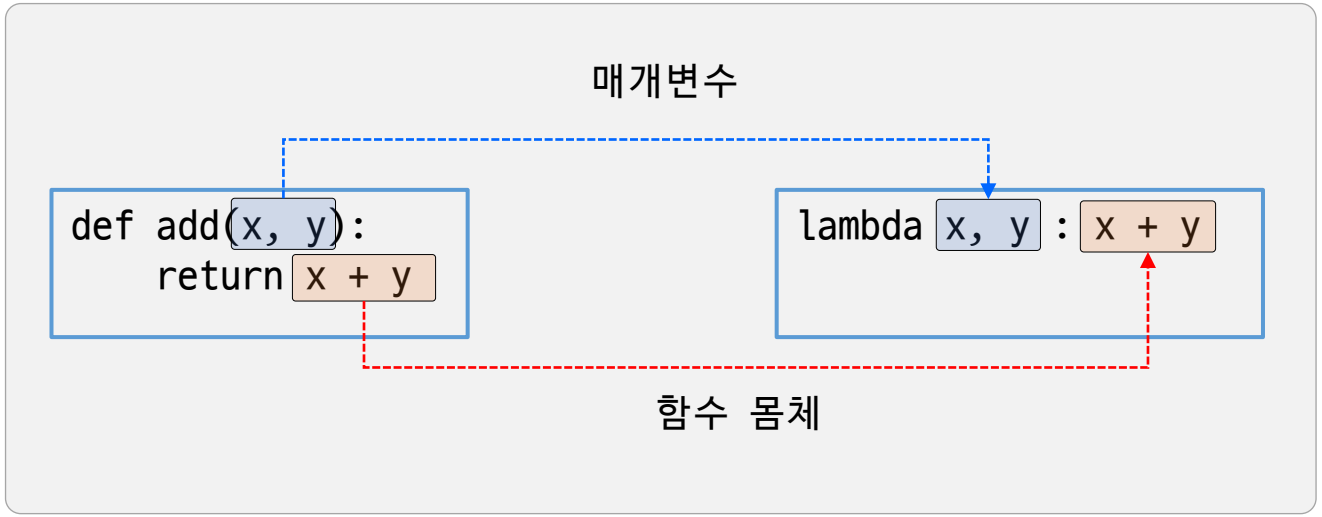
파이썬 람다

람다식 또는 람다 함수

- 람다식(무명 함수)
  - 이름은 없고 몸체만 있는 함수: 1회용으로 간단한 기능의 함수를 만드는 것
  - **lambda** 키워드로 생성
  - 콜론(:)을 기준으로 매개 변수와 수식(함수 몸체)으로 나뉨
  - 여러 개의 매개 변수를 가질 수 있지만, 반환값은 하나만 허용
  - 람다식은 결과를 자동으로 return

```
lambda 매개변수1, 매개변수2 : 함수 몸체(매개변수를 이용한 표현식)
```



lambda 함수

- 일반 함수 사용

```
def get_sum(x, y):
    return x + y

print("정수의 합:", get_sum(10, 20))
```

```
정수의 합: 30
```

- 람다식 사용

```
sum = lambda a, b : a + b # a + b의 결과를 리턴 (sum 변수가 받음)
```

```
print(sum(3, 7))
```

```
10
```

key 매개변수 #1

- 정렬에 사용되는 키를 개발자가 변경
  - sorted() 함수: 정렬된 리스트를 새롭게 생성하고 리턴

```
sorted(iterable, key=None, reverse=False)
```

  - ➢ key=함수
  - ➢ 정렬 방식: reverse=False (오름 차순: 기본값), reverse=True (내림 차순)

  - key=str.lower
    - str.lower()를 key로 사용: 문자열을 소문자로 변경 후 정렬

<lambda_ex01.py>

```
msg = "The health know not of their health, but only the sick"
sorted_list = sorted(msg.split(), key=str.lower)
print(sorted_list)
```

```
['but', 'health', 'health,', 'know', 'not', 'of', 'only', 'sick', 'The', 'the', 'their']
```

  - ➢ ‘The’와 ‘the’는 소문자로 변경해서 동일한 문자열로 취급
  - ➢ 단순히 ‘The’가 문장에서 먼저 나오기 때문에 ‘The’, ‘the’ 순서로 정렬됨

key 매개변수 #2

- 정렬 기준 및 정렬 방식
  - key=len
    - 문자열의 길이를 정렬 기준으로 설정
  - reverse=True
    - 내림 차순 정렬

<lambda_ex01.py>

```
# 문자열의 길이를 기준으로 내림차순 정렬
msg = "The health know not of their health, but only the sick"
descending_sorted_list = sorted(msg.split(), key=len, reverse=True)
print(descending_sorted_list)
```

문자열 길이의 역순

```
['health,', 'health', 'their', 'know', 'only', 'sick', 'The', 'not', 'but', 'the', 'of']
```

key 매개변수 #3: 여러 항목을 가지는 리스트 정렬

- 여러 항목을 가지는 리스트 정렬
  - key 매개변수에 lambda 사용
    - lambda 함수: 다양한 정렬 기준을 설정

<lambda_ex01.py>

```
students = [('Alice', 3.9, 20160303),
            ('Bob', 3.0, 20160302),
            ('Charlie', 4.3, 20160301)]

# 학번(students[2])을 기준으로 오름차순 정렬
sorted_students1 = sorted(students, key = lambda s: s[2])
print(sorted_students1)

# 학점(students[1])을 기준으로 내림 차순 정렬
sorted_students2 = sorted(students, key = lambda s: s[1], reverse=True)
print(sorted_students2)
```

입력 파라미터 (students)

s[1]을 key로 설정

```
[('Charlie', 4.3, 20160301), ('Bob', 3.0, 20160302), ('Alice', 3.9, 20160303)]
[('Charlie', 4.3, 20160301), ('Alice', 3.9, 20160303), ('Bob', 3.0, 20160302)]
```

```
lambda s: s[2]
```

```
def temp(s):
    return s[2]
```

key 매개변수 #4: 람다식을 활용한 객체 정렬

- 객체 리스트의 정렬 기준 설정
  - Student 객체의 name 값을 기준으로 리스트를 오름차순 정렬

<lambda_ex02.py>

```
class Student:
    def __init__(self, name, grade, number):
        self.name = name
        self.grade = grade
        self.number = number

    def __repr__(self):
        return f'({self.name}, {self.grade}, {self.number})'

# Student 객체 리스트 생성
students = [Student('홍길동', 3.9, 20240303),
            Student('김유신', 3.0, 20240302),
            Student('박문수', 4.3, 20240301)]

print(students[0])

sorted_list = sorted(students, key=lambda s: s.name)
print(sorted_list)
```

카드 게임 과제에서 엄청 써야됨!!!!!!!!!!

객체를 문자열로 표현하는 함수

Student 객체 출력:
Student 클래스에 __repr__()에 정의된 형태로 출력

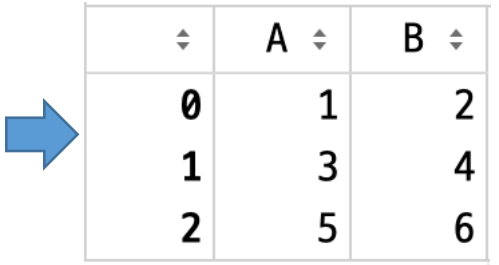
```
(홍길동, 3.9, 20240303)
[(김유신, 3.0, 20240302), (박문수, 4.3, 20240301), (홍길동, 3.9, 20240303)]
```

DataFrame에 일반 함수 적용(apply)

- apply(함수명)함수에 일반 함수 적용
  - DataFrame의 컬럼에 연산 일괄 적용

<lambda_ex03.py>

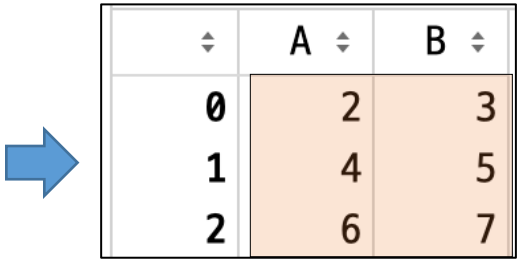
```
import pandas as pd
import numpy as np

df = pd.DataFrame([[1, 2], [3, 4], [5, 6]], columns=['A', 'B '])
print(df)
```



	A	B
0	1	2
1	3	4
2	5	6

```
def plus_one(x):
    x = x + 1
    return x
```

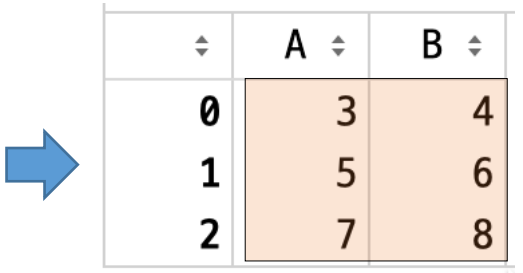
```
df['A'] = df['A'].apply(plus_one)
df['B'] = df['B'].apply(plus_one)
print(df)
```



	A	B
0	2	3
1	4	5
2	6	7

  - DataFrame 전체에 plus_one()함수 일괄 적용

```
df = df.apply(plus_one)
df
```

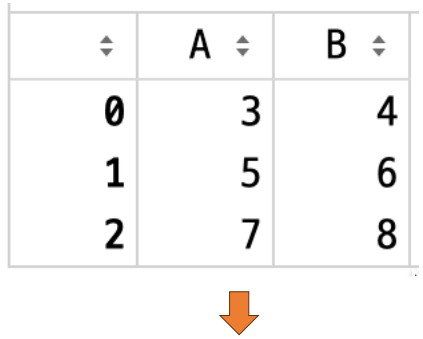


	A	B
0	3	4
1	5	6
2	7	8

DataFrame에 lambda 함수 적용 #1

- apply() 및 lambda 함수 적용
  - apply(람다함수)

```
DataFrame.apply(람다함수)
```

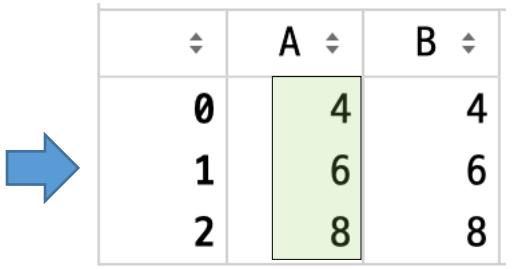


	A	B
0	3	4
1	5	6
2	7	8

  - 컬럼['A']에 lambda 함수 적용

<lambda_ex03.py>

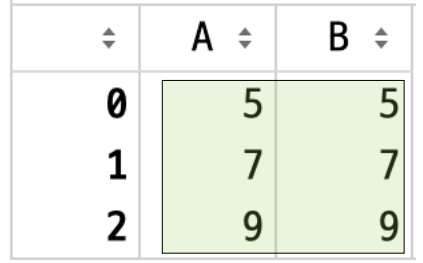
```
df['A'] = df['A'].apply(lambda x : x + 1)
print(df)
```



	A	B
0	4	4
1	6	6
2	8	8

  - 데이터프레임 전체에 lambda 함수 적용

```
df = df.apply(lambda x : x + 1)
print(df)
```



	A	B
0	5	5
1	7	7
2	9	9

DataFrame에 lambda 함수 적용 #2

- 특정 컬럼들에 apply(lambda 함수) 적용
  - 새로운 컬럼 추가(['C'])

<lambda_ex03.py>

```
df['C'] = [10, 20, 30] # 새로운 컬럼 추가
print(df)
```

	A	B	C
0	5	5	10
1	7	7	20
2	9	9	30

  - 두 개의 컬럼(['A', 'C'])에 lambda 함수 적용

```
df[['A', 'C']] = df[['A', 'C']].apply(lambda x : x * 10)
df
```

	A	B	C
0	50	5	100
1	70	7	200
2	90	9	300

딕셔너리에 lambda 적용

- 딕셔너리 get(key)함수
  - 딕셔너리의 key에 해당하는 값(value)를 리턴

```
dict.get(key)
```

➢ 'key'에 대응하는 값이 없으면 None을 리턴하고 KeyError 발생

```
dict.get(key[, default])
```

➢ key에 대응하는 값이 없으면, default에 지정한 값을 리턴

get(*key*[, *default*])

key 가 딕셔너리에 있는 경우 *key* 에 대응하는 값을 돌려주고, 그렇지 않으면 *default* 를 돌려줍니다. *default* 가 주어지지 않으면 기본값 `None` 이 사용됩니다. 그래서 이 메서드는 절대로 KeyError 를 일으키지 않습니다.

https://docs.python.org/ko/3/library/stdtypes.html#dict

딕셔너리에 lambda 적용 예제 #1

- 딕셔너리 생성

<lambda_ex04.py>

```
import pandas as pd
addr_aliases = {'경기':'경기도', '경남':'경상남도', '경북':'경상북도', '충북':'충청북도',
                '서울시':'서울특별시', '부산특별시':'부산광역시', '대전시':'대전광역시 ',
                '부산시':'부산광역시', '충남':'충청남도', '전남':'전라남도', '전북':'전라북도'}
```

- dict.get(key)
  - ➢ 'key'에 대응하는 값이 없으면 None을 리턴

```
print(addr_aliases.get('경기'))
print(addr_aliases.get('대전')) # None을 리턴
print(addr_aliases.get('부산')) # '부산' key는 없음
print(addr_aliases.get('부산', '부산광역시 ')) # key에 '부산'이 없으면 '부산광역시' 리턴
```

```
경기도
None
None
부산광역시
```

딕셔너리에 lambda 적용 예제 #2

- DataFrame 생성

<lambda_ex04.py>

```
addr_df = pd.DataFrame([ ' 경기', '대전광역시', '경남', '경북', '충북', '충남',
                         '전북', '전남', '경상북도 ' ], columns=['시도'])
print(addr_df)
```

	시도
0	경기
1	대전광역시
2	경남
3	경북
4	충북
5	충남
6	전북
7	전남
8	경상북도

- lambda v: dict.get(key, default) 사용
  - dictionary에서 key에 해당하는 값이 없으면, default에 지정한 값을 리턴

```
addr_df['시도']= addr_df['시도'].apply(lambda v: addr_aliases.get(v, v))
addr_df
```

key

default 값

	시도
0	경기도
1	대전광역시
2	경상남도
3	경상북도
4	충청북도
5	충청남도
6	전라북도
7	전라남도
8	경상북도

  - addr_df에 있는 '대전광역시'의 경우, addr_aliases.get('대전광역시', '대전광역시')를 검색
  - addr_alises 딕셔너리에는 key에 '대전광역시' 항목이 없음
  - None을 리턴하지 않고, 두 번째 파라미터(디폴트값)인 '대전광역시'를 리턴

lambda를 대신할 일반 함수 구현

- lambda v : addr_aliases.get(v, v)를 일반 함수로 구현
  - get_dict_value(key)

<lambda_ex04.py>

```
def get_dict_value(key):
    if not addr_aliases.get(key): # key에 해당하는 value(값)이 없으면, key를 리턴
        print('key:{}에 해당되는 값이 없어서 {}를 반환함'.format(key, key))
        return key
    else:
        value = addr_aliases.get(key)
        #print('key:{}, value:{}'.format(key, value))
        return value

print(get_dict_value('대구'))
```

```
key:대구에 해당되는 값이 없어서 대구를 반환함
대구
```

일반 함수를 apply()에 적용 예제

- DataFrame에 apply(get_dict_value) 적용
  - addr_df1['시도']에서 한 라인씩 get_dict_value()함수에 전달

<lambda_ex04.py>

```
addr_df1 = pd.DataFrame(['경기', '대전광역시', '경남', '경북', '충북', '충남',
                         '전북', '전남', '경상북도'], columns=['시도'])

# lambda식 대신에. get_dict_value()함수 호출
addr_df1['시도']= addr_df1['시도'].apply(get_dict_value)
addr_df1
```

	시도
0	경기도
1	대전광역시
2	경상남도
3	경상북도
4	충청북도
5	충청남도
6	전라북도
7	전라남도
8	경상북도

map 함수

- map(function, iterable)
  - 반복 가능한 객체(리스트, 튜플: iterable)의 각 항목에 주어진 function을 적용한 다음 결과를 반환하는 함수

<lambda_ex05.py>

```
def square(n):
    return n*n


mylist = [1, 2, 3, 4, 5]
result = list(map(square, mylist))
print(result)
```

```
[1, 4, 9, 16, 25]
```

  - 문자열을 공백 기준으로 분리하고 각 항목에 내장함수 int()를 수행

```
int_list = list(map(int, input('정수를 입력하세요: ').split()))
print(int_list)
```

```
정수를 입력하세요: 1 2 3 4 5 6
[1, 2, 3, 4, 5, 6]
```

간단한 딕셔너리에 lambda 적용

- map(function, iterable) 함수
  - 리스트의 요소를 지정된 lambda 함수로 처리

<lambda_ex05.py>

```
d = {'a': 1, 'b': 2}

# map의 첫 번째 파라미터 function은 lambda 함수로 대체
values = map(lambda key: d[key], d.keys())
print(list(values))
```

```
[1, 2]
```

  - 369 게임

```
list_369 = list(map(lambda x : '짝' if x % 3 == 0 else x, range(1, 10)))
print(list_369)
```

```
[1, 2, '짝', 4, 5, '짝', 7, 8, '짝']
```

Questions?