )

tp2 = (1, )

tp3 = (1, 2 , 3 , 4 , 5)

tp4 = (1, 2 , (3 , 4 , 5))

tp5 = 1, 2 , 3
```

Tuple은 조금 특이한 List라고 해도 무방할 정도로 List와 성격이 비슷합니다.
List는 [ ] 대괄호로 묶이지만 Tuple은 ( ) 소괄호로 묶입니다.
1개의 요소만을 가질 때 튜플은 tp2와 같이 뒤에 반드시 콤마(,)가 와야 합니다.
또한 tp5처럼 괄호를 생략해도 괜찮습니다.
Tuple과 List의 가장 큰 차이점은 Tuple은 값을 변경할 수 없다 입니다.
그래서 값의 변화를 원하지 않는 List의 경우에는 Tuple로 선언하는 것이 바람직합니다.

Tuple(튜플 자료형)

```
tp1 = (1, 2 , 3)
tp2 = (4 , 5, 6)

print(tp1 [2])              # 3
print(tp1 [1: ])            # (2, 3)
print(tp1+ tp2)             # (1, 2, 3, 4, 5, 6)
print(tp2 * 2)              # (4, 5, 6, 4, 5, 6)
```

Tuple은 인덱싱, 슬라이싱, 병합, 반복 모두 가능합니다.

Dictionary(딕셔너리 자료형)

```
dic1 = dict( )
dic2 = { 'k1' : 'v1' , 'k2' : 'v2' , 'k3' : 'v3' }
dic3 = dict([( 'name' , 'L3pold7' ) , ( 'phone' , '010-1234-5678' )])
dic4 = dict(firstname = 'Myungseo' , lastname = 'Kang' )
dic5 = { 'ls' : [ 'a' , 'b' , 'c' ]}

printf(dic 2)                  #  {'k1' : 'v1' , 'k2' : 'v2' , 'k3' : 'v3'}
print(dic2[ 'k2' ])            # v2
print(dic3)                    # {'phone' : '010-1234-5678' , 'name' : 'L3pold7'}
print(dic3[ 'name' ]           # L3pold7
print(dic4)                    # {'firstname' : 'Myungseo' , 'lastname' : 'Kang'}
print(dic4[ 'firstname' ])     # Myungseo
print(dic5[ 'ls' ])            # ['a' , 'b' , 'c']
```

Dictionary는 키 = 값 형태로 이루어진 자료형입니다.
이렇게 대응 관계를 나내는 자료형을 연관 배열 혹은 Hash라고 합니다. 대표적인 예로는 루비의 Hash가 있습니다.
빈 Dictionary를 만들 땐 dict( ) 함수를 사용하면 됩니다.
그리고 value 값을 호출할 때는 Dictionary 이름['키값']으로 호출하게 되면 값을 얻을 수 있습니다.
또한 Dictionary의 값으로 List도 넣을 수 있습니다.

Dictionary(딕셔너리 자료형)

```
test = { 1 : 'first' }
test[2] = 'second'

print(test)          # {2 : 'second' , 1: 'first'}
```

Dictionary는 간단하게 키 값을 지정해주고 추가해주면 됩니다.

```
test = { 1 : 'first' , 2 : 'second' , 3 : 'third' }

del test[2]
print(test)          # {1 : 'first' , 3: 'third'}
```

삭제는 List에서도 사용했듯이 del( ) 함수를 사용하면 됩니다.

Dictionary(딕셔너리 자료형)

```
test = { 'name' : 'Myungseo' , 'nickname' : 'L3opold7' , 'birthday' : '0523' }

print(test.keys( ))          # dict_keys([ 'name' , 'nickname' , 'birthday'])
print(test.values( ))        # dict_values([ 'Myungseo' , 'L3opold7' , '0523'])
print(test.items( ))         # dict_items([('nickname', 'L3opold7') , ('name' , 'Myungseo') , ('birthday' , '0523')])
```

keys( ), values( ) 함수를 통해서 딕셔너리의 key 혹은 value를 dict_key 혹은 dict_values 객체로 얻을 수 있습니다.
items( ) 함수는 key와 value를 Tuple을 사용해서 묶은 값을 dict_items라는 객체로 반환해줍니다.

```
test = { 'name' : 'Myungseo' , 'nickname' : 'L3opold7' , 'birthday' : '0523' }

test.clear( )
print(test)          # { }
```

clear( ) 함수를 이용해서 모두 지워버릴 수 있다.

Dictionary(딕셔너리 자료형)

```
test = { 'name' : 'Myungseo' , 'nickname' : 'L3opold7' , 'birthday' : '0523' }

print(test.get( 'no_key' ))          # None
print(test.get( 'name' )             # Myungseo
print(test[ 'name' ])                # Myungseo
print(test[ 'no_key' ])              # Error
```

test['no_key']의 경우에는 Error를 내뱉지만 test.get('no_key')는 None 객체를 반환하기 때문에
get(x,y) 함수를 쓰는 것이 더 적절하다.
get(x , y) 함수는 Dictionary안에 x라는 키 값이 없을 경우 y라는 디폴트 값을 반환해줍니다.

```
test = { 'name' : 'Myungseo' , 'nickname' : 'L3opold7' , 'birthday' : '0523' }

print( 'name' in test)          # True
print( 'no_key' in test)        # False
```

Set(집합 자료형)

```
s = set ([ 1, 2, 3, 4, 5 ])
print(s)                          # {1, 2, 3, 4, 5}

hello = set ( 'Hello World!' )
print(hello)                      # {' ', 'H', '!', 'e', 'l', 'o', 'd', 'w', 'r'}
```

집합 자료형인 Set 입니다. 말 그래도 집합을 나타내기 위한 자료형입니다.
특징으로는 중복을 허용하지 않고, 순서가 없다는 것이 있습니다.
List와 Tuple은 순서가 있기 때문에 인덱싱을 통해 원하는 값을 가져올 수 있었지만,
Set은 Dictionary와 비슷하게 순서가 없는 자료형이기 때문에 인덱싱이 불가능합니다.
만약 Set에서 인덱싱을 하고 싶다면 List나 Tuple로 형 변환을 시킨 뒤에 해야합니다.

아무래도 Set은 집합 자료형이다보니 교집합, 차집합, 합집합 등 집합 연산에 있어 매우 유리합니다.

Set(집합 자료형)

```
set1 = set([ 1, 2, 3, 4, 5])
set2 = set([ 5, 6, 7, 8, 9, 0])

print(set1  & set2)          # { 5, 6 }
print(set1  | set2)          # { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
print(set1  -  set2)         # { 1, 2, 3, 4 }
print(set2  -  set1)         # { 0, 8, 9, 7 }
```

차례대로 교집합, 합집합, set1-set2 차집합, set2-set1 차집합 입니다.

```
set1 = set([ 1, 2, 3, 4, 5])
set2 = set([ 5, 6, 7, 8, 9, 0])

print(set1.intersection(set2))          # { 5, 6 }
print(set1.union(set2))                 # { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
print(set1.difference(set2))            # { 1, 2, 3, 4 }
print(set2.difference(set1))            # { 0, 8, 9, 7 }
```

이렇게 Set 자료형의 내장 함수를 통해서 교집합, 차집합, 합집합을 구할 수 있습니다.

Set(집합 자료형)

```
set1 = set(1, 2 , 3, 4, 5)

set1.remove(3)
print(set1)                    # { 1, 2, 4, 5}
```

특정 값을 제거하고 싶을 경우에는 remove(x) 함수를 사용하면 됩니다.
x의 위치에는 제거하고 싶은 값을 적어줍니다.

for, while(반복문)

```
test = [ 1, 2, 3, 4, 5]

for i in test:
        print(i)
        ...
        1
        2
        3
        4
        5
        ...
```

for, while 문은 반복문입니다.
다음과 같이 i부분에는 변수, test 부분에는 List나 Tuple 혹은 String 같은 반복가능한 변수가 옵니다.

for, while(반복문)

```
test = [ (1, 2) , (3, 4 )]

for (i, j) in test:
        print(i + j)
        ...
        3
        7
        ...
```

이렇게도 사용이 가능합니다. C언어의 for문보다는 간편하게 사용 가능합니다.

```
for i in range( 0, 10 );
        print( i )
```

range 객체를 이용해서 쉽게 for문을 만들 수도 있습니다.

for, while(반복문)

```
test_list = [ 1, 2, 3, 4, 5 ]
result = [ ]

for num in test_list:
        result.append( num*3 )

print(result)                    # [ 3, 6, 9, 12, 15 ]
```

간단한 List 내장 함수와 for문을 이용한 예제입니다.
이런 코드를 아래와 같이 요약할 수 있습니다.

```
test_list = [ 1, 2, 3, 4, 5 ]

result = [ num * 3 for in test_list ]

print(result)                    # [ 3, 6, 9, 12, 15 ]
```

Function(함수)

```
def fuction_name( parameter ) :
        code here
```

함수(Function)는 여러 프로그래밍 언어에서 등장하는 개념입니다.
Python에서는 위와 같이 함수를 정의합니다.

```
function_name ( parameter )
```

호출할 때도 간단합니다.

Function(함수)

```
def hello( num ) :
        for i in range( 0 , num) :
                print( 'hello, ' +str( i ))
```

이렇게 함수를 설정해주면 hello 함수를 인자값으로 5를 넘겨서 실행해주게 되면

```
hello, 0
hello, 1
hello, 2
hello, 3
hello, 4
```

이렇게 나오게 됩니다.
range 객체로 0, num 까지의 iterable 객체를 만들어줬고, print 함수로 i 를 str 함수를 이용해 문자열로 바꾼 뒤, 출력해주고 있습니다.

Lambda(익명 함수)

```
multiply_number  = lambda x: x*1500
print(multiply_number( 5 ))              #7500
```

Lambda(람다)라는 익명함수는 일종의 작은 함수입니다.
지금까지 알아본 함수는 어떤 코드를 실행하고 함수를 정의하고, 인자를 전달하면서 함수를 호출합니다.
하지만 Lambda(람다)는 일반 함수와 달리 함수명이 없습니다.
그리고 여러 번 호출하지도 않고 1회성으로 사용하는 함수입니다.

```
func_range = list ( map ( lambda x: x*1000 , ( range ( 1, 6 ))))
print(func_range                  #[1000, 2000, 3000, 4000, 5000]
```

이 코드는 lambda 키워드로 Lambda(익명 함수)를 만들고, range 함수로 1~5 객체를 만들었습니다.
두개의 값을 map( )이라는 함수를 사용해서 Mapping(매핑)을 해주고 있습니다.
즉, x의 값이 1, 2, 3, 4, 5가 됩니다. 그 다음 list( ) 함수로 List 객체를 만듭니다.

File I/O(파일 입출력)

```
f = open ( “filename.txt” , ‘w’ )
f.write( “이 문자열은 파일에 기록됩니다.” )
f. close( )

# 파일 객체 = open(파일 이름, 모드)
```

파일 입출력은 다양하게 많이 쓰입니다.
예를 들면 엑셀 파일 같은 것들을 읽어들일 때나 텍스트 파일에서 원하는 문자열을 추출할 때 등등 쓰입니다.

모드의 종류에는 3가지가 있습니다.

r : 읽기 모드 - 파일을 읽어들일 때 사용함
w : 쓰기 모드 - 파일에 내용을 쓸 때 사용함
a : 추가 모드 - 파일의 끝에 새로운 내용을 추가할 때 사용함

File I/O(파일 입출력)

```
# w 모드 예제

f = open ( "example.txt" , 'w' )
for i in range( 1, 6 ) :                    # 1 이상부터 6 미만까지
        data = "%d번째 줄입니다. \n" % i
        f.write(data)
f.close( )
```

w모드로 열게 되면 해당 파일이 이미 존재하면 원래 있던 내용은 다 지워지고 새로 작성됩니다.
또한 해당 파일이 없을 경우 새로 생성을 하게 됩니다.

```
# r모드 예제

f = open ( "example.txt" , 'r' )
data = f.read( )
print(data)
f.close( )
```

파일을 전부 문자열로 리턴하는 read 메서드를 사용해서 파일 전체를 읽어들일 수 있다.

File I/O(파일 입출력)

```
# a모드 예제

f = open ( "example.txt" , 'a' )
for i in range( 6, 11 ) :
        data = "%d번째 줄입니다. \n" % i
        f.write(data)
f.close( )
```

이렇게 example.txt 뒤에 추가해 줄 수 있습니다.

```
with open( "example.txt" , "w" ) as f:
        f.write( " Keep Calm and Code Python" )
```

f.open, f.close를 따로 써줄 필요 없이 with 키워드로 이렇게 작성해 줄 수 있습니다.

try ~ except, finally(예외 처리)

```
try:
        num = 10 / 0
        print( num )
except Exception :
        printf( "0으로 나눌 수 없습니다." )
```

먼저 예외가 발생할 수 있는 부분은 try 문으로 묶은 다음에
except 문으로 예외를 지정해서 처리해 줄 수 있습니다.

```
try:
        num = 10 / 0
        print( num )
except ZeroDivisonError as e :
        printf( "0으로 나눌 수 없습니다." )
```

as e 부분은 말 그래도 alias e 라는 뜻으로 except 문 안에서
ZeroDivisionError 라는 이름을 e라는 이름으로 쓰겠다는 말입니다.
ZeroDivisionError는 Exception 클래스를 상속받기 때문에 Exception으로 호출해도 불리게 됩니다.

try ~ except, finally(예외 처리)

```
try:
        num = 10 / 5
        print( num )
except ZeroDivisonError as e :
        printf( "0으로 나눌 수 없습니다." )
finally:
        print( "실행 완료" )
```

finally 키워드는 예외가 발생하든 안하든 무조건 실행시킬 코드입니다.